This manual defines the core Racket language and describes its most prominent libraries. The companion manual *The Racket Guide* provides a friendlier (though less precise and less complete) overview of the language.

```
#lang racket/base package: base
#lang racket
```

Unless otherwise noted, the bindings defined in this manual are exported by the `racket/base` and `racket` languages.
<table>
<thead>
<tr>
<th>Section</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>4.6</td>
<td>Characters</td>
<td>281</td>
</tr>
<tr>
<td>4.6.1</td>
<td>Characters and Scalar Values</td>
<td>281</td>
</tr>
<tr>
<td>4.6.2</td>
<td>Character Comparisons</td>
<td>282</td>
</tr>
<tr>
<td>4.6.3</td>
<td>Classifications</td>
<td>286</td>
</tr>
<tr>
<td>4.6.4</td>
<td>Character Conversions</td>
<td>288</td>
</tr>
<tr>
<td>4.7</td>
<td>Symbols</td>
<td>289</td>
</tr>
<tr>
<td>4.7.1</td>
<td>Additional Symbol Functions</td>
<td>292</td>
</tr>
<tr>
<td>4.8</td>
<td>Regular Expressions</td>
<td>293</td>
</tr>
<tr>
<td>4.8.1</td>
<td>Regexp Syntax</td>
<td>294</td>
</tr>
<tr>
<td>4.8.2</td>
<td>Additional Syntactic Constraints</td>
<td>299</td>
</tr>
<tr>
<td>4.8.3</td>
<td>Regexp Constructors</td>
<td>300</td>
</tr>
<tr>
<td>4.8.4</td>
<td>Regexp Matching</td>
<td>304</td>
</tr>
<tr>
<td>4.8.5</td>
<td>Regexp Splitting</td>
<td>315</td>
</tr>
<tr>
<td>4.8.6</td>
<td>Regexp Substitution</td>
<td>316</td>
</tr>
<tr>
<td>4.9</td>
<td>Keywords</td>
<td>319</td>
</tr>
<tr>
<td>4.9.1</td>
<td>Additional Keyword Functions</td>
<td>321</td>
</tr>
<tr>
<td>4.10</td>
<td>Pairs and Lists</td>
<td>321</td>
</tr>
<tr>
<td>4.10.1</td>
<td>Pair Constructors and Selectors</td>
<td>321</td>
</tr>
<tr>
<td>4.10.2</td>
<td>List Operations</td>
<td>325</td>
</tr>
<tr>
<td>4.10.3</td>
<td>List Iteration</td>
<td>327</td>
</tr>
<tr>
<td>4.10.4</td>
<td>List Filtering</td>
<td>330</td>
</tr>
<tr>
<td>4.10.5</td>
<td>List Searching</td>
<td>333</td>
</tr>
<tr>
<td>4.10.6</td>
<td>Pair Accessor Shorthands</td>
<td>336</td>
</tr>
<tr>
<td>4.10.7</td>
<td>Additional List Functions and Synonyms</td>
<td>343</td>
</tr>
<tr>
<td>4.10.8</td>
<td>Immutable Cyclic Data</td>
<td>363</td>
</tr>
<tr>
<td>4.11</td>
<td>Immutable Pairs and Lists</td>
<td>365</td>
</tr>
<tr>
<td>4.11.1</td>
<td>Immutable Pair Constructors and Selectors</td>
<td>365</td>
</tr>
<tr>
<td>4.12</td>
<td>Vectors</td>
<td>366</td>
</tr>
<tr>
<td>4.12.1</td>
<td>Additional Vector Functions</td>
<td>370</td>
</tr>
<tr>
<td>4.13</td>
<td>Boxes</td>
<td>377</td>
</tr>
<tr>
<td>4.14</td>
<td>Hash Tables</td>
<td>378</td>
</tr>
<tr>
<td>4.14.1</td>
<td>Additional Hash Table Functions</td>
<td>393</td>
</tr>
<tr>
<td>4.15</td>
<td>Sequences and Streams</td>
<td>395</td>
</tr>
<tr>
<td>4.15.1</td>
<td>Sequences</td>
<td>395</td>
</tr>
<tr>
<td>4.15.2</td>
<td>Streams</td>
<td>417</td>
</tr>
<tr>
<td>4.15.3</td>
<td>Generators</td>
<td>424</td>
</tr>
<tr>
<td>4.16</td>
<td>Dictionaries</td>
<td>429</td>
</tr>
<tr>
<td>4.16.1</td>
<td>Dictionary Predicates and Contracts</td>
<td>430</td>
</tr>
<tr>
<td>4.16.2</td>
<td>Generic Dictionary Interface</td>
<td>432</td>
</tr>
<tr>
<td>4.16.3</td>
<td>Dictionary Sequences</td>
<td>445</td>
</tr>
<tr>
<td>4.16.4</td>
<td>Contracted Dictionaries</td>
<td>446</td>
</tr>
<tr>
<td>4.16.5</td>
<td>Custom Hash Tables</td>
<td>447</td>
</tr>
<tr>
<td>4.16.6</td>
<td>Passing keyword arguments in dictionaries</td>
<td>452</td>
</tr>
<tr>
<td>4.17</td>
<td>Sets</td>
<td>453</td>
</tr>
<tr>
<td>4.17.1</td>
<td>Hash Sets</td>
<td>453</td>
</tr>
<tr>
<td>4.17.2</td>
<td>Set Predicates and Contracts</td>
<td>456</td>
</tr>
</tbody>
</table>
12.11 File Inclusion ...................................................... 885
12.12 Syntax Utilities .................................................. 886
  12.12.1 Creating formatted identifiers ........................... 886
  12.12.2 Pattern variables ......................................... 888
  12.12.3 Error reporting ............................................. 888
  12.12.4 Recording disappeared uses ............................ 889
  12.12.5 Miscellaneous utilities ................................. 890

13 Input and Output .................................................... 893
  13.1 Ports .................................................................. 893
  13.1.1 Encodings and Locales .................................... 894
  13.1.2 Managing Ports .............................................. 895
  13.1.3 Port Buffers and Positions ............................... 897
  13.1.4 Counting Positions, Lines, and Columns ............ 900
  13.1.5 File Ports .................................................... 902
  13.1.6 String Ports ................................................ 909
  13.1.7 Pipes .......................................................... 913
  13.1.8 Structures as Ports ......................................... 914
  13.1.9 Custom Ports ................................................ 914
  13.1.10 More Port Constructors, Procedures, and Events . 934
  13.2 Byte and String Input ........................................ 952
  13.3 Byte and String Output ....................................... 964
  13.4 Reading .......................................................... 967
  13.5 Writing ........................................................... 975
  13.6 Pretty Printing .................................................. 982
    13.6.1 Basic Pretty-Print Options .............................. 984
    13.6.2 Per-Symbol Special Printing ........................... 985
    13.6.3 Line-Output Hook ......................................... 987
    13.6.4 Value Output Hook ....................................... 988
    13.6.5 Additional Custom-Output Support .................. 989
  13.7 Reader Extension ............................................... 990
    13.7.1 Readtables ................................................ 991
    13.7.2 Reader-Extension Procedures ........................ 996
    13.7.3 Special Comments ....................................... 997
  13.8 Printer Extension ............................................... 997
  13.9 Serialization ................................................... 1000
  13.10 Fast-Load Serialization ........................................ 1010
  13.11 Cryptographic Hashing ....................................... 1012

14 Reflection and Security ............................................. 1014
  14.1 Namespaces ..................................................... 1014
  14.2 Evaluation and Compilation .................................. 1024
  14.3 The racket/load Language ..................................... 1036
  14.4 Module Names and Loading ................................... 1037
    14.4.1 Resolving Module Names ............................... 1037
    14.4.2 Compiled Modules and References .................. 1041
1 Language Model

1.1 Evaluation Model

Racket evaluation can be viewed as the simplification of expressions to obtain values. For example, just as an elementary-school student simplifies

\[
1 + 1 = 2
\]

Racket evaluation simplifies

\[
(+ 1 1) \rightarrow 2
\]

The arrow \(\rightarrow\) replaces the more traditional \(=\) to emphasize that evaluation proceeds in a particular direction toward simpler expressions. In particular, a value, such as the number 2, is an expression that evaluation simplifies no further.

1.1.1 Sub-expression Evaluation and Continuations

Some simplifications require more than one step. For example:

\[
(- 4 (+ 1 1)) \rightarrow (- 4 2) \rightarrow 2
\]

An expression that is not a value can always be partitioned into two parts: a redex (“reducible expression”), which is the part that can change in a single-step simplification (highlighted), and the continuation, which is the evaluation context surrounding the redex. In \((- 4 (+ 1 1))\), the redex is \((+ 1 1)\), and the continuation is \((- 4 []\)), where [] takes the place of the redex as it is reduced. That is, the continuation says how to “continue” after the redex is reduced to a value.

Before some expressions can be evaluated, some or all of their sub-expressions must be evaluated. For example, in the application \((- 4 (+ 1 1))\), the application of \(-\) cannot be reduced until the sub-expression \((+ 1 1)\) is reduced. Thus, the specification of each syntactic form specifies how (some of) its sub-expressions are evaluated and then how the results are combined to reduce the form away.

The dynamic extent of an expression is the sequence of evaluation steps during which the expression contains the redex.

1.1.2 Tail Position

An expression \(expr1\) is in tail position with respect to an enclosing expression \(expr2\) if, whenever \(expr1\) becomes a redex, its continuation is the same as was the enclosing \(expr2\)’s
continuation.

For example, the \((+ 1 1)\) expression is \textit{not} in tail position with respect to \((- 4 (+ 1 1))\). To illustrate, we use the notation \(C[expr]\) to mean the expression that is produced by substituting \(expr\) in place of [] in some continuation \(C\):

\[
C[(- 4 (+ 1 1))] \rightarrow C[(- 4 2)]
\]

In this case, the continuation for reducing \((+ 1 1)\) is \(C[(- 4 [])]\), not just \(C\). The requirement specified in the first paragraph above is not met.

In contrast, \((+ 1 1)\) is in tail position with respect to \((\text{if} \ (\text{zero?} \ 0) (+ 1 1) 3)\) because, for any continuation \(C\),

\[
C[(\text{if} \ (\text{zero?} \ 0) (+ 1 1) 3)] \rightarrow C[(\text{if} \ #t (+ 1 1) 3)] \rightarrow C[(+ 1 1)]
\]

The requirement specified in the first paragraph is met. The steps in this reduction sequence are driven by the definition of \(\text{if}\), and they do not depend on the continuation \(C\). The “then” branch of an \(\text{if}\) form is always in tail position with respect to the \(\text{if}\) form. Due to a similar reduction rule for \(\text{if}\) and \(\#f\), the “else” branch of an \(\text{if}\) form is also in tail position.

Tail-position specifications provide a guarantee about the asymptotic space consumption of a computation. In general, the specification of tail positions accompanies the description of each syntactic form, such as \(\text{if}\).

### 1.1.3 Multiple Return Values

A Racket expression can evaluate to \textit{multiple values}, to provide symmetry with the fact that a procedure can accept multiple arguments.

Most continuations expect a certain number of result values, although some continuations can accept an arbitrary number. Indeed, most continuations, such as \((+ []) 1)\), expect a single value. The continuation \((\text{let-values} \ ([(x y) []]) expr)\) expects two result values; the first result replaces \(x\) in the body \(expr\), and the second replaces \(y\) in \(expr\). The continuation \((\text{begin} [] (+ 1 2))\) accepts any number of result values, because it ignores the result(s).

In general, the specification of a syntactic form indicates the number of values that it produces and the number that it expects from each of its sub-expressions. In addition, some procedures (notably \texttt{values}) produce multiple values, and some procedures (notably \texttt{call-with-values}) create continuations internally that accept a certain number of values.
1.1.4 Top-Level Variables

Given

\[ x = 10 \]

then an algebra student simplifies \( x + 1 \) as follows:

\[ x + 1 = 10 + 1 = 11 \]

Racket works much the same way, in that a set of top-level variables (see also §1.1.8 "Variables and Locations") are available for substitutions on demand during evaluation. For example, given

\[
\text{(define } x \ 10) \]

then

\[
(+ \ x \ 1) \rightarrow (+ \ 10 \ 1) \rightarrow 11
\]

In Racket, the way definitions are created is just as important as the way they are used. Racket evaluation thus keeps track of both definitions and the current expression, and it extends the set of definitions in response to evaluating forms such as `define`.

Each evaluation step, then, transforms the current set of definitions and program into a new set of definitions and program. Before a `define` can be moved into the set of definitions, its expression (i.e., its right-hand side) must be reduced to a value. (The left-hand side is not an expression position, and so it is not evaluated.)

Using `set!`, a program can change the value associated with an existing top-level variable:

\[
\text{(define } x \ 10) \ \\
\text{(begin } (\text{set! } x \ 8) \ x) \]
1.1.5 Objects and Imperative Update

In addition to set! for imperative update of top-level variables, various procedures enable the modification of elements within a compound data structure. For example, vector-set! modifies the content of a vector.

To explain such modifications to data, we must distinguish between values, which are the results of expressions, and objects, which hold the data referenced by a value.

A few kinds of objects can serve directly as values, including booleans, (void), and small exact integers. More generally, however, a value is a reference to an object stored somewhere else. For example, a value can refer to a particular vector that currently holds the value 10 in its first slot. If an object is modified via one value, then the modification is visible through all the values that reference the object.

In the evaluation model, a set of objects must be carried along with each step in evaluation, just like the definition set. Operations that create objects, such as vector, add to the set of objects:

```
→ objects: (define <o1> (vector 10 20))
defined:
  evaluate: (begin
              (define x (vector 10 20))
              (define y x)
              (vector-set! x 0 11)
              (vector-ref y 0))
→ objects: (define <o1> (vector 10 20))
defined:
  evaluate: (begin
              (define x <o1>)
              (define y x)
              (vector-set! x 0 11)
              (vector-ref y 0))
→ objects: (define <o1> (vector 10 20))
defined:
  evaluate: (begin (void)
                    (define y x)
                    (vector-set! x 0 11)
                    (vector-ref y 0))
```

15
→ objects: (define <o1> (vector 10 20))
   defined: (define x <o1>)
evaluate: (begin (define y x)
                 (vector-set! x 0 11)
                 (vector-ref y 0))
→ objects: (define <o1> (vector 10 20))
   defined: (define x <o1>)
evaluate: (begin (define y <o1>)
                 (vector-set! x 0 11)
                 (vector-ref y 0))
→ objects: (define <o1> (vector 10 20))
   defined: (define x <o1>)
   (define y <o1>)
evaluate: (begin (void)
                 (vector-set! x 0 11)
                 (vector-ref y 0))
→ objects: (define <o1> (vector 10 20))
   defined: (define x <o1>)
   (define y <o1>)
evaluate: (begin (vector-set! <o1> 0 11)
                 (vector-ref y 0))
→ objects: (define <o1> (vector 11 20))
   defined: (define x <o1>)
   (define y <o1>)
evaluate: (begin (void)
                 (vector-ref y 0))
→ objects: (define <o1> (vector 11 20))
   defined: (define x <o1>)
   (define y <o1>)
evaluate: (vector-ref <o1> 0)
→ objects: (define <o1> (vector 11 20))
   defined: (define x <o1>)
   (define y <o1>)
evaluate: 11
The distinction between a top-level variable and an object reference is crucial. A top-level variable is not a value, so it must be evaluated. Each time a variable expression is evaluated, the value of the variable is extracted from the current set of definitions. An object reference, in contrast, is a value and therefore needs no further evaluation. The evaluation steps above use angle-bracketed \(<o1>\) for an object reference to distinguish it from a variable name.

An object reference can never appear directly in a text-based source program. A program representation created with \texttt{datum->syntax}, however, can embed direct references to existing objects.

### 1.1.6 Garbage Collection

In the program state

\begin{verbatim}
objects: (define <o1> (vector 10 20))
         (define <o2> (vector 0))
defined: (define x <o1>)
evaluate: (+ 1 x)
\end{verbatim}

evaluation cannot depend on \(<o2>\), because it is not part of the program to evaluate, and it is not referenced by any definition that is accessible by the program. The object is said to not be \textit{reachable}. The object \(<o2>\) may therefore be removed from the program state by \textit{garbage collection}.

A few special compound datatypes hold \textit{weak references} to objects. Such weak references are treated specially by the garbage collector in determining which objects are reachable for the remainder of the computation. If an object is reachable \textit{only} via a weak reference, then the object can be reclaimed, and the weak reference is replaced by a different value (typically \texttt{#f}).

As a special case, a fixnum is always considered reachable by the garbage collector. Many other values are always reachable due to the way they are implemented and used: A character in the Latin-1 range is always reachable, because \texttt{equal?} Latin-1 characters are always \texttt{eq?}, and all of the Latin-1 characters are referenced by an internal module. Similarly, \texttt{null}, \texttt{#t}, \texttt{#f}, \texttt{eof}, and \texttt{#<void>} are always reachable. Values produced by \texttt{quote} remain reachable when the \texttt{quote} expression itself is reachable.

### 1.1.7 Procedure Applications and Local Variables

Given

\[ f(x) = x + 10 \]

an algebra student simplifies \( f(7) \) as follows:
\[ f(7) = 7 + 10 = 17 \]

The key step in this simplification is to take the body of the defined function \( f \) and replace each \( x \) with the actual value 7.

Racket procedure application works much the same way. A procedure is an object, so evaluating \( (f \ 7) \) starts with a variable lookup:

- **objects:** (define <p1> (lambda (x) (+ x 10)))
  - defined: (define f <p1>)
  - evaluate: (f 7)
- **objects:** (define <p1> (lambda (x) (+ x 10)))
  - defined: (define f <p1>)
  - evaluate: (<p1> 7)

Unlike in algebra, however, the value associated with a procedure argument variable can be changed in the body of a procedure by using \( \text{set!} \), as in the example \((\text{lambda } (x) (\begin{align*} \text{set! } & x \ 3) \ x)\))

Since the value associated with argument variable \( x \) can be changed, the value cannot be substituted for \( x \) when the procedure is first applied.

Instead, a new **location** is created for each variable on each application. The argument value is placed in the location, and each instance of the variable in the procedure body is replaced with the new location:

- **objects:** (define <p1> (lambda (x) (+ x 10)))
  - defined: (define f <p1>)
  - evaluate: (p1 7)
- **objects:** (define <p1> (lambda (x) (+ x 10)))
  - defined: (define f <p1>)
  - evaluate: (+ xloc 7)
- **objects:** (define <p1> (lambda (x) (+ x 10)))
  - defined: (define f <p1>)
  - evaluate: (+ 7 10)
- **objects:** (define <p1> (lambda (x) (+ x 10)))
  - defined: (define f <p1>)
  - evaluate: 17

A location is the same as a top-level variable, but when a location is generated, it (conceptually) uses a name that has not been used before and that cannot be generated again or accessed directly.

Generating a location in this way means that \( \text{set!} \) evaluates for local variables, including argument variables, in the same way as for top-level variables, because the local variable is always replaced with a location by the time the \( \text{set!} \) form is evaluated.
The location-generation and substitution step of procedure application requires that the argument is a value. Therefore, in \(((\text{lambda } (x) (+ x 10)) (+ 1 2))\), the \((+ 1 2)\) sub-expression must be simplified to the value 3, and then 3 can be placed into a location for \(x\). In other words, Racket is a call-by-value language.

Evaluation of a local-variable form, such as \((\text{let } ([x (+ 1 2)]) \text{ expr})\), is the same as for a procedure call. After \((+ 1 2)\) produces a value, it is stored in a fresh location that replaces every instance of \(x\) in \(\text{expr}\).

### 1.1.8 Variables and Locations

A variable is a placeholder for a value, and expressions in an initial program refer to variables. A top-level variable is both a variable and a location. Any other variable is always replaced by a location at run-time; thus, evaluation of expressions involves only locations. A single local variable (i.e., a non-top-level, non-module-level variable), such as an argument variable, can correspond to different locations during different applications.

For example, in the program

\[(\text{define } y (+ (\text{let } ([x 5]) x) 6))\]
both $y$ and $x$ are variables. The $y$ variable is a top-level variable, and the $x$ is a local variable. When this code is evaluated, a location is created for $x$ to hold the value 5, and a location is also created for $y$ to hold the value 11.

The replacement of a variable with a location during evaluation implements Racket’s *lexical scoping*. For example, when an argument variable $x$ is replaced by the location $x\text{loc}$, it is replaced *throughout* the body of the procedure, including any nested $\text{lambda}$ forms. As a result, future references to the variable always access the same location.

### 1.1.9 Modules and Module-Level Variables

Most definitions in Racket are in *modules*. In terms of evaluation, a module is essentially a prefix on a defined name, so that different modules can define the same name. That is, a module-level variable is like a top-level variable from the perspective of evaluation.

One difference between a module and a top-level definition is that a module can be *declared* without instantiating its module-level definitions. Evaluation of a $\text{require instantiates}$ (i.e., triggers the instantiation of) the declared module, which creates variables that correspond to its module-level definitions.

For example, given the module declaration

```
(module m racket
  (define x 10))
```

the evaluation of $(\text{require } m)$ creates the variable $x$ and installs 10 as its value. This $x$ is unrelated to any top-level definition of $x$ (as if it were given a unique, module-specific prefix).

**Phases**

The purpose of *phases* is to address the necessary separation of names defined at execution time versus names defined at expansion time.

A module can be instantiated in multiple phases. A phase is an integer that, like a module name, is effectively a prefix on the names of module-level definitions. Phase 0 is the execution-time phase.

A top-level $\text{require}$ instantiates a module at phase 0, if the module is not already instantiated at phase 0. A top-level $(\text{require } (\text{for-syntax } ...))$ instantiates a module at phase 1 (if it is not already instantiated at that phase); $\text{for-syntax}$ also has a different binding effect on further program parsing, as described in §1.2.3.4 “Introducing Bindings”.

Within a module, some definitions are already shifted by a phase: the $\text{begin-for-syntax}$ form is similar to $\text{begin}$, but it shifts expressions and definitions by a relative phase +1.
Likewise, the `define-for-syntax` form is similar to `define`, but shifts the definition by +1. Thus, if the module is instantiated at phase 1, the variables defined with `begin-for-syntax` are created at phase 2, and so on. Moreover, this relative phase acts as another layer of prefixing, so that `x` defined with `define` and `x` defined with `define-for-syntax` can co-exist in a module without colliding. A `begin-for-syntax` form can be nested within a `begin-for-syntax` form, in which case the inner definitions and expressions are in relative phase +2, and so on. Higher phases are mainly related to program parsing instead of normal evaluation.

If a module instantiated at phase `n` requires another module, then the required module is first instantiated at phase `n`, and so on transitively. (Module requirements cannot form cycles.) If a module instantiated at phase `n` requires another module `M for-syntax`, then `M` becomes available at phase `n+1`, and it later may be instantiated at phase `n+1`. If a module that is available at phase `n` (for `n>0`) requires another module `M for-template`, then `M` becomes available at phase `n-1`, and so on. Instantiations of available modules above phase 0 are triggered on demand as described in §1.2.3.9 “Module Expansion, Phases, and Visits”.

A final distinction among module instantiations is that multiple instantiations may exist at phase 1 and higher. These instantiations are created by the parsing of module forms (see §1.2.3.9 “Module Expansion, Phases, and Visits”), and are, again, conceptually distinguished by prefixes.

Top-level variables can exist in multiple phases in the same way as within modules. For example, `define` within `begin-for-syntax` creates a phase 1 variable. Furthermore, reflective operations like `make-base-namespace` and `eval` provide access to top-level variables in higher phases, while module instantiations (triggered by `require`) relative to such top-levels are in correspondingly higher phases.

**The Separate Compilation Guarantee**

When a module is compiled, its phase 1 is instantiated. This can, in turn, trigger the transitive instantiation of many other modules at other phases, including phase 1. Racket provides a very strong guarantee about this instantiation called “The Separate Compilation Guarantee”:

> Any effects of the instantiation of the module’s phase 1 due to compilation on the Racket runtime system are discarded.

The guarantee concerns *effects*. There are two different kinds of effects: internal and external.

Internal effects are exemplified by mutation. Mutation is the action of a function such as `set-box!`, which changes the value contained in the box. The modified box is not observable outside Racket, so the effect is said to be “internal.” By definition, internal effects are not detectable outside the Racket program.

External effects are exemplified by input/output (I/O). I/O is the action of a function such as
as `tcp-connect`, which communicates with the operating system to send network packets outside the machine running Racket. The transmission of these packets is observable outside Racket, in particular by the receiving computer or any routers in between. External effects exist to be detectable outside the Racket program and are often detectable using physical processes.

An effect is *discarded* when it is no longer detectable. For instance, the mutation of a box from 3 to 4 is discarded when it ceases to be detectable that it was ever changed and thus would still contain 3. Because external effects are intrinsically observable outside Racket, they are irreversible and cannot be discarded.

Thus, The Separate Compilation Guarantee only concerns effects like mutation, because they are exclusively effects “on the Racket runtime system” and not “on the physical universe.”

There are many things a Racket program can do that appear to be internal effects but are actually external effects. For instance, `bytes-set!` is typically an internal effect, except when the bytes are created by `make-shared-bytes`, which allocates in space observable by other processes. Thus, effects which modify those bytes are not discardable, so `bytes-set!`, in this case, has an external effect.

The opposite is also true: some things which appear to be external are actually internal. For instance, if a Racket program starts multiple threads and uses mutation to communicate between them, that mutation is purely internal, because Racket’s threads are defined entirely internally (they are not related to operating system threads).

Furthermore, whenever a Racket program calls an unsafe function, the Racket runtime system makes no promises about its effects. For instance, all foreign calls use `ffi/unsafe`, so all foreign calls are unsafe and their effects cannot be discarded by Racket.

Finally, The Separate Compilation Guarantee only concerns instantiations at phase 1 during compilation and not all phase 1 instantiations generally, such as when its phase 1 is required and used for effects via reflective mechanisms.

The practical consequence of this guarantee is that because effects are never visible, no module can detect whether a module it `require`es is already compiled. Thus, it can never change the compilation of one module to have already compiled a different module. In particular, if module A is shared by the phase 1 portion of modules X and Y, then any internal effects while X is compiled are not visible during the compilation of Y, regardless of whether X and Y are compiled during the same execution of Racket’s runtime system and regardless of the order of compilation.

The following set of modules demonstrate this guarantee. First, we define a module with the ability to observe effects via a box:

```
(module box racket/base
  (provide (all-defined-out))
  (define b (box 0)))
```
Next, we define two syntax transformers that use and mutate this box:

```racket
(module transformers racket/base
  (provide (all-defined-out))
  (require (for-syntax racket/base 'box))
  (define-syntax (sett stx)
    (set-box! b 2)
    #'(void))
  (define-syntax (gett stx)
    #'#,(unbox b)))
```

Next, we define a module that uses these transformers:

```racket
(module user racket/base
  (provide (all-defined-out))
  (require 'transformers)
  (sett)
  (define gott (gett)))
```

Finally, we define a second module that uses these transformers and the `user` module:

```racket
(module test racket/base
  (require 'box 'transformers 'user)
  (displayln gott)
  (displayln (gett))

  (sett)
  (displayln (gett))

  (displayln (unbox b)))
```

This module displays:

- 2, because the `(gett)` in module `user` expanded to 2.
- 0, because the effects of compiling `user` were discarded.
- 2, because the effect of `(sett)` inside `test` has not yet been discarded.
- 0, because the effects of `sett` at phase 1 are irrelevant to the phase 0 use of `b` in `(unbox b)`.

Furthermore, this display will never change, regardless of which order these modules are compiled in or whether they are compiled at the same time or separately.
In contrast, if these modules were changed to store the value of \( b \) in a file on the filesystem, then the program would only display 2.

The Separate Compilation Guarantee is described in more detail in the paper “Composable and Compilable Macros” [Flatt02], including informative examples. The paper “Advanced Macrology and the implementation of Typed Scheme” [Culpepper07] also contains an extended example of why it is important and how to design effectful syntactic extensions in its presence.

Cross-Phase Persistent Modules

Module declarations that fit a highly constrained form—including a `( #%declare #:cross-phase-persistent)` form in the module body—create cross-phase persistent modules. A cross-phase persistent module’s instantiations across all phases share the variables produced by the first instantiation of the module. Additionally, cross-phase persistent module instantiations persist across module registries when they share a common module declaration.

Examples:

```scheme
> (module cross '#%kernel
  (#%declare #:cross-phase-persistent)
  (#%provide x)
  (define-values (x) (gensym)))
> (module noncross '#%kernel
  (#%provide x)
  (define-values (x) (gensym)))
> (define ns (current-namespace))
> (define (same-instance? mod)
  (namespace-require mod)
  (define a
    (parameterize ([current-namespace (make-base-namespace)])
      (namespace-attach-module-declaration ns mod)
      (namespace-require mod)
      (namespace-variable-value 'x)))
> (define b
    (parameterize ([current-namespace (make-base-namespace)])
      (namespace-attach-module-declaration ns mod)
      (namespace-require mod)
      (namespace-variable-value 'x)))
> (eq? a b))
> (same-instance? 'noncross)
#f
> (same-instance? 'cross)
#t
```

The intent of a cross-phase persistent module is to support values that are recognizable after
phase crossings. For example, when a macro transformer running in phase 1 raises a syntax error as represented by an `exn:fail:syntax` instance, the instance is recognizable by a phase-0 exception handler wrapping a call to `eval` or `expand` that triggered the syntax error, because the `exn:fail:syntax` structure type is defined by a cross-phase persistent module.

A cross-phase persistent module imports only other cross-phase persistent modules, and it contains only definitions that bind variables to functions, structure types and related functions, or structure-type properties and related functions. A cross-phase persistent module never includes syntax literals (via `quote-syntax`) or variable references (via `#%variable-reference`). See §1.2.7 "Cross-Phase Persistent Module Declarations" for the syntactic specification of a cross-phase persistent module declaration.

A documented module should be assumed non–cross-phase persistent unless it is specified as cross-phase persistent (such as `racket/kernel`).

**Module Redeclarations**

When a module is declared using a name with which a module is already declared, the new declaration’s definitions replace and extend the old declarations. If a variable in the old declaration has no counterpart in the new declaration, the old variable continues to exist, but its binding is not included in the lexical information for the module body. If a new variable definition has a counterpart in the old declaration, it effectively assigns to the old variable.

If a module is instantiated in the current namespace’s base phase before the module is redeclared, the redeclaration of the module is immediately instantiated in that phase.

If the current inspector does not manage a module’s declaration inspector (see §14.10 “Code Inspectors”), then the module cannot be redeclared. Similarly, a cross-phase persistent module cannot be redeclared. Even if redeclaration succeeds, instantiation of a module that is previously instantiated may fail if instantiation for the redeclaration attempts to modify variables that are constant (see `compile-enforce-module-constants`).

**Submodules**

A `module` or `module*` form within a top-level `module` form declares a *submodule*. A submodule is accessed relative to its enclosing module, usually with a `submod` path. Submodules can be nested to any depth.

Although a submodule is lexically nested within a module, it cannot necessarily access the bindings of its enclosing module directly. More specifically, a submodule declared with `module cannot require` from its enclosing module, but the enclosing module can require the submodule. In contrast, a submodule declared with `module*` conceptually follows its enclosing module, so can `require` from its enclosing module, but the enclosing module cannot `require` the submodule. Unless a submodule imports from its enclosing module or vice versa, then visits or instantiations of the two modules are independent, and their implementations may even be loaded from bytecode sources at different times.
A submodule declared with `module` can import any preceding submodule declared with `module`. A submodule declared with `module*` can import any preceding module declared with `module*` and any submodule declared with `module`.

When a submodule declaration has the form `(module* name #f ... )`, then all of the bindings of the enclosing module’s bodies are visible in the submodule’s body, and the submodule implicitly imports the enclosing module. The submodule can provide any bindings that it inherits from its enclosing module.

### 1.1.10 Continuation Frames and Marks

Every continuation $C$ can be partitioned into continuation frames $C_1, C_2, ..., C_n$ such that $C = C_1[C_2[...[C_n]]]$, and no frame $C_i$ can be itself partitioned into smaller continuations. Evaluation steps add frames to and remove frames from the current continuation, typically one at a time.

Each frame is conceptually annotated with a set of continuation marks. A mark consists of a key and its value. The key is an arbitrary value, and each frame includes at most one mark for any given key. Various operations set and extract marks from continuations, so that marks can be used to attach information to a dynamic extent. For example, marks can be used to record information for a “stack trace” to be presented when an exception is raised, or to implement dynamic scope.

### 1.1.11 Prompts, Delimited Continuations, and Barriers

A prompt is a special kind of continuation frame that is annotated with a specific prompt tag (essentially a continuation mark). Various operations allow the capture of frames in the continuation from the redex position out to the nearest enclosing prompt with a particular prompt tag; such a continuation is sometimes called a delimited continuation. Other operations allow the current continuation to be extended with a captured continuation (specifically, a composable continuation). Yet other operations abort the computation to the nearest enclosing prompt with a particular tag, or replace the continuation to the nearest enclosing prompt with another one. When a delimited continuation is captured, the marks associated with the relevant frames are also captured.

A continuation barrier is another kind of continuation frame that prohibits certain replacements of the current continuation with another. Specifically, a continuation can be replaced by another only when the replacement does not introduce any continuation barriers. A continuation barrier thus prevents “downward jumps” into a continuation that is protected by a barrier. Certain operations install barriers automatically; in particular, when an exception handler is called, a continuation barrier prohibits the continuation of the handler from capturing the continuation past the exception point.
An escape continuation is essentially a derived concept. It combines a prompt for escape purposes with a continuation for mark-gathering purposes. As the name implies, escape continuations are used only to abort to the point of capture.

### 1.1.12 Threads

Racket supports multiple threads of evaluation. Threads run concurrently, in the sense that one thread can preempt another without its cooperation, but threads currently all run on the same processor (i.e., the same underlying operating system process and thread).

Threads are created explicitly by functions such as `thread`. In terms of the evaluation model, each step in evaluation actually deals with multiple concurrent expressions, up to one per thread, rather than a single expression. The expressions all share the same objects and top-level variables, so that they can communicate through shared state, and **sequential consistency** [Lamport79] is guaranteed (i.e., the result is consistent with some global sequence imposed on all evaluation steps across threads). Most evaluation steps involve a single step in a single thread, but certain synchronization primitives require multiple threads to progress together in one step; for example, an exchange of a value through a channel progresses in two threads simultaneously.

Unless otherwise noted, all constant-time procedures and operations provided by Racket are thread-safe in the sense that they are **atomic**: they happen as a single evaluation step. For example, `set!` assigns to a variable as an atomic action with respect to all threads, so that no thread can see a “half-assigned” variable. Similarly, `vector-set!` assigns to a vector atomically. Note that the evaluation of a `set!` expression with its subexpression is not necessarily atomic, because evaluating the subexpression involves a separate step of evaluation. Only the assignment action itself (which takes after the subexpression is evaluated to obtain a value) is atomic. Similarly, a procedure application can involve multiple steps that are not atomic, even if the procedure itself performs an atomic action.

The `hash-set!` procedure is not atomic, but the table is protected by a lock: see §4.14 "Hash Tables" for more information. Port operations are generally not atomic, but they are thread-safe in the sense that a byte consumed by one thread from an input port will not be returned also to another thread, and procedures like `port-commit-peeked` and `write-bytes-avail` offer specific concurrency guarantees.

In addition to the state that is shared among all threads, each thread has its own private state that is accessed through thread cells. A thread cell is similar to a normal mutable object, but a change to the value inside a thread cell is seen only when extracting a value from that cell in the same thread. A thread cell can be **preserved**: when a new thread is created, the creating thread’s value for a preserved thread cell serves as the initial value for the cell in the created thread. For a non-preserved thread cell, a new thread sees the same initial value (specified when the thread cell is created) as all other threads.

Futures and places offer different kinds of concurrency and parallelism, and they have
weaker guarantees about shared state. (Places can share state through functions like `make-shared-bytes`.) Each thread of evaluation in a future or place is constrained to behave consistent with the possibility of some other thread that might inspect any shared data starting at any point that a future or place starts. In the case that two futures or two places share state, each read or write operation to shared state corresponds to a read or write operation at the virtual-memory level, and the operations are constrained to the order they could be observed or affected by a thread. However, Racket does not enforce additional guarantees about reordering that might be performed at the virtual-memory level or below, except in the case of operations that specify such guarantees explicitly (e.g., `box-cas`).

### 1.1.13 Parameters

Parameters are essentially a derived concept in Racket; they are defined in terms of continuation marks and thread cells. However, parameters are also “built in,” due to the fact that some primitive procedures consult parameter values. For example, the default output stream for primitive output operations is specified by a parameter.

A parameter is a setting that is both thread-specific and continuation-specific. In the empty continuation, each parameter corresponds to a preserved thread cell; a corresponding parameter procedure accesses and sets the thread cell’s value for the current thread.

In a non-empty continuation, a parameter’s value is determined through a parameterization that is associated with the nearest enclosing continuation frame via a continuation mark (whose key is not directly accessible). A parameterization maps each parameter to a preserved thread cell, and the combination of the thread cell and the current thread yields the parameter’s value. A parameter procedure sets or accesses the relevant thread cell for its parameter.

Various operations, such as `parameterize` or `call-with-parameterization`, install a parameterization into the current continuation’s frame.

### 1.1.14 Exceptions

Exceptions are essentially a derived concept in Racket; they are defined in terms of continuations, prompts, and continuation marks. However, exceptions are also “built in,” due to the fact that primitive forms and procedures may raise exceptions.

An exception handler to catch exceptions can be associated with a continuation frame through a continuation mark (whose key is not directly accessible). When an exception is raised, the current continuation’s marks determine a chain of exception handler procedures that are consulted to handle the exception. A handler for uncaught exceptions is designated through a built-in parameter.

One potential action of an exception handler is to abort the current continuation up to an
enclosing prompt with a particular prompt tag. The default handler for uncaught exceptions, in particular, aborts to a particular tag for which a prompt is always present, because the prompt is installed in the outermost frame of the continuation for any new thread.

1.1.15 Custodians

A custodian manages a collection of threads, file-stream ports, TCP ports, TCP listeners, UDP sockets, byte converters, and places. Whenever a thread, etc., is created, it is placed under the management of the current custodian as determined by the current-custodian parameter.

Except for the root custodian, every custodian itself is managed by a custodian, so that custodians form a hierarchy. Every object managed by a subordinate custodian is also managed by the custodian’s owner.

When a custodian is shut down via custodian-shutdown-all, it forcibly and immediately closes the ports, TCP connections, etc., that it manages, as well as terminating (or suspending) its threads. A custodian that has been shut down cannot manage new objects. After the current custodian is shut down, if a procedure is called that attempts to create a managed resource (e.g., open-input-file, thread), then the exn:fail:contract exception is raised.

A thread can have multiple managing custodians, and a suspended thread created with thread/suspend-to-kill can have zero custodians. Extra custodians become associated with a thread through thread-resume (see §11.1.2 “Suspending, Resuming, and Killing Threads”). When a thread has multiple custodians, it is not necessarily killed by a custodian-shutdown-all. Instead, shut-down custodians are removed from the thread’s managing custodian set, and the thread is killed when its managing set becomes empty.

The values managed by a custodian are semi-weakly held by the custodian: a will can be executed for a value that is managed by a custodian; in addition, weak references via weak hash tables, ephemeron, or weak boxes can be dropped on the BC implementation of Racket, but not on the CS implementation. For all variants, a custodian only weakly references its subordinate custodians; if a subordinate custodian is unreferenced but has its own subordinates, then the custodian may be garbage collected, at which point its subordinates become immediately subordinate to the collected custodian’s superordinate (owner) custodian.

In addition to the other entities managed by a custodian, a custodian box created with make-custodian-box strongly holds onto a value placed in the box until the box’s custodian is shut down. However, the custodian only weakly retains the box itself, so the box and its content can be collected if there are no other references to them.

When Racket is compiled with support for per-custodian memory accounting (see custodian-memory-accounting-available?), the current-memory-use procedure can report a custodian-specific result. This result determines how much memory is occupied
by objects that are reachable from the custodian’s managed values, especially its threads, and including its sub-custodians’ managed values. If an object is reachable from two custodians where neither is an ancestor of the other, an object is arbitrarily charged to one or the other, and the choice can change after each collection; objects reachable from both a custodian and its descendant, however, are reliably charged to the custodian and not to the descendants, unless the custodian can reach the objects only through a descendant custodian or a descendant’s thread. Reachability for per-custodian accounting does not include weak references, references to threads managed by other custodians, references to other custodians, or references to custodian boxes for other custodians.

1.2 Syntax Model

The syntax of a Racket program is defined by

- a read pass that processes a character stream into a syntax object; and
- an expand pass that processes a syntax object to produce one that is fully parsed.

For details on the read pass, see §1.3 “The Reader”. Source code is normally read in read-syntax mode, which produces a syntax object.

The expand pass recursively processes a syntax object to produce a complete parse of the program. Binding information in a syntax object drives the expansion process, and when the expansion process encounters a binding form, it extends syntax objects for sub-expression with new binding information.

1.2.1 Identifiers, Binding, and Scopes

An identifier is a source-program entity. Parsing (i.e., expanding) a Racket program reveals that some identifiers correspond to variables, some refer to syntactic forms (such as lambda, which is the syntactic form for functions), some refer to transformers for macro expansion, and some are quoted to produce symbols or syntax objects. An identifier binds another (i.e., it is a binding) when the former is parsed as a variable or syntactic form and the latter is parsed as a reference to the former; the latter is bound.

For example, as a fragment of source, the text

\[(\text{let } ([x 5]) \ x)\]

includes two identifiers: \texttt{let} and \texttt{x} (which appears twice). When this source is parsed in a context where \texttt{let} has its usual meaning, the first \texttt{x} binds the second \texttt{x}.

§4.2 “Identifiers and Binding” in The Racket Guide introduces binding.
Bindings and references are determined through scope sets. A *scope* corresponds to a region of the program that is either in part of the source or synthesized through elaboration of the source. Nested binding contexts (such as nested functions) create nested scopes, while macro expansion creates scopes that overlap in more complex ways. Conceptually, each scope is represented by a unique token, but the token is not directly accessible. Instead, each scope is represented by a value that is internal to the representation of a program.

A *form* is a fragment of a program, such as an identifier or a function call. A form is represented as a syntax object, and each syntax object has an associated set of scopes (i.e., a *scope set*). In the above example, the representations of the `x`s include the scope that corresponds to the `let` form.

When a form parses as the binding of a particular identifier, parsing updates a global table that maps a combination of an identifier’s symbol and scope set to its meaning: a variable, a syntactic form, or a transformer. An identifier refers to a particular binding when the reference’s symbol and the identifier’s symbol are the same, and when the reference’s scope set is a superset of the binding’s scope set. For a given identifier, multiple bindings may have scope sets that are subsets of the identifier’s; in that case, the identifier refers to the binding whose set is a superset of all others; if no such binding exists, the reference is ambiguous (and triggers a syntax error if it is parsed as an expression). A binding *shadows* any binding (i.e., it is *shadowing* any binding) with the same symbol but a subset of scopes.

For example, in

```
(let ([x 5]) x)
```

in a context where `let` corresponds to the usual syntactic form, the parsing of `let` introduces a new scope for the binding of `x`. Since the second `x` receives that scope as part of the `let` body, the first `x` binds the second `x`. In the more complex case

```
(let ([x 5])
  (let ([x 6])
    x))
```

the inner `let` creates a second scope for the second `x`s, so its scope set is a superset of the first `x`’s scope set—which means that the binding for the second `x` shadows the one for the first `x`, and the third `x` refers to the binding created by the second one.

A *top-level binding* is a binding from a definition at the top-level; a *module binding* is a binding from a definition in a module; all other bindings are *local bindings*. Within a module, references to top-level bindings are disallowed. An identifier without a binding is *unbound*.

Throughout the documentation, identifiers are typeset to suggest the way that they are parsed. A hyperlinked identifier like `lambda` indicates a reference to a syntactic form or variable. A plain identifier like `x` is a variable or a reference to an unspecified top-level variable.

Every binding has a *phase level* in which it can be referenced, where a phase level normally
corresponds to an integer (but the special label phase level does not correspond to an integer).
Phase level 0 corresponds to the run time of the enclosing module (or the run time of top-level expressions). Bindings in phase level 0 constitute the base environment. Phase level 1 corresponds to the time during which the enclosing module (or top-level expression) is expanded; bindings in phase level 1 constitute the transformer environment. Phase level -1 corresponds to the run time of a different module for which the enclosing module is imported for use at phase level 1 (relative to the importing module); bindings in phase level -1 constitute the template environment. The label phase level does not correspond to any execution time; it is used to track bindings (e.g., to identifiers within documentation) without implying an execution dependency.

An identifier can have different bindings in different phase levels. More precisely, the scope set associated with a form can be different at different phase levels; a top-level or module context implies a distinct scope at every phase level, while scopes from macro expansion or other syntactic forms are added to a form’s scope sets at all phases. The context of each binding and reference determines the phase level whose scope set is relevant.

Changed in version 6.3 of package base: Changed local bindings to have a specific phase level, like top-level and module bindings.

1.2.2 Syntax Objects

A syntax object combines a simpler Racket value, such as a symbol or pair, with lexical information, source-location information, syntax properties, and tamper status. The lexical information of a syntax object comprises a set of scope sets, one for each phase level. In particular, an identifier is represented as a syntax object containing a symbol, and its lexical information can be combined with the global table of bindings to determine its binding (if any) at each phase level.

For example, a car identifier might have lexical information that designates it as the car from the racket/base language (i.e., the built-in car). Similarly, a lambda identifier’s lexical information may indicate that it represents a procedure form. Some other identifier’s lexical information may indicate that it references a top-level variable.

When a syntax object represents a more complex expression than an identifier or simple constant, its internal components can be extracted. Even for extracted identifiers, detailed information about binding is available mostly indirectly; two identifiers can be compared to determine whether they refer to the same binding (i.e., free-identifier=?), or whether the identifiers have the same scope set so that each identifier would bind the other if one were in a binding position and the other in an expression position (i.e., bound-identifier=?).

For example, when the program written as

\[
(\text{let } ([x 5]) (+ x 6))
\]
is represented as a syntax object, then two syntax objects can be extracted for the two \( x \)s. Both the \texttt{free-identifier=} and \texttt{bound-identifier=} predicates will indicate that the \( x \)s are the same. In contrast, the \texttt{let} identifier is not \texttt{free-identifier=} or \texttt{bound-identifier=} to either \( x \).

The lexical information in a syntax object is independent of the rest of the syntax object, and it can be copied to a new syntax object in combination with an arbitrary other Racket value. Thus, identifier-binding information in a syntax object is predicated on the symbolic name of the identifier as well as the identifier’s lexical information; the same question with the same lexical information but different base value can produce a different answer.

For example, combining the lexical information from \texttt{let} in the program above to \texttt{'}\( x \) would not produce an identifier that is \texttt{free-identifier=} to either \( x \), since it does not appear in the scope of the \( x \) binding. Combining the lexical context of the \texttt{6} with \texttt{'}\( x \), in contrast, would produce an identifier that is \texttt{bound-identifier=} to both \( x \).

The \texttt{quote-syntax} form bridges the evaluation of a program and the representation of a program. Specifically, \texttt{(quote-syntax datum #:local)} produces a syntax object that preserves all of the lexical information that \texttt{datum} had when it was parsed as part of the \texttt{quote-syntax} form. Note that \texttt{(quote-syntax datum)} form is similar, but it removes certain scopes from the \texttt{datum}’s scope sets; see \texttt{quote-syntax} for more information.

1.2.3 Expansion (Parsing)

Expansion recursively processes a syntax object in a particular phase level, starting with phase level 0. Bindings from the syntax object’s lexical information drive the expansion process, and cause new bindings to be introduced for the lexical information of sub-expressions. In some cases, a sub-expression is expanded in a deeper phase than the enclosing expression.

Fully Expanded Programs

A complete expansion produces a syntax object matching the following grammar:

\[
\text{top-level-form} = \text{general-top-level-form} \\
| (\texttt{\%expression expr}) \\
| (\texttt{module id module-path} \\
| (\texttt{\%plain-module-begin} \\
| \text{module-level-form} ...)) \\
| (\texttt{begin top-level-form ...}) \\
| (\texttt{begin-for-syntax top-level-form ...})
\]

\[
\text{module-level-form} = \text{general-top-level-form} \\
| (\texttt{\%provide raw-provide-spec ...}) \\
| (\texttt{begin-for-syntax module-level-form ...}) \\
| \text{submodule-form}
\]
A fully-expanded syntax object corresponds to a parse of a program (i.e., a parsed program), and lexical information on its identifiers indicates the parse.

More specifically, the typesetting of identifiers in the above grammar is significant. For
example, the second case for `expr` is a syntax-object list whose first element is an identifier, where the identifier’s lexical information specifies a binding to the `#%plain-lambda` of the `racket/base` language (i.e., the identifier is `free-identifier=?` to one whose binding is `#%plain-lambda`). In all cases, identifiers above typeset as syntactic-form names refer to the bindings defined in §3 "Syntactic Forms".

In a fully expanded program for a namespace whose base phase is 0, the relevant phase level for a binding in the program is \( N \) if the bindings has \( N \) surrounding `begin-for-syntax` and `define-syntaxes` forms—not counting any `begin-for-syntax` forms that wrap a module or module* form for the body of the module or module*, unless a module* form has `#f` in place of a `module-path` after the `id`. The `datum` in a `quote-syntax` form preserves its information for all phase levels.

A reference to a local binding in a fully expanded program has a scope set that matches its binding identifier exactly. Additional scopes, if any, are removed. As a result, `bound-identifier=?` can be used to correlate local binding identifiers with reference identifiers, while `free-identifier=?` must be used to relate references to module bindings or top-level bindings.

In addition to the grammar above, `#%expression` can appear in a fully local-expanded expression position. For example, `#%expression` can appear in the result from `local-expand` when the stop list is empty. Reference-identifier scope sets are reduced in local-expanded expressions only when the `local-expand` stop list is empty.

Changed in version 6.3 of package `base`: Added the `#:local` variant of `quote-syntax`; removed `letrec-syntaxes+values` from possibly appearing in a fully local-expanded form.

**Expansion Steps**

In a recursive expansion, each single step in expanding a syntax object at a particular phase level depends on the immediate shape of the syntax object being expanded:

- If it is an identifier (i.e., a syntax-object symbol), then a binding is determined by the identifier’s lexical information. If the identifier has a binding, that binding is used to continue. If the identifier is unbound, a new syntax-object symbol ‘`#%top` is created using the lexical information of the identifier with implicit-made-explicit properties; if this `#%top` identifier has no binding, then parsing fails with an `exn:fail:syntax` exception. Otherwise, the new identifier is combined with the original identifier in a new syntax-object pair (also using the same lexical information as the original identifier), and the `#%top` binding is used to continue.

  Changed in version 6.3 of package `base`: Changed the introduction of `#%top` in a top-level context to unbound identifiers only.

- If it is a syntax-object pair whose first element is an identifier, and if the identifier has a binding other than as a top-level variable, then the identifier’s binding is used to continue.
• If it is a syntax-object pair of any other form, then a new syntax-object symbol '#%app is created using the lexical information of the pair with implicit-made-explicit properties. If the resulting #%app identifier has no binding, parsing fails with an exn:fail:syntax exception. Otherwise, the new identifier is combined with the original pair to form a new syntax-object pair (also using the same lexical information as the original pair), and the #%app binding is used to continue.

• If it is any other syntax object, then a new syntax-object symbol '#%datum is created using the lexical information of the original syntax object with implicit-made-explicit properties. If the resulting #%datum identifier has no binding, parsing fails with an exn:fail:syntax exception. Otherwise, the new identifier is combined with the original syntax object in a new syntax-object pair (using the same lexical information as the original pair), and the #%datum binding is used to continue.

Thus, the possibilities that do not fail lead to an identifier with a particular binding. This binding refers to one of three things:

• A transformer, such as introduced by define-syntax or let-syntax. If the associated value is a procedure of one argument, the procedure is called as a syntax transformer (described below), and parsing starts again with the syntax-object result. If the transformer binding is to any other kind of value, parsing fails with an exn:fail:syntax exception. The call to the syntax transformer is parameterized to set current-namespace to a namespace that shares bindings and variables with the namespace being used to expand, except that its base phase is one greater.

• A variable binding, such as introduced by a module-level define or by let. In this case, if the form being parsed is just an identifier, then it is parsed as a reference to the corresponding variable. If the form being parsed is a syntax-object pair, then an #%app is added to the front of the syntax-object pair in the same way as when the first item in the syntax-object pair is not an identifier (third case in the previous enumeration), and parsing continues.

• A core syntactic form, which is parsed as described for each form in §3 “Syntactic Forms”. Parsing a core syntactic form typically involves recursive parsing of sub-forms, and may introduce bindings that determine the parsing of sub-forms.

When a #%top, #%app, or #%datum identifier is added by the expander, it is given implicit-made-explicit properties: an 'implicit-made-explicit syntax property whose value is #t, and a hidden property to indicate that the implicit identifier is original in the sense of syntax-original? if the syntax object that gives the identifier its lexical information has that property.

Changed in version 7.9.0.13 of package base: Added implicit-made-explicit properties.

Expansion Context
Each expansion step occurs in a particular context, and transformers and core syntactic forms may expand differently for different contexts. For example, a module form is allowed only in a top-level context, and it fails in other contexts. The possible contexts are as follows:

- **top-level context**: outside of any module, definition, or expression, except that sub-expressions of a top-level begin form are also expanded as top-level forms.
- **module-begin context**: inside the body of a module, as the only form within the module.
- **module context**: in the body of a module (inside the module-begin layer).
- **internal-definition context**: in a nested context that allows both definitions and expressions.
- **expression context**: in a context where only expressions are allowed.

Different core syntactic forms parse sub-forms using different contexts. For example, a let form always parses the right-hand expressions of a binding in an expression context, but it starts parsing the body in an internal-definition context.

### Introducing Bindings

Bindings are introduced during expansion when certain core syntactic forms are encountered:

- When a require form is encountered at the top level or module level, each symbol specified by the form is paired with the scope set of the specification to introduce new bindings. If not otherwise indicated in the require form, bindings are introduced at the phase levels specified by the exporting modules: phase level 0 for each normal provide, phase level 1 for each for-syntax provide, and so on. The for-meta provide form allows exports at an arbitrary phase level (as long as a binding exists within the module at the phase level).

  A for-syntax sub-form within require imports similarly, but the resulting bindings have a phase level that is one more than the exported phase levels, when exports for the label phase level are still imported at the label phase level. More generally, a for-meta sub-form within require imports with the specified phase level shift; if the specified shift is #f, or if for-label is used to import, then all bindings are imported into the label phase level.

- When a define, define-values, define-syntax, or define-syntaxes form is encountered at the top level or module level, a binding is added phase level 0 (i.e., the base environment is extended) for each defined identifier.

- When a begin-for-syntax form is encountered at the top level or module level, bindings are introduced as for define-values and define-syntaxes, but at phase
level 1 (i.e., the transformer environment is extended). More generally, begin-for-syntax forms can be nested, an each begin-for-syntax shifts its body definition by one phase level.

- When a let-values form is encountered, the body of the let-values form is extended (by creating new syntax objects) with a fresh scope. The scope is added to the identifiers themselves, so that the identifiers in binding position are bound-identifier=? to uses in the fully expanded form, and so they are not bound-identifier=? to other identifiers. The new bindings are at the phase level at which the let-values form is expanded.

- When a letrec-values or letrec-syntaxes+values form is encountered, bindings are added as for let-values, except that the right-hand-side expressions are also extended with the new scope.

- Definitions in internal-definition contexts introduce new scopes and bindings as described in §1.2.3.8 “Internal Definitions”.

For example, in

\[\text{let-values (([(x) 10]) (+ x y))}\]

the binding introduced for x applies to the x in the body, because a fresh scope is created and added to both the binding x and reference x. The same scope is added to the y, but since it has a different symbol than the binding x, it does not refer to the new binding. Any x outside of this let-values form does not receive the fresh scope and therefore does not refer to the new binding.

**Transformer Bindings**

In a top-level context or module context, when the expander encounters a define-syntaxes form, the binding that it introduces for the defined identifiers is a transformer binding. The value of the binding exists at expansion time, rather than run time (though the two times can overlap), though the binding itself is introduced with phase level 0 (i.e., in the base environment).

The value for the binding is obtained by evaluating the expression in the define-syntaxes form. This expression must be expanded (i.e., parsed) before it can be evaluated, and it is expanded at phase level 1 (i.e., in the transformer environment) instead of phase level 0.

If the resulting value is a procedure of one argument or the result of make-set!-transformer on a procedure, then it is used as a syntax transformer (a.k.a. macro). The procedure is expected to accept a syntax object and return a syntax object. A use of the binding (at phase level 0) triggers a call of the syntax transformer by the expander; see §1.2.3.2 “Expansion Steps”.

Before the expander passes a syntax object to a transformer, the syntax object is extended with a fresh macro-introduction scope (that applies to all sub-syntax objects) to distinguish
syntax objects at the macro’s use site from syntax objects that are introduced by the macro; in the result of the transformer the presence of the scope is flipped, so that introduced syntax objects retain the scope, and use-site syntax objects do not have it. In addition, if the use of a transformer is in the same definition context as its binding, the use-site syntax object is extended with an additional fresh use-site scope that is not flipped in the transformer’s result, so that only use-site syntax objects have the use-site scope.

The scope-introduction process for macro expansion helps keep binding in an expanded program consistent with the lexical structure of the source program. For example, the expanded form of the program

```
(define x 12)
(define-syntax m
  (syntax-rules ()
    [(_ id) (let ([x 10]) id)])
  (m x))
```

is

```
(define x 12)
(define-syntax m ....)
(let ([x 10]) x)
```

However, the result of the last expression is 12, not 10. The reason is that the transformer bound to \texttt{m} introduces the binding \texttt{x}, but the referencing \texttt{x} is present in the argument to the transformer. The introduced \texttt{x} is left with one fresh scope, while the reference \texttt{x} has a different fresh scope, so the binding \texttt{x} is not \texttt{bound-identifier=7} to the body \texttt{x}.

A use-site scope on a binding identifier is ignored when the definition is in the same context where the use-site scope was introduced. This special treatment of use-site scopes allows a macro to expand to a visible definition. For example, the expanded form of the program

```
(define-syntax m
  (syntax-rules ()
    [(_ id) (define id 5)])
  (m x))
```

is

```
(define-syntax m ....)
(define x 5)
x
```

where the \texttt{x} in the \texttt{define} form has a use-site scope that is not present on the final \texttt{x}. The final \texttt{x} nevertheless refers to the definition, because the use-site scope is effectively removed before installing the definition’s binding. In contrast, the expansion of
(define-syntax m
  (syntax-rules ()
    [_ id] (let ([x 4]
                  (let ([id 5]
                        x)))))
(m x)

is

(define-syntax m ....)
(let ([x 4])
  (let ([x 5])
    x))

where the second x has a use-site scope that prevents it from binding the final x. The use-site scope is not ignored in this case, because the binding is not part of the definition context where (m x) was expanded.

The set! form works with the make-set!-transformer and prop:set!-transformer property to support assignment transformers that transform set! expressions. An assignment transformer contains a procedure that is applied by set! in the same way as a normal transformer by the expander.

The make-rename-transformer procedure or prop:rename-transformer property creates a value that is also handled specially by the expander and by set! as a transformer binding’s value. When id is bound to a rename transformer produced by make-rename-transformer, it is replaced with the target identifier passed to make-rename-transformer. In addition, as long as the target identifier does not have a true value for the 'not-free-identifier=? syntax property, the binding table is extended to indicate that id is an alias for the identifier in the rename transformer. The free-identifier=? function follows aliasing chains to determine equality of bindings, the identifier-binding function similarly follows aliasing chains, and the provide form exports id as the target identifier. Finally, the syntax-local-value function follows rename transformer chains even when binding aliases are not installed.

In addition to using scopes to track introduced identifiers, the expander tracks the expansion history of a form through syntax properties such as 'origin. See §12.7 “Syntax Object Properties” for more information.

Finally, the expander uses a tamper status to control the way that unexported and protected module bindings are used. See §12.8 “Syntax Taints” for more information on a tamper status.

The expander’s handling of letrec-syntaxes+values is similar to its handling of define-syntaxes. A letrec-syntaxes+values can be expanded in an arbitrary phase level n (not just 0), in which case the expression for the transformer binding is expanded at
phase level $n+1$.

The expressions in a `begin-for-syntax` form are expanded and evaluated in the same way as for `define-syntaxes`. However, any introduced bindings from definition within `begin-for-syntax` are at phase level 1 (not a transformer binding at phase level 0).

**Local Binding Context**

Although the binding of an identifier can be uniquely determined from the combination of its lexical information and the global binding table, the expander also maintains a *local binding context* that records additional information about local bindings to ensure they are not used outside of the lexical region in which they are bound.

Due to the way local binding forms like `let` add a fresh scope to both bound identifiers and body forms, it isn’t ordinarily possible for an identifier to reference a local binding without appearing in the body of the `let`. However, if macros use compile-time state to stash bound identifiers, or use `local-expand` to extract identifiers from an expanded binding form, they can violate this constraint. For example, the following `stash-id` and `unstash-id` macros cooperate to move a reference to a locally-bound `x` identifier outside of the lexical region in which it is bound:

```
> (begin-for-syntax
  (define stashed-id #f))
> (define-syntax (stash-id stx)
  (syntax-case stx ()
    [(_ id)
      (begin
        (set! stashed-id #'id)
        #'(void))]))
> (define-syntax (unstash-id stx)
  stashed-id)
> (let ([x 42])
  (stash-id x)
  (unstash-id))
42
> (unstash-id)
identifier used out of context: #<syntax:eval:5:0 x>
```

In general, an identifier’s lexical information is not sufficient to know whether or not its binding is available in the enclosing context, since the scope set for the identifier stored in `stashed-id` unambiguously refers to a binding in the global binding table. This can be observed by the fact that `identifier-binding` produces `'lexical`, not `#f`:

```
> (define-syntax (stashed-id-binding stx)
  #'(#,(identifier-binding stashed-id))
> (stashed-id-binding)
'lexical
```
However, the reference produced by (unstash-id) in the above program is still illegal, even if it isn’t technically unbound. To record the fact that x’s binding is in scope only within the body of its corresponding let form, the expander adds x’s binding to the local binding context while expanding the let body. More generally, the expander adds all local variable bindings to the local binding context while expanding expressions in which a reference to the variable would be legal. When the expander encounters an identifier bound to a local variable, and the associated binding is not in the current local binding context, it raises a syntax error.

The local binding context also tracks local transformer bindings (i.e. bindings bound by forms like let-syntax) in a similar way, except that the context also stores the compile-time value associated with the transformer. When an identifier that is locally bound as a transformer is used in application position as a syntax transformer, or its compile-time value is looked up using syntax-local-value, the local binding context is consulted to retrieve the value. If the binding is in scope, its associated compile-time value is used; otherwise, the expander raises a syntax error.

Examples:

> (define-syntax (stashed-id-local-value stx)  
  #'#, (syntax-local-value stashed-id))
> (let-syntax ([y 42])  
  (stash-id y)  
  (stashed-id-local-value))  
42
> (stashed-id-local-value)  
syntax-local-value: identifier is not bound to syntax:  
#<syntax:eval:11:0 y>

Partial Expansion

In certain contexts, such as an internal-definition context or module context, partial expansion is used to determine whether forms represent definitions, expressions, or other declaration forms. Partial expansion works by cutting off the normal recursive expansion when the relevant binding is for a primitive syntactic form.

As a special case, when expansion would otherwise add an #%app, #%datum, or #%top identifier to an expression, and when the binding turns out to be the primitive #%app, #%datum, or #%top form, then expansion stops without adding the identifier.

Internal Definitions

An internal-definition context supports local definitions mixed with expressions. Forms that allow internal definitions document such positions using the body meta-variable. Definitions in an internal-definition context are equivalent to local binding via letrec-syntaxes+values; macro expansion converts internal definitions to a letrec-syntaxes+values form.
Expansion relies on partial expansion of each body in an internal-definition sequence. Partial expansion of each body produces a form matching one of the following cases:

- A define-values form: The binding table is immediately enriched with bindings for the define-values form. Further expansion of the definition is deferred, and partial expansion continues with the rest of the body.
- A define-syntaxes form: The right-hand side is expanded and evaluated (as for a letrec-syntaxes+values form), and a transformer binding is installed for the body sequence before partial expansion continues with the rest of the body.
- A primitive expression form other than begin: Further expansion of the expression is deferred, and partial expansion continues with the rest of the body.
- A begin form: The sub-forms of the begin are spliced into the internal-definition sequence, and partial expansion continues with the first of the newly-spliced forms (or the next form, if the begin had no sub-forms).

After all body forms are partially expanded, if no definitions were encountered, then the expressions are collected into a begin form as the internal-definition context’s expansion. Otherwise, at least one expression must appear after the last definition, and any expr that appears between definitions is converted to (define-values () (begin expr (values))); the definitions are then converted to bindings in a letrec-syntaxes+values form, and all expressions after the last definition become the body of the letrec-syntaxes+values form.

Before partial expansion begins, expansion of an internal-definition context begins with the introduction of a fresh outside-edge scope on the content of the internal-definition context. This outside-edge scope effectively identifies syntax objects that are present in the original form. An inside-edge scope is also created and added to the original content; furthermore, the inside-edge scope is added to the result of any partial expansion. This inside-edge scope ensures that all bindings introduced by the internal-definition context have a particular scope in common.

Module Expansion, Phases, and Visits

Expansion of a module form proceeds in a similar way to expansion of an internal-definition context: an outside-edge scope is created for the original module content, and an inside-edge scope is added to both the original module and any form that appears during a partial expansion of the module’s top-level forms to uncover definitions and imports.

A require form not only introduces bindings at expansion time, but also visits the referenced module when it is encountered by the expander. That is, the expander instantiates any variables defined in the module within begin-for-syntax, and it also evaluates all expressions for define-syntaxes transformer bindings.

Module visits propagate through requires in the same way as module instantiation. Moreover, when a module is visited at phase 0, any module that it requires for-syntax is
instantiate at phase 1, while further requires for-template leading back to phase 0 causes the required module to be visited at phase 0 (i.e., not instantiated).

During compilation, the top-level of module context is itself implicitly visited. Thus, when the expander encounters (require (for-syntax ....)), it immediately instantiates the required module at phase 1, in addition to adding bindings at phase level 1 (i.e., the transformer environment). Similarly, the expander immediately evaluates any form that it encounters within begin-for-syntax.

Phases beyond 0 are visited on demand. For example, when the right-hand side of a phase-0 let-syntax is to be expanded, then modules that are available at phase 1 are visited. More generally, initiating expansion at phase \( n \) visits modules at phase \( n \), which in turn instantiates modules at phase \( n+1 \). These visits and instantiations apply to available modules in the enclosing namespace’s module registry; a per-registry lock prevents multiple threads from concurrently instantiating and visiting available modules. On-demand instantiation of available modules uses the same reentrant lock as namespace-call-with-registry-lock.

When the expander encounters require and (require (for-syntax ....)) within a module context, the resulting visits and instantiations are specific to the expansion of the enclosing module, and are kept separate from visits and instantiations triggered from a top-level context or from the expansion of a different module. Along the same lines, when a module is attached to a namespace through namespace-attach-module, modules that it requires are transitively attached, but instances are attached only at phases at or below the namespace’s base phase.

**Macro-Introduced Bindings**

When a top-level definition binds an identifier that originates from a macro expansion, the definition captures only uses of the identifier that are generated by the same expansion due to the fresh scope that is generated for the expansion.

Examples:

```
> (define-syntax def-and-use-of-x
  (syntax-rules ()
    [(def-and-use-of-x val)
      ; x below originates from this macro:
      (begin (define x val) x)]))
> (define x 1)
> x
1
> (def-and-use-of-x 2)
2
> x
1
> (define-syntax def-and-use
```
For a top-level definition (outside of a module), the order of evaluation affects the binding of a generated definition for a generated identifier use. If the use precedes the definition, then the use is resolved with the bindings that are in place that at point, which will not be a macro-generated binding. (No such dependency on order occurs within a module, since a module binding covers the entire module body.) To support the declaration of an identifier before its use, the define-syntaxes form avoids binding an identifier if the body of the define-syntaxes declaration produces zero results.

Examples:

```scheme
> (define bucket-1 0)
> (define bucket-2 0)
> (define-syntax def-and-set!-use-of-x
   (syntax-rules ()
     [(def-and-set!-use-of-x val)
      (begin (set! bucket-1 x) (define x val) (set! bucket-2 x))]))
> (define x 1)
> (def-and-set!-use-of-x 2)
> x
1
> bucket-1
1
> bucket-2
2
> (define-syntax defs-and-uses/fail
   (syntax-rules ()
     [(def-and-use)
      (begin
        ; Initial reference to even precedes definition:
        (define (odd x) (if (zero? x) #f (even (sub1 x))))
        (define (even x) (if (zero? x) #t (odd (sub1 x))))
        (odd 17)))))
> (defs-and-uses/fail)
```

even: undefined;
cannot reference an identifier before its definition
in module: top-level
Macro-generated require and provide clauses also introduce and reference generation-specific bindings (due to the added scope) with the same ordering effects as for definitions. The bindings depend on the scope set attached to specific parts of the form:

- In require, for a require-spec of the form (rename-in [orig-id bind-id]) or (only-in .... [orig-id bind-id]), the bind-id supplies the scope set for the binding. In require for other require-specs, the generator of the require-spec determines the scope set.

- In provide, for a provide-spec of the form id, the exported identifier is the one that binds id, but the external name is the plain, symbolic part of id. The exceptions for all-except-out are similarly determined, as is the orig-id binding of a rename-out form, and plain symbols are used for the external names. For all-defined-out, only identifiers with definitions having only the scopes of (all-defined-out) form are exported; the external name is the plain symbol from the definition.

1.2.4 Compilation

Before expanded code is evaluated, it is first compiled. A compiled form has essentially the same information as the corresponding expanded form, though the internal representation naturally dispenses with identifiers for syntactic forms and local bindings. One significant difference is that a compiled form is almost entirely opaque, so the information that it contains cannot be accessed directly (which is why some identifiers can be dropped). At the same time, a compiled form can be marshaled to and from a byte string, so it is suitable for saving and re-loading code.

Although individual read, expand, compile, and evaluate operations are available, the operations are often combined automatically. For example, the eval procedure takes a syntax object and expands it, compiles it, and evaluates it.
1.2.5 Namespaces

A namespace is both a starting point for parsing and a starting point for running compiled code. A namespace also has a module registry that maps module names to module declarations (see §1.1.9 “Modules and Module-Level Variables”). This registry is shared by all phase levels, and it applies both to parsing and to running compiled code.

As a starting point for parsing, a namespace provides scopes (one per phase level, plus one that spans all phase levels). Operations such as namespace-require create initial bindings using the namespace’s scopes, and the further expansion and evaluation in the namespace can create additional bindings. Evaluation of a form with a namespace always adds the namespace’s phase-specific scopes to the form and to any result of expanding the top-level form; as a result, every binding identifier has at least one scope. The namespace’s additional scope, which is added at all phase levels, is added only on request (e.g., by using eval as opposed to eval-syntax). Except for namespaces generated by a module (see module->namespace), every namespace uses the same scope as the one added to all phase levels, while the scopes specific to a phase level are always distinct.

As a starting point evaluating compiled code, each namespace encapsulates a distinct set of top-level variables at various phases, as well as a potentially distinct set of module instances in each phase. That is, even though module declarations are shared for all phase levels, module instances are distinct for each phase. Each namespace has a base phase, which corresponds to the phase used by reflective operations such as eval and dynamic-require. In particular, using eval on a require form instantiates a module in the namespace’s base phase.

After a namespace is created, module instances from existing namespaces can be attached to the new namespace. In terms of the evaluation model, top-level variables from different namespaces essentially correspond to definitions with different prefixes, but attaching a module uses the same prefix for the module’s definitions in namespaces where it is attached. The first step in evaluating any compiled expression is to link its top-level variable and module-level variable references to specific variables in the namespace.

At all times during evaluation, some namespace is designated as the current namespace. The current namespace has no particular relationship, however, with the namespace that was used to expand the code that is executing, or with the namespace that was used to link the compiled form of the currently evaluating code. In particular, changing the current namespace during evaluation does not change the variables to which executing expressions refer. The current namespace only determines the behavior of reflective operations to expand code and to start evaluating expanded/compiled code.

Examples:

> (define x 'orig) ; define in the original namespace
; The following let expression is compiled in the original
; namespace, so direct references to x see 'orig.
> (let ([n (make-base-namespace)]) ; make new namespace
(parameterize ([current-namespace n])
  (eval '(define x 'new)); evals in the new namespace
  (display x); displays 'orig
  (display (eval 'x))); displays 'new

orignew

If an identifier is bound to syntax or to an import, then defining the identifier as a variable shadows the syntax or import in future uses of the environment. Similarly, if an identifier is bound to a top-level variable, then binding the identifier to syntax or an import shadows the variable; the variable’s value remains unchanged, however, and may be accessible through previously evaluated expressions.

Examples:

> (define x 5)
> (define (f) x)
> x
5
> (f)
5
> (define-syntax x (syntax-id-rules () [_ 10]))
> x
10
> (f)
5
> (define x 7)
> x
7
> (f)
7
> (module m racket (define x 8) (provide x))
> (require 'm)
> x
8
> (f)
7

Like a top-level namespace, each module form has an associated scope to span all phase levels of the module’s content, plus a scope at each phase level. The latter is added to every form, original or appearing through partial macro expansion, within the module’s immediate body. Those same scopes are propagated to a namespace created by module->namespace for the module. Meanwhile, parsing of a module form begins by removing the all scopes that correspond to the enclosing top-level or (in the case of submodules) module and module* forms.
1.2.6 Inferred Value Names

To improve error reporting, names are inferred at compile-time for certain kinds of values, such as procedures. For example, evaluating the following expression:

\[
\text{let } (\text{f (lambda () 0)}) (\text{f 1 2 3})
\]

produces an error message because too many arguments are provided to the procedure. The error message is able to report \text{f} as the name of the procedure. In this case, Racket decides, at compile-time, to name as \text{'f} all procedures created by the let-bound lambda.

Names are inferred whenever possible for procedures. Names closer to an expression take precedence. For example, in

\[
\text{define my-f (let (f (lambda () 0)) f)}
\]

the procedure bound to \text{my-f} will have the inferred name \text{'f}.

When an \text{inferred-name} property is attached to a syntax object for an expression (see §12.7 “Syntax Object Properties”), the property value is used for naming the expression, and it overrides any name that was inferred from the expression’s context. Normally, the property value should be a symbol. A \text{inferred-name} property value of #<void> hides a name that would otherwise be inferred from context (perhaps because a binding identifier’s was automatically generated and should not be exposed).

To support the propagation and merging of consistent properties during expansions, the value of the \text{inferred-name} property can be a tree formed with cons where all of the leaves are the same. For example, (cons \text{'name 'name}) is equivalent to \text{'name}, and (cons (void) (void)) is equivalent to #<void>.

When an inferred name is not available, but a source location is available, a name is constructed using the source location information. Inferred and property-assigned names are also available to syntax transformers, via syntax-local-name.

1.2.7 Cross-Phase Persistent Module Declarations

A module is cross-phase persistent only if it fits the following grammar, which uses non-terminals from §1.2.3.1 “Fully Expanded Programs” only if it includes (#%declare #:cross-phase-persistent), only it includes no uses of quote-syntax or #%variable-reference, and only if no module-level binding is set!ed.

\[
cross-module = (\text{module id module-path}
\begin{array}{l}
\text{(#%plain-module-begin}
\text{cross-form ...))}
\end{array}
\]

See procedure-rename to override a procedure’s inferred name at runtime.
This grammar applies after expansion, but because a cross-phase persistent module imports only from other cross-phase persistent modules, the only relevant expansion steps are the implicit introduction of #%plain-module-begin, implicit introduction of #%plain-app, and implicit introduction and/or expansion of #%datum.

Changed in version 7.5.0.12 of package base: Allow (#%plain-app variable-reference-from-unsafe? (#%variable-reference)).

1.3 The Reader

Racket’s reader is a recursive-descent parser that can be configured through a readtable and various other parameters. This section describes the reader’s parsing when using the default readtable.
Reading from a stream produces one datum. If the result datum is a compound value, then reading the datum typically requires the reader to call itself recursively to read the component data.

The reader can be invoked in either of two modes: read mode, or read-syntax mode. In read-syntax mode, the result is always a syntax object that includes source-location and (initially empty) lexical information wrapped around the sort of datum that read mode would produce. In the case of pairs, vectors, and boxes, the content is also wrapped recursively as a syntax object. Unless specified otherwise, this section describes the reader’s behavior in read mode, and read-syntax mode does the same modulo wrapping of the final result.

Reading is defined in terms of Unicode characters; see §13.1 “Ports” for information on how a byte stream is converted to a character stream.

Symbols, keywords, strings, byte strings, regexps, characters, and numbers produced by the reader in read-syntax mode are interned, which means that such values in the result of read-syntax are always eq? when they are equal? (whether from the same call or different calls to read-syntax). Symbols and keywords are interned in both read and read-syntax mode. When a quoted value is in compiled code that written and then read back in (see §1.4.16 “Printing Compiled Code”), only strings and byte strings are interned when reading the code. Sending an interned value across a place channel does not necessarily produce an interned value at the receiving place. See also datum-intern-literal and datum->syntax.

1.3.1 Delimiters and Dispatch

Along with whitespace and a BOM character, the following characters are delimiters:

```
( ) [ ] { } , . ; :
```

A delimited sequence that starts with any other character is typically parsed as either a symbol, number, or extflonum, but a few non-delimiter characters play special roles:

- # has a special meaning as an initial character in a delimited sequence; its meaning depends on the characters that follow; see below.
- | starts a subsequence of characters to be included verbatim in the delimited sequence (i.e., they are never treated as delimiters, and they are not case-folded when case-insensitivity is enabled); the subsequence is terminated by another |, and neither the initial nor terminating | is part of the subsequence.
- \ outside of a | pair causes the following character to be included verbatim in a delimited sequence.
More precisely, after skipping whitespace and \texttt{\uFEFF} BOM characters, the reader dispatches based on the next character or characters in the input stream as follows:

- \texttt{\{} starts a pair or list; see §1.3.6 “Reading Pairs and Lists”
- \texttt{[} starts a pair or list; see §1.3.6 “Reading Pairs and Lists”
- \texttt{\{} starts a pair or list; see §1.3.6 “Reading Pairs and Lists”
- ) matches \texttt{\{} or raises \texttt{exn:fail:read}
- ] matches \texttt{[} or raises \texttt{exn:fail:read}
- } matches \texttt{\} or raises \texttt{exn:fail:read}
- starts a string; see §1.3.7 “Reading Strings”
- starts a quote; see §1.3.8 “Reading Quotes”
- starts a quasiquote; see §1.3.8 “Reading Quotes”
- starts a [splicing] unquote; see §1.3.8 “Reading Quotes”
- starts a line comment; see §1.3.9 “Reading Comments”
- \texttt{#t} or \texttt{#T} true; see §1.3.5 “Reading Booleans”
- \texttt{#f} or \texttt{#F} false; see §1.3.5 “Reading Booleans”
- \texttt{#(} starts a vector; see §1.3.10 “Reading Vectors”
- \texttt{[#} starts a vector; see §1.3.10 “Reading Vectors”
- \texttt{#)} starts a vector; see §1.3.10 “Reading Vectors”
- \texttt{#f(} or \texttt{#Fl(} starts a flvector; see §1.3.10 “Reading Vectors”
- \texttt{#f[} or \texttt{#Fl[} starts a flvector; see §1.3.10 “Reading Vectors”
- \texttt{#f{} or \texttt{#Fx{} starts a fxvector; see §1.3.10 “Reading Vectors”
- \texttt{#fx} or \texttt{#Fx} starts a fxvector; see §1.3.10 “Reading Vectors”
- \texttt{#s(} starts a structure literal; see §1.3.11 “Reading Structures”
- \texttt{#s[} starts a structure literal; see §1.3.11 “Reading Structures”
- \texttt{#s{} starts a structure literal; see §1.3.11 “Reading Structures”
- \texttt{#\} starts a character; see §1.3.14 “Reading Characters”
- \texttt{#\\} starts a byte string; see §1.3.7 “Reading Strings”
- \texttt{\#} starts a symbol; see §1.3.2 “Reading Symbols”
- \texttt{\#\} starts a keyword; see §1.3.15 “Reading Keywords”
- \texttt{#\} starts a box; see §1.3.13 “Reading Boxes”
- \texttt{#\} starts a block comment; see §1.3.9 “Reading Comments”
- \texttt{#\} starts an S-expression comment; see §1.3.9 “Reading Comments”
- \texttt{#\} starts a syntax quote; see §1.3.8 “Reading Quotes”
- \texttt{#\} starts a line comment; see §1.3.9 “Reading Comments”
- \texttt{#\} starts a line comment; see §1.3.9 “Reading Comments”
- \texttt{#\} may start a reader extension; see §1.3.18 “Reading via an Extension”
- \texttt{#\} starts a syntax quasiquote; see §1.3.8 “Reading Quotes”
- \texttt{#\} starts a syntax [splicing] unquote; see §1.3.8 “Reading Quotes”
- \texttt{#\} starts compiled code; see §1.4.16 “Printing Compiled Code”
- \texttt{#t} or \texttt{#T} starts a number; see §1.3.3 “Reading Numbers”
- \texttt{#e} or \texttt{#E} starts a number; see §1.3.3 “Reading Numbers”
- \texttt{#x} or \texttt{#X} starts a number or extflonum; see §1.3.3 “Reading Numbers”
- \texttt{#o} or \texttt{#U} starts a number or extflonum; see §1.3.3 “Reading Numbers”
A sequence that does not start with a delimiter or # is parsed as either a symbol, a number (see §1.3.3 “Reading Numbers”), or an extflonum (see §1.3.4 “Reading Extflonums”), except that . by itself is never parsed as a symbol or number (unless the read-accept-dot parameter is set to #f). A #\% also starts a symbol. The resulting symbol is interned. A successful number or extflonum parse takes precedence over a symbol parse.

When the read-case-sensitive parameter is set to #f, characters in the sequence that are not quoted by #\| or #\\ are first case-normalized. If the reader encounters #ci, #CI, #Ci, or #cI, then it recursively reads the following datum in case-insensitive mode. If the reader encounters #cs, #CS, #Cs, or #cS, then it recursively reads the following datum in case-
Examples:

Apple  reads equal to (string->symbol "Apple")
Ap#ple  reads equal to (string->symbol "Ap#ple")
Ap ple  reads equal to (string->symbol "Ap ple")
Ap|ple   reads equal to (string->symbol "Ap ple")
Ap\ple  reads equal to (string->symbol "Ap ple")
#ci Apple reads equal to (string->symbol "apple")
#ci |Apple reads equal to (string->symbol "Apple")
#ci \Apple reads equal to (string->symbol "Apple")
#ci#cs Apple reads equal to (string->symbol "#ci Apple")
#%Apple  reads equal to (string->symbol "#%Apple")

1.3.3 Reading Numbers

A sequence that does not start with a delimiter is parsed as a number when it matches the following grammar case-insensitively for \(\langle\text{number}_{10}\rangle\) (decimal), where \(n\) is a meta-meta-variable in the grammar. The resulting number is interned in read-syntax mode.

A number is optionally prefixed by an exactness specifier, \(#e\) (exact) or \(#i\) (inexact), which specifies its parsing as an exact or inexact number; see §4.3 “Numbers” for information on number exactness. As the non-terminal names suggest, a number that has no exactness specifier and matches only \(\langle\text{inexact-number}\rangle\) is normally parsed as an inexact number, otherwise it is parsed as an exact number. If the read-decimal-as-inexact parameter is set to \(#f\), then all numbers without an exactness specifier are instead parsed as exact.

If the reader encounters \(#b\) (binary), \(#o\) (octal), \(#d\) (decimal), or \(#x\) (hexadecimal), it must be followed by a sequence that is terminated by a delimiter or end-of-file, and that is either an extflonum (see §1.3.4 “Reading Extflonums”) or matches the \(\langle\text{general-number}\rangle\), \(\langle\text{general-number}_{8}\rangle\), \(\langle\text{general-number}_{10}\rangle\), or \(\langle\text{general-number}_{16}\rangle\) grammar, respectively.

A \(#e\) or \(#i\) followed immediately by \(#b\), \(#o\), \(#d\), or \(#x\) is treated the same as the reverse order: \(#b\), \(#o\), \(#d\), or \(#x\) followed by \(#e\) or \(#i\).

An \(\langle\text{exponent-mark}\rangle\) in an inexact number serves both to specify an exponent and to specify a numerical precision. If single-flonums are supported (see §4.3 “Numbers”) and the read-single-flonum parameter is set to \(#t\), the marks \(f\) and \(s\) specify single-flonums. If read-single-flonum is set to \(#f\), or with any other mark, a double-precision flonum is produced. If single-flonums are not supported and read-single-flonum is set to \(#t\), then the exn:fail:unsupported exception is raised when a single-flonum would otherwise be produced. Special infinity and not-a-number flonums and single-flonums are distinct; specials with the \(+\) suffix, like +nan.0, are double-precision flonums, while specials with the \(+\) suffix, like +nan.0, are single-flonums if enabled though read-single-flonum.

§3.2 “Numbers” in The Racket Guide introduces the syntax of numbers.
A # in an \textit{inexact} number is the same as 0, but # can be used to suggest that the digit’s actual value is unknown.

All letters in a number representation are parsed case-insensitively, independent of the \texttt{read-case-sensitive} parameter. For example, \#I\#D+InF.F+3I is parsed the same as \#i#d+inf.f+3i. In the grammar below, each literal lowercase letter stands for both itself and its uppercase form.

\begin{verbatim}
number := exact | inexact
exact := exact-rational | exact-complex
exact-rational := [sign] \(\text{unsigned-rational}\)

\[ \begin{align*}
\text{exact-complex} & := \text{exact-rational} \ \text{sign} \ \text{unsigned-rational} \ \text{i} \\
\text{inexact} & := \text{inexact-real} | \text{inexact-complex} \\
\text{inexact-real} & := [\text{sign}] \ \text{inexact-normal} \\
\text{inexact-unsigned} & := \text{inexact-normal} | \text{inexact-special} \\
\text{inexact-normal} & := \text{inexact-simple} \ \text{[exp-mark]} \ \text{exact-integer} \\
\text{inexact-simple} & := \text{digits}^+ \ | \ \text{[exp-mark]} \ \text{digits}^+ \\
\text{digits}^+ & := \text{digits}\ [.] \ #^* \ | \ \text{[digits}\ [.] \ #^* \\
\text{[sign]} & := * | = \\
\text{digits}_{16} & := \text{digits}_{10} | a \ | b \ | c \ | d \ | e \ | f \\
\text{digits}_{10} & := \text{digits}_{8} | 8 \ | 9 \\
\text{digits}_{8} & := \text{digits}_{2} | 2 \ | 3 \ | 4 \ | 5 \ | 6 \ | 7 \\
\text{digits}_{2} & := 0 \ | 1 \\
\text{exp-mark}_{16} & := s \ | l \\
\text{exp-mark}_{10} & := \text{exp-mark}_{16} | d \ | e \ | f \\
\text{exp-mark}_{8} & := \text{exp-mark}_{10} \\
\text{exp-mark}_{2} & := \text{exp-mark}_{10} \\
\text{general-number} & := [\text{exactness}] \ [number] \\
\text{exactness} & := \#e \ | \ #i
\end{align*} \]
\end{verbatim}

Examples:

\begin{verbatim}
-1 reads equal to -1
1/2 reads equal to (/ 1 2)
1.0 reads equal to (exact->inexact 1)
1+2i reads equal to (make-rectangular 1 2)
1/2+3/4i reads equal to (make-rectangular (/ 1 2) (/ 3 4))
\end{verbatim}
1.0+3.0e71 reads equal to (exact->inexact (make-rectangular 1 3000000))
2e5 reads equal to (exact->inexact 200000)
#i5 reads equal to (exact->inexact 5)
#e2e5 reads equal to 200000
#x2e5 reads equal to 741
#b101 reads equal to 5

1.3.4 Reading Extflonums

An extflonum has the same syntax as an ⟨inexact-realn⟩ that includes an ⟨exp-markn⟩, but
with t or T in place of the ⟨exp-markn⟩. In addition, +inf.t, =inf.t, +nan.t, =nan.t are
extflonums. A #b (binary), #o (octal), #d (decimal), or #x (hexadecimal) radix specifi-
cation can prefix an extflonum, but #i or #e cannot, and a extflonum cannot be used to form
a complex number. The read-decimal-as-inexact parameter has no effect on extflonum
reading.

1.3.5 Reading Booleans

A #true, #t, #T followed by a delimiter is the input syntax for the boolean constant “true,”
and #false, #f, or #F followed by a delimiter is the complete input syntax for the boolean
constant “false.”

1.3.6 Reading Pairs and Lists

When the reader encounters a ⟨, ⟩, or {, it starts parsing a pair or list; see §4.10 “Pairs and
Lists” for information on pairs and lists.

To parse the pair or list, the reader recursively reads data until a matching ⟨⟩, ⟩, or } (respec-
tively) is found, and it specially handles a delimited . . Pairs ⟨⟩, ⟩, and } are treated the
same way, so the remainder of this section simply uses “parentheses” to mean any of these
pair.

If the reader finds no delimited . . among the elements between parentheses, then it produces
a list containing the results of the recursive reads.

If the reader finds two data between the matching parentheses that are separated by a de-
limited . . , then it creates a pair. More generally, if it finds two or more data where the last
datum is preceded by a delimited . . , then it constructs nested pairs: the next-to-last element
is paired with the last, then the third-to-last datum is paired with that pair, and so on.

If the reader finds three or more data between the matching parentheses, and if a pair of
delimited . . s surrounds any other than the first and last elements, the result is a list containing
the element surrounded by .s as the first element, followed by the others in the read order. This convention supports a kind of infix notation at the reader level.

In read-syntax mode, the recursive reads for the pair/list elements are themselves in read-syntax mode, so that the result is a list or pair of syntax objects that is itself wrapped as a syntax object. If the reader constructs nested pairs because the input included a single delimited ., then only the innermost pair and outermost pair are wrapped as syntax objects.

Whether wrapping a pair or list, if the pair or list was formed with [ and ], then a 'paren-shape property is attached to the result with the value #\[. If the read-square-bracket-with-tag parameter is set to #t, then the resulting pair or list is wrapped by the equivalent of (cons '#%brackets pair-or-list).

Similarly, if the list or pair was formed with { and }, then a 'paren-shape property is attached to the result with the value #\{. If the read-curly-brace-with-tag parameter is set to #t, then the resulting pair or list is wrapped by the equivalent of (cons '#%braces pair-or-list).

If a delimited . appears in any other configuration, then the exn:fail:read exception is raised. Similarly, if the reader encounters a ), ], or } that does not end a list being parsed, then the exn:fail:read exception is raised.

Examples:

<table>
<thead>
<tr>
<th>Input</th>
<th>Output</th>
</tr>
</thead>
<tbody>
<tr>
<td>()</td>
<td>(list)</td>
</tr>
<tr>
<td>(1 2 3)</td>
<td>(list 1 2 3)</td>
</tr>
<tr>
<td>{1 2 3}</td>
<td>(list 1 2 3)</td>
</tr>
<tr>
<td>[1 2 3]</td>
<td>(list 1 2 3)</td>
</tr>
<tr>
<td>(1 (2) 3)</td>
<td>(list 1 (list 2) 3)</td>
</tr>
<tr>
<td>(1 , 3)</td>
<td>(cons 1 3)</td>
</tr>
<tr>
<td>(1 , (3))</td>
<td>(list 1 3)</td>
</tr>
<tr>
<td>(1 . 2 . 3)</td>
<td>(list 2 1 3)</td>
</tr>
</tbody>
</table>

If the read-square-bracket-as-paren and read-square-bracket-with-tag parameters are set to #f, then when the reader encounters [ and ], the exn:fail:read exception is raised. Similarly, if the read-curly-brace-as-paren and read-curly-brace-with-tag parameters are set to #f, then when the reader encounters { and }, the exn:fail:read exception is raised.

If the read-accept-dot parameter is set to #f, then a delimited . triggers an exn:fail:read exception. If the read-accept-infix-dot parameter is set to #f, then multiple delimited .s trigger an exn:fail:read exception, instead of the infix conversion.
1.3.7 Reading Strings

When the reader encounters "$", it begins parsing characters to form a string. The string continues until it is terminated by another "$" (that is not escaped by \\). The resulting string is interned in read-syntax mode.

Within a string sequence, the following escape sequences are recognized:

- \a: alarm (ASCII 7)
- \b: backspace (ASCII 8)
- \t: tab (ASCII 9)
- \n: linefeed (ASCII 10)
- \v: vertical tab (ASCII 11)
- \f: formfeed (ASCII 12)
- \r: return (ASCII 13)
- \e: escape (ASCII 27)
- "\": double-quotes (without terminating the string)
- "\": quote (i.e., the backslash has no effect)
- \\: backslash (i.e., the second is not an escaping backslash)
- \d\{1,3\}: Unicode for the octal number specified by \d\{1,3\} (i.e., 1 to 3 \d\{1,3\}'s), where each \d\{1,3\} is 0, 1, 2, 3, 4, 5, 6, or 7. A longer form takes precedence over a shorter form, and the resulting octal number must be between 0 and 255 decimal, otherwise the exn:fail:read exception is raised.
- \x\{1,2\}: Unicode for the hexadecimal number specified by \x\{1,2\}, where each \x\{1,2\} is 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, a, b, c, d, e, or f (case-insensitive). The longer form takes precedence over the shorter form.
- \u\{4,4\}: like \u, but with up to four hexadecimal digits (longer sequences take precedence). The resulting hexadecimal number must be a valid argument to integer->char, otherwise the exn:fail:read exception is raised—unless the encoding continues with another \u to form a surrogate-style encoding.
- \u\{4,4\}\u\{4,4\}: like \u, but for two hexadecimal numbers, where the first is in the range #xD800 to #xDBFF and the second is in the range #xDC00 to #xDFFF; the resulting character is the one represented by the numbers as a UTF-16 surrogate pair.
• \(\langle{\text{digit}_{16}}\rangle^{[1,8]}\): like \(\langle{\text{x}}\rangle\), but with up to eight hexadecimal digits (longer sequences take precedence). The resulting hexadecimal number must be a valid argument to \texttt{integer->char}, otherwise the \texttt{exn:fail:read} exception is raised.

• \(\langle{\text{newline}}\rangle\): elided, where \(\langle{\text{newline}}\rangle\) is either a linefeed, carriage return, or carriage return–linefeed combination. This convention allows single-line strings to span multiple lines in the source.

If the reader encounters any other use of a backslash in a string constant, the \texttt{exn:fail:read} exception is raised.

A string constant preceded by \# is parsed as a byte string. (That is, \#\texttt{nn} starts a byte-string literal.) See §4.5 “Byte Strings” for information on byte strings. The resulting byte string is interned in \texttt{read-syntax} mode. Byte-string constants support the same escape sequences as character strings, except \texttt{\u} and \texttt{\U}. Otherwise, each character within the byte-string quotes must have a Unicode code-point number in the range 0 to 255, which is used as the corresponding byte’s value; if a character is not in that range, the \texttt{exn:fail:read} exception is raised.

When the reader encounters \#<<, it starts parsing a here string. The characters following \#<< until a newline character define a terminator for the string. The content of the string includes all characters between the \#<< line and a line whose only content is the specified terminator. More precisely, the content of the string starts after a newline following \#<<, and it ends before a newline that is followed by the terminator, where the terminator is itself followed by either a newline or end-of-file. No escape sequences are recognized between the starting and terminating lines; all characters are included in the string (and terminator) literally. A return character is not treated as a line separator in this context. If no characters appear between \#<< and a newline or end-of-file, or if an end-of-file is encountered before a terminating line, the \texttt{exn:fail:read} exception is raised.

Examples:

"Apple"    reads equal to "Apple"
"\x41pple" reads equal to "Apple"
"\"Apple\"" reads equal to "\x22Apple\x22"
"\\"      reads equal to "\x5C"
#"Apple"   reads equal to (bytes 65 112 112 108 101)

### 1.3.8 Reading Quotes

When the reader encounters \#, it recursively reads one datum and forms a new list containing the symbol \texttt{quote} and the following datum. This convention is mainly useful for reading Racket code, where \texttt{"s} can be used as a shorthand for \texttt{(quote \texttt{s})}.

Several other sequences are recognized and transformed in a similar way. Longer prefixes take precedence over short ones:
adds quote
adds quasiquote
adds unquote
@ adds unquote-splicing
#\ adds syntax
#_ adds quasisyntax
#\ adds unsyntax
#,@ adds unsyntax-splicing

Examples:
apple reads equal to (list 'quote 'apple)
(1 ,2) reads equal to (list 'quasiquote (list 1 (list 'unquote 2)))

The &, , and ,@ forms are disabled when the read-accept-quasiquote parameter is set to #f, in which case the exn:fail:read exception is raised instead.

1.3.9 Reading Comments

A ; starts a line comment. When the reader encounters ;, it skips past all characters until the next linefeed (ASCII 10), carriage return (ASCII 13), next-line (Unicode 133), line-separator (Unicode 8232), or paragraph-separator (Unicode 8233) character.

A #\ starts a nestable block comment. When the reader encounters #\, it skips past all characters until a closing #\. Pairs of matching #\ and #\ can be nested.

A #; starts an S-expression comment. When the reader encounters #; it recursively reads one datum, and then discards it (continuing on to the next datum for the read result).

A #! (which is #! followed by a space) or #!/ starts a line comment that can be continued to the next line by ending a line with \\. This form of comment normally appears at the beginning of a Unix script file.

Examples:
; comment reads equal to nothing
#\ a #\ 1 reads equal to 1
#\ #\ a #\ 1 #\ 2 reads equal to 2
#; 2 reads equal to 2
#!/bin/sh reads equal to nothing
#!/bin/sh reads equal to nothing
1.3.10 Reading Vectors

When the reader encounters a #, #l, or #f, it starts parsing a vector; see §4.12 “Vectors” for information on vectors. A #l in place of # starts an flvector, but is not allowed in read-syntax mode; see §4.3.3.2 “Flonum Vectors” for information on flvectors. A #f in place of # starts an fxvector, but is not allowed in read-syntax mode; see §4.3.4.2 “Fixnum Vectors” for information on fxvectors. The #l, #f, #fl, #fl, #fx, and #fx forms can be disabled through the read-square-bracket-as-paren and read-curly-brace-as-paren parameters.

The elements of the vector are recursively read until a matching ), ], or } is found, just as for lists (see §1.3.6 “Reading Pairs and Lists”). A delimited . is not allowed among the vector elements. In the case of flvectors, the recursive read for element is implicitly prefixed with #i and must produce a flonum. In the case of fxvectors, the recursive read for element is implicitly prefixed with #e and must produce a fixnum.

An optional vector length can be specified between #, #fl, #fx and (, [, or {. The size is specified using a sequence of decimal digits, and the number of elements provided for the vector must be no more than the specified size. If fewer elements are provided, the last provided element is used for the remaining vector slots; if no elements are provided, then 0 is used for all slots.

In read-syntax mode, each recursive read for vector elements is also in read-syntax mode, so that the wrapped vector’s elements are also wrapped as syntax objects, and the vector is immutable.

Examples:

#(1 apple 3) reads equal to (vector 1 'apple 3)
#3("apple" "banana") reads equal to (vector "apple" "banana" "banana")
#3() reads equal to (vector 0 0 0)

1.3.11 Reading Structures

When the reader encounters a #s, #s, or #s, it starts parsing an instance of a prefab structure type; see §5 “Structures” for information on structure types. The #s and #s forms can be disabled through the read-square-bracket-as-paren and read-curly-brace-as-paren parameters.

The elements of the structure are recursively read until a matching ), ], or } is found, just as for lists (see §1.3.6 “Reading Pairs and Lists”). A single delimited . is not allowed among the elements, but two ..s can be used as in a list for an infix conversion.

The first element is used as the structure descriptor, and it must have the form (when quoted) of a possible argument to make-prefab-struct; in the simplest case, it can be a symbol. The remaining elements correspond to field values within the structure.
In **read-syntax** mode, the structure type must not have any mutable fields. The structure’s elements are read in **read-syntax** mode, so that the wrapped structure’s elements are also wrapped as syntax objects.

If the first structure element is not a valid prefab structure type key, or if the number of provided fields is inconsistent with the indicated prefab structure type, the **exn:fail:read** exception is raised.

### 1.3.12 Reading Hash Tables

A **#hash** starts an immutable hash-table constant with key matching based on **equal?**. The characters after **hash** must parse as a list of pairs (see §1.3.6 "Reading Pairs and Lists") with a specific use of delimited `::` it must appear between the elements of each pair in the list and nowhere in the sequence of list elements. The first element of each pair is used as the key for a table entry, and the second element of each pair is the associated value.

A **#hasheq** starts a hash table like **#hash**, except that it constructs a hash table based on **eq?** instead of **equal?**.

A **#hasheqv** starts a hash table like **#hash**, except that it constructs a hash table based on **eqv?** instead of **equal?**.

In all cases, the table is constructed by adding each mapping to the hash table from left to right, so later mappings can hide earlier mappings if the keys are equivalent.

Examples, where **make-...** stands for **make-immutable-hash**:

- **#hash()** reads equal to (**make-... '()'**)
- **#hasheq()** reads equal to (**make-... eq '()'**)
- **#hash(("a" . 5))** reads equal to (**make-... '(('"a" . 5))'**)
- **#hasheq((a . 5) (b . 7))** reads equal to (**make-... eq '((b . 7) (a . 5))'**)
- **#hasheqv((a . 5) (a . 7))** reads equal to (**make-... eqv '((a . 7))'**)

### 1.3.13 Reading Boxes

When the reader encounters a **#&**, it starts parsing a box; see §4.13 "Boxes" for information on boxes. The content of the box is determined by recursively reading the next datum.

In **read-syntax** mode, the recursive read for the box content is also in **read-syntax** mode, so that the wrapped box’s content is also wrapped as a syntax object, and the box is immutable.

Examples:

- **#&17** reads equal to (**box 17**)
1.3.14 Reading Characters

A \#\n starts a character constant, which has one of the following forms:

- \#\null or \#\null: NUL (ASCII 0); the next character must not be alphabetic.
- \#\backspace: backspace (ASCII 8); the next character must not be alphabetic.
- \#\tab: tab (ASCII 9); the next character must not be alphabetic.
- \#\newline or \#\linefeed: linefeed (ASCII 10); the next character must not be alphabetic.
- \#\vtab: vertical tab (ASCII 11); the next character must not be alphabetic.
- \#\page: page break (ASCII 12); the next character must not be alphabetic.
- \#\return: carriage return (ASCII 13); the next character must not be alphabetic.
- \#\space: space (ASCII 32); the next character must not be alphabetic.
- \#\rubout: delete (ASCII 127); the next character must not be alphabetic.
- \#\digit{3,3}: Unicode for the octal number specified by three octal digits—as in string escapes (see §1.3.7 “Reading Strings”), but constrained to exactly three digits.
- \#\u\digit{1,4}: Unicode for the hexadecimal number specified by \digit{1,4}, as in string escapes (see §1.3.7 “Reading Strings”).
- \#\\U\digit{1,8}: like \#\u, but with up to eight hexadecimal digits (although only six digits are actually useful).
- \#\c: the character \c, as long as \c and the characters following it do not match any of the previous cases, as long as \c or the character after \c is not alphabetic, and as long as \c is not an octal digit or is not followed by an octal digit (i.e., two octal digits commit to a third).

Examples:
\#\newline reads equal to (integer->char 10)
\#\ reads equal to (integer->char 110)
\#\3BBB reads equal to (integer->char 955)
\#\ reads equal to (integer->char 955)

1.3.15 Reading Keywords

A \#\: starts a keyword. The parsing of a keyword after the \#: is the same as for a symbol, including case-folding in case-insensitive mode, except that the part after \#\: is never parsed as a number. The resulting keyword is interned.
Examples:
#:Apple reads equal to (string->keyword "Apple")
#:1 reads equal to (string->keyword "1")

1.3.16 Reading Regular Expressions

A #rx or #px starts a regular expression. The characters immediately after #rx or #px must parse as a string or byte string (see §1.3.7 "Reading Strings"). A #rx prefix starts a regular expression as would be constructed by regexp, #px as constructed by pregexp, #rx# as constructed by byte-regexp, and #px# as constructed by byte-pregexp. The resulting regular expression is interned in read-syntax mode.

Examples:
#rx".*" reads equal to (regexp ".*")
#px"[\s]*" reads equal to (pregexp "[\s]*")
#rx#".*" reads equal to (byte-regexp #".*")
#px#"[\s]*" reads equal to (byte-pregexp #"[\s]*")

1.3.17 Reading Graph Structure

A #<digit10>\[^{1,8}]# tags the following datum for reference via #<digit10>\[^{1,8}]#, which allows the reader to produce a datum that has graph structure. Neither form is allowed in read-syntax mode.

For a specific #<digit10>\[^{1,8}]# in a single read result, each #<digit10>\[^{1,8}]# reference is replaced by the datum read for the corresponding #<digit10>\[^{1,8}]#; the definition #<digit10>\[^{1,8}]# also produces just the datum after it. A #<digit10>\[^{1,8}]# definition can appear at most once, and a #<digit10>\[^{1,8}]# definition must appear before a #<digit10>\[^{1,8}]# reference appears, otherwise the exn:fail:read exception is raised. If the read-accept-graph parameter is set to #f, then #<digit10>\[^{1,8}]# or #<digit10>\[^{1,8}]# triggers a exn:fail:read exception.

Although a comment parsed via #; discards the datum afterward, #<digit10>\[^{1,8}]# definitions in the discarded datum still can be referenced by other parts of the reader input, as long as both the comment and the reference are grouped together by some other form (i.e., some recursive read); a top-level #; comment neither defines nor uses graph tags for other top-level forms.

Examples:
(#1=100 #1# #1#) reads equal to (list 100 100 100)
#0=(1 . #0#) reads equal to (let* ([ph (make-placeholder #f)]
  [v (cons 1 ph)])
  (placeholder-set! ph v)
  (make-reader-graph v))
1.3.18 Reading via an Extension

When the reader encounters #reader, it loads an external reader procedure and applies it to the current input stream.

The reader recursively reads the next datum after #reader, and passes it to the procedure that is the value of the current-reader-guard parameter; the result is used as a module path. The module path is passed to dynamic-require with either 'read or 'read-syntax (depending on whether the reader is in read or read-syntax mode). The module is loaded in a root namespace of the current namespace.

The arity of the resulting procedure determines whether it accepts extra source-location information: a read procedure accepts either one argument (an input port) or five, and a read-syntax procedure accepts either two arguments (a name value and an input port) or six. In either case, the four optional arguments are the reader’s module path (as a syntax object in read-syntax mode) followed by the line (positive exact integer or #f), column (non-negative exact integer or #f), and position (positive exact integer or #f) of the start of the #reader form. The input port is the one whose stream contained #reader, where the stream position is immediately after the recursively read module path.

The procedure should produce a datum result. If the result is a syntax object in read mode, then it is converted to a datum using syntax->datum; if the result is not a syntax object in read-syntax mode, then it is converted to one using datum->syntax. See also §13.7.2 "Reader-Extension Procedures" for information on the procedure’s results.

If the read-accept-reader parameter is set to #f, then if the reader encounters #reader, the exn:fail:read exception is raised.

The #lang reader form is similar to #reader, but more constrained: the #lang must be followed by a single space (ASCII 32), and then a non-empty sequence of alphanumeric ASCII, +, -, _, and/or / characters terminated by whitespace or an end-of-file. The sequence must not start or end with /. A sequence #lang <name> is equivalent to either #reader (submod <name> reader) or #reader <name>/lang/reader, where the former is tried first guarded by a module-declared? check (but after filtering by current-reader-guard, so both are passed to the value of current-reader-guard if the latter is used). Note that the terminating whitespace (if any) is not consumed before the external reading procedure is called.

Finally, #! is an alias for #lang followed by a space when #! is followed by alphanumeric ASCII, +, -, or _. Use of this alias is discouraged as needed to construct programs that conform to certain grammars, such as that of R6RS [Sperber07].

By convention, #lang normally appears at the beginning of a file, possibly after comment forms, to specify the syntax of a module.

If the read-accept-reader or read-accept-lang parameter is set to #f, then if the
reader encounters #lang or equivalent #!, the exn:fail:read exception is raised.

1.3.19 Reading with C-style Infix-Dot Notation

When the read-cdot parameter is set to #t, then a variety of changes occur in the reader.

First, symbols can no longer include the character ., unless the . is quoted with | or \

Second, numbers can no longer include the character ., unless the number is prefixed with #e, #i, #b, #o, #d, #x, or an equivalent prefix as discussed in §1.3.3 “Reading Numbers”. If these numbers are followed by a . intended to be read as a C-style infix dot, then a delimiter must precede the .

Finally, after reading any datum x, the reader will seek through whitespace, BOM characters, and comments and look for zero or more sequences of a . followed by another datum y. It will then group x and y together in a #%dot form so that x.y reads equal to (#%dot x y).

If x.y has another . after it, the reader will accumulate more .-separated datums, grouping them from left-to-right. For example, x.y.z reads equal to (#%dot (#%dot x y) z).

In read-syntax mode, the #%dot symbol has the source location information of the . character and the entire list has the source location information spanning from the start of x to the end of y.

S-Expression Reader Language

#lang s-exp package: base

The s-exp “language” is a kind of meta-language. It reads the S-expression that follows #lang s-exp and uses it as the language of a module form. It also reads all remaining S-expressions until an end-of-file, using them for the body of the generated module.

That is,

#lang s-exp module-path

form ...

is equivalent to

(module name-id module-path

form ...)

where name-id is derived from the source input port’s name: if the port name is a filename path, the filename without its directory path and extension is used for name-id, otherwise name-id is anonymous-module.
Chaining Reader Language

#lang reader package: base

The reader “language” is a kind of meta-language. It reads the S-expression that follows #lang reader and uses it as a module path (relative to the module being read) that effectively takes the place of reader. In other words, the reader meta-language generalizes the syntax of the module specified after #lang to be a module path, and without the implicit addition of /lang/reader to the path.

1.4 The Printer

The Racket printer supports three modes:

- write mode prints core datatypes in such a way that using read on the output produces a value that is equal? to the printed value;
- display mode prints core datatypes in a more “end-user” style rather than “programmer” style; for example, a string displays as its content characters without surrounding "s or escapes;
- print mode by default—when print-as-expression is #t—prints most datatypes in such a way that evaluating the output as an expression produces a value that is equal? to the printed value; when print-as-expression is set to #f, then print mode is like write mode.

In print mode when print-as-expression is #t (as is the default), a value prints at a quoting depth of either 0 (unquoted) or 1 (quoted). The initial quoting depth is accepted as an optional argument by print, and printing of some compound datatypes adjusts the print depth for component values. For example, when a list is printed at quoting depth 0 and all of its elements are quotable, the list is printed with a ' prefix, and the list’s elements are printed at quoting depth 1.

When the print-graph parameter is set to #t, then the printer first scans an object to detect cycles. The scan traverses the components of pairs, mutable pairs, vectors, boxes (when print-box is #t), hash tables (when print-hash-table is #t and when key are held strongly), fields of structures exposed by struct->vector (when print-struct is #t), and fields of structures exposed by printing when the structure’s type has the prop:custom-write property. If print-graph is #t, then this information is used to print sharing through graph definitions and references (see §1.3.17 “Reading Graph Structure”). If a cycle is detected in the initial scan, then print-graph is effectively set to #t automatically.

With the exception of displaying byte strings, printing is defined in terms of Unicode characters; see §13.1 “Ports” for information on how a character stream is written to a port’s underlying byte stream.
1.4.1 Printing Symbols

Symbols containing spaces or special characters write using escaping \ and quoting |s. When the read-case-sensitive parameter is set to #f, then symbols containing upper-case characters also use escaping \ and quoting |s. In addition, symbols are quoted with |s or leading \ when they would otherwise print the same as a numerical constant or as a delimited . (when read-accept-dot is #t).

When read-accept-bar-quote is #t, |s are used in printing when one | at the beginning and one | at the end suffice to correctly print the symbol. Otherwise, \s are always used to escape special characters, instead of quoting them with |s.

When read-accept-bar-quote is #f, then | is not treated as a special character. The following are always special characters:

  ( ) [ ] { } # . , + - ; \\

In addition, # is a special character when it appears at the beginning of the symbol, and when it is not followed by %.

Symbols display without escaping or quoting special characters. That is, the display form of a symbol is the same as the display form of symbol->string applied to the symbol.

Symbols print the same as they write, unless print-as-expression is set to #t (as is the default) and the current quoting depth is 0. In that case, the symbol’s printed form is prefixed with #. For the purposes of printing enclosing datatypes, a symbol is quotable.

1.4.2 Printing Numbers

A number prints the same way in write, display, and print modes. For the purposes of printing enclosing datatypes, a number is quotable.

A complex number that is not a real number always prints as \langle m\rangle+\langle n\rangle or \langle m\rangle-\langle n\rangle, where \langle m\rangle and \langle n\rangle (for a non-negative imaginary part) or \langle n\rangle (for a negative imaginary part) are the printed forms of the number’s numerator and denominator (as determined by numerator and denominator). A negative exact number prints with a - prefix on the printed form of the number’s exact negation. When printing a number as hexadecimal (e.g., via number->string), digits a through f are printed in lowercase. A #e or radix marker such as #d does not prefix the number.

A double-precision inexact number (i.e., a flonum) that is a rational number prints with
either a \( \cdot \) decimal point, an \( e \) exponent marker and non-zero exponent, or both. The form is selected to keep the output short, with the constraint that reading the printed form back in produces an \texttt{equal?} number. A \texttt{#i} does not prefix the number, and \#i is never used in place of a digit. A \texttt{+} does not prefix a positive number, but a \texttt{+} or \texttt{-} is printed before the exponent if \( e \) is present. Positive infinity prints as \texttt{+inf.0}, negative infinity prints as \texttt{-inf.0}, and not-a-number prints as \texttt{+nan.0}.

A single-precision inexact number that is a rational number prints like a double-precision number, but always with an exponent, using \texttt{f} in place of \( e \) to indicate the number’s precision; if the number would otherwise print without an exponent, \texttt{0} (with no \texttt{+}) is printed as the exponent part. Single-precision positive infinity prints as \texttt{+inf.f}, negative infinity prints as \texttt{-inf.f}, and not-a-number prints as \texttt{+nan.f}.

1.4.3 Printing Extflonums

An extflonum prints the same way in \texttt{write}, \texttt{display}, and \texttt{print} modes. For the purposes of printing enclosing datatypes, an extflonum is quotable.

An extflonum prints in the same way a single-precision inexact number (see §1.4.2 “Printing Numbers”), but always with a \texttt{t} or \texttt{T} exponent marker or as a suffix for \texttt{+inf.t}, \texttt{-inf.t}, or \texttt{+nan.t}. When extflonum operations are supported, printing always uses lowercase \texttt{t}; when extflonum operations are not supported, an extflonum prints the same as its reader (see §1.3 “The Reader”) source, since reading is the only way to produce an extflonum.

1.4.4 Printing Booleans

The boolean constant \texttt{#t} prints as \texttt{#true} or \texttt{#t} in all modes (\texttt{display}, \texttt{write}, and \texttt{print}), depending on the value of \texttt{print-boolean-long-form}, and the constant \texttt{#f} prints as \texttt{#false} or \texttt{#f}. For the purposes of printing enclosing datatypes, a symbol is quotable.

1.4.5 Printing Pairs and Lists

In \texttt{write} and \texttt{display} modes, an empty list prints as \texttt{()}. A pair normally prints starting with \((\) followed by the printed form of its \texttt{car}. The rest of the printed form depends on the \texttt{cdr}:

- If the \texttt{cdr} is a pair or the empty list, then the printed form of the pair completes with the printed form of the \texttt{cdr}, except that the leading \( ( \) in the \texttt{cdr}’s printed form is omitted.
- Otherwise, the printed form of the pair continues with a space, \( , \), another space, the printed form of the \texttt{cdr}, and a \( ) \).
If `print-reader-abbreviations` is set to #t, then pair printing in `write` mode is adjusted in the case of a pair that starts a two-element list whose first element is 'quote, 'quasiquote, 'unquote, 'unquote-splicing, 'syntax, 'quasisyntax, 'unsyntax, or 'unsyntax-splicing. In that case, the pair is printed with the corresponding reader syntax: "", "", \#, \#, \#, \#, or \#,@, respectively. After the reader syntax, the second element of the list is printed. When the list is a tail of an enclosing list, the tail is printed after a . in the enclosing list (after which the reader abbreviations work), instead of including the tail as two elements of the enclosing list. If the reader syntax , or #, is followed by a symbol that prints with a leading @, then the printer adds an extra space before the @.

The printed form of a pair is the same in both `write` and `display` modes, except as the printed form of the pair’s car and cdr vary with the mode. The `print` form is also the same if `print-as-expression` is #f or the quoting depth is 1.

For `print` mode when `print-as-expression` is #t and the quoting depth is 0, then the empty list prints as (()). For a pair whose car and cdr are quotable, the pair prints in `write` mode but with a # prefix; the pair’s content is printed with quoting depth 1. Otherwise, when the car or cdr is not quotable, then pair prints with either `cons` (when the cdr is not a pair), `list` (when the pair is a list), or `list*` (otherwise) after the opening (, any . that would otherwise be printed is suppressed, and the pair content is printed at quoting depth 0. In all cases, when `print-as-expression` is #t for `print` mode, then the value of `print-reader-abbreviations` is ignored and reader abbreviations are always used for lists printed at quoting depth 1.

By default, mutable pairs (as created with `mcons`) print the same as pairs for `write` and `display`, except that { and } are used instead of ( and ). Note that the reader treats {...} and (...)) equivalently on input, creating immutable pairs in both cases. Mutable pairs in `print` mode with `print-as-expression` as #f or a quoting depth of 1 also use { and }.

In `print` mode with `print-as-expression` as #t and a quoting depth of 0, a mutable pair prints as (mcons , the mcdr and mcar printed at quoting depth 0 and separated by a space, and a closing ).

If the `print-pair-curly-braces` parameter is set to #t, then pairs print using { and } when not using `print` mode with `print-as-expression` as #t and a quoting depth of 0. If the `print-mpair-curly-braces` parameter is set to #f, then mutable pairs print using ( and ) in that mode.

For the purposes of printing enclosing datatypes, an empty list is always quotable, a pair is quotable when its car and cdr are quotable, and a mutable list is never quotable.

Changed in version 6.9.0.6 of package base: Added a space when printing , or #, followed by a symbol that prints with a leading @.
1.4.6 Printing Strings

All strings display as their literal character sequences.

The write or print form of a string starts with " and ends with another ". Between the "s, each character is represented. Each graphic or blank character is represented as itself, with two exceptions: " is printed as \", and \ is printed as \\.

Each non-graphic, non-blank character (according to char-graphic? and char-blank?) is printed using the escape sequences described in §1.3.7 “Reading Strings” using \a, \b, \t, \n, \v, \f, \r, or \e if possible, otherwise using \u with four hexadecimal digits or \U with eight hexadecimal digits (using the latter only if the character value does not fit into four digits).

All byte strings display as their literal byte sequence; this byte sequence may not be a valid UTF-8 encoding, so it may not correspond to a sequence of characters.

The write or print form of a byte string starts with #" and ends with a ".

Between the "s, each byte is written using the corresponding ASCII decoding if the byte is between 0 and 127 and the character is graphic or blank (according to char-graphic? and char-blank?). Otherwise, the byte is written using \a, \b, \t, \n, \v, \f, \r, or \e if possible, otherwise using \ followed by one to three octal digits (only as many as necessary).

For the purposes of printing enclosing datatypes, a string or a byte string is quotable.

1.4.7 Printing Vectors

In display mode, the printed form of a vector is # followed by the printed form of vector->list applied to the vector. In write mode, the printed form is the same, except that when the print-vector-length parameter is #t, a decimal integer is printed after the #, and a repeated last element is printed only once.

Vectors print the same as they write, unless print-as-expression is set to #t and the current quoting depth is 0. In that case, if all of the vector’s elements are quotable, then the vector’s printed form is prefixed with # and its elements printed with quoting depth 1. If its elements are not all quotable, then the vector prints as (vector, the elements at quoting depth 0, and a closing )). A vector is quotable when all of its elements are quotable.

In write or display mode, a flvector prints like a vector, but with a #f prefix instead of #. A fxvector similarly prints with a #fx prefix instead of #. The print-vector-length parameter affects flvector and fxvector printing the same as vector printing. In print mode, flvectors and fxvectors are not quotable, and they print like a vector at quoting depth 0 using a (flvector or (fxvector) prefix, respectively.
1.4.8 Printing Structures

When the `print-struct` parameter is set to #t, then the way that structures print depends on details of the structure type for which the structure is an instance:

- If the structure type is a prefab structure type, then it prints in `write` or `display` mode using `#s( followed by the prefab structure type key, then the printed form of each field in the structure, and then `)`. 

  In `print` mode when `print-as-expression` is set to #t and the current quoting depth is 0, if the structure’s content is all quotable, then the structure’s `printed` form is prefixed with `' and its content is printed with quoting depth 1. If any of its content is not quotable, then the structure type prints the same as a non-prefab structure type. An instance of a prefab structure type is quotable when all of its content is quotable.

- If the structure has a `prop:custom-write` property value, then the associated procedure is used to print the structure, unless the `print-unreadable` parameter is set to #f.

  For `print` mode, an instance of a structure type with a `prop:custom-write` property is treated as quotable if it has the `prop:custom-print-quotable` property with a value of `'always`. If it has `'maybe` as the property value, then the structure is treated as quotable if its content is quotable, where the content is determined by the values recursively printed by the structure’s `prop:custom-write` procedure. Finally, if the structure has `'self` as the property value, then it is treated as quotable.

  In `print` mode when `print-as-expression` is #t, the structure’s `prop:custom-write` procedure is called with either 0 or 1 as the quoting depth, normally depending on the structure’s `prop:custom-print-quotable` property value. If the property value is `'always`, the quoting depth is normally 1. If the property value is `'maybe`, then the quoting depth is 1 if the structure is quotable, or normally 0 otherwise. If the property value is `'self, then the quoting depth may be 0 or 1; it is normally 0 if the structure is not printed as a part of an enclosing quotable value, even though the structure is treated as quotable. Finally, if the property value is `'never, then the quoting depth is normally 0. The quoting depth can vary from its normal value if the structure is printed with an explicit quoting depth of 1.

- If the structure’s type is transparent or if any ancestor is transparent (i.e., `struct?` on the instance produces #t), then the structure prints as the vector produced by `struct-vector in display` mode, in `write` mode, or in `print` mode when `print-as-expression` is set to #f or when the quoting depth is 0.

  In `print` mode with `print-as-expression` as #t and a quoting depth of 0, the structure content is printed with a `{ followed by the structure’s type name (as determined by `object-name`) in `write` mode; the remaining elements are `printed` at quoting depth 0 and separated by a space, and finally a closing `}`.

  A transparent structure type that is not a prefab structure type is never quotable.
For any other structure type, the structure prints as an unreadable value; see §1.4.15 “Printing Unreadable Values” for more information.

If the print-struct parameter is set to #f, then all structures without a prop:custom-write property print as unreadable values (see §1.4.15 “Printing Unreadable Values”) and count as quotable.

1.4.9 Printing Hash Tables

When the print-hash-table parameter is set to #t, in write and display modes, a hash table prints starting with #hash(), #hasheqv(), or #hasheq( for a table using equal?, eqv?, or eq? key comparisons, respectively, as long as the hash table retains keys strongly. After the prefix, each key–value mapping is shown as (the printed form of a key, a space, .., a space, the printed form the corresponding value, and ), with an additional space if the key–value pair is not the last to be printed. After all key–value pairs, the printed form completes with ).

In print mode when print-as-expression is #f or the quoting depth is 1, the printed form is the same as for write. Otherwise, if the hash table’s keys and values are all quotable, the table prints with a # prefix, and the table’s key and values are printed at quoting depth 1. If some key or value is not quotable, the hash table prints as (hash, (hasheqv), or (hasheq) followed by alternating keys and values printed at quoting depth 1 and separated by spaces, and finally a closing ). A hash table is quotable when all of its keys and values are quotable.

When the print-hash-table parameter is set to #f or when a hash table retains its keys weakly, a hash table prints as #<hash> and counts as quotable.

1.4.10 Printing Boxes

When the print-box parameter is set to #t, a box prints as #& followed by the printed form of its content in write, display, or print mode when print-as-expression is #f or the quoting depth is 1.

In print mode when print-as-expression is #t and the quoting depth is 0, a box prints with a # prefix and its value is printed at quoting depth 1 when its content is quotable, otherwise the box prints a (box) followed by the content at quoting depth 0 and a closing ). A box is quotable when its content is quotable.

When the print-box parameter is set to #f, a box prints as #<box> and counts as quotable.
1.4.11 Printing Characters

Characters with the special names described in §1.3.14 “Reading Characters” write and print using the same name. (Some characters have multiple names; the \newline and \null names are used instead of \linefeed and \null.) Other graphic characters (according to char-graphic?) write as \ followed by the single character, and all others characters are written in \u notation with four digits or \U notation with eight digits (using the latter only if the character value does not fit in four digits).

All characters display directly as themselves (i.e., a single character).

For the purposes of printing enclosing datatypes, a character is quotable.

1.4.12 Printing Keywords

Keywords write, print, and display the same as symbols (see §1.4.1 “Printing Symbols”) except with a leading #: (after any ' prefix added in print mode), and without special handling for an initial # or when the printed form would match a number or a delimited . (since #: distinguishes the keyword).

For the purposes of printing enclosing datatypes, a keyword is quotable.

1.4.13 Printing Regular Expressions

Regexp values write, display, and print starting with #px (for pregexp-based regexps) or #rx (for regexp-based regexps) followed by the write form of the regexp’s source string or byte string.

For the purposes of printing enclosing datatypes, a regexp value is quotable.

1.4.14 Printing Paths

Paths write and print as #<path:....>. A path displays the same as the string produced by path->string. For the purposes of printing enclosing datatypes, a path counts as quotable.

Although a path can be converted to a string with path->string or to a byte string with path->bytes, neither is clearly the right choice for printing a path and reading it back. If the path value is meant to be moved among platforms, then a string is probably the right choice, despite the potential for losing information when converting a path to a string. For a path that is intended to be re-read on the same platform, a byte string is probably the right choice,
since it preserves information in an unportable way. Paths do not print in a readable way so that programmers are not misled into thinking that either choice is always appropriate.

### 1.4.15 Printing Unreadable Values

For any value with no other printing specification, assuming that the `print-unreadable` parameter is set to `#t`, the output form is `#<something>`, where `<something>` is specific to the type of the value and sometimes to the value itself. If `print-unreadable` is set to `#f`, then attempting to print an unreadable value raises `exn:fail`.

For the purposes of printing enclosing datatypes, a value that prints unreadably nevertheless counts as quotable.

### 1.4.16 Printing Compiled Code

Compiled code as produced by `compile` prints using `#~`. Compiled code printed with `#~` is essentially assembly code for Racket, and reading such a form produces a compiled form when the `read-accept-compiled` parameter is set to `#t`.

Compiled code parsed from `#~` is marked as non-runnable if the current code inspector (see `current-code-inspector`) is not the original code inspector; on attempting to evaluate or reoptimize non-runnable bytecode, `exn:fail` exception is raised. Otherwise, compiled code parsed from `#~` may contain references to unexported or protected bindings from a module. Conceptually, the references in bytecode are associated with the current code inspector, where the code will only execute if that inspector controls the relevant module invocation (see §14.10 “Code Inspectors”)—but the original code inspector controls all other inspectors, anyway.

A compiled-form object may contain uninterned symbols (see §4.7 “Symbols”) that were created by `gensym` or `string->uninterned-symbol`. When the compiled object is read via `#~`, each uninterned symbol in the original form is mapped to a new uninterned symbol, where multiple instances of a single symbol are consistently mapped to the same new symbol. The original and new symbols have the same printed representation. Unreadable symbols, which are typically generated indirectly during expansion and compilation, are saved and restored consistently through `#~`.

The dynamic nature of uninterned symbols and their localization within `#~` can cause problems when `gensym` or `string->uninterned-symbol` is used to construct an identifier for a top-level or module binding (depending on how the identifier and its references are compiled). To avoid problems, generate distinct identifiers either with `generate-temporaries` or by applying the result of `make-syntax-introducer` to an existing identifier; those functions lead to top-level and module variables with unreadable symbolic names, and the names are deterministic as long as expansion is otherwise deterministic.
When a compiled-form object has string and byte string literals, they are interned using `datum-intern-literal` when the compiled-object for is read back in. Numbers and other values that `read-syntax` would intern, however, are not interned when read back as quoted literals in a compiled object.

A compiled form may contain path literals. Although paths are not normally printed in a way that can be read back in, path literals can be written and read as part of compiled code. The `current-write-relative-directory` parameter is used to convert the path to a relative path as it is written, and then `current-load-relative-directory` parameter (falling back to `current-directory`) is used to convert any relative path back as it is read.

For a path in a syntax object’s source, if the `current-write-relative-directory` parameter is not set or the path is not relative to the value of the `current-write-relative-directory` parameter, then the path is coerced to a string that preserves only part of the path (an in effort to make it less tied to the build-time filesystem, which can be different than the run-time filesystem).

Finally, a compiled form may contain `srcloc` structures if the source field of the structure is a path for some system, a string, a byte string, a symbol, or `#f`. For a path value (matching the current platform’s convention), if the path cannot be recorded as a relative path based on `current-write-relative-directory`, then it is converted to a string with at most two path elements; if the path contains more than two elements, then the string contains `.../`, the next-to-last element, `/` and the last element. The intent of the constraints on `srcloc` values and the conversion of the source field is to preserve some source information but not expose or record a path that makes no sense on a different filesystem or platform.

For internal testing purposes in the BC implementation of Racket, when the `PLT_VALIDATE_LOAD` environment variable is set, the reader runs a validator on bytecode parsed from `#`. The validator may catch miscompilations or bytecode-file corruption. The validator may run lazily, such as checking a procedure only when the procedure is called.

Changed in version 6.90.0.21 of package base: Adjusted the effect of changing the code inspector on parsed bytecode, causing the reader to mark the loaded code as generally unrunnable instead of rejecting at read time references to unsafe operations.

Changed in version 7.0: Allowed some `srcloc` values embedded in compiled code.

### 1.5 Implementations

The definition of Racket aims for determinism and independence from its implementation. Nevertheless, some details inevitably vary with the implementation. Racket currently has two main implementations:

- The CS implementation is the default implementation as of Racket version 8.0. This variant is called “CS” because it uses Chez Scheme as its core compiler and runtime system.
The CS implementation typically provides the best performance for Racket programs. Compiled Racket CS code in a ".zo" file normally contains machine code that is specific to an operating system and architecture.

- The BC implementation was the default implementation up until version 7.9. The “BC” label stands for “before Chez” or “bytecode.” Compiled Racket BC code in a ".zo" file normally contains platform-independent bytecode that is further compiled to machine code “just in time” as the code is loaded.

Racket BC has two variants: 3m and CGC. The difference is the garbage collection implementation, where 3m uses a garbage collector that moves objects in memory (an effect that is visible to foreign libraries, for example) and keeps precise track of allocated objects, while CGC uses a “conservative” collector that requires less cooperation from an embedding foreign environment. The 3m subvariant tends to perform much better than CGC, and it became the default variant in version 370 (which would be v3.7 in the current versioning convention).

Most Racket programs run the same in all implementation variants, but some Racket features are available only on some implementation variants, and the interaction of Racket and foreign functions is significantly different across the variants. Use system-type to get information about the current running implementation.
2 Notation for Documentation

This chapter introduces essential terminology and notation that is used throughout Racket documentation.

2.1 Notation for Module Documentation

Since Racket programs are organized into modules, documentation reflects that organization with an annotation at the beginning of a section or subsection that describes the bindings that a particular module provides.

For example, the section that describes the functionality provided by `racket/list` starts

```
(require racket/list)  package: base
```

Instead of `require`, some modules are introduced with `#lang`:

```
#lang racket/base  package: base
```

Using `#lang` means that the module is normally used as the language of a whole module—that is, by a module that starts `#lang` followed by the language—instead of imported with `require`. Unless otherwise specified, however, a module name documented with `#lang` can also be used with `require` to obtain the language’s bindings.

The module annotation also shows the package that the module belongs to on the right-hand side. For more details about packages, see Package Management in Racket.

Sometimes, a module specification appears at the beginning of a document or at the start of a section that contains many subsections. The document’s section or section’s subsections are meant to “inherit” the module declaration of the enclosing document or section. Thus, bindings documented in `The Racket Reference` are available from `racket` and `racket/base` unless otherwise specified in a section or subsection.

2.2 Notation for Syntactic Form Documentation

Syntactic forms are specified with a grammar. Typically, the grammar starts with an open parenthesis followed by the syntactic form’s name, as in the grammar for `if`:

```
(if test-expr then-expr else-expr)
```
Since every form is expressed in terms of syntax objects, parentheses in a grammar specification indicate a syntax object wrapping a list, and the leading if is an identifier that starts the list whose binding is the if binding of the module being documented—in this case, racket/base. Square brackets in the grammar indicate a syntax-object list in the same way as parentheses, but in places square brackets are normally used by convention in a program’s source.

Italic identifiers in the grammar are metavariables that correspond to other grammar productions. Certain metavariable names have implicit grammar productions:

- A metavariable that ends in id stands for an identifier.
- A metavariable that ends in keyword stands for a syntax-object keyword.
- A metavariable that ends with expr stands for any form, and the form will be parsed as an expression.
- A metavariable that ends with body stands for any form; the form will be parsed as either a local definition or an expression. A body can parse as a definition only if it is not preceded by any expression, and the last body must be an expression; see also §1.2.3.8 “Internal Definitions”.
- A metavariable that ends with datum stands for any form, and the form is normally uninterpreted (e.g., quoted).
- A metavariable that ends with number or boolean stands for any syntax-object (i.e., literal) number or boolean, respectively.

In a grammar, form ... stands for any number of forms (possibly zero) matching form, while form ...+ stands for one or more forms matching form.

Metavariables without an implicit grammar are defined by productions alongside the syntactic form’s overall grammar. For example, in

```
(lambda formals body ...+)
```

```
formals = id
  | (id ...)
  | (id ...+. rest-id)
```

the formals metavariable stands for either an identifier, zero or more identifiers in a syntax-object list, or a syntax object corresponding to a chain of one or more pairs where the chain ends in an identifier instead of an empty list.

Some syntactic forms have multiple top-level grammars, in which case the documentation of the syntactic forms shows multiple grammars. For example,
indicates that init-rest can either be alone in its syntax-object list or followed by a single identifier.

Finally, a grammar specification that includes expr metavariables may be augmented with run-time contracts on some of the metavariables, which indicate a predicate that the result of the expression must satisfy at run time. For example,

```
(parameterize ([parameter-expr value-expr] ...) 
  body ...+)
```

indicates that the result of each parameter-expr must be a value \( v \) for which (parameter? \( v \)) returns true.

### 2.3 Notation for Function Documentation

Procedures and other values are described using a notation based on contracts. In essence, these contracts describe the interfaces of the documented library using Racket predicates and expressions.

For example, the following is the header of the definition of a typical procedure:

```
(char->integer char) -> exact-integer?
  char : char?
```

The function being defined, char->integer, is typeset as if it were being applied. The metavariables that come after the function name stand in for arguments. The white text in the corner identifies the kind of value that is being documented.

Each metavariable is described with a contract. In the preceding example, the metavariable char has the contract char?. This contract specifies that any argument char that answers true to the char? predicate is valid. The documented function may or may not actually check this property, but the contract signals the intent of the implementer.

The contract on the right of the arrow, exact-integer? in this case, specifies the expected result that is produced by the function.
Contract specifications can be more expressive than just names of predicates. Consider the following header for `argmax`:

```scheme
(argmax proc lst) -> any
  proc : (-> any/c real?)
  lst : (and/c pair? list?)
```

The contract `(-> any/c real?)` denotes a function contract specifying that `proc`'s argument can be any single value and the result should be a real number. The contract `(and/c pair? list?)` for `lst` specifies that `lst` should pass both `pair?` and `list?` (i.e., that it is a non-empty list).

Both `->` and `and/c` are examples of contract combinators. Contract combinators such as `or/c`, `cons/c`, `listof`, and others are used throughout the documentation. Clicking on the hyperlinked combinator name will provide more information on its meaning.

A Racket function may be documented as having one or more optional arguments. The `read` function is an example of such a function:

```scheme
(read [in]) -> any
  in : input-port? = (current-input-port)
```

The brackets surrounding the `in` argument in the application syntax indicates that it is an optional argument.

The header for `read` specifies a contract for the parameter `in` as usual. To the right of the contract, it also specifies a default value `(current-input-port)` that is used if `read` is called with no arguments.

Functions may also be documented as accepting mandatory or optional keyword-based arguments. For example, the `sort` function has two optional, keyword-based arguments:

```scheme
(sort lst
  less-than?
  #:key extract-key
  #:cache-keys? cache-keys?) -> list?
  lst : list?
  less-than? : (any/c any/c . -> . any/c)
  extract-key : (any/c . -> . any/c) = (lambda (x) x)
  cache-keys? : boolean? = #f
```

The brackets around the `extract-key` and `cache-keys?` arguments indicate that they are optional as before. The contract section of the header shows the default values that are provided for these keyword arguments.
2.4 Notation for Structure Type Documentation

A structure type is also documented using contract notation:

```
(struct color (red green blue alpha))
red : (and/c natural-number/c (<=/c 255))
green : (and/c natural-number/c (<=/c 255))
blue : (and/c natural-number/c (<=/c 255))
alpha : (and/c natural-number/c (<=/c 255))
```

The structure type is typeset as it were declared in the source code of a program using the `struct` form. Each field of the structure is documented with a corresponding contract that specifies the values that are accepted for that field.

In the example above, the structure type `color` has four fields: `red`, `green`, `blue`, and `alpha`. The constructor for the structure type accepts field values that satisfy `(and/c natural-number/c (<=/c 255))`, i.e., non-negative exact integers up to 255.

Additional keywords may appear after the field names in the documentation for a structure type:

```
(struct data-source (connector args extensions)
  #:mutable)
  connector : (or/c 'postgresql 'mysql 'sqlite3 'odbc)
  args : list?
  extensions : (listof (list/c symbol? any/c))
```

Here, the `#:mutable` keyword indicates that the fields of instances of the `data-source` structure type can be mutated with their respective setter functions.

2.5 Notation for Parameter Documentation

A parameter is documented the same way as a function:

```
(current-command-line-arguments) -> (vectorof string?)
(current-command-line-arguments argv) -> void?
argv : (vectorof (and/c string? immutable?))
```

Since parameters can be referenced or set, there are two entries in the header above. Calling `current-command-line-arguments` with no arguments accesses the parameter’s value,
which must be a vector whose elements pass both `string?` and `immutable?`. Calling `current-command-line-arguments` with a single argument sets the parameter’s value, where the value must be a vector whose elements pass `string?` (and a guard on the parameter coerces the strings to immutable form, if necessary).

### 2.6 Notation for Other Documentation

Some libraries provide bindings to constant values. These values are documented with a separate header:

```
object% : class?
```

The `racket/class` library provides the `object%` value, which is the root of the class hierarchy in Racket. Its documentation header just indicates that it is a value that satisfies the predicate `class?`. 
3 Syntactic Forms

This section describes the core syntax forms that appear in a fully expanded expression, plus many closely related non-core forms. See §1.2.3.1 “Fully Expanded Programs” for the core grammar.

Notation

Each syntactic form is described by a BNF-like notation that describes a combination of (syntax-wrapped) pairs, symbols, and other data (not a sequence of characters). These grammatical specifications are shown as in the following specification of a something form:

\[
\text{(something } \text{id thing-expr } \ldots)\\
\text{thing-expr : number?}
\]

Within such specifications,

- \(\ldots\) indicates zero or more repetitions of the preceding datum; more generally, \(N\) consecutive \(\ldots\) a row indicate a consecutive repetition of the preceding \(N\) datums.
- \(\ldots+\) indicates one or more repetitions of the preceding datum.
- Italic meta-identifiers play the role of non-terminals. Some meta-identifier names imply syntactic constraints:
  - A meta-identifier that ends in \(id\) stands for an identifier.
  - A meta-identifier that ends in \(keyword\) stands for a keyword.
  - A meta-identifier that ends with \(expr\) (such as \(\text{thing-expr}\)) stands for a sub-form that is expanded as an expression.
  - A meta-identifier that ends with \(body\) stands for a sub-form that is expanded in an internal-definition context (see §1.2.3.8 “Internal Definitions”).
- Contracts indicate constraints on sub-expression results. For example, \(\text{thing-expr : number?}\) indicates that the expression \(\text{thing-expr}\) must produce a number.

3.1 Modules: \(\text{module, module*}, \ldots\)

\[
\text{(module } \text{id module-path form } \ldots)
\]

Declares a top-level module or a submodule. For a top-level module, if the \(\text{current-module-declare-name}\) parameter is set, the parameter value is used for the module name
and id is ignored, otherwise (quote id) is the name of the declared module. For a submodule, id is the name of the submodule to be used as an element within a submod module path.

The module-path form must be as for require, and it supplies the initial bindings for the body forms. That is, it is treated like a (require module-path) prefix before the forms, except that the bindings introduced by module-path can be shadowed by definitions and requires in the module body forms.

If a single form is provided, then it is partially expanded in a module-begin context. If the expansion leads to #%plain-module-begin, then the body of the #%plain-module-begin is the body of the module. If partial expansion leads to any other primitive form, then the form is wrapped with #%module-begin using the lexical context of the module body; this identifier must be bound by the initial module-path import, and its expansion must produce a #%plain-module-begin to supply the module body. Finally, if multiple forms are provided, they are wrapped with #%module-begin, as in the case where a single form does not expand to #%plain-module-begin.

After such wrapping, if any, and before any expansion, an 'enclosing-module-name property is attached to the #%module-begin syntax object (see §12.7 "Syntax Object Properties"); the property’s value is a symbol corresponding to id.

Each form is partially expanded (see §1.2.3.7 "Partial Expansion") in a module context. Further action depends on the shape of the form:

- If it is a begin form, the sub-forms are flattened out into the module’s body and immediately processed in place of the begin.
- If it is a define-syntaxes form, then the right-hand side is evaluated (in phase 1), and the binding is immediately installed for further partial expansion within the module. Evaluation of the right-hand side is parameterized to set current-namespace as in let-syntax.
- If it is a begin-for-syntax form, then the body is expanded (in phase 1) and evaluated. Expansion within a begin-for-syntax form proceeds with the same partial-expansion process as for a module body, but in a higher phase, and saving all #%provide forms for all phases until the end of the module’s expansion. Evaluation of the body is parameterized to set current-namespace as in let-syntax.
- If the form is a #%require form, bindings are introduced immediately, and the imported modules are instantiated or visited as appropriate.
- If the form is a #%provide form, then it is recorded for processing after the rest of the body.
- If the form is a define-values form, then the binding is installed immediately, but the right-hand expression is not expanded further.
• If the form is a `module` form, then it is immediately expanded and declared for the extent of the current top-level enclosing module’s expansion.

• If the form is a `module*` form, then it is not expanded further.

• Similarly, if the form is an expression, it is not expanded further.

After all forms have been partially expanded this way, then the remaining expression forms (including those on the right-hand side of a definition) are expanded in an expression context. After all expression forms, `%%provide` forms are processed in the order in which they appear (independent of phase) in the expanded module. Finally, all `module*` forms are expanded in order, so that each becomes available for use by subsequent `module*` forms; the enclosing module itself is also available for use by `module*` submodules.

The scope of all imported identifiers covers the entire module body, except for nested `module` and `module*` forms (assuming a non-#f `module-path` in the latter case). The scope of any identifier defined within the module body similarly covers the entire module body except for such nested `module` and `module*` forms. The ordering of syntax definitions does not affect the scope of the syntax names; a transformer for A can produce expressions containing B, while the transformer for B produces expressions containing A, regardless of the order of declarations for A and B. However, a syntactic form that produces syntax definitions must be defined before it is used.

No identifier can be imported or defined more than once at any phase level within a single module, except that a definition via `define-values` or `define-syntaxes` can shadow a preceding import via `%%require`; unless the shadowed import is from the module’s initial `module-path`, a warning is logged to the initial logger. Every exported identifier must be imported or defined. No expression can refer to a top-level variable. A `module*` form in which the enclosing module’s bindings are visible (i.e., a nested `module*` with #f instead of a `module-path`) can define or import bindings that shadow the enclosing module’s bindings.

The evaluation of a `module` form does not evaluate the expressions in the body of the module (except sometimes for redeclarations; see §1.1.9.4 “Module Redeclarations”). Evaluation merely declares a module, whose full name depends both on id or (current-module-declare-name).

A module body is executed only when the module is explicitly instantiated via `require` or `dynamic-require`. On invocation, imported modules are instantiated in the order in which they are required into the module (although earlier instantiations or transitive requires can trigger the instantiation of a module before its order within a given module). Then, expressions and definitions are evaluated in order as they appear within the module. Each evaluation of an expression or definition is wrapped with a continuation prompt (see `call-with-continuation-prompt`) for the default prompt tag and using a prompt handler that re-aborts and propagates its argument to the next enclosing prompt. Each evaluation of a definition is followed, outside of the prompt, by a check that each of the definition’s variables
has a value; if the portion of the prompt-delimited continuation that installs values is skipped, then the `exn:fail:contract:variable?` exception is raised.

Accessing a module-level variable before it is defined signals a run-time error, just like accessing an undefined global variable. If a module (in its fully expanded form) does not contain a set! for an identifier that defined within the module, then the identifier is a constant after it is defined; its value cannot be changed afterward, not even through reflective mechanisms. The `compile-enforce-module-constants` parameter, however, can be used to disable enforcement of constants.

When a syntax object representing a `module` form has a `module-language` syntax property attached, and when the property value is a vector of three elements where the first is a module path (in the sense of `module-path?`) and the second is a symbol, then the property value is preserved in the corresponding compiled and/or declared module. The third component of the vector should be printable and readable, so that it can be preserved in marshaled bytecode. The `racket/base` and `racket` languages attach `'#(racket/language-info get-info #f)` to a module form. See also `module-compiled-language-info`, `module->language-info`, and `racket/language-info`.

See also §1.1.9 “Modules and Module-Level Variables”, §1.2.3.9 “Module Expansion, Phases, and Visits” and §12.9.1 “Information on Expanded Modules”.

Example:

```
> (module duck racket/base
  (provide num-eggs quack)
  (define num-eggs 2)
  (define (quack n)
    (unless (zero? n)
      (printf "quack\n")
      (quack (sub1 n))))
```

Changed in version 6.3 of package `base`: Changed `define-syntaxes` and `define-values` to shadow any preceding import, and dropped the use of `submodule` syntax property values on nested `module` or `module*` forms.

```
(module* id module-path form ...)
(module* id #f form ...)
```

Like `module`, but only for declaring a submodule within a module, and for submodules that may require the enclosing module.

Instead of a `module-path` after `id`, `#f` indicates that all bindings from the enclosing module are visible in the submodule. In that case, `begin-for-syntax` forms that wrap the `module*` form shift the phase level of the enclosing module’s bindings relative to the submodule. The macro expander handles such nesting by shifting the phase level of the `module*` form so that its body starts at phase level 0, expanding, and then reverting the phase level shift; beware...

§6.2.3 “Submodules” in The Racket Guide introduces `module*`. 

87
that this process can leave syntax objects as \texttt{origin} syntax property values out-of-sync with the expanded module.

When a \texttt{module*} form has a \texttt{module-path}, the submodule expansion starts by removing the scopes of the enclosing module, the same as the \texttt{module} form. No shifting compensates for any begin-for-syntax forms that may wrap the submodule.

\begin{verbatim}
(module+ id form ...)
\end{verbatim}

Declares and/or adds to a submodule named \texttt{id}.

Each addition for \texttt{id} is combined in order to form the entire submodule using (\texttt{module* id #f ...}) at the end of the enclosing module. If there is only one \texttt{module+} for a given \texttt{id}, then (\texttt{module+ id form ...}) is equivalent to (\texttt{module* id #f form ...}), but still moved to the end of the enclosing module.

When a module contains multiple submodules declared with \texttt{module+}, then the relative order of the initial \texttt{module+} declarations for each submodule determines the relative order of the \texttt{module*} declarations at the end of the enclosing module.

A submodule must not be defined using \texttt{module+ and module} or \texttt{module*}. That is, if a submodule is made of \texttt{module+} pieces, then it must be made only of \texttt{module+} pieces.

\begin{verbatim}
(#%module-begin form ...)
\end{verbatim}

Legal only in a module begin context, and handled by the \texttt{module} and \texttt{module*} forms.

The \texttt{#%module-begin} form of \texttt{racket/base} wraps every top-level expression to print non-\texttt{#<void>} results using \texttt{current-print}.

The \texttt{#%module-begin} form of \texttt{racket/base} also declares a \texttt{configure-runtime} submodule (before any other \texttt{form}), unless some \texttt{form} is either an immediate \texttt{module} or \texttt{module*} form with the name \texttt{configure-runtime}. If a \texttt{configure-runtime} submodule is added, the submodule calls the \texttt{configure} function of \texttt{racket/runtime-config}.

\begin{verbatim}
(#%printing-module-begin form ...)
\end{verbatim}

Legal only in a module begin context.

Like \texttt{#%module-begin}, but without adding a \texttt{configure-runtime} submodule.

\begin{verbatim}
(#%plain-module-begin form ...)
\end{verbatim}

Legal only in a module begin context, and handled by the \texttt{module} and \texttt{module*} forms.

\begin{verbatim}
(#%declare declaration-keyword ...)
\end{verbatim}
Declarations that affect run-time or reflective properties of the module:

• #:cross-phase-persistent — declares the module as cross-phase persistent, and reports a syntax error if the module does not meet the import or syntactic constraints of a cross-phase persistent module.

• #:empty-namespace — declares that module->namespace for this module should produce a namespace with no bindings; limiting namespace support in this way can reduce the lexical information that otherwise must be preserved for the module.

• #:unsafe — declares that the module can be compiled without checks that could trigger exn:fail:contract, and the resulting behavior is unspecified for an evaluation where exn:fail:contract should have been raised; see also §17 “Unsafe Operations”. For example, a use of car can be compiled as a use of unsafe-car, and the behavior is unspecified if unsafe-car is applied to a non-pair. The #:unsafe declaration keyword is allowed only when the current code inspector is the initial one. Macros can generate conditionally unsafe code, depending on the expansion context, by expanding to a use of (variable-reference-from-unsafe? (#%variable-reference)).

A #%declare form must appear in a module context or a module-begin context. Each declaration-keyword can be declared at most once within a module body.

Changed in version 6.3 of package base: Added #:empty-namespace.

Changed in version 7.9.0.5: Added #:unsafe.

3.2 Importing and Exporting: require and provide

(require require-spec ...)
require-spec = module-path \\
| (only-in require-spec id-maybe-renamed ...) \\
| (except-in require-spec id ...) \\
| (prefix-in prefix-id require-spec) \\
| (rename-in require-spec [orig-id bind-id] ...) \\
| (combine-in require-spec ...) \\
| (relative-in module-path require-spec ...) \\
| (only-meta-in phase-level require-spec ...) \\
| (for-syntax require-spec ...) \\
| (for-template require-spec ...) \\
| (for-label require-spec ...) \\
| (for-meta phase-level require-spec ...) \\
| derived-require-spec

module-path = root-module-path \\
| (submod root-module-path submod-path-element ...) \\
| (submod ".." submod-path-element ...) \\
| (submod "." submod-path-element ...)

root-module-path = (quote id) \\
| rel-string \\
| (lib rel-string ...+) \\
| id \\
| (file string) \\
| (planet id) \\
| (planet string) \\
| (planet rel-string \\
| (user-string pkg-string vers) \\
| rel-string ...)

submod-path-element = id \\
| "." \\

id-maybe-renamed = id \\
| [orig-id bind-id]

phase-level = exact-integer \\
| #f 

vers = \\
| nat \\
| nat minor-vers

minor-vers = nat \\
| (nat nat) \\
| (= nat) \\
| (+ nat) \\
| (- nat)
In a top-level context, require instantiates modules (see §1.1.9 “Modules and Module-Level Variables”). In a top-level context or module context, expansion of require visits modules (see §1.2.3.9 “Module Expansion, Phases, and Visits”). In both contexts and both evaluation and expansion, require introduces bindings into a namespace or a module (see §1.2.3.4 “Introducing Bindings”). A require form in a expression context or internal-definition context is a syntax error.

A require-spec designates a particular set of identifiers to be bound in the importing context. Each identifier is mapped to a particular export of a particular module; the identifier to bind may be different from the symbolic name of the originally exported identifier. Each identifier also binds at a particular phase level.

No identifier can be bound multiple times in a given phase level by an import, unless all of the bindings refer to the same original definition in the same module. In a module context, an identifier can be either imported or defined for a given phase level, but not both.

The syntax of require-spec can be extended via define-require-syntax, and when multiple require-specs are specified in a require, the bindings of each require-spec are visible for expanding later require-specs. The pre-defined forms (as exported by racket/base) are as follows:

```scheme
(module-path
  Imports all exported bindings from the named module, using the export identifiers as the local identifiers. (See below for information on module-path.)
  The lexical context of the module-path form determines the context of the introduced identifiers.

(only-in require-spec id-maybe-renamed ...)
  Like require-spec, but constrained to those exports for which the identifiers to bind match id-maybe-renamed: as id or as orig-id in [orig-id bind-id]. If the id or orig-id of any id-maybe-renamed is not in the set that require-spec describes, a syntax error is reported.

Examples:
> (require (only-in racket/tcp
              tcp-listen
              [tcp-accept my-accept]))
> tcp-listen
<procedure:tcp-listen>
> my-accept
<procedure:tcp-accept>
> tcp-accept
```

91
tcp-accept: undefined;
cannot reference an identifier before its definition
in module: top-level

(\texttt{except-in require-spec id \ldots})

Like \texttt{require-spec}, but omitting those imports for which \texttt{id}s are the identifiers to bind; if any \texttt{id} is not in the set that \texttt{require-spec} describes, a syntax error is reported.

Examples:

\begin{verbatim}
> (require (except-in racket/tcp tcp-listen))
> tcp-accept
#<procedure:tcp-accept>
> tcp-listen
tcp-listen: undefined;
cannot reference an identifier before its definition
in module: top-level
\end{verbatim}

(prefix-in prefix-id require-spec)

Like \texttt{require-spec}, but adjusting each identifier to be bound by prefixing it with \texttt{prefix-id}. The lexical context of the \texttt{prefix-id} is ignored, and instead preserved from the identifiers before prefixing.

Examples:

\begin{verbatim}
> (require (prefix-in tcp: racket/tcp))
> tcp:tcp-accept
#<procedure:tcp-accept>
> tcp:tcp-listen
#<procedure:tcp-listen>
\end{verbatim}

(rename-in require-spec [orig-id bind-id] \ldots)

Like \texttt{require-spec}, but replacing the identifier to bind \texttt{orig-id} with \texttt{bind-id}; if any \texttt{orig-id} is not in the set that \texttt{require-spec} describes, a syntax error is reported.

Examples:
\[
\begin{align*}
\textit{The union of the require-specs. If two or more imports from the require-specs have the same identifier name but they do not refer to the same original binding, a syntax error is reported.}
\end{align*}
\]

Examples:

\[
\begin{align*}
\textit{Like the combination of require-specs, but removing any binding that is not for phase-level, where #f for phase-level corresponds to the label phase level.}
\end{align*}
\]

The following example imports bindings only at phase level 1, the transform phase:

\[
\begin{align*}
\textit{Like the union of the require-specs, but each relative module path in a require-spec is treated as relative to module-path instead of the enclosing context.}
\end{align*}
\]

The require transformer that implements relative-in sets current-require-module-path to adjust module paths in the require-specs.
(define-for-syntax meta-eggs 2)
(define-for-syntax meta-chicks 3)
(define num-eggs 2))
> (require (only-meta-in 1 'nest))
> (define-syntax (desc stx)
  (printf "s ~n meta-eggs meta-chicks)
#'(void))
> (desc)
  2 3
> num-eggs
  num-eggs: undefined;
  cannot reference an identifier before its definition
  in module: top-level

The following example imports only bindings at phase level 0, the normal phase.

> (require (only-meta-in 0 'nest))
> num-eggs
  2

[[for-meta phase-level require-spec ...]]

Like the combination of require-specs, but the binding specified by each require-spec is shifted by phase-level. The label phase level corresponds to #f, and a shifting combination that involves #f produces #f.

Examples:

> (module nest racket
  (provide num-eggs)
  (define num-eggs 2))
> (require (for-meta 0 'nest))
> num-eggs
  2
> (require (for-meta 1 'nest))
> (define-syntax (roost stx)
  (datum->syntax stx num-eggs))
> (roost)
  2

[[for-syntax require-spec ...]]

Same as (for-meta 1 require-spec ...).
(for-template require-spec ...)  
Same as (for-meta -1 require-spec ...).

(for-label require-spec ...)  
Same as (for-meta #f require-spec ...). If an identifier in any of the require-specs is bound at more than one phase level, a syntax error is reported.

derived-require-spec

See define-require-syntax for information on expanding the set of require-spec forms.

A module-path identifies a module, either a root module or a submodule that is declared lexically within another module. A root module is identified either through a concrete name in the form of an identifier, or through an indirect name that can trigger automatic loading of the module declaration. Except for the (quote id) case below, the actual resolution of a root module path is up to the current module name resolver (see current-module-name-resolver), and the description below corresponds to the default module name resolver.

(quote id)

Refers to a submodule previously declared with the name id or a module previously declared interactively with the name id. When id refers to a submodule, (quote id) is equivalent to (submod "." id).

Examples:

; a module declared interactively as test:  
> (require 'test)

rel-string

A path relative to the containing source (as determined by current-load-relative-directory or current-directory). Regardless of the current platform, rel-string is always parsed as a Unix-format relative path: / is the path delimiter (multiple adjacent /s are not allowed), .. accesses the parent directory, and . accesses the current directory. The path cannot be empty or contain a leading or trailing slash, path elements before than the last one cannot
include a file suffix (i.e., a . in an element other than . or ..), and the only allowed characters are ASCII letters, ASCII digits, =, +, _, /, and %. Furthermore, a % is allowed only when followed by two lowercase hexadecimal digits, and the digits must form a number that is not the ASCII value of a letter, digit, =, +, or _.

If rel-string ends with a "..ss" suffix, it is converted to a "..rkt" suffix. The compiled-load handler may reverse that conversion if a "..rkt" file does not exist and a "..ss" exists.

Examples:

; a module named "x.rkt" in the same
; directory as the enclosing module's file:
> (require "x.rkt")

; a module named "x.rkt" in the parent directory
; of the enclosing module file's directory:
> (require "../x.rkt")

(lib rel-string ...+)

A path to a module installed into a collection (see §18.2 “Libraries and Collections”). The rel-strings in lib are constrained similar to the plain rel-string case, with the additional constraint that a rel-string cannot contain . or .. directory indicators.

The specific interpretation of the path depends on the number and shape of the rel-strings:

• If a single rel-string is provided, and if it consists of a single element (i.e., no /) with no file suffix (i.e., no .), then rel-string names a collection, and "main.rkt" is the library file name.

Examples:

; the main swindle library:
> (require (lib "swindle"))

; the same:
> (require (lib "swindle/main.rkt"))

• If a single rel-string is provided, and if it consists of multiple /-separated elements, then each element up to the last names a collection, subcollection, etc., and the last element names a file. If the last element has no file suffix, ".rkt" is added, while a ".ss" suffix is converted to ".rkt".

Examples:

; "turbo.rkt" from the "swindle" collection:
; (require (lib "swindle/turbo"))
; the same:
; (require (lib "swindle/turbo.rkt"))
; the same:
; (require (lib "swindle/turbo.ss"))

• If a single rel-string is provided, and if it consists of a single element with a file suffix (i.e., with a .), then rel-string names a file within the "mzlib" collection. A ".ss" suffix is converted to ".rkt". (This convention is for compatibility with older version of Racket.)

Examples:

; "tar.rkt" module from the "mzlib" collection:
; (require (lib "tar.ss"))

• Otherwise, when multiple rel-strings are provided, the first rel-string is effectively moved after the others, and all rel-strings are appended with / separators. The resulting path names a collection, then subcollection, etc., ending with a file name. No suffix is added automatically, but a ".ss" suffix is converted to ".rkt". (This convention is for compatibility with older version of Racket.)

Examples:

; "tar.rkt" module from the "mzlib" collection:
; (require (lib "tar.ss" "mzlib"))

id

A shorthand for a lib form with a single rel-string whose characters are the same as in the symbolic form of id. In addition to the constraints of a lib rel-string, id must not contain ..

Example:

; (require racket/tcp)

(file string)

Similar to the plain rel-string case, but string is a path—possibly absolute—using the current platform's path conventions and expand-user-path. A ".ss" suffix is converted to ".rkt".

Example:

; (require (file "~/tmp/x.rkt"))
(planet id)
(planet string)
(planet rel-string (user-string pkg-string vers)
  rel-string ...)

Specifies a library available via the PLaneT server.

The first form is a shorthand for the last one, where the id's character sequence
must match the following spec grammar:

\[
\begin{align*}
\text{spec} & ::= \text{owner} \mid \text{pkg} \mid \text{lib} \\
\text{owner} & ::= \text{elem} \\
\text{pkg} & ::= \text{elem} \mid \text{elem} \mid \text{version} \\
\text{version} & ::= \text{int} \mid \text{int} \mid \text{minor} \\
\text{minor} & ::= \text{int} \mid \text{int} \mid \text{int} \mid \text{int} \\
\text{lib} & ::= \text{empty} \mid \text{path} \\
\text{path} & ::= \text{elem} \mid \text{elem} \mid \text{path}
\end{align*}
\]

and where an elem is a non-empty sequence of characters that are ASCII letters, ASCII digits, \_ , \+, \-, or \% followed by lowercase hexadecimal digits (that do not encode one of the other allowed characters), and an int is a non-
empty sequence of ASCII digits. As this shorthand is expended, a .plt extension is added to pkg, and a .rkt extension is added to path; if no path is included, "main.rkt" is used in the expansion.

A (planet string) form is like a (planet id) form with the identifier
collapsed to a string, except that the string can optionally end with a file
extension (i.e., a .) for a path. A .ss file extension is converted to .rkt.

In the more general last form of a planet module path, the rel-strings are
similar to the lib form, except that the (user-string pkg-string vers) names a PLaneT-based package instead of a collection. A version specification
in the more general last form of a planet module path, the rel-strings are
similar to the lib form, except that the (user-string pkg-string vers) names a PLaneT-based package instead of a collection. A version specification
include an optional major and minor version, where the minor version can
be a specific number or a constraint: (nat nat) specifies an inclusive range,
(= nat) specifies an exact match, (+ nat) specifies a minimum version and
is equivalent to just nat, and (- nat) specifies a maximum version. The =, +, and - identifiers in a minor-version constraint are recognized symbolically.

Examples:

; "main.rkt" in package "farm" by "mcdonald":
> (require (planet mcdonald/farm))
; "main.rkt" in version >= 2.0 of "farm" by "mcdonald":
> (require (planet mcdonald/farm:2))
; "main.rkt" in version >= 2.5 of "farm" by "mcdonald":
> (require (planet mcdonald/farm:2:5))
; "duck.rkt" in version >= 2.5 of "farm" by "mcdonald":
> (require (planet mcdonald/farm:2:5/duck))
(submod \textit{root-module-path} \textit{submod-path-element} ...) \\
(submod "." \textit{submod-path-element} ...) \\
(submod ".." \textit{submod-path-element} ...)

Identifies a submodule within the module specified by \textit{root-module-path} or relative to the current module in the case of (submod "." ...), where (submod ".." \textit{submod-path-element} ...) is equivalent to (submod "." ".." \textit{submod-path-element} ...). Submodules have symbolic names, and a sequence of identifiers as \textit{submod-path-elements} determine a path of successively nested submodules with the given names. A ".." as a \textit{submod-path-element} names the enclosing module of a submodule, and it’s intended for use in (submod "." ...) and (submod ".." ...) forms.

As \texttt{require} prepares to handle a sequence of \texttt{require-specs}, it logs a “prefetch” message to the current logger at the '\texttt{info}' level, using the name '\texttt{module-prefetch}', and including message data that is a list of two elements: a list of module paths that appear to be imported, and a directory path to use for relative module paths. The logged list of module paths may be incomplete, but a compilation manager can use approximate prefetch information to start on compilations in parallel.

Changed in version 6.0.1.10 of package base: Added prefetch logging.

(local-require \texttt{require-spec} ...)

Like \texttt{require}, but for use in a internal-definition context to import just into the local context. Only bindings from phase level 0 are imported.

Examples:

```
> (let ()
  (local-require racket/control)
  fcontrol)
<procedure:fcontrol>
> fcontrol
fcontrol: undefined;
cannot reference an identifier before its definition
in module: top-level
```

(provide \texttt{provide-spec} ...)

§6.5 “Exports: provide” in \textit{The Racket Guide} introduces \texttt{provide}.
\[
\text{provide-spec} = \text{id} \\
\text{| (all-defined-out)} \\
\text{| (all-from-out module-path ...)} \\
\text{| (rename-out [orig-id export-id] ...)} \\
\text{| (except-out provide-spec provide-spec ...)} \\
\text{| (prefix-out prefix-id provide-spec)} \\
\text{| (struct-out id)} \\
\text{| (combine-out provide-spec ...)} \\
\text{| (protect-out provide-spec ...)} \\
\text{| (for-meta phase-level provide-spec ...)} \\
\text{| (for-syntax provide-spec ...)} \\
\text{| (for-template provide-spec ...)} \\
\text{| (for-label provide-spec ...)} \\
\text{| derived-provide-spec}
\]

\[
\text{phase-level} = \text{exact-integer} \\
\text{| #f}
\]

Declares exports from a module. A provide form must appear in a module context or a module-begin context.

A provide-spec indicates one or more bindings to provide. For each exported binding, the external name is a symbol that can be different from the symbolic form of the identifier that is bound within the module. Also, each export is drawn from a particular phase level and exported at the same phase level; by default, the relevant phase level is the number of begin-for-syntax forms that enclose the provide form.

The syntax of provide-spec can be extended by bindings to provide transformers or provide pre-transformers, such as via define-provide-syntax, but the pre-defined forms are as follows.

\[
\text{id}
\]

Exports id, which must be bound within the module (i.e., either defined or imported) at the relevant phase level. The symbolic form of id is used as the external name, and the symbolic form of the defined or imported identifier must match (otherwise, the external name could be ambiguous).

Examples:

\[
> \ (\text{module nest racket} \\
> (\text{provide num-eggs}) \\
> (\text{define num-eggs 2})) \\
> \ (\text{require 'nest}) \\
> \ \text{num-eggs}
\]

2

100
If \( id \) has a transformer binding to a rename transformer, then the transformer affects the exported binding. See \texttt{make-rename-transformer} for more information.

\section{(all-defined-out)}

Exports all identifiers that are defined at the relevant phase level within the exporting module, and that have the same lexical context as the \( \text{(all-defined-out)} \) form, excluding bindings to rename transformers where the target identifier has the \texttt{not-provide-all-defined} syntax property. The external name for each identifier is the symbolic form of the identifier. Only identifiers accessible from the lexical context of the \( \text{(all-defined-out)} \) form are included; that is, macro-introduced imports are not re-exported, unless the \( \text{(all-defined-out)} \) form was introduced at the same time.

Examples:

\begin{verbatim}
> (module nest racket
  (provide (all-defined-out))
  (define num-eggs 2))
> (require 'nest)
> num-eggs
2
\end{verbatim}

\section{(all-from-out module-path ...)}

Exports all identifiers that are imported into the exporting module using a \texttt{require-spec} built on each \texttt{module-path} (see §3.2 "Importing and Exporting: \texttt{require} and \texttt{provide}") with no phase-level shift. The symbolic name for export is derived from the name that is bound within the module, as opposed to the symbolic name of the export from each \texttt{module-path}. Only identifiers accessible from the lexical context of the \texttt{module-path} are included; that is, macro-introduced imports are not re-exported, unless the \texttt{module-path} was introduced at the same time.

Examples:

\begin{verbatim}
> (module nest racket
  (provide num-eggs)
  (define num-eggs 2))
> (module hen-house racket
  (require 'nest)
  (provide (all-from-out 'nest)))
> (require 'hen-house)
> num-eggs
2
\end{verbatim}
(rename-out [orig-id export-id] ...)

Exports each orig-id, which must be bound within the module at the relevant phase level. The symbolic name for each export is export-id instead of orig-id.

Examples:

> (module nest racket
      (provide (rename-out [count num-eggs]))
      (define count 2))
> (require 'nest)
> num-eggs
2
> count
  count: undefined;
cannot reference an identifier before its definition
  in module: top-level

(export-out provide-spec provide-spec ...)

Like the first provide-spec, but omitting the bindings listed in each subsequent provide-spec. If one of the latter bindings is not included in the initial provide-spec, a syntax error is reported. The symbolic export name information in the latter provide-specs is ignored; only the bindings are used.

Examples:

> (module nest racket
      (provide (except-out (all-defined-out)
                          num-chicks))
      (define num-eggs 2)
      (define num-chicks 3))
> (require 'nest)
> num-eggs
2
> num-chicks
  num-chicks: undefined;
cannot reference an identifier before its definition
  in module: top-level

(prefix-out prefix-id provide-spec)

Like provide-spec, but with each symbolic export name from provide-spec prefixed with prefix-id.

Examples:
(module nest racket
  (provide (prefix-out chicken: num-eggs))
  (define num-eggs 2))

> (require 'nest)
> chicken:num-eggs
2

(struct-out id)

Exports the bindings associated with a structure type id. Typically, id is bound with (struct id ...); more generally, id must have a transformer binding of structure-type information at the relevant phase level; see §5.7 “Structure Type Transformer Binding”. Furthermore, for each identifier mentioned in the structure-type information, the enclosing module must define or import one identifier that is free-identifier=. If the structure-type information includes a super-type identifier, and if the identifier has a transformer binding of structure-type information, the accessor and mutator bindings of the super-type are not included by struct-out for export.

Examples:

> (module nest racket
  (provide (struct-out egg))
  (struct egg (color wt)))
> (require 'nest)
> (egg-color (egg 'blue 10))
'blue

(combine-out provide-spec ...)

The union of the provide-specs.

Examples:

> (module nest racket
  (provide (combine-out num-eggs num-chicks))
  (define num-eggs 2)
  (define num-chicks 1))
> (require 'nest)
> num-eggs
2
> num-chicks
1
(protect-out provide-spec ...)

Like the union of the provide-specs, except that the exports are protected; re-
quiring modules may refer to these bindings, but may not extract these bindings
from macro expansions or access them via eval without access privileges. For
more details, see §14.10 “Code Inspectors”. The provide-spec must specify
only bindings that are defined within the exporting module.

Examples:

> (module nest racket
     (provide num-eggs (protect-out num-chicks))
     (define num-eggs 2)
     (define num-chicks 3))
> (define weak-inspector (make-inspector (current-code-
       inspector)))
> (define (weak-eval x)
      (parameterize ([current-code-inspector weak-
                      inspector])
        (define weak-ns (make-base-namespace))
        (namespace-attach-module (current-namespace)
          'nest
          weak-ns)
        (parameterize ([current-namespace weak-ns])
          (namespace-require 'nest)
          (eval x))))
> (require 'nest)
> (list num-eggs num-chicks)
  '(2 3)
> (weak-eval 'num-eggs)
  2
> (weak-eval 'num-chicks)
  ?: access disallowed by code inspector to protected variable
  : from module: 'nest
  : at: num-chicks

See also §15.4 “Code Inspectors for Trusted and Untrusted Code”.

(for-meta phase-level provide-spec ...)

Like the union of the provide-specs, but adjusted to apply to the phase level
specified by phase-level relative to the current phase level (where #f corre-
sponds to the label phase level). In particular, an id or rename-out form as
a provide-spec refers to a binding at phase-level relative to the current
level, an all-defined-out exports only definitions at phase-level relative
to the current phase level, and an all-from-out exports bindings imported with a shift by phase-level.

Examples:

```
> (module nest racket
  (begin-for-syntax
   (define eggs 2))
  (define chickens 3)
  (provide (for-syntax eggs)
   chickens))

> (require 'nest)

> (define-syntax (test-eggs stx)
   (printf "Eggs are ~a\n" eggs)
   #'0)

> (test-eggs)
Eggs are 2
0
> chickens
3

> (module broken-nest racket
  (define eggs 2)
  (define chickens 3)
  (provide (for-syntax eggs)
   chickens))

eval:7:0: provide: provided identifier is not defined or required
   at: eggs
   in: (#%provide (expand (provide-trampoline (for-syntax eggs) chickens)))

> (module nest2 racket
  (begin-for-syntax
   (define eggs 2))
  (provide (for-syntax eggs)))

> (require (for-meta 2 racket/base)
   (for-syntax 'nest2))

> (define-syntax (test stx)
   (define-syntax (show-eggs stx)
     (printf "Eggs are ~a\n" eggs)
     #'0)
   (begin
     (show-eggs)
     #'0))

Eggs are 2
> (test)
0
```
(for-syntax provide-spec ...)  
Same as (for-meta 1 provide-spec ...).

(for-template provide-spec ...)  
Same as (for-meta -1 provide-spec ...).

(for-label provide-spec ...)  
Same as (for-meta #f provide-spec ...).

derived-provide-spec  
See define-provide-syntax for information on expanding the set of provide-spec forms.

Each export specified within a module must have a distinct symbolic export name, though the same binding can be specified with the multiple symbolic names.

(for-meta phase-level require-spec ...)  
See require and provide.

(for-syntax require-spec ...)  
See require and provide.

(for-template require-spec ...)  
See require and provide.

(for-label require-spec ...)  
See require and provide.
The primitive import form, to which `require` expands. A `raw-require-spec` is similar to a `require-spec` in a `require` form, except that the syntax is more constrained, not composable, and not extensible. Also, sub-form names like `for-syntax` and `lib` are recognized symbolically, instead of via bindings. Although not formalized in the grammar above, a `just-meta` form cannot appear within a `just-meta` form, but it can appear under `for-meta`, `for-syntax`, `for-template`, or `for-label`.

Each `raw-require-spec` corresponds to the obvious `require-spec`, but the `rename` sub-form has the identifiers in reverse order compared to `rename-in`.

For most `raw-require-specs`, the lexical context of the `raw-require-spec` determines the context of introduced identifiers. The exception is the `rename` sub-form, where the
lexical context of the local-id is preserved.

A literal-path as a raw-root-module-path corresponds to a path in the sense of path?. Since path values are never produced by read-syntax, they appear only in programmatic constructs such as namespace-require, with otherwise take a quoted raw-module-spec.

```
(#%provide raw-provide-spec ...)

raw-provide-spec = phaseless-spec
  | (for-meta phase-level phaseless-spec ...)
  | (for-syntax phaseless-spec ...)
  | (for-label phaseless-spec ...)
  | (protect raw-provide-spec ...)

phase-level = exact-integer
  | #f

phaseless-spec = id
  | (rename local-id export-id)
  | (struct struct-id (field-id ...))
  | (all-from raw-module-path)
  | (all-from-except raw-module-path id ...)
  | (all-defined)
  | (all-defined-except id ...)
  | (prefix-all-defined prefix-id)
  | (prefix-all-defined-except prefix-id id ...)
  | (protect phaseless-spec ...)
  | (expand (id . datum))
```

The primitive export form, to which provide expands. A raw-module-path is as for #%require. A protect sub-form cannot appear within a protect sub-form.

Like #%require, the sub-form keywords for #%provide are recognized symbolically, and nearly every raw-provide-spec has an obvious equivalent provide-spec via provide, with the exception of the struct and expand sub-forms.

A (struct struct-id (field-id ...)) sub-form expands to struct-id, make-struct-id, struct:struct-id, struct?:struct-id, struct-id-field-id for each field-id, and set-struct-id-field-id! for each field-id. The lexical context of the struct-id is used for all generated identifiers.

Unlike #%require, the #%provide form is macro-extensible via an explicit expand sub-form; the (id . datum) part is locally expanded as an expression (even though it is not actually an expression), stopping when a begin form is produced; if the expansion result is (begin raw-provide-spec ...), it is spliced in place of the expand form, otherwise a
syntax error is reported. The `expand` sub-form is not normally used directly; it provides a hook for implementing `provide` and `provide transformers`.

The `all-from` and `all-from-except` forms re-export only identifiers that are accessible in lexical context of the `all-from` or `all-from-except` form itself. That is, macro-introduced imports are not re-exported, unless the `all-from` or `all-from-except` form was introduced at the same time. Similarly, `all-defined` and its variants export only definitions accessible from the lexical context of the `phaseless-spec` form.

### 3.2.1 Additional `require` Forms

```
(require racket/require)  ; package: base
```

The bindings documented in this section are provided by the `racket/require` library, not `racket/base` or `racket`.

The following forms support more complex selection and manipulation of sets of imported identifiers.

```
(matching-identifiers-in regexp require-spec)
```

Like `require-spec`, but including only imports whose names match `regexp`. The `regexp` must be a literal regular expression (see §4.8 "Regular Expressions").

Examples:

```
> (module zoo racket/base
   (provide tunafish swordfish blowfish
            monkey lizard ant)
   (define tunafish 1)
   (define swordfish 2)
   (define blowfish 3)
   (define monkey 4)
   (define lizard 5)
   (define ant 6))
> (require racket/require)
> (require (matching-identifiers-in #rx"\w*fish" 'zoo))
> tunafish
1
> swordfish
2
> blowfish
3
> monkey
monkey: undefined:
```
cannot reference an identifier before its definition in module: top-level

(subtract-in require-spec subtracted-spec ...)

Like `require-spec`, but omitting those imports that would be imported by one of the `subtracted-specs`.

Examples:

```
> (module earth racket
  (provide land sea air)
  (define land 1)
  (define sea 2)
  (define air 3))
```

```
> (module mars racket
  (provide aliens)
  (define aliens 4))
```

```
> (module solar-system racket
  (require 'earth 'mars)
  (provide (all-from-out 'earth)
            (all-from-out 'mars)))
```

```
> (require racket/require)
> (require (subtract-in 'solar-system 'earth))
```

```
> land
land: undefined:
cannot reference an identifier before its definition in module: top-level
```

```
> aliens
4
```

(filtered-in proc-expr require-spec)

Applies an arbitrary transformation on the import names (as strings) of `require-spec`. The `proc-expr` must evaluate at expansion time to a single-argument procedure, which is applied on each of the names from `require-spec`. For each name, the procedure must return either a string for the import’s new name or `#f` to exclude the import.

For example,

```
(require (filtered-in
  (lambda (name)
    (and (regexp-match? #rx"^[a-z-]+$" name)
         (regexp-replace #rx"-" (string-titlecase name) "")))
  racket/base))
```
imports only bindings from racket/base that match the pattern \#rx"^[a-z-]+$", and it converts the names to “camel case.”

\(\text{path-up rel-string ...}\)

Specifies paths to modules named by the rel-strings similar to using the rel-strings directly, except that if a required module file is not found relative to the enclosing source, it is searched for in the parent directory, and then in the grand-parent directory, etc., all the way to the root directory. The discovered path relative to the enclosing source becomes part of the expanded form.

This form is useful in setting up a “project environment.” For example, using the following "config.rkt" file in the root directory of your project:

```racket
#lang racket/base
(require racket/require-syntax
 (for-syntax "utils/in-here.rkt"))

(provide utils-in)
(define-require-syntax utils-in in-here-transformer)
```

and using "utils/in-here.rkt" under the same root directory:

```racket
#lang racket/base
(require racket/runtime-path)
(provide in-here-transformer)
(define-runtime-path here ".")
(define (in-here-transformer stx)
  (syntax-case stx ()
    [(_ sym)
      (identifier? #'sym)
      (let ([path (build-path here (format "a.rkt" (syntax-e #'sym)))]]
        (datum->syntax stx `(file , (path->string path) stx)))]
  ))
```

then path-up works for any other module under the project directory to find "config.rkt":

```racket
(require racket/require
 (path-up "config.rkt")
 (utils-in foo))
```

Note that the order of requires in the example is important, as each of the first two bind the identifier used in the following.

An alternative in this scenario is to use path-up directly to find the utility module:
but then sub-directories that are called "utils" override the one in the project’s root. In other words, the previous method requires only a single unique name.

```racket
(multi-in subs ...+)

subs = sub-path
    | (sub-path ...)
sub-path = rel-string
    | id
```

Specifies multiple files to be required from a hierarchy of directories or collections. The set of required module paths is computed as the Cartesian product of the `subs` groups, where each `sub-path` is combined with other `sub-path`s in order using a `/` separator. A `sub-path` as a `subs` is equivalent to `(sub-path)`. All `sub-path`s in a given multi-in form must be either strings or identifiers.

Examples:

```racket
(require (multi-in racket (dict list)))

is equivalent to (require racket/dict racket/list)

(require (multi-in "math" "matrix" "utils.rkt"))

is equivalent to (require "math(matrix/utils.rkt")

(require (multi-in "utils" ("math.rkt" "matrix.rkt")))

is equivalent to (require "utils/math.rkt" "utils/matrix.rkt")

(require (multi-in ("math" "matrix") "utils.rkt"))

is equivalent to (require "math/utils.rkt" "matrix/utils.rkt")

(require (multi-in ("math" "matrix") ("utils.rkt" "helpers.rkt")))

is equivalent to (require "math/utils.rkt" "math/helpers.rkt"
    "matrix/utils.rkt" "matrix/helpers.rkt")
```
3.2.2 Additional provide Forms

(require racket/provide) package: base

The bindings documented in this section are provided by the racket/provide library, not racket/base or racket.

(matching-identifiers-out regexp provide-spec)

Like provide-spec, but including only exports of bindings with an external name that matches regexp. The regexp must be a literal regular expression (see §4.8 “Regular Expressions”).

(filtered-out proc-expr provide-spec)

Analogous to filtered-in, but for filtering and renaming exports.

For example,

(provide (filtered-out
(lambda (name)
  (and (regexp-match? #rx"^[a-z-]+$" name)
       (regexp-replace
        #rx"-" (string-titlecase name) ")
      (all-defined-out)))

exports only bindings that match the pattern #rx"^[a-z-]+$", and it converts the names to "camel case."

3.3 Literals: quote and #\%datum

Many forms are implicitly quoted (via #\%datum) as literals. See §1.2.3.2 “Expansion Steps” for more information.

(quote datum)

Produces a constant value corresponding to datum (i.e., the representation of the program fragment) without its lexical information, source location, etc. Quoted pairs, vectors, and boxes are immutable.

Examples:

> (quote x)

'x
Expands to (quote datum), as long as datum is not a keyword. If datum is a keyword, a syntax error is reported.

See also §1.2.3.2 "Expansion Steps" for information on how the expander introduces #%datum identifiers.

Examples:

> (#%datum . 10)
10
> (#%datum . x)
x
> (#%datum . #:x)

The #%datum form is helpful in recursive definition contexts where expanding a subsequent definition can provide compile-time information for the current expression. For example, consider a define-sym-case macro that simply records some symbols at compile-time in a given identifier.
(define-syntax (define-sym-case stx)
  (syntax-case stx ()
    [(_ id sym ...)
      (andmap identifier? (syntax->list #'(sym ...)))
      #'(define-syntax id
         #'(sym ...))])

and then a variant of case that checks to make sure the symbols used in the expression match those given in the earlier definition:

(define-syntax (sym-case stx)
  (syntax-case stx ()
    [(_ id val-expr [(sym) expr] ...) 
      (let ()
        (define expected-ids
          (syntax-local-value #'id
            (#
              (lambda ()
                (raise-syntax-error 'sym-case
                  "expected an identifier bound via define-sym-case" stx
                  #'id)))))
        (define actual-ids (syntax->datum #'(sym ...)))
        (unless (equal? expected-ids actual-ids)
          (raise-syntax-error 'sym-case
            (format "expected the symbols ~s" expected-ids)
            stx))
        #'(case val-expr [(sym expr] ...)])])

If the definition follows the use like this, then the define-sym-case macro does not have a chance to bind id and the sym-case macro signals an error:

> (let ()
   (sym-case land-creatures 'bear
     [(bear) 1]
     [(fox) 2])
   (define-sym-case land-creatures bear fox))

eval:11:0: sym-case: expected an identifier bound via define-sym-case
at: land-creatures
in: (sym-case land-creatures (quote bear) ((bear) 1) ((fox) 2))

115
But if the `sym-case` is wrapped in an `#%expression`, then the expander does not need to expand it to know it is an expression and it moves on to the `define-sym-case` expression.

```lisp
> (let ()
  (#%expression (sym-case sea-creatures 'whale
               [(whale) 1]
               [(squid) 2]))
  (define-sym-case sea-creatures whale squid)
'more...)
'more...
```

Of course, a macro like `sym-case` should not require its clients to add `#%expression`; instead it should check the basic shape of its arguments and then expand to `#%expression` wrapped around a helper macro that calls `syntax-local-value` and finishes the expansion.

### 3.5 Variable References and `#%top`

```
id
```

Refers to a top-level, module-level, or local binding, when `id` is not bound as a transformer (see §1.2.3 “Expansion”). At run-time, the reference evaluates to the value in the location associated with the binding.

When the expander encounters an `id` that is not bound by a module-level or local binding, it converts the expression to (`#%top . id`) giving `#%top` the lexical context of the `id`; typically, that context refers to `#%top`. See also §1.2.3.2 “Expansion Steps”.

Examples:

```lisp
> (define x 10)
> x
10
> (let ([x 5]) x)
5
> ((lambda (x) x) 2)
2
```

```
(#%top . id)
```

Equivalent to `id` when `id` is bound to a module-level or top-level variable. In a top-level context, (`#%top . id`) always refers to a top-level variable, even if `id` is unbound or otherwise bound.
Within a module form, $(\texttt{#\%top . id})$ expands to just $id$—with the obligation that $id$ is defined within the module and has no local binding in its context. At phase level 0, $(\texttt{#\%top . id})$ is an immediate syntax error if $id$ is not bound. At phase level 1 and higher, a syntax error is reported if $id$ is not defined at the corresponding phase by the end of module-body partial expansion.

See also §1.2.3.2 “Expansion Steps” for information on how the expander introduces #%top identifiers.

Examples:

```racket
> (define x 12)
> (let ([x 5]) (\texttt{#\%top . x}))
5
```

Changed in version 6.3 of package base: Changed the introduction of #%top in a top-level context to unbound identifiers only.

### 3.6 Locations: #\%variable-reference

```
(#\%variable-reference id)
(#\%variable-reference (#\%top . id))
(#\%variable-reference)
```

Produces an opaque variable reference value representing the location of $id$, which must be bound as a variable. If no $id$ is supplied, the resulting value refers to an “anonymous” variable defined within the enclosing context (i.e., within the enclosing module, or at the top level if the form is not inside a module).

A variable reference can be used with variable-reference->empty-namespace, variable-reference->resolved-module-path, and variable-reference->namespace, but facilities like define-name-space-anchor and namespace-anchor->namespace wrap those to provide a clearer interface. A variable reference is also useful to low-level extensions; see Inside: Racket C API.

### 3.7 Procedure Applications and #\%app

```
(\texttt{proc-expr arg ...})
```

Applies a procedure, when $\texttt{proc-expr}$ is not an identifier that has a transformer binding (see §1.2.3 “Expansion”).
More precisely, the expander converts this form to `(app proc-expr arg ...), giving `app the lexical context that is associated with the original form (i.e., the pair that combines `proc-expr and its arguments). Typically, the lexical context of the pair indicates the procedure-application `app that is described next. See also §1.2.3.2 “Expansion Steps”

Examples:

```lisp
> (+ 1 2)
3
> ((lambda (x #:arg y) (list y x)) #:arg 2 1)
'(2 1)
```

Applies a procedure. Each `arg is one of the following:

- `arg-expr

  The resulting value is a non-keyword argument.

- `keyword arg-expr

  The resulting value is a keyword argument using `keyword. Each `keyword in the application must be distinct.

The `proc-expr and `arg-exprs are evaluated in order, left to right. If the result of `proc-expr is a procedure that accepts as many arguments as non-keyword `arg-exprs, if it accepts arguments for all of the `keywords in the application, and if all required keyword-based arguments are represented among the `keywords in the application, then the procedure is called with the values of the `arg-exprs. Otherwise, the `exn:fail:contract exception is raised.

The continuation of the procedure call is the same as the continuation of the application expression, so the results of the procedure are the results of the application expression.

The relative order of `keyword-based arguments matters only for the order of `arg-expr evaluations; the arguments are associated with argument variables in the applied procedure based on the `keywords, and not their positions. The other `arg-expr values, in contrast, are associated with variables according to their order in the application form.

See also §1.2.3.2 “Expansion Steps” for information on how the expander introduces `app identifiers.

Examples:
> (#%app + 1 2)
3
> (#%app (lambda (x #:arg y) (list y x)) #:arg 2 1)
'(2 1)
> (#%app cons)
cons: arity mismatch:
the expected number of arguments does not match the given number
expected: 2
given: 0

(#%plain-app proc-expr arg-expr ...)
(#%plain-app)

Like #%app, but without support for keyword arguments. As a special case, (#%plain-app) produces '().

3.8 Procedure Expressions: lambda and case-lambda

(lambda kw-formals body ...+)
(\kw-formals body ...+)

kw-formals = (arg ...)
| (arg ...+ . rest-id)
| rest-id

arg = id
| [id default-expr]
| keyword id
| keyword [id default-expr]

Produces a procedure. The kw-formals determines the number of arguments and which keyword arguments that the procedure accepts.

Considering only the first arg case, a simple kw-formals has one of the following three forms:

(id ...)

The procedure accepts as many non-keyword argument values as the number of ids. Each id is associated with an argument value by position.
The procedure accepts any number of non-keyword arguments greater or equal to the number of ids. When the procedure is applied, the ids are associated with argument values by position, and all leftover arguments are placed into a list that is associated to rest-id.

The procedure accepts any number of non-keyword arguments. All arguments are placed into a list that is associated with rest-id.

More generally, an arg can include a keyword and/or default value. Thus, the first two cases above are more completely specified as follows:

Each arg has the following four forms:

- id
  Adds one to both the minimum and maximum number of non-keyword arguments accepted by the procedure. The id is associated with an actual argument by position.

- [id  default-expr]
  Adds one to the maximum number of non-keyword arguments accepted by the procedure. The id is associated with an actual argument by position, and if no such argument is provided, the default-expr is evaluated to produce a value associated with id. No arg with a default-expr can appear before an id without a default-expr and without a keyword.

- keyword  id
  The procedure requires a keyword-based argument using keyword. The id is associated with a keyword-based actual argument using keyword.
**keyword [id default-expr]**

The procedure accepts a keyword-based argument using *keyword*. The *id* is associated with a keyword-based actual argument using *keyword*, if supplied in an application; otherwise, the *default-expr* is evaluated to obtain a value to associate with *id*.

The position of a *keyword arg* in *kw-formals* does not matter, but each specified *keyword* must be distinct.

**{(arg ...+ . rest-id)**

Like the previous case, but the procedure accepts any number of non-keyword arguments beyond its minimum number of arguments. When more arguments are provided than non-*keyword* arguments among the *args*, the extra arguments are placed into a list that is associated to *rest-id*.

The *kw-formals* identifiers are bound in the *bodys*. When the procedure is applied, a new location is created for each identifier, and the location is filled with the associated argument value. The locations are created and filled in order, with *default-exprs* evaluated as needed to fill locations.

If any identifier appears in the *bodys* that is not one of the identifiers in *kw-formals*, then it refers to the same location that it would if it appeared in place of the *lambda* expression. (In other words, variable reference is lexically scoped.)

When multiple identifiers appear in a *kw-formals*, they must be distinct according to *bound-identifier=*?

If the procedure produced by *lambda* is applied to fewer or more by-position or by-keyword arguments than it accepts, to by-keyword arguments that it does not accept, or without required by-keyword arguments, then the *exn:fail:contract* exception is raised.

The last *body* expression is in tail position with respect to the procedure body.

Examples:

```lisp
> ((lambda (x) x) 10)
10
> ((lambda (x y) (list y x)) 1 2)
'(2 1)
> ((lambda (x [y 5]) (list y x)) 1 2)
'(2 1)
> (let ([f (lambda (x #:arg y) (list y x))]
      (list (f 1 #:arg 2)
            (f #:arg 2 1)))
```
When compiling a lambda or case-lambda expression, Racket looks for a `method-arity-error` property attached to the expression (see §12.7 “Syntax Object Properties”). If it is present with a true value, and if no case of the procedure accepts zero arguments, then the procedure is marked so that an `exn:fail:contract:arity` exception involving the procedure will hide the first argument, if one was provided. (Hiding the first argument is useful when the procedure implements a method, where the first argument is implicit in the original source). The property affects only the format of `exn:fail:contract:arity` exceptions, not the result of `procedure-arity`.

When a keyword-accepting procedure is bound to an identifier in certain ways, and when the identifier is used in the function position of an application form, then the application form may be expanded in such a way that the original binding is obscured as the target of the application. To help expose the connection between the function application and function declaration, an identifier in the expansion of the function application is tagged with a syntax property accessible via `syntax-procedure-alias-property` if it is effectively an alias for the original identifier. An identifier in the expansion is tagged with a syntax property accessible via `syntax-procedure-converted-arguments-property` if it is like the original identifier except that the arguments are converted to a flattened form: keyword arguments, required by-position arguments, by-position optional arguments, and rest arguments—all as required, by-position arguments; the keyword arguments are sorted by keyword name, each optional keyword argument is followed by a boolean to indicate whether a value is provided, and #f is used for an optional keyword argument whose value is not provided; optional by-position arguments include #f for each non-provided argument, and then the sequence of optional-argument values is followed by a parallel sequence of booleans to indicate whether each optional-argument value was provided.

```
(case-lambda [formals body ...+] ...)
```

```
formals = (id ...)
   | (id ...+ . rest-id)
   | rest-id
```

Produces a procedure. Each `[formals body ...+]` clause is analogous to a single lambda procedure; applying the case-lambda-generated procedure is the same as applying a procedure that corresponds to one of the clauses—the first procedure that accepts the given number of arguments. If no corresponding procedure accepts the given number of arguments, the `exn:fail:contract` exception is raised.

Note that a case-lambda clause supports only `formals`, not the more general `kw-formals` of lambda. That is, case-lambda does not directly support keyword and optional arguments.

Example:
> (let ([f (case-lambda
    [() 10]
    [(x) x]
    [(x y) (list y x)]
    [r r)])
  (list (f)
        (f 1)
        (f 1 2)
        (f 1 2 3)))
'(10 1 (2 1) (1 2 3))

Like lambda, but without support for keyword or optional arguments.

3.9 Local Binding: let, let*, letrec, ...

(l%plain-lambda formals body ...+)

The first form evaluates the val-exps left-to-right, creates a new location for each id, and places the values into the locations. It then evaluates the bodys, in which the ids are bound. The last body expression is in tail position with respect to the let form. The ids must be distinct according to bound-identifier=?.

Examples:
> (let ([x 5]) x)
5
> (let ([x 5])
   (let ([x 2]
         [y x])
     (list y x)))
'(5 2)

The second form evaluates the init-exps; the resulting values become arguments in an application of a procedure (lambda (id ...) body ...+), where proc-id is bound within the bodys to the procedure itself.

Example:
> (let fac ([n 10])
    (if (zero? n)
\begin{verbatim}
  (let* ([id val-expr] ...) body ...+)
\end{verbatim}

Like \texttt{let}, but evaluates the \texttt{val-expr}s one by one, creating a location for each \texttt{id} as soon as the value is available. The \texttt{id}s are bound in the remaining \texttt{val-expr}s as well as the \texttt{body}s, and the \texttt{id}s need not be distinct; later bindings shadow earlier bindings.

Example:

```scheme
> (let* ([x 1]
        [y (+ x 1)])
  (list y x))
'(2 1)
```

\begin{verbatim}
(letrec ([id val-expr] ...) body ...+)
\end{verbatim}

Like \texttt{let}, including left-to-right evaluation of the \texttt{val-expr}s, but the locations for all \texttt{id}s are created first, all \texttt{id}s are bound in all \texttt{val-expr}s as well as the \texttt{body}s, and each \texttt{id} is initialized immediately after the corresponding \texttt{val-expr} is evaluated. The \texttt{id}s must be distinct according to \texttt{bound-identifier=?}.

Referencing or assigning to an \texttt{id} before its initialization raises \texttt{exn:fail:contract:variable}. If an \texttt{id} (i.e., the binding instance or \texttt{id}) has an `'undefined-error-name` syntax property whose value is a symbol, the symbol is used as the name of the variable for error reporting, instead of the symbolic form of \texttt{id}.

Example:

```scheme
> (letrec ([is-even? (lambda (n)
      (or (zero? n)
          (is-odd? (sub1 n)))
      [is-odd? (lambda (n)
        (and (not (zero? n))
          (is-even? (sub1 n)))])]
  (is-odd? 11))  
#t
```

Changed in version 6.0.1.2 of package \texttt{base}: Changed reference or assignment of an uninitialized \texttt{id} to an error.

\begin{verbatim}
(let-values ([(id ... ) val-expr] ...) body ...+)
\end{verbatim}
Like `let`, except that each `val-expr` must produce as many values as corresponding `ids`, otherwise the `exn:fail:contract` exception is raised. A separate location is created for each `id`, all of which are bound in the `bodys`.

Example:

```scheme
> (let-values ([(x y) (quotient/remainder 10 3)])
  (list y x))
'(1 3)
```

```
(let-values ([(id ...) val-expr] ...) body ...+)
```

Like `let*`, except that each `val-expr` must produce as many values as corresponding `ids`. A separate location is created for each `id`, all of which are bound in the later `val-expr`s and in the `bodys`.

Example:

```scheme
> (let*-values ([(x y) (quotient/remainder 10 3)]
   [(z) (list y x)])
z)
'(1 3)
```

```
(let*-values ([(id ...) val-expr] ...) body ...+)
```

Like `letrec`, except that each `val-expr` must produce as many values as corresponding `ids`. A separate location is created for each `id`, all of which are bound in all `val-expr`s and in the `bodys`.

Example:

```scheme
> (letrec-values ([(is-even? is-odd?)
   (values
    (lambda (n)
      (or (zero? n)
        (is-odd? (sub1 n))))
    (lambda (n)
      (or (= n 1)
        (is-even? (sub1 n))))))]
  (is-odd? 11))
#t
```

```
(let-rec-values ([(id ...) val-expr] ...) body ...+)
```

```
(let-syntax ([id trans-expr] ...) body ...+)
```
Creates a transformer binding (see §1.2.3.5 "Transformer Bindings") of each id with the value of trans-expr, which is an expression at phase level 1 relative to the surrounding context. (See §1.2.1 "Identifiers, Binding, and Scopes" for information on phase levels.)

The evaluation of each trans-expr is parameterized to set current-namespace to a namespace that shares bindings and variables with the namespace being used to expand the let-syntax form, except that its base phase is one greater.

Each id is bound in the bodies, and not in other trans-exprs.

\[
\text{(letrec-syntax (\[id \ trans-expr \] ...) body ...+)}
\]

Like let-syntax, except that each id is also bound within all trans-exprs.

\[
\text{(let-syntaxes (\[(id \ ...) \ trans-expr \] ...) body ...+)}
\]

Like let-syntax, but each trans-expr must produce as many values as corresponding ids, each of which is bound to the corresponding value.

\[
\text{(let-syntaxes (\[(id \ ...) \ trans-expr \] ...) body ...+)}
\]

Like let-syntax, except that each id is also bound within all trans-exprs.

\[
\text{(letrec-syntaxes+values (\[(trans-id \ ...) \ trans-expr \] ...) \[(val-id \ ...) \ val-expr \] ...)}
\]

\[
\text{body ...+)}
\]

Combines letrec-syntaxes with a variant of letrec-values: each trans-id and val-id is bound in all trans-exprs and val-exprs.

The letrec-syntaxes+values form is the core form for local compile-time bindings, since forms like letrec-syntax and internal-definition contexts expand to it. In a fully expanded expression (see §1.2.3.1 "Fully Expanded Programs"), the trans-id bindings are discarded and the form reduces to a combination of letrec-values or let-values.

For variables bound by letrec-syntaxes+values, the location-creation rules differ slightly from letrec-values. The \[(val-id \ ...) \ val-expr\] binding clauses are partitioned into minimal sets of clauses that satisfy the following rule: if a clause has a val-id binding that is referenced (in a full expansion) by the val-expr of an earlier clause, the two clauses and all in between are in the same set. If a set consists of a single clause whose val-expr does not refer to any of the clause’s val-ids, then locations for the val-ids are created after the val-expr is evaluated. Otherwise, locations for all val-ids in a set are created just before the first val-expr in the set is evaluated. For the purposes of forming sets, a (quote-syntax datum #:local) form counts as a reference to all bindings in the letrec-syntaxes+values form.
The end result of the location-creation rules is that scoping and evaluation order are the same as for \texttt{letrec-values}, but the compiler has more freedom to optimize away location creation. The rules also correspond to a nesting of \texttt{let-values} and \texttt{letrec-values}, which is how \texttt{letrec-syntaxes+values} for a fully-expanded expression.

See also \texttt{local}, which supports local bindings with \texttt{define}, \texttt{define-syntax}, and more.

### 3.10 Local Definitions: local

\begin{verbatim}
(require racket/local)  package: base
\end{verbatim}

The bindings documented in this section are provided by the \texttt{racket/local} and \texttt{racket} libraries, but not \texttt{racket/base}.

\begin{verbatim}
(local [definition ...] body ...+)
\end{verbatim}

Like \texttt{letrec-syntaxes+values}, except that the bindings are expressed in the same way as in the top-level or in a module body: using \texttt{define}, \texttt{define-values}, \texttt{define-syntax}, \texttt{struct}, etc. Definitions are distinguished from non-definitions by partially expanding \texttt{definition} forms (see §1.2.3.7 “Partial Expansion”). As in the top-level or in a module body, a \texttt{begin}-wrapped sequence is spliced into the sequence of \texttt{definitions}.

### 3.11 Constructing Graphs: shared

\begin{verbatim}
(require racket/shared)  package: base
\end{verbatim}

The bindings documented in this section are provided by the \texttt{racket/shared} and \texttt{racket} libraries, but not \texttt{racket/base}.

\begin{verbatim}
(shared ([id expr] ...) body ...+)
\end{verbatim}

Binds \texttt{ids} with shared structure according to \texttt{exprs} and then evaluates the \texttt{bodys}, returning the result of the last expression.

The \texttt{shared} form is similar to \texttt{letrec}, except that special forms of \texttt{expr} are recognized (after partial macro expansion) to construct graph-structured data, where the corresponding \texttt{letrec} would instead produce a use-before-initialization error.

Each \texttt{expr} (after partial expansion) is matched against the following \texttt{shared-expr} grammar, where earlier variants in a production take precedence over later variants:

\[
\text{shared-expr} = \text{shell-expr} \\
| \text{plain-expr}
\]
The `prefix:make-id` identifier above matches three kinds of references. The first kind is any binding whose name has `make-` in the middle, and where `prefix:id` has a transformer binding to structure information with a full set of mutator bindings; see §5.7 "Structure Type Transformer Binding". The second kind is an identifier that itself has a transformer binding to structure information. The third kind is an identifier that has a 'constructor-for syntax property whose value is an identifier with a transformer binding to structure information. A `shell-id`, meanwhile, must be one of the `id`s bound by the `shared` form to a `shell-expr`.

When the `exprs` of the `shared` form are parsed as `shared-expr` (taking into account the order of the variants for parsing precedence), the sub-expressions that were parsed via `early-expr` will be evaluated first when the `shared` form is evaluated. Among such expressions, they are evaluated in the order as they appear within the `shared` form. However, any reference to an `id` bound by `shared` produces a use-before-initialization error, even if the binding for the `id` appears before the corresponding `early-expr` within the `shared` form.

The `shell-ids` and `shell-exprs` (not counting `patchable-expr` and `early-expr` sub-expressions) are effectively evaluated next:

- A `shell-id` reference produces the same value as the corresponding `id` will produce within the `bodys`, assuming that `id` is never mutated with `set!`. This special handling of a `shell-id` reference is one way in which `shared` supports the creation of cyclic data, including immutable cyclic data.
• A *shell-expr* of the form \((mcons \ \text{patchable-expr} \ \text{patchable-expr})\), \((\text{vector} \ \text{patchable-expr} \ ...\), \((\text{box} \ \text{patchable-expr})\), or \((\text{prefix}:\text{make-id} \ \text{patchable-expr} \ ...\)) produces a mutable value whose content positions are initialized to *undefined*. Each content position is *patched* (i.e., updated) after the corresponding *patchable-expr* expression is later evaluated.

Next, the *plain-exprs* are evaluated as for letrec, where a reference to an *id* raises *exn:fail:contract:variable* if it is evaluated before the right-hand side of the *id* binding.

Finally, the *patchable-exprs* are evaluated and their values replace *undefineds* in the results of *shell-exprs*. At this point, all *ids* are bound, so *patchable-exprs* can create data cycles (but only with cycles that can be created via mutation).

Examples:

```
> (shared ([a (cons 1 a)])
  a)
#0='(1 . #0#)
> (shared ([a (cons 1 b)]
   [b (cons 2 a)])
  a)
#0='(1 2 . #0#)
> (shared ([a (cons 1 b)]
   [b 7])
  a)
'(1 . 7)
> (shared ([a a]) ; no indirection...
  a)
a: undefined;
  cannot use before initialization
> (shared ([a (cons 1 b)] ; b is early...
   [b a])
  a)
a: undefined;
  cannot use before initialization
> (shared ([a (mcons 1 b)] ; b is patchable...
   [b a])
  a)
#0=(mcons 1 #0#)
> (shared ([a (vector b b b)]
   [b (box 1)])
   (set-box! b 5)
  a)
'#(#&5 #&5 #&5)
> (shared ([a (box b)])
```
3.12 Conditionals: if, cond, and, and or

\[
\text{if test-expr then-expr else-expr}
\]

Evaluates test-expr. If it produces any value other than \#f, then then-expr is evaluated, and its results are the result for the if form. Otherwise, else-expr is evaluated, and its results are the result for the if form. The then-expr and else-expr are in tail position with respect to the if form.

Examples:

```scheme
> (if (positive? -5) (error "doesn't get here") 2)
2
> (if (positive? 5) 1 (error "doesn't get here"))
1
> (if 'we-have-no-bananas "yes" "no")
"yes"
```

\[
\text{cond cond-clause ...}
\]

cond-clause = [test-expr then-body ...]+ |
\[\text{else then-body ...}+\]
\[\text{test-expr => proc-expr}\]
\[\text{test-expr}\]

A cond-clause that starts with else must be the last cond-clause.

If no cond-clauses are present, the result is \#<void>.

If only a [else then-body ...+] is present, then the then-bodys are evaluated. The results from all but the last then-body are ignored. The results of the last then-body, which is in tail position with respect to the cond form, are the results for the whole cond form.

Otherwise, the first test-expr is evaluated. If it produces \#f, then the result is the same as a cond form with the remaining cond-clauses, in tail position with respect to the original cond form. Otherwise, evaluation depends on the form of the cond-clause:
[**test-expr then-body ...+**]

The *then-bodies* are evaluated in order, and the results from all but the last *then-body* are ignored. The results of the last *then-body*, which is in tail position with respect to the cond form, provides the result for the whole cond form.

[**test-expr => proc-expr**]

The *proc-expr* is evaluated, and it must produce a procedure that accepts one argument, otherwise the `exn:fail:contract` exception is raised. The procedure is applied to the result of *test-expr* in tail position with respect to the cond expression.

[**test-expr**]

The result of the *test-expr* is returned as the result of the cond form. The *test-expr* is not in tail position.

Examples:

```scheme
> (cond)
> (cond
   [else 5])
5
> (cond
   [(positive? -5) (error "doesn't get here")]
   [(zero? -5) (error "doesn't get here, either")]
   [(positive? 5) 'here])
'here
> (cond
   [(member 2 '(1 2 3)) => (lambda (l) (map - l))]
   '(-2 -3)
> (cond
   [(member 2 '(1 2 3))] '2 3)
else
```

Recognized specially within forms like cond. An else form as an expression is a syntax error.
Recognized specially within forms like cond. A \( \text{\texttt{=>}} \) form as an expression is a syntax error.

\[
\text{\texttt{(and \ expr \ ...)}}
\]

If no \texttt{expr}s are provided, then result is \texttt{#t}.

If a single \texttt{expr} is provided, then it is in tail position, so the results of the \texttt{and} expression are the results of the \texttt{expr}.

Otherwise, the first \texttt{expr} is evaluated. If it produces \texttt{#f}, the result of the \texttt{and} expression is \texttt{#f}. Otherwise, the result is the same as an \texttt{and} expression with the remaining \texttt{expr}s in tail position with respect to the original \texttt{and} form.

Examples:

\[
\begin{array}{l}
> \text{(and)}\\
\quad \texttt{#t}\\
> \text{(and 1)}\\
\quad 1\\
> \text{(and \texttt{(values 1 2)})}\\
\quad 1\\
\quad 2\\
> \text{(and \texttt{#f \ (error \ "doesn't get here")})}\\
\quad \texttt{#f}\\
> \text{(and \texttt{#t 5})}\\
\quad 5
\end{array}
\]

\[
\text{\texttt{(or \ expr \ ...)}}
\]

If no \texttt{expr}s are provided, then result is \texttt{#f}.

If a single \texttt{expr} is provided, then it is in tail position, so the results of the \texttt{or} expression are the results of the \texttt{expr}.

Otherwise, the first \texttt{expr} is evaluated. If it produces a value other than \texttt{#f}, that result is the result of the \texttt{or} expression. Otherwise, the result is the same as an \texttt{or} expression with the remaining \texttt{expr}s in tail position with respect to the original \texttt{or} form.

Examples:

\[
\begin{array}{l}
> \text{(or)}\\
\quad \texttt{#f}\\
> \text{(or 1)}\\
\quad 1\\
> \text{(or \texttt{(values 1 2)})}
\end{array}
\]
> (or 5 (error "doesn't get here"))
> (or #f 5)

3.13 Dispatch: case

\[
\text{(case } \text{val-expr case-clause ...)}
\]

\[
\text{case-clause } = \begin{cases} 
\text{[(datum ...)} \text{ then-body ...]} \\
\text{[else then-body ...]} 
\end{cases}
\]

Evaluates \text{val-expr} and uses the result to select a \text{case-clause}. The selected clause is the first one with a \text{datum} whose quoted form is \text{equal?} to the result of \text{val-expr}. If no such \text{datum} is present, the \text{else case-clause} is selected; if no \text{else case-clause} is present, either, then the result of the \text{case} form is \#<void>.

For the selected \text{case-clause}, the results of the last \text{then-body}, which is in tail position with respect to the \text{case} form, are the results for the whole \text{case} form.

A \text{case-clause} that starts with \text{else} must be the last \text{case-clause}.

The \text{case} form can dispatch to a matching \text{case-clause} in $O(\log N)$ time for $N$ \text{datums}.

Examples:

> (case (+ 7 5)
  [[(1 2 3) 'small]
   [(10 11 12) 'big]]
'big
> (case (- 7 5)
  [[(1 2 3) 'small]
   [(10 11 12) 'big]]
'small
> (case (string-append "do" "g")
  [("cat" "dog" "mouse") "animal"]
  [else "mineral or vegetable"])
"animal"
> (case (list 'y 'x)
  [[[a b] (x y)] 'forwards]
   [[[b a] (y x)] 'backwards])
'backwards
> (case 'x
  [(x) "ex"]
  [(\'x) "quoted ex"])
"ex"

> (case (list 'quote 'x)
  [(x) "ex"]
  [(\'x) "quoted ex"])
"quoted ex"

(define (classify c)
  (case (char-general-category c)
    [(ll lu lt ln lo) "letter"]
    [(nd nl no) "number"]
    [else "other"])))

> (classify #\A)
"letter"
> (classify #\1)
"number"
> (classify #\!)
"other"

3.14 Definitions: define, define-syntax, ...


The first form binds id to the result of expr, and the second form binds id to a procedure. In the second case, the generated procedure is (CVT (head args) body ...+), using the CVT meta-function defined as follows:

(CVT (id . kw-formals) . datum) = (lambda kw-formals . datum)
(CVT (head . kw-formals) . datum) = (lambda kw-formals expr)
    if (CVT head . datum) = expr
In an internal-definition context, a define form introduces a local binding; see §1.2.3.8 “Internal Definitions”. At the top level, the top-level binding for id is created after evaluating expr, if it does not exist already, and the top-level mapping of id (in the namespace linked with the compiled definition) is set to the binding at the same time.

In a context that allows liberal expansion of define, id is bound as syntax if expr is an immediate lambda form with keyword arguments or args include keyword arguments.

Examples:

```
(define x 10)
> x
10

(define (f x)
  (+ x 1))
> (f 10)
11

(define ((f x) [y 20])
  (+ x y))
> ((f 10) 30)
40
> ((f 10))
30
```

```
(define-values (id ...) expr)
```

Evaluates the expr, and binds the results to the ids, in order, if the number of results matches the number of ids; if expr produces a different number of results, the exn:fail:contract exception is raised.

In an internal-definition context (see §1.2.3.8 “Internal Definitions”), a define-values form introduces local bindings. At the top level, the top-level binding for each id is created after evaluating expr, if it does not exist already, and the top-level mapping of each id (in the namespace linked with the compiled definition) is set to the binding at the same time.

Examples:

```
> (define-values () (values))
> (define-values (x y z) (values 1 2 3))
> z
3
```

135
If a define-values form for a function definition in a module body has a `compiler-hint:cross-module-inline` syntax property with a true value, then the Racket treats the property as a performance hint. See §19.5 “Function-Call Optimizations” in The Racket Guide for more information, and see also `begin-encourage-inline`.

```
(define-syntax id expr)
(define-syntax (head args) body ...+)
```

The first form creates a transformer binding (see §1.2.3.5 “Transformer Bindings”) of `id` with the value of `expr`, which is an expression at phase level 1 relative to the surrounding context. (See §1.2.1 “Identifiers, Binding, and Scopes” for information on phase levels.) Evaluation of `expr` side is parameterized to set `current-namespace` as in `let-syntax`.

The second form is a shorthand the same as for `define`; it expands to a definition of the first form where the `expr` is a lambda form.

In an internal-definition context (see §1.2.3.8 “Internal Definitions”) a `define-syntax` form introduces a local binding.

Examples:

```
> (define-syntax foo
  (syntax-rules ()
    ((_ a ...) (printf "~a\n" (list a ...)))))
> (foo 1 2 3 4)
(1 2 3 4)
> (define-syntax (bar syntax-object)
  (syntax-case syntax-object ()
    ((_ a ...) #'(printf "~a\n" (list a ...)))))
> (bar 1 2 3 4)
(1 2 3 4)
```

```
(define-syntaxes (id ...) expr)
```

Like `define-syntax`, but creates a transformer binding for each `id`. The `expr` should produce as many values as `ids`, and each value is bound to the corresponding `id`.

When `expr` produces zero values for a top-level `define-syntaxes` (i.e., not in a module or internal-definition position), then the `ids` are effectively declared without binding; see §1.2.3.10 “Macro-Introduced Bindings”.

In an internal-definition context (see §1.2.3.8 “Internal Definitions”), a `define-syntaxes` form introduces local bindings.

Examples:
(define-syntaxes (foo1 foo2 foo3)
  (let ([transformer1 (lambda (syntax-object)
      (syntax-case syntax-object ()
        [(_) #'1]))]
      [transformer2 (lambda (syntax-object)
      (syntax-case syntax-object ()
        [(_) #'2]))]
      [transformer3 (lambda (syntax-object)
      (syntax-case syntax-object ()
        [(_) #'3]))])
    (values transformer1 transformer2 transformer3)))

> (foo1)
1
> (foo2)
2
> (foo3)
3

(define-for-syntax id expr)
  (define-for-syntax (head args) body ...+)

Like define, except that the binding is at phase level 1 instead of phase level 0 relative to its context. The expression for the binding is also at phase level 1. (See §1.2.1 "Identifiers, Binding, and Scopes" for information on phase levels.) The form is a shorthand for (begin-for-syntax (define id expr)) or (begin-for-syntax (define (head args) body ...+)).

Within a module, bindings introduced by define-for-syntax must appear before their uses or in the same define-for-syntax form (i.e., the define-for-syntax form must be expanded before the use is expanded). In particular, mutually recursive functions bound by define-for-syntax must be defined by the same define-for-syntax form.

Examples:

> (define-for-syntax helper 2)
> (define-syntax (make-two syntax-object)
  (printf "helper is ~a\n" helper)
  #'2)
> (make-two)
helper is 2
2
; 'helper' is not bound in the runtime phase
> helper
helper: undefined;
cannot reference an identifier before its definition in module: top-level

> (define-for-syntax (filter-ids ids)
  (filter identifier? ids))
> (define-syntax (show-variables syntax-object)
  (syntax-case syntax-object ()
  [(_ expr ...)
    (with-syntax ([[(only-ids ...)]
      (filter-ids (syntax->list #'(expr ...)))]))
    #'(list only-ids ...))])
> (let ([a 1] [b 2] [c 3])
  (show-variables a 5 2 b c))
'(1 2 3)

(define-values-for-syntax (id ... expr)

Like define-for-syntax, but expr must produce as many values as supplied ids, and all of the ids are bound (at phase level 1).

Examples:

> (define-values-for-syntax (foo1 foo2) (values 1 2))
> (define-syntax (bar syntax-object)
  (printf "foo1 is \n foo2 is \n" foo1 foo2)
  #2)
> (bar)
foo1 is 1 foo2 is 2
2

3.14.1 require Macros

(require racket/require-syntax) package:base

The bindings documented in this section are provided by the racket/require-syntax library, not racket/base or racket.

(define-require-syntax id proc-exp)  
(define-require-syntax (id args ...) body ...+)

The first form is like define-syntax, but for a require sub-form. The proc-exp must produce a procedure that accepts and returns a syntax object representing a require sub-form.

This form expands to define-syntax with a use of make-require-transformer (see §12.4.1 “require Transformers” for more information).
The second form is a shorthand the same as for define-syntax; it expands to a definition of the first form where the proc-expr is a lambda form.

\[
\text{(syntax-local-require-introduce stx)} \rightarrow \text{syntax?}
\]
\[
stx : \text{syntax?}
\]

For backward compatibility only; equivalent to syntax-local-introduce.

Changed in version 6.90.0.29 of package base: Made equivalent to syntax-local-introduce.

3.15 Sequencing: begin, begin0, and begin-for-syntax

\[
\text{(begin form ...)}
\]
\[
\text{(begin expr ...+)}
\]

§4.8 “Sequencing” in The Racket Guide introduces begin and begin0.
The first form applies when `begin` appears at the top level, at module level, or in an internal-definition position (before any expression in the internal-definition sequence). In that case, the `begin` form is equivalent to splicing the `forms` into the enclosing context.

The second form applies for `begin` in an expression position. In that case, the `exprs` are evaluated in order, and the results are ignored for all but the last `expr`. The last `expr` is in tail position with respect to the `begin` form.

Examples:

```
> (begin
  (define x 10)
  x)
10
> (+ 1 (begin
      (printf "hi\n")
      2))
hi
3
> ;(let-values ([ (x y) (begin
      (values 1 2 3)
      (values 1 2))]
      (list x y))
'(1 2)
```

```
| (begin0 expr ...+)

Evaluates the first `expr`, then evaluates the other `exprs` in order, ignoring their results. The results of the first `expr` are the results of the `begin0` form; the first `expr` is in tail position only if no other `exprs` are present.

Example:

```
> (begin0
  (values 1 2)
  (printf "hi\n"))
hi
1
2
```

```
| (begin-for-syntax form ...)

Allowed only in a top-level context or module context, shifts the phase level of each `form` by one:

```
140
```
• expressions reference bindings at a phase level one greater than in the context of the begin-for-syntax form;

• define, define-values, define-syntax, and define-syntaxes forms bind at a phase level one greater than in the context of the begin-for-syntax form;

• in require and provide forms, the default phase level is greater, which is roughly like wrapping the content of the require form with for-syntax;

• expression form expr: converted to (define-values-for-syntax () (begin expr (values))), which effectively evaluates the expression at expansion time and, in the case of a module context, preserves the expression for future visits of the module.

See also module for information about expansion order and partial expansion for begin-for-syntax within a module context. Evaluation of an expr within begin-for-syntax is parameterized to set current-namespace as in let-syntax.

3.16 Guarded Evaluation: when and unless

|| (when test-expr body ...+)

Evaluates test-expr. If the result is #f, then the result of the when expression is #<void>. Otherwise, the bodys are evaluated, and the last body is in tail position with respect to the when form.

Examples:

> (when (positive? -5)
   (display "hi"))
> (when (positive? 5)
   (display "hi")
   (display " there"))
hi there

|| (unless test-expr body ...+)

Equivalent to (when (not test-expr) body ...+).

Examples:

> (unless (positive? 5)
   (display "hi"))
> (unless (positive? -5)
   (display "hi")
   (display " there"))
hi there
3.17 Assignment: set! and set!-values

(set! id expr)

If id has a transformer binding to an assignment transformer, as produced by make-set!-transformer or as an instance of a structure type with the prop:set!-transformer property, then this form is expanded by calling the assignment transformer with the full expressions. If id has a transformer binding to a rename transformer as produced by make-rename-transformer or as an instance of a structure type with the prop:rename-transformer property, then this form is expanded by replacing id with the target identifier (e.g., the one provided to make-rename-transformer). If a transformer binding has both prop:set!-transformer and prop:rename-transformer properties, the latter takes precedence.

Otherwise, evaluates expr and installs the result into the location for id, which must be bound as a local variable or defined as a top-level variable or module-level variable. If id refers to an imported binding, a syntax error is reported. If id refers to a top-level variable that has not been defined, the exn:fail:contract exception is raised.

See also compile-allow-set!-undefined.

Examples:

> (define x 12)
> (set! x (add1 x))
> x
13
> (let ([x 5])
  (set! x (add1 x))
  x)
6
> (set! i-am-not-defined 10)
set!: assignment disallowed;
cannot set variable before its definition
variable: i-am-not-defined
in module: top-level

(set!-values (id ...) expr)

Assuming that all ids refer to variables, this form evaluates expr, which must produce as many values as supplied ids. The location of each id is filled with the corresponding value from expr in the same way as for set!.

Example:
> (let ([a 1]
    [b 2])
  (set!-values (a b) (values b a))
  (list a b))
'(2 1)

More generally, the set!-values form is expanded to

(let-values ([(tmp-id ...) expr])
  (set! id tmp-id) ...)

which triggers further expansion if any id has a transformer binding to an assignment transformer.

3.18 Iterations and Comprehensions: for, for/list, ...

The for iteration forms are based on SRFI-42 [SRFI-42].

3.18.1 Iteration and Comprehension Forms

(for (for-clause ...) body-or-break ... body)

  for-clause = [id seq-expr]
  | [(id ...) seq-expr]
  | #:when guard-expr
  | #:unless guard-expr
  | break-clause

  break-clause = #:break guard-expr
  | #:final guard-expr

  body-or-break = body
  | break-clause

  seq-expr : sequence?

Iteratively evaluates bodys. The for-clauses introduce bindings whose scope includes body and that determine the number of times that body is evaluated. A break-clause either among the for-clauses or bodys stops further iteration.

In the simple case, each for-clause has one of its first two forms, where [id seq-expr] is a shorthand for [(id) seq-expr]. In this simple case, the seq-exprs are evaluated left-to-right, and each must produce a sequence value (see §4.15.1 "Sequences").
The for form iterates by drawing an element from each sequence; if any sequence is empty, then the iteration stops, and #<void> is the result of the for expression. Otherwise a location is created for each id to hold the values of each element; the sequence produced by a seq-expr must return as many values for each iteration as corresponding ids.

The ids are then bound in the body, which is evaluated, and whose results are ignored. Iteration continues with the next element in each sequence and with fresh locations for each id.

A for form with zero for-clauses is equivalent to a single for-clause that binds an unreferenced id to a sequence containing a single element. All of the ids must be distinct according to bound-identifier=?.

If any for-clause has the form #:when guard-expr, then only the preceding clauses (containing no #:when or #:unless) determine iteration as above, and the body is effectively wrapped as

```scheme
(when guard-expr
  (for (for-clause ...) body ...+))
```

using the remaining for-clauses. A for-clause of the form #:unless guard-expr corresponds to the same transformation with unless in place of when.

A #:break guard-expr clause is similar to a #:unless guard-expr clause, but when #:break avoids evaluation of the bodys, it also effectively ends all sequences within the for form. A #:final guard-expr clause is similar to #:break guard-expr, but instead of immediately ending sequences and skipping the bodys, it allows at most one more element from each later sequence and at most one more evaluation of the following bodys. Among the bodys, besides stopping the iteration and preventing later body evaluations, a #:break guard-expr or #:final guard-expr clause starts a new internal-definition context.

In the case of list and stream sequences, the for form itself does not keep each element reachable. If a list or stream produced by a seq-expr is otherwise unreachable, and if the for body can no longer reference an id for a list element, then the element is subject to garbage collection. The make-do-sequence sequence constructor supports additional sequences that behave like lists and streams in this way.

If a seq-expr is a quoted literal list, vector, exact integer, string, byte string, immutable hash, or expands to such a literal, then it may be treated as if a sequence transformer such as in-list was used, unless the seq-expr has a true value for the 'for:no-implicit-optimization syntax property; in most cases this improves performance.

Examples:

```scheme
> (for ([i '(1 2 3)]
      [j "abc"]
      #:when (odd? i)
```
(for ([i j k])
  (display (list i j k)))
(1 a #t)(1 a #f)(3 c #t)(3 c #f)
> (for ([i j] #hash("a" . 1)("b" . 20))
  (display (list i j)))
(a 1)(b 20)
> (for ([i '(1 2 3)]
    [j "abc"]
    #:break (not (odd? i))
    [k #(#t #f)])
  (display (list i j k)))
(1 a #t)(1 a #f)
> (for ([i '(1 2 3)]
    [j "abc"]
    #:final (not (odd? i))
    [k #(#t #f)])
  (display (list i j k)))
(1 a #t)(1 a #f)(2 b #t)
> (for ([i '(1 2 3)]
    [j "abc"]
    [k #(#t #f)])
  #:break (not (or (odd? i) k))
  (display (list i j k)))
(1 a #t)
> (for ()
  (display "here"))
here
> (for ([i '()])
  (error "doesn't get here"))
(list i j k))
'((1 #\a #t) (1 #\a #f) (3 #\c #t) (3 #\c #f))
> (for/list ([i '(1 2 3)]
  [j "abc"]
  #:break (not (odd? i))
  [k (#(t #f)])
  (list i j k))
'((1 #\a #t) (1 #\a #f))
> (for/list () 'any)
'(any)
> (for/list ([i ()]))
'(error "doesn't get here")
()

(for/vector maybe-length (for-clause ...) body-or-break ... body)

maybe-length =
  #:length length-expr
  #:length length-expr #:fill fill-expr

length-expr : exact-nonnegative-integer?

Iterates like for/list, but results are accumulated into a vector instead of a list.

If the optional #:length clause is specified, the result of length-expr determines the
length of the result vector. In that case, the iteration can be performed more efficiently, and
it terminates when the vector is full or the requested number of iterations have been per-
formed, whichever comes first. If length-expr specifies a length longer than the number
of iterations, then the remaining slots of the vector are initialized to the value of fill-expr,
which defaults to 0 (i.e., the default argument of make-vector).

Examples:
> (for/vector ([i '(1 2 3)]) (number->string i))
'(#"1" "2" "3")
> (for/vector #:length 2 ([i '(1 2 3)]) (number->string i))
'(#"1" "2")
> (for/vector #:length 4 ([i '(1 2 3)]) (number->string i))
'(#"1" "2" "3" 0)
> (for/vector #:length 4 #:fill "?" ([i '(1 2 3)]) (number->string i))
'(#"1" "2" "3" ")

The for/vector form may allocate a vector and mutate it after each iteration of body,
which means that capturing a continuation during body and applying it multiple times may
mutate a shared vector.
Like `for/list`, but the result is an immutable hash table; `for/hash` creates a table using `equal?` to distinguish keys, `for/hasheq` produces a table using `eq?`, and `for/hasheqv` produces a table using `eqv?`. The last expression in the body must return two values: a key and a value to extend the hash table accumulated by the iteration.

Example:

```scheme
> (for/hash ([i '(1 2 3)])
    (values i (number->string i)))
'(1 . "1") (2 . "2") (3 . "3")
```

```scheme```
(for/and (for-clause ...) body-or-break ... body)
```

Iterates like `for`, but when last expression of body produces `#f`, then iteration terminates, and the result of the `for/and` expression is `#f`. If the body is never evaluated, then the result of the `for/and` expression is `#t`. Otherwise, the result is the (single) result from the last evaluation of body.

Examples:

```scheme
> (for/and ([i '(1 2 3 "x")])
    (i . < . 3))
#f
> (for/and ([i '(1 2 3 4)])
    i)
4
> (for/and ([i '(1 2 3 4)])
    #:break (= i 3)
    i)
2
> (for/and ([i '()])
    (error "doesn't get here"))
#t
```

```scheme```
(for/or (for-clause ...) body-or-break ... body)
```

Iterates like `for`, but when last expression of body produces a value other than `#f`, then iteration terminates, and the result of the `for/or` expression is the same (single) value. If the body is never evaluated, then the result of the `for/or` expression is `#f`. Otherwise, the result is `#f`.

Examples:
> (for/or ([i '(1 2 3)]
   (i . < . 3))
#t
> (for/or ([i '(1 2 3 4)]
   i)
1
> (for/or ([i '()]
   (error "doesn't get here"))
#f

(for/sum (for-clause ...) body-or-break ... body)

Iterates like for, but each result of the last body is accumulated into a result with +.

Example:
> (for/sum ([i '(1 2 3 4)] i)
10

(for/product (for-clause ...) body-or-break ... body)

Iterates like for, but each result of the last body is accumulated into a result with *.

Example:
> (for/product ([i '(1 2 3 4)] i)
24

(for/lists (id ... maybe-result)
  (for-clause ...)
  body-or-break ... body)

maybe-result =
  #:result result-expr

Similar to for/list, but the last body expression should produce as many values as given ids. The ids are bound to the reversed lists accumulated so far in the for-clauses and bodys.

If a result-expr is provided, it is used as with for/fold when iteration terminates; otherwise, the result is as many lists as supplied ids.

The scope of id bindings is the same as for accumulator identifiers in for/fold. Mutating a id affects the accumulated lists, and mutating it in a way that produces a non-list can cause a final reverse for each id to fail.

Examples:
> (for/lists (l1 l2 l3)
  ([i '(1 2 3)]
   [j "abc"]
   #:when (odd? i)
   [k #(#t #f)]))
(values i j k))
'(1 1 3 3)
'(#\a #\a #\c #\c)
'(#t #f #t #f)
> (for/lists (acc)
  ([x '(tvp tofu seitan tvp tofu)]
   #:unless (member x acc)))
  x)
'(tvp tofu seitan)
> (for/lists (firsts seconds #:result (list firsts seconds))
  ([pr '((1 . 2) (3 . 4) (5 . 6))])
  (values (car pr) (cdr pr)))
'(1 3 5) (2 4 6))

Changed in version 7.1.0.2 of package base: Added the #:result form.

### (for/first (for-clause ...) body-or-break ... body)

Iterates like for, but after body is evaluated the first time, then the iteration terminates, and the for/first result is the (single) result of body. If the body is never evaluated, then the result of the for/first expression is #f.

Examples:

> (for/first ([i '(1 2 3 "x")]
  #:when (even? i))
  (number->string i))
"2"
> (for/first ([i '()])
  (error "doesn't get here"))
#f

### (for/last (for-clause ...) body-or-break ... body)

Iterates like for, but the for/last result is the (single) result of the last evaluation of body. If the body is never evaluated, then the result of the for/last expression is #f.

Examples:

> (for/last ([i '(1 2 3 4 5)]
#:when (even? i)
  (number->string i))
"4"
> (for/last ([i ()])
  (error "doesn't get here"))
#f

(for/fold ([acc-id init-expr] ... maybe-result) (for-clause ...)
  body-or-break ... body)

maybe-result =
  | #:result result-expr

Iterates like for. Before iteration starts, the init-exprs are evaluated to produce initial accumulator values. At the start of each iteration, a location is generated for each accum-id, and the corresponding current accumulator value is placed into the location. The last expression in body must produce as many values as accum-ids, and those values become the current accumulator values. When iteration terminates, if a result-expr is provided then the result of the for/fold is the result of evaluating result-expr (with accum-ids in scope and bound to their final values), otherwise the results of the for/fold expression are the accumulator values.

Examples:
> (for/fold ([sum 0]
  [rev-roots null])
  ([i '(1 2 3 4)])
  (values (+ sum i) (cons (sqrt i) rev-roots)))
10
'(2 1.7320508075688772 1.4142135623730951 1)
> (for/fold ([acc '()]
  [seen (hash)])
  #:result (reverse acc))
  ([x (in-list '(0 1 1 2 3 4 4 4))])
(cond
  [(hash-ref seen x #:f)
   (values acc seen)]
  [else (values (cons x acc)
    (hash-set seen x #:t))])
'(0 1 2 3 4)

The binding and evaluation order of accum-ids and init-exprs do not completely follow the textual, left-to-right order relative to the for-clauses. Instead, the sequence expressions in for-clauses that determine the outermost iteration are evaluated first, then the init-exprs are evaluated and the accum-ids are bound, and finally the outermost iteration’s identifiers are bound. One consequence is that the accum-ids are not bound in
for-clauses for the outermost initialization. At the same time, when a accum-id is used as a for-clause binding for the outermost iteration, the for-clause binding shadows the accum-id binding in the loop body (which is what you would expect syntactically). A fresh variable for each accum-id (at a fresh location) is bound in each nested iteration that is created by a later group for for-clauses (after a #:when or #:unless, for example).

Changed in version 6.11.0.1 of package base: Added the #:result form.

```
(for/foldr ([accum-id init-expr] ... accum-option ...)
 (for-clause ...)
 body-or-break ... body)
```

Like for/fold, but analogous to foldr rather than foldl: the given sequences are still iterated in the same order, but the loop body is evaluated in reverse order. Evaluation of a for/foldr expression uses space proportional to the number of iterations it performs, and all elements produced by the given sequences are retained until backwards evaluation of the loop body begins (assuming the element is, in fact, referenced in the body).

Examples:

```
> (define (in-printing seq)
  (sequence-map (lambda (v) (println v) v) seq))
> (for/foldr ([acc '()])
  ([v (in-printing (in-range 1 4))])
  (println v)
  (cons v acc))
1
2
3
3
2
1
'(1 2 3)
```

Furthermore, unlike for/fold, the accum-ids are not bound within guard-exprs or body-or-break forms that appear before a break-clause.

While the aforementioned limitations make for/foldr less generally useful than for/fold, for/foldr provides the additional capability to iterate lazily via the #:delay, #:delay-as, and #:delay-with options, which can mitigate many of for/foldr’s disadvantages. If at least one such option is specified, the loop body is given explicit control
over when iteration continues: by default, each \texttt{accum-id} is bound to a promise that, when forced, produces the \texttt{accum-id}'s current value.

In this mode, iteration does not continue until one such promise is forced, which triggers any additional iteration necessary to produce a value. If the loop body is lazy in its \texttt{accum-ids}—that is, it returns a value without forcing any of them—then the loop (or any of its iterations) will produce a value before iteration has completely finished. If a reference to at least one such promise is retained, then forcing it will resume iteration from the point at which it was suspended, even if control has left the dynamic extent of the loop body.

Examples:

\begin{verbatim}
> (for/foldr ([acc '()] #:delay)
   ([v (in-range 1 4)])
   (printf "--> ~v
" v)
(begi0
   (cons v (force acc))
   (printf "<-- ~v
" v)))
--> 1
--> 2
--> 3
<-- 3
<-- 2
<-- 1
'(1 2 3)
> (define resume
   (for/foldr ([acc '()] #:delay)
   ([v (in-range 1 5)])
   (printf "--> ~v
" v)
(begi0
   (cond
    [(= v 1) (force acc)]
    [(= v 2) acc]
    [else (cons v (force acc))])
   (printf "<-- ~v
" v)))
--) 3
--) 4
<-- 4
<-- 3
'(3 4)
\end{verbatim}
This extra control over iteration order allows `for/foldr` to both consume and construct infinite sequences, so long as it is at least sometimes lazy in its accumulators.

Examples:

```scheme
> (define squares (for/foldr ([s empty-stream] #:delay)
  ([n (in-naturals)])
  (stream-cons (* n n) (force s))))
> (stream->list (stream-take squares 10))
'(0 1 4 9 16 25 36 49 64 81)
```

The suspension introduced by the `#:delay` option does not ordinarily affect the loop’s eventual return value, but if `#:delay` and `#:result` are combined, the `accum-ids` will be delayed in the scope of the `result-expr` in the same way they are delayed within the loop body. This can be used to introduce an additional layer of suspension around the evaluation of the entire loop, if desired.

Examples:

```scheme
> (define evaluated-yet? #f)
> (for/foldr ([acc (set! evaluated-yet? #t)] #:delay) ()
  (force acc))
> evaluated-yet?
#t

> (define evaluated-yet? #f)
> (define start
  (for/foldr ([acc (set! evaluated-yet? #t)] #:delay #:result acc) ()
    (force acc)))
> evaluated-yet?
#f
> (force start)
> evaluated-yet?
#t
```

If the `#:delay-as` option is provided, then `delayed-id` is bound to an additional promise that returns the values of all `accum-ids` at once. When multiple `accum-ids` are provided, forcing this promise can be slightly more efficient than forcing the promises bound to the `accum-ids` individually.

If the `#:delay-with` option is provided, the given `delayer-id` is used to suspend nested iterations (instead of the default, `delay`). A form of the shape `(delayer-id recur-expr)` is constructed and placed in expression position, where `recur-expr` is an expression that, when evaluated, will perform the next iteration and return its result (or results). Sensible choices for `delayer-id` include lazy, delay-sync, delay/thread, or any of the other `delay` forms.

See also
`for/stream` for a more convenient (albeit less flexible) way to lazily transform infinite sequences. (Internally, `for/stream` is defined in terms of `for/foldr`.)
promise constructors from racket/promise, as well as thunk from racket/function. However, beware that choices such as thunk or delay/name may evaluate their subexpression multiple times, which can lead to nonsensical results for sequences that have state, as the state will be shared between all evaluations of the recur-expr.

If multiple accum-ids are given, the #:delay-with option is provided, and delayer-id is not bound to one of delay, lazy, delay/strict, delay/sync, delay/thread, or delay/idle, the accum-ids will not be bound at all, even within the loop body. Instead, the #:delay-as option must be specified to access the accumulator values via delayed-id.

Added in version 7.3.0.3 of package base.

```
(for* (for-clause ...) body-or-break ... body)
```

Like for, but with an implicit #:when #t between each pair of for-clauses, so that all sequence iterations are nested.

Example:

```
> (for* ([i '(1 2)]
         [j "ab"]
       (display (list i j))
   (1 a)(1 b)(2 a)(2 b)
```

```
(for*/list (for-clause ...) body-or-break ... body)
(for*/lists (id ... maybe-result) (for-clause ...) body-or-break ... body)
(for*/vector maybe-length (for-clause ...) body-or-break ... body)
(for*/hash (for-clause ...) body-or-break ... body)
(for*/hasheq (for-clause ...) body-or-break ... body)
(for*/and (for-clause ...) body-or-break ... body)
(for*/or (for-clause ...) body-or-break ... body)
(for*/sum (for-clause ...) body-or-break ... body)
(for*/product (for-clause ...) body-or-break ... body)
(for*/first (for-clause ...) body-or-break ... body)
(for*/last (for-clause ...) body-or-break ... body)
(for*/fold ([accum-id init-expr] ... maybe-result) (for-clause ...) body-or-break ... body)
(for*/foldr ([accum-id init-expr] ... accum-option ...) (for-clause ...) body-or-break ... body)
```

Like for/list, etc., but with the implicit nesting of for*.

Example:
> (for*/list ([i '(1 2)]
   [j "ab"])
   (list i j))
'((1 #\a) (1 #\b) (2 #\a) (2 #\b))

Changed in version 7.3.0.3 of package base: Added the for*/foldr form.

3.18.2 Deriving New Iteration Forms

(for/fold/derived orig-datum
 ([accum-id init-expr] ... maybe-result) (for-clause ...)
 body-or-break ... body)

Like for/fold, but the extra orig-datum is used as the source for all syntax errors.

A macro that expands to for/fold/derived should typically use split-for-body to handle the possibility of macros and other definitions mixed with keywords like #:break.

Examples:

> (require (for-syntax syntax/for-body))
> (define-syntax (for/digits stx)
  (syntax-case stx ()
   [(_ clauses body ... tail-expr)
    (with-syntax ([original stx]
      [((pre-body ...) (post-body ...))
       (split-for-body stx #'(body ... tail-
                                 expr))])
    #'(let-values
       ([[(n k)
        (for/fold/derived
         original ([n 0] [k 1])
         clauses
         pre-body ...
         (values (+ n (* (let () post-
                          body ...) k)) (* k 10))))
      n))])

; If we misuse for/digits, we can get good error reporting
; because the use of orig-datum allows for source correlation:
> (for/digits
   [a (in-list '(1 2 3))]
   [b (in-list '(4 5 6))]
   (+ a b))
eval:4:0: for/digits: bad sequence binding clause
at: a
in: (for/digits (a (in-list (quote (1 2 3)))) (b (in-list (quote (4 5 6)))) (+ a b))
> (for/digits
   ([a (in-list '(1 2 3)])
    [b (in-list '(2 4 6)])
   (+ a b))
963
; Another example: compute the max during iteration:
> (define-syntax (for/max stx)
   (syntax-case stx ()
      [(_ clauses body ... tail-expr)
       (with-syntax ([original stx]
          [((pre-body ...) (post-body ...))
           (split-for-body stx #'(body ... tail-expr))])
        #'(for/fold/derived original
           ([current-max -inf.0])
            clauses
            pre-body ...
            (define maybe-new-max (let () post-body ...))
            (if (> maybe-new-max current-max)
                maybe-new-max
                current-max)))]))
> (for/max ([n '(3.14159 2.71828 1.61803)]
            [s '(-1 1 1)])
   (* n s))
2.71828

Changed in version 6.11.0.1 of package base: Added the #:result form.

(for*/fold/derived orig-datum
  ([accum-id init-expr] ... maybe-result) (for-clause ...)
  body-or-break ... body)

Like for*/fold, but the extra orig-datum is used as the source for all syntax errors.

Examples:

> (require (for-syntex syntax/for-body))
> (define-syntax (for*/digits stx)
   (syntax-case stx ()
      [(_ clauses body ... tail-expr)
       (with-syntax ([original stx]
          [((pre-body ...) (post-body ...))))
        156
(split-for-body stx #'(body ... tail-expr))]

(let-values
  ([(n k)
    (for*/fold/derived original ([n 0] [k 1])
    clauses
    pre-body ...
    (values (+ n (* (let () post-body ...) k)) (* k 10)))
    (n)))]))

> (for*/digits
   ([ds (in-list '((8 3) (1 1)))]
    [d (in-list ds)]
    d)

**eval:10:0: for*/digits: bad sequence binding clause**
**at: ds**
**in: (for*/digits (ds (in-list quote ((8 3) (1 1)))) (d (in-list ds)) d)**
> (for*/digits
   ([ds (in-list '((8 3) (1 1)))]
    [d (in-list ds)]
    d)

1138

Changed in version 6.11.0.1 of package base: Added the #:result form.

(for/foldr/derived orig-datum
  ([accum-id init-expr] ... accum-option ...) (for-clause ...)
  body-or-break ... body)
(for*/foldr/derived orig-datum
  ([accum-id init-expr] ... accum-option ...) (for-clause ...)
  body-or-break ... body)

Like for/foldr and for*/foldr, but the extra orig-datum is used as the source for all syntax errors as in for/fold/derived and for*/fold/derived.

Added in version 7.3.0.3 of package base.

(define-sequence-syntax id
  expr-transform-expr
  clause-transform-expr)

  expr-transform-expr : (or/c (-> identifier?)
                      (syntax? . -> . syntax?))

  clause-transform-expr : (syntax? . -> . syntax?)
Defines \texttt{id} as syntax. An \texttt{(id . rest)} form is treated specially when used to generate a sequence in a \texttt{for-clause} of \texttt{for} (or one of its variants). In that case, the procedure result of \texttt{clause-transform-expr} is called to transform the clause.

When \texttt{id} is used in any other expression position, the result of \texttt{expr-transform-expr} is used. If it is a procedure of zero arguments, then the result must be an identifier \texttt{other-id}, and any use of \texttt{id} is converted to a use of \texttt{other-id}. Otherwise, \texttt{expr-transform-expr} must produce a procedure (of one argument) that is used as a macro transformer.

When the \texttt{clause-transform-expr} transformer is used, it is given a \texttt{for-clause} as an argument, where the clause’s form is normalized so that the left-hand side is a parenthesized sequence of identifiers. The right-hand side is of the form \texttt{(id . rest)}. The result can be either \texttt{#f}, to indicate that the forms should not be treated specially (perhaps because the number of bound identifiers is inconsistent with the \texttt{(id . rest)} form), or a new \texttt{for-clause} to replace the given one. The new clause might use \texttt{:do-in}. To protect identifiers in the result of \texttt{clause-transform-expr}, use \texttt{for-clause-syntax-protect} instead of \texttt{syntax-protect}.

Examples:

```scheme
> (define (check-nat n)
  (unless (exact-nonnegative-integer? n)
    (raise-argument-error 'in-digits "exact-nonnegative-integer?" n)))
> (define-sequence-syntax in-digits
  (lambda () #'in-digits/proc)
  (lambda (stx)
    (syntax-case stx ()
      [(d) (_ nat)]
      [#'[d]
        (:do-in
          ([n] nat]
          (check-nat n)
          ([i n])
          (not (zero? i))
          ([j d] (quotient/remainder i 10))
          #t
          #t
          [j]])]
      [else #f])))
> (define (in-digits/proc n)
  (for/list ([d (in-digits n)]) d))
> (for/list ([d (in-digits 1138)]) d)
'(8 3 1 1)
> (map in-digits (list 137 216))
'((7 3 1) (6 1 2))
```
A form that can only be used as a seq-expr in a for-clause of for (or one of its variants).

Within a for, the pieces of the :do-in form are spliced into the iteration essentially as follows:

```scheme
(let-values ([([outer-id ...] outer-expr] ...) outer-check
  ([loop-id loop-expr] ...) pos-guard
  ([([inner-id ...] inner-expr] ...) pre-guard
    post-guard
    (loop-arg ...)))
```

where body-bindings and done-expr are from the context of the :do-in use. The identifiers bound by the for clause are typically part of the ([([inner-id ...] inner-expr] ...) section.

Beware that body-bindings and done-expr can contain arbitrary expressions, potentially including set! on outer-id or inner-id identifiers if they are visible in the original for form, so beware of depending on such identifiers in post-guard and loop-arg.

The actual loop binding and call has additional loop arguments to support iterations in parallel with the :do-in form, and the other pieces are similarly accompanied by pieces from parallel iterations.

For an example of :do-in, see define-sequence-syntax.

```
(for-clause-syntax-protect stx) -> syntax?
```

Provided for-syntax: Like syntax-protect, but allows the for expander to disarm the result syntax object, and arms the pieces of a clause instead of the entire syntax object.
Use this function to protect the result of a `clause-transform-expr` that is bound by `define-sequence-syntax`.

### 3.18.3 Do Loops

```
(do ([id init-expr step-expr-maybe] ...)  ; (id init-expr step-expr-maybe) ...
    (stop?-expr finish-expr ...)        ; (stop?-expr finish-expr ...) ...
    expr ...)                          ; expr ...
```

`step-expr-maybe = step-expr`

Iteratively evaluates the `expr`s for as long as `stop?-expr` returns `#f`.

To initialize the loop, the `init-expr`s are evaluated in order and bound to the corresponding `id`s. The `id`s are bound in all expressions within the form other than the `init-expr`s.

After the `id`s have been bound, the `stop?-expr` is evaluated. If it produces `#f`, each `expr` is evaluated for its side-effect. The `id`s are then effectively updated with the values of the `step-expr`s, where the default `step-expr` for `id` is just `id`; more precisely, iteration continues with fresh locations for the `id`s that are initialized with the values of the corresponding `step-expr`s.

When `stop?-expr` produces a true value, then the `finish-expr`s are evaluated in order, and the last one is evaluated in tail position to produce the overall value for the do form. If no `finish-expr` is provided, the value of the do form is `#<void>`.

### 3.19 Continuation Marks: `with-continuation-mark`

```
(with-continuation-mark key-expr val-expr result-expr)
```

The `key-expr`, `val-expr`, and `result-expr` expressions are evaluated in order. After `key-expr` is evaluated to obtain a key and `val-expr` is evaluated to obtain a value, the key is mapped to the value as a continuation mark in the current continuation’s initial continuation frame. If the frame already has a mark for the key, the mark is replaced. Finally, the `result-expr` is evaluated; the continuation for evaluating `result-expr` is the continuation of the `with-continuation-mark` expression (so the result of the `result-expr` is the result of the `with-continuation-mark` expression, and `result-expr` is in tail position for the `with-continuation-mark` expression).

### 3.20 Quasiquoting: `quasiquote`, `unquote`, and `unquote-splicing`

```
(quasiquote datum)
```
The same as `datum if datum does not include (unquote expr) or (unquote-splicing expr). An (unquote expr) form escapes from the quote, however, and the result of the expr takes the place of the (unquote expr) form in the quasiquote result. An (unquote-splicing expr) similarly escapes, but the expr produces a list whose elements are spliced as multiple values place of the (unquote-splicing expr).

An unquote or unquote-splicing form is recognized in any of the following escaping positions within datum: in a pair, in a vector, in a box, in a prefab structure field after the name position, and in hash table value position (but not in a hash table key position). Such escaping positions can be nested to an arbitrary depth.

An unquote-splicing form must appear as the car of a quoted pair, as an element of a quoted vector, or as an element of a quoted prefab structure. In the case of a pair, if the cdr of the relevant quoted pair is empty, then expr need not produce a list, and its result is used directly in place of the quoted pair (in the same way that append accepts a non-list final argument).

If unquote or unquote-splicing appears within quasiquote in an escaping position but in a way other than as (unquote expr) or (unquote-splicing expr), a syntax error is reported.

Examples:

```lisp
> (quasiquote (0 1 2))
'(0 1 2)

> (quasiquote (0 (unquote (+ 1 2)) 4))
'(0 3 4)

> (quasiquote (0 (unquote-splicing (list 1 2)) 4))
'(0 1 2 4)

> (quasiquote (0 (unquote-splicing 1) 4))
unquote-splicing: contract violation
  expected: list?
given: 1

> (quasiquote (0 (unquote-splicing 1))))
'(0 . 1)
```

A quasiquote, unquote, or unquote-splicing form is typically abbreviated with `\', `\n', or `\n', respectively. See also §1.3.8 "Reading Quotes".

Examples:

```lisp
> `(0 1 2)
'(0 1 2)

> `(1 ,(+ 1 2) 4)
'(1 3 4)

> `#s(stuff 1 ,(+ 1 2) 4)
'#s(stuff 1 3 4)
```
A quasiquote form within the original `datum` increments the level of quasiquotation: within the quasiquote form, each unquote or unquote-splicing is preserved, but a further nested unquote or unquote-splicing escapes. Multiple nestings of quasiquote require multiple nestings of unquote or unquote-splicing to escape.

Examples:

```scheme
> `(1 ,,@,,,@,(list (+ 1 2)) 4)
'(1 1 2 4)
> `(1 1 2 4)
'(1 1 2 4)
```

The quasiquote form allocates only as many fresh cons cells, vectors, and boxes as are needed without analyzing unquote and unquote-splicing expressions. For example, in `'(1 2 3)` a single tail `'(2 3)` is used for every evaluation of the quasiquote expression. When allocating fresh data, the quasiquote form allocates mutable vectors, mutable boxes and immutable hashes.

Examples:

```scheme
> (immutable? `@((,0))
#f
> (immutable? `@(#hash((a ,0)))
#t
```

See quasiquote, where unquote is recognized as an escape. An unquote form as an expression is a syntax error.

See quasiquote, where unquote-splicing is recognized as an escape. An unquote-splicing form as an expression is a syntax error.
3.21 Syntax Quoting: quote-syntax

```scheme
(quote-syntax datum)
(quote-syntax datum #:local)
```

Similar to `quote`, but produces a syntax object that preserves the lexical information and source-location information attached to `datum` at expansion time.

When `#:local` is specified, then all scopes in the syntax object’s lexical information are preserved. When `#:local` is omitted, then the scope sets within `datum` are pruned to omit the scope for any binding form that appears between the `quote-syntax` form and the enclosing top-level context, module body, or phase level crossing, whichever is closer.

Unlike `syntax (#)`), `quote-syntax` does not substitute pattern variables bound by `with-syntax`, `syntax-parse`, or `syntax-case`.

Examples:

```scheme
> (syntax? (quote-syntax x))
#t
> (quote-syntax (1 2 3))
#<syntax:eval:78:0 (1 2 3)>
> (with-syntax ([a #
#5])
  (quote-syntax (a b c)))
#<syntax:eval:79:0 (a b c)>
> (free-identifier=? (let ([x 1]) (quote-syntax x))
  (quote-syntax x))
#t
> (free-identifier=? (let ([x 1]) (quote-syntax x #:local))
  (quote-syntax x))
#f
```

Changed in version 6.3 of package `base`: Added scope pruning and support for `#:local`.

3.22 Interaction Wrapper: #%top-interaction

```scheme
(%top-interaction . form)
```

Expands to simply `form`. The `#%top-interaction` form is similar to `#%app` and `#%module-begin`, in that it provides a hook to control interactive evaluation through `load` (more precisely, the default load handler) or `read-eval-print-loop`.
3.23 Blocks: block

```
(require racket/block)   package: base
```

The bindings documented in this section are provided by the `racket/block` library, not `racket/base` or `racket`.

```
(block defn-or-expr ...)
```

Supports a mixture of expressions and mutually recursive definitions, as in a module body. Unlike an internal-definition context, the last `defn-or-expr` need not be an expression.

The result of the block form is the result of the last `defn-or-expr` if it is an expression, `<void>` otherwise. If no `defn-or-expr` is provided (after flattening `begin` forms), the result is `<void>`.

The final `defn-or-expr` is executed in tail position, if it is an expression.

Examples:
```
> (define (f x)
  (block
    (define y (add1 x))
    (displayln y)
    (define z (* 2 y))
    (+ 3 z)))
> (f 12)
13 29
```

3.24 Internal-Definition Limiting: `#%stratified-body`

```
(#%stratified-body defn-or-expr ...)
```

Like `(let () defn-or-expr ...) for an internal-definition context sequence, except that an expression is not allowed to precede a definition, and all definitions are treated as referring to all other definitions (i.e., locations for variables are all allocated first, like `letrec` and unlike `letrec-syntaxes+values`).

The `#%stratified-body` form is useful for implementing syntactic forms or languages that supply a more limited kind of internal-definition context.

3.25 Performance Hints: `begin-encourage-inline`

```
(require racket/performance-hint)   package: base
```

164
The bindings documented in this section are provided by the `racket/performance-hint` library, not `racket/base` or `racket`.

```racket
(begin-encourage-inline form ...)
```

Attaches a `'compiler-hint:cross-module-inline` syntax property to each `form`, which is useful when a `form` is a function definition. See `define-values`.

The `begin-encourage-inline` form is also provided by the `(submod racket/performance-hint begin-encourage-inline)` module, which has fewer dependencies than `racket/performance-hint`.

Changed in version 6.2 of package `base`: Added the `(submod racket/performance-hint begin-encourage-inline)` submodule.

```racket
(define-inline id expr)
(define-inline (head args) body ...+)
```

```racket
head = id
 | (head args)
args = arg ...
 | arg ... . rest-id
arg = arg-id
 | [arg-id default-expr]
 | keyword arg-id
 | keyword [arg-id default-expr]
```

Like `define`, but ensures that the definition will be inlined at its call sites. Recursive calls are not inlined, to avoid infinite inlining. Higher-order uses are supported, but also not inlined. Misapplication (by supplying the wrong number of arguments or incorrect keyword arguments) is also not inlined and left as a run-time error.

The `define-inline` form may interfere with the Racket compiler’s own inlining heuristics, and should only be used when other inlining attempts (such as `begin-encourage-inline`) fail.

Changed in version 8.1.0.5 of package `base`: Changed to treat misapplication as a run-time error.

### 3.26 Importing Modules Lazily: `lazy-require`

```
(require racket/lazy-require)  package:base
```

The bindings documented in this section are provided by the `racket/lazy-require` library, not `racket/base` or `racket`.

165
(lazy-require [module-path (fun-import ...)] ...)

fun-import = fun-id
  | (orig-fun-id fun-id)

Defines each fun-id as a function that, when called, dynamically requires the export named orig-fun-id from the module specified by module-path and calls it with the same arguments. If orig-fun-id is not given, it defaults to fun-id.

If the enclosing relative phase level is not 0, then module-path is also placed in a submodule (with a use of define-runtime-module-path-index at phase level 0 within the submodule). Introduced submodules have the names lazy-require-auxn-m, where n is a phase-level number and m is a number.

When the use of a lazily-required function triggers module loading, it also triggers a use of register-external-module to declare an indirect compilation dependency (in case the function is used in the process of compiling a module).

Examples:

> (lazy-require [racket/list (partition)])
> (partition even? '(1 2 3 4 5))
  '(2 4)
  '(1 3 5)
> (module hello racket/base
  (provide hello)
  (printf "starting hello server\n")
  (define (hello) (printf "hello!\n")))
> (lazy-require ['hello ([hello greet])])
> (greet)
starting hello server
hello!

(lazy-require-syntax [module-path (macro-import ...)] ...)

macro-import = macro-id
  | (orig-macro-id macro-id)

Like lazy-require but for macros. That is, it defines each macro-id as a macro that, when used, dynamically loads the macro’s implementation from the given module-path. If orig-macro-id is not given, it defaults to macro-id.

Use lazy-require-syntax in the implementation of a library with large, complicated macros to avoid a dependence from clients of the library on the macro “compilers.”
that only macros with exceptionally large compile-time components (such as Typed Racket, which includes a type checker and optimizer) benefit from lazy-require-syntax; typical macros do not.

**Warning:** lazy-require-syntax breaks the invariants that Racket’s module loader and linker rely on; these invariants normally ensure that the references in code produced by a macro are loaded before the code runs. Safe use of lazy-require-syntax requires a particular structure in the macro implementation. (In particular, lazy-require-syntax cannot simply be introduced in the client code.) The macro implementation must follow these rules:

1. the interface module must require the runtime-support module
2. the compiler module must require the runtime-support module via an *absolute* module path rather than a *relative* path

To explain the concepts of “interface, compiler, and runtime-support modules”, here is an example module that exports a macro:

```racket
(module original racket/base
  (define (ntimes-proc n thunk)
    (for ([i (in-range n)] (thunk)))
  (define-syntax-rule (ntimes n expr)
    (ntimes-proc n (lambda () expr)))
  (provide ntimes))
```

Suppose we want to use lazy-require-syntax to lazily load the implementation of the ntimes macro transformer. The original module must be split into three parts:

```racket
(module runtime-support racket/base
  (define (ntimes-proc n thunk)
    (for ([i (in-range n)] (thunk)))
  (provide ntimes-proc))
(module compiler racket/base
  (require 'runtime-support)
  (define-syntax-rule (ntimes n expr)
    (ntimes-proc n (lambda () expr)))
  (provide ntimes))
(module interface racket/base
  (require racket/lazy-require)
  (require 'runtime-support)
  (lazy-require-syntax ['Compiler (ntimes)]
  (provide ntimes))
```

The runtime support module contains the function and value definitions that the macro refers to. The compiler module contains the macro definition(s) themselves—the part of the code
that “disappears” after compile time. The interface module lazily loads the macro transformer, but it makes sure the runtime support module is defined at run time by requiring it normally. In a larger example, of course, the runtime support and compiler may both consist of multiple modules.

Here what happens when we don’t separate the runtime support into a separate module:

```lisp
> (module bad-no-runtime racket/base
  (define (ntimes-proc n thunk)
    (for ([i (in-range n)]) (thunk)))
  (define-syntax-rule (ntimes n expr)
    (ntimes-proc n (lambda () expr)))
  (provide ntimes))

> (module bad-client racket/base
  (require racket/lazy-require)
  (lazy-require-syntax ['bad-no-runtime (ntimes)])
  (ntimes 3 (printf "hello?\n")))

> (require 'bad-client)
require: namespace mismatch;
  reference to a module that is not instantiated
  module: 'bad-no-runtime
  phase: 0
```

A similar error occurs when the interface module doesn’t introduce a dependency on the runtime support module.
4 Datatypes

Each pre-defined datatype comes with a set of procedures for manipulating instances of the datatype.

4.1 Equality

Equality is the concept of whether two values are “the same.” Racket supports a few different kinds of equality by default, though equal? is preferred for most use cases.

\[
\text{equal?} \quad (v1 \ v2) \rightarrow \text{boolean?}
\]

\[
\begin{align*}
v1 &: \text{any/c} \\
v2 &: \text{any/c}
\end{align*}
\]

Two values are equal? if and only if they are eqv?, unless otherwise specified for a particular datatype.

Datatypes with further specification of equal? include strings, byte strings, pairs, mutable pairs, vectors, boxes, hash tables, and inspectable structures. In the last six cases, equality is recursively defined; if both \(v1\) and \(v2\) contain reference cycles, they are equal when the infinite unfoldings of the values would be equal. See also gen:equal+hash and prop:impersonator-of.

Examples:

\[
\begin{align*}
> (\text{equal?} \ 'yes \ 'yes) & \Rightarrow \text{#t} \\
> (\text{equal?} \ 'yes \ 'no) & \Rightarrow \text{#f} \\
> (\text{equal?} \ (* \ 6 \ 7) \ 42) & \Rightarrow \text{#t} \\
> (\text{equal?} \ (\text{expt} \ 2 \ 100) \ (\text{expt} \ 2 \ 100)) & \Rightarrow \text{#t} \\
> (\text{equal?} \ 2 \ 2.0) & \Rightarrow \text{#f} \\
> (\text{let} \ ((v \ (\text{mcons} \ 1 \ 2))) \ (\text{equal?} \ v \ v)) & \Rightarrow \text{#t} \\
> (\text{equal?} \ (\text{mcons} \ 1 \ 2) \ (\text{mcons} \ 1 \ 2)) & \Rightarrow \text{#t} \\
> (\text{equal?} \ (\text{integer->char} \ 955) \ (\text{integer->char} \ 955)) & \Rightarrow \text{#t} \\
> (\text{equal?} \ (\text{make-string} \ 3 \ #\z) \ (\text{make-string} \ 3 \ #\z)) & \Rightarrow \text{#t} \\
> (\text{equal?} \ #t \ #t) & \Rightarrow \text{#t}
\end{align*}
\]
Two values are `eqv?` if and only if they are `eq?`, unless otherwise specified for a particular datatype.

The number and character datatypes are the only ones for which `eqv?` differs from `eq?`. Two numbers are `eqv?` when they have the same exactness, precision, and are both equal and non-zero, both `+0.0`, both `+0.0f0`, both `-0.0`, both `-0.0f0`, both `+nan.0`, or both `+nan.f`—considering real and imaginary components separately in the case of complex numbers. Two characters are `eqv?` when their `char->integer` results are equal.

Generally, `eqv?` is identical to `equal?` except that the former cannot recursively compare the contents of compound data types (such as lists and structs) and cannot be customized by user-defined data types. The use of `eqv?` is lightly discouraged in favor of `equal?`.

Examples:

```
> (eqv? 'yes 'yes)
#t
> (eqv? 'yes 'no)
#f
> (eqv? (* 6 7) 42)
#t
> (eqv? (expt 2 100) (expt 2 100))
#t
> (eqv? 2 2.0)
#f
> (let ([v (mcons 1 2)]) (eqv? v v))
#t
> (eqv? (mcons 1 2) (mcons 1 2))
#f
> (eqv? (integer->char 955) (integer->char 955))
#t
> (eqv? (make-string 3 #\z) (make-string 3 #\z))
#f
> (eqv? #t #t)
#t
```

(eq? v1 v2) → boolean?

v1 : any/c
v2 : any/c
Return #t if \( v_1 \) and \( v_2 \) refer to the same object, #f otherwise. As a special case among numbers, two fixnums that are \( = \) are also the same according to eq?. See also §4.1.1 “Object Identity and Comparisons”.

Examples:

\[
\begin{align*}
> & \ (\text{eq?} \ 'yes \ 'yes) \\
& \ #t \\
> & \ (\text{eq?} \ 'yes \ 'no) \\
& \ #f \\
> & \ (\text{eq?} \ (* \ 6 \ 7) \ 42) \\
& \ #t \\
> & \ (\text{eq?} \ (\text{expt} \ 2 \ 100) \ (\text{expt} \ 2 \ 100)) \\
& \ #f \\
> & \ (\text{eq?} \ 2 \ 2.0) \\
& \ #f \\
> & \ (\text{let} \ ([v \ (\text{mcons} \ 1 \ 2)]) \ (\text{eq?} \ v \ v)) \\
& \ #t \\
> & \ (\text{eq?} \ (\text{mcons} \ 1 \ 2) \ (\text{mcons} \ 1 \ 2)) \\
& \ #f \\
> & \ (\text{eq?} \ (\text{integer->char} \ 955) \ (\text{integer->char} \ 955)) \\
& \ #t \\
> & \ (\text{eq?} \ (\text{make-string} \ 3 \ #\z) \ (\text{make-string} \ 3 \ #\z)) \\
& \ #f \\
> & \ (\text{eq?} \ #t \ #t) \\
& \ #t
\end{align*}
\]

\[
(\text{equal?}/\text{recur} \ v_1 \ v_2 \ \text{recur-proc}) \to \text{boolean?} \\
\begin{align*}
v_1 : & \ \text{any/c} \\
v_2 : & \ \text{any/c} \\
\text{recur-proc} : & \ (\text{any/c} \ \text{any/c} \to \text{any/c})
\end{align*}
\]

Like equal?, but using recur-proc for recursive comparisons (which means that reference cycles are not handled automatically). Non-#f results from recur-proc are converted to #t before being returned by equal?/recur.

Examples:

\[
\begin{align*}
> & \ (\text{equal?}/\text{recur} \ 1 \ 1 \ (\text{lambda} \ (a \ b) \ #f)) \\
& \ #t \\
> & \ (\text{equal?}/\text{recur} \ '(1) \ '(1) \ (\text{lambda} \ (a \ b) \ #f)) \\
& \ #f \\
> & \ (\text{equal?}/\text{recur} \ '#(1 \ 1 \ 1) \ '#(1 \ 1.2 \ 3/4) \\
& \ (\text{lambda} \ (a \ b) \ (\leq (\text{abs} \ (- a \ b)) \ 0.25))) \\
& \ #t
\end{align*}
\]
4.1.1 Object Identity and Comparisons

The `eq?` operator compares two values, returning `#t` when the values refer to the same object. This form of equality is suitable for comparing objects that support imperative update (e.g., to determine that the effect of modifying an object through one reference is visible through another reference). Also, an `eq?` test evaluates quickly, and `eq?`-based hashing is more lightweight than `equal?`-based hashing in hash tables.

In some cases, however, `eq?` is unsuitable as a comparison operator, because the generation of objects is not clearly defined. In particular, two applications of `+` to the same two exact integers may or may not produce results that are `eq?`, although the results are always `equal?`. Similarly, evaluation of a `lambda` form typically generates a new procedure object, but it may re-use a procedure object previously generated by the same source `lambda` form.

The behavior of a datatype with respect to `eq?` is generally specified with the datatype and its associated procedures.

4.1.2 Equality and Hashing

All comparable values have at least one hash code — an arbitrary integer (more specifically a fixnum) computed by applying a hash function to the value. The defining property of these hash codes is that **equal values have equal hash codes**. Note that the reverse is not true: two unequal values can still have equal hash codes. Hash codes are useful for various indexing and comparison operations, especially in the implementation of hash tables. See §4.14 “Hash Tables” for more information.

```
(equal-hash-code v) ∈ fixnum?
| v : any/c
```

Returns a hash code consistent with `equal?`. For any two calls with `equal?` values, the returned number is the same. A hash code is computed even when `v` contains a cycle through pairs, vectors, boxes, and/or inspectable structure fields. Additionally, user-defined data types can customize how this hash code is computed by implementing `gen:equal+hash`.

For any `v` that could be produced by `read`, if `v2` is produced by `read` for the same input characters, the `(equal-hash-code v)` is the same as `(equal-hash-code v2)` — even if `v` and `v2` do not exist at the same time (and therefore could not be compared by calling `equal?`).

Changed in version 6.4.0.12 of package base: Strengthened guarantee for readable values.

```
(equal-secondary-hash-code v) ∈ fixnum?
| v : any/c
```

Like `equal-hash-code`, but computes a secondary hash code suitable for use in double
(eq-hash-code v) → fixnum?
  v : any/c

Returns a hash code consistent with eq?. For any two calls with eq? values, the returned number is the same.

(eqv-hash-code v) → fixnum?
  v : any/c

Returns a hash code consistent with eqv?. For any two calls with eqv? values, the returned number is the same.

### 4.1.3 Implementing Equality for Custom Types

| gen:equal+hash : any/c |

A generic interface (see §5.4 “Generic Interfaces”) for types that can be compared for equality using equal?. The following methods must be implemented:

- **equal-proc** : (-> any/c any/c (-> any/c any/c boolean?) any/c) — tests whether the first two arguments are equal, where both values are instances of the structure type to which the generic interface is associated (or a subtype of the structure type).

  The third argument is an equal? predicate to use for recursive equality checks; use the given predicate instead of equal? to ensure that data cycles are handled properly and to work with equal?/recur (but beware that an arbitrary function can be provided to equal?/recur for recursive checks, which means that arguments provided to the predicate might be exposed to arbitrary code).

  The equal-proc is called for a pair of structures only when they are not eq?, and only when they both have a gen:equal+hash value inherited from the same structure type. With this strategy, the order in which equal? receives two structures does not matter. It also means that, by default, a structure sub-type inherits the equality predicate of its parent, if any.

- **hash-proc** : (-> any/c (-> any/c exact-integer?) exact-integer?) — computes a hash code for the given structure, like equal-hash-code. The first argument is an instance of the structure type (or one of its subtypes) to which the generic interface is associated.

  The second argument is an equal-hash-code-like procedure to use for recursive hash-code computation; use the given procedure instead of equal-hash-code to ensure that data cycles are handled properly.
Although the result of hash-proc can be any exact integer, it will be truncated for most purposes to a fixnum (e.g., for the result of equal-hash-code). Roughly, truncation uses bitwise-and to take the lower bits of the number. Thus, variation in the hash-code computation should be reflected in the fixnum-compatible bits of hash-proc’s result. Consumers of a hash code are expected to use variation within the fixnum range appropriately, and producers are not responsible to reflect variation in hash codes across the full range of bits that fit within a fixnum.

- **hash2-proc**: 
  
  \[(-> any/c (-> any/c exact-integer?) exact-integer?)]

  — computes a secondary hash code for the given structure. This procedure is like hash-proc, but analogous to equal-secondary-hash-code.

Take care to ensure that hash-proc and hash2-proc are consistent with equal-proc. Specifically, hash-proc and hash2-proc should produce the same value for any two structures for which equal-proc produces a true value.

When a structure type has no gen:equal+hash implementation, then transparent structures (i.e., structures with an inspector that is controlled by the current inspector) are equal? when they are instances of the same structure type (not counting sub-types), and when they have equal? field values. For transparent structures, equal-hash-code and equal-secondary-hash-code derive hash code using the field values. For opaque structure types, equal? is the same as eq?, and equal-hash-code and equal-secondary-hash-code results are based only on eq-hash-code. If a structure has a prop:impersonator-of property, then the prop:impersonator-of property takes precedence over gen:equal+hash if the property value’s procedure returns a non-#f value when applied to the structure.

Examples:

```scheme
(define (farm=? farm1 farm2 recursive-equal?)
  (and (= (farm-apples farm1) (farm-apples farm2))
       (= (farm-oranges farm1) (farm-oranges farm2))
       (= (farm-sheep farm1) (farm-sheep farm2))))

(define (farm-hash-code farm recursive-equal-hash)
  (+ (* 10000 (farm-apples farm))
      (* 100 (farm-oranges farm))
      (* 1 (farm-sheep farm))))

(define (farm-secondary-hash-code farm recursive-equal-hash)
  (+ (* 10000 (farm-sheep farm))
      (* 100 (farm-apples farm))
      (* 1 (farm-oranges farm))))
```

174
(struct farm (apples oranges sheep)
 #:methods gen:equal+hash
 [(define equal-proc farm=?)
  (define hash-proc farm-hash-code)
  (define hash2-proc farm-secondary-hash-code)]
)

(define eastern-farm (farm 5 2 20))
(define western-farm (farm 18 6 14))
(define northern-farm (farm 5 20 20))
(define southern-farm (farm 18 6 14))

> (equal? eastern-farm western-farm)
#f
> (equal? eastern-farm northern-farm)
#f
> (equal? western-farm southern-farm)
#t

(prop:equal+hash : struct-type-property?)

A structure type property (see §5.3 “Structure Type Properties”) that supplies an equality predicate and hashing functions for a structure type. Using the prop:equal+hash property is discouraged; the gen:equal+hash generic interface should be used instead. A prop:equal+hash property value is a list of three procedures that correspond to the methods of gen:equal+hash:

- equal-proc : (-> any/c any/c (-> any/c any/c boolean?) any/c)
- hash-proc : (-> any/c (-> any/c exact-integer?) exact-integer?)
- hash2-proc : (-> any/c (-> any/c exact-integer?) exact-integer?)

4.2 Booleans

True and false booleans are represented by the values #t and #f, respectively, though operations that depend on a boolean value typically treat anything other than #f as true. The #t value is always eq? to itself, and #f is always eq? to itself.

See §1.3.5 “Reading Booleans” for information on reading booleans and §1.4.4 “Printing Booleans” for information on printing booleans.

See also and, or, andmap, and ormap.

(boolean? v) → boolean?

v : any/c
Returns #t if \( v \) is #t or #f, #f otherwise.

Examples:

\[
\begin{align*}
> (\text{boolean? } #f) & \quad #t \\
> (\text{boolean? } #t) & \quad #t \\
> (\text{boolean? } '\text{true}) & \quad #f \\
\end{align*}
\]

\[
(\text{not } v) \rightarrow \text{boolean}? \\
v : \text{any/c}
\]

Returns #t if \( v \) is #f, #f otherwise.

Examples:

\[
\begin{align*}
> (\text{not } #f) & \quad #t \\
> (\text{not } #t) & \quad #f \\
> (\text{not } '\text{we-have-no-bananas}) & \quad #f \\
\end{align*}
\]

\[
(\text{immutable? } v) \rightarrow \text{boolean}? \\
v : \text{any/c}
\]

Returns #t if \( v \) is an immutable string, byte string, vector, hash table, or box, #f otherwise.

Note that \texttt{immutable?} is not a general predicate for immutability (despite its name). It works only for a handful of datatypes for which a single predicate—\texttt{string?}, \texttt{vector?}, etc.—recognizes both mutable and immutable variants of the datatype. In particular, \texttt{immutable?} produces #f for a pair, even though pairs are immutable, since \texttt{pair?} implies immutability.

Examples:

\[
\begin{align*}
> (\text{immutable? } '\text{hello}) & \quad #f \\
> (\text{immutable? } "\text{a string}") & \quad #t \\
> (\text{immutable? } (\text{box } 5)) & \quad #f \\
> (\text{immutable? } #(0 1 2 3)) & \quad #f \\
\end{align*}
\]
#t
> (immutable? (make-hash))
#f
> (immutable? (make-immutable-hash '(a b)))
#t
> (immutable? #t)
#f

## 4.2.1 Boolean Aliases

```racket
(require racket/bool)  ; package: base
```

The bindings documented in this section are provided by the `racket/bool` and `racket` libraries, but not `racket/base`.

### true : boolean?

An alias for `#t`.

### false : boolean?

An alias for `#f`.

### (symbol=? a b) → boolean?

```
| a  : symbol? |
| b  : symbol? |
```

Returns `(equal? a b)` (if `a` and `b` are symbols).

### (boolean=? a b) → boolean?

```
| a  : boolean? |
| b  : boolean? |
```

Returns `(equal? a b)` (if `a` and `b` are booleans).

### (false? v) → boolean?

```
| v  : any/c |
```

Returns `(not v)`.

### (nand expr ...)

Same as `(not (and expr ...))`.

Examples:
> (nand #f #t)
  #t
> (nand #f (error 'ack "we don't get here")
  #t

(nor expr ...)

Same as (not (or expr ...)).
In the two argument case, returns #t if neither of the arguments is a true value.
Examples:

> (nor #f #t)
  #f
> (nor #t (error 'ack "we don't get here")
  #f

(implies expr1 expr2)

Checks to be sure that the first expression implies the second.
Same as (if expr1 expr2 #t).
Examples:

> (implies #f #t)
  #t
> (implies #f #f)
  #t
> (implies #t #f)
  #f
> (implies #f (error 'ack "we don't get here")
  #t

(xor b1 b2) → any
  | b1 : any/c
  | b2 : any/c

Returns the exclusive or of b1 and b2.
If exactly one of b1 and b2 is not #f, then return it. Otherwise, returns #f.
Examples:
\begin{verbatim}
> (xor 11 #f)
11
> (xor #f 22)
22
> (xor 11 22)
#f
> (xor #f #f)
#f
\end{verbatim}

### 4.3 Numbers

All numbers are complex numbers. Some of them are real numbers, and all of the real numbers that can be represented are also rational numbers, except for \(+inf.0\) (positive infinity), \(+inf.f\) (single-precision variant, when enabled via \texttt{read-single-flonum}), \(-inf.0\) (negative infinity), \(-inf.f\) (single-precision variant, when enabled), \(+nan.0\) (not-a-number), and \(+nan.f\) (single-precision variant, when enabled). Among the rational numbers, some are integers, because \texttt{round} applied to the number produces the same number.

Orthogonal to those categories, each number is also either an exact number or an inexact number. Unless otherwise specified, computations that involve an inexact number produce inexact results. Certain operations on inexact numbers, however, produce an exact number, such as multiplying an inexact number with an exact \texttt{0}. Operations that mathematically produce irrational numbers for some rational arguments (e.g., \texttt{sqrt}) may produce inexact results even for exact arguments.

In the case of complex numbers, either the real and imaginary parts are both exact or inexact with the same precision, or the number has an exact zero real part and an inexact imaginary part; a complex number with an exact zero imaginary part is a real number.

Inexact real numbers are implemented as double-precision IEEE floating-point numbers, also known as flonums, or as single-precision IEEE floating-point numbers, also known as single-flonums. Single-flonums are supported only when \texttt{(single-flonum-available?)} reports \#t. Although we write \(+inf.f\), \(-inf.f\), and \(+nan.f\) to mean single-flonums, those forms read as double-precision flonums by default, since \texttt{read-single-flonum} is \#f by default. When single-flonums are supported, inexact numbers are still represented as flonums by default, and single precision is used only when a computation starts with single-flonums.

Inexact numbers can be coerced to exact form, except for the inexact numbers \(+inf.0\), \(+inf.f\), \(-inf.0\), \(-inf.f\), \(+nan.0\), and \(+nan.f\), which have no exact form. Dividing a number by exact zero raises an exception; dividing a non-zero number other than \(+nan.0\) or \(+nan.f\) by an inexact zero returns \(+inf.0\), \(+inf.f\), \(-inf.0\) or \(-inf.f\), depending on the sign and precision of the dividend. The \(+nan.0\) value is not = to itself, but \(+nan.0\) is \texttt{eqv?} to itself, and \(+nan.f\) is similarly \texttt{eqv?} but not = to itself. Conversely, \texttt{(= 0.0 -0.0)} is \#t, but (\texttt{eqv? 0.0 -0.0}) is \#f, and the same for \texttt{0.0f0} and \texttt{-0.0f0} (which are single-precision variants). The datum \(-nan.0\) refers to the same constant as \(+nan.0\), and \(-nan.f\)
is the same as `+nan.f`.

Calculations with infinities produce results consistent with IEEE double- or single-precision floating point where IEEE specifies the result; in cases where IEEE provides no specification, the result corresponds to the limit approaching infinity, or `+nan.0` or `+nan.f` if no such limit exists.

The precision and size of exact numbers is limited only by available memory (and the precision of operations that can produce irrational numbers). In particular, adding, multiplying, subtracting, and dividing exact numbers always produces an exact result.

A fixnum is an exact integer whose two’s complement representation fits into 30 or 31 bits (depending on the Racket variant) on a 32-bit platform or 61 or 63 bits (depending on the Racket variant) on a 64-bit platform. No allocation is required when computing with fixnums. See also the `racket/fixnum` module, below.

Two fixnums that are `=` are also the same according to `eq?`. Otherwise, the result of `eq?` applied to two numbers is undefined, except that numbers produced by the default reader in `read-syntax` mode are interned and therefore `eq?` when they are `eqv?`.

Two real numbers are `eqv?` when they are both inexact with the same precision or both exact, and when they are `=` (except for `+nan.0`, `+nan.f`, `0.0`, `+0.0f0`, `-0.0`, and `-0.0f0`, as noted above). Two complex numbers are `eqv?` when their real and imaginary parts are `eqv?`. Two numbers are `equal?` when they are `eqv?`.

See §1.3.3 “Reading Numbers” for information on reading numbers and §1.4.2 “Printing Numbers” for information on printing numbers.

### 4.3.1 Number Types

```scheme
(number? v) -> boolean?
  v : any/c
```

Returns `#t` if `v` is a number, `#f` otherwise.

Examples:

```scheme
> (number? 1)
#t
> (number? 2+3i)
#t
> (number? "hello")
#f
> (number? +nan.0)
#t
```

180
(complex? v) → boolean?
  v : any/c

Returns (number? v), because all numbers are complex numbers.

(real? v) → boolean?
  v : any/c

Returns #t if v is a real number, #f otherwise.

Examples:
> (real? 1)
#t
> (real? +inf.0)
#t
> (real? 2+3i)
#f
> (real? 2.0+0.0i)
#f
> (real? "hello")
#f

(rational? v) → boolean?
  v : any/c

Returns #t if v is a rational number, #f otherwise.

Examples:
> (rational? 1)
#t
> (rational? +inf.0)
#f
> (rational? "hello")
#f

(integer? v) → boolean?
  v : any/c

Returns #t if v is a number that is an integer, #f otherwise.

Examples:
> (integer? 1)  
#t
> (integer? 2.3)  
#f
> (integer? 4.0)  
#t
> (integer? +inf.0)  
#f
> (integer? 2+3i)  
#f
> (integer? "hello")  
#f

(exact-integer? v) → boolean?
  v : any/c

Returns (and (integer? v) (exact? v)).

Examples:

> (exact-integer? 1)  
#t
> (exact-integer? 4.0)  
#f

(exact-nonnegative-integer? v) → boolean?
  v : any/c

Returns (and (exact-integer? v) (not (negative? v))).

Examples:

> (exact-nonnegative-integer? 0)  
#t
> (exact-nonnegative-integer? -1)  
#f

(exact-positive-integer? v) → boolean?
  v : any/c

Returns (and (exact-integer? v) (positive? v)).

Examples:
> (exact-positive-integer? 1)
#t
> (exact-positive-integer? 0)
#f

(inexact-real? v) → boolean?
  v : any/c

Returns (and (real? v) (inexact? v)).

(fixnum? v) → boolean?
  v : any/c

Return #t if v is a fixnum, #f otherwise.

Note: the result of this function is platform-dependent, so using it in syntax transformers can lead to platform-dependent bytecode files. See also fixnum-for-every-system?.

(flonum? v) → boolean?
  v : any/c

Return #t if v is a flonum, #f otherwise.

(double-flonum? v) → boolean?
  v : any/c

Identical to flonum?.

(single-flonum? v) → boolean?
  v : any/c

Return #t if v is a single-flonum (i.e., a single-precision floating-point number), #f otherwise.

(single-flonum-available?) → boolean?

Returns #t if single-flonums are supported on the current platform, #f otherwise.

Currently, single-flonum-available? produces #t when (system-type 'vm) produces 'racket, and single-flonum-available? produces #f otherwise.

If the result is #f, then single-flonum? also produces #f for all arguments.

Added in version 7.3.0.5 of package base.
\[(\text{zero? } z) \rightarrow \text{boolean?}\]
\[z : \text{number?}\]

Returns \(= 0 z\).

Examples:
\[
\begin{align*}
> & (\text{zero? } 0) \\
& \texttt{#t}
\end{align*}
\]
\[
\begin{align*}
> & (\text{zero? } -0.0) \\
& \texttt{#t}
\end{align*}
\]

\[(\text{positive? } x) \rightarrow \text{boolean?}\]
\[x : \text{real?}\]

Returns \(> x 0\).

Examples:
\[
\begin{align*}
> & (\text{positive? } 10) \\
& \texttt{#t}
\end{align*}
\]
\[
\begin{align*}
> & (\text{positive? } -10) \\
& \texttt{#f}
\end{align*}
\]
\[
\begin{align*}
> & (\text{positive? } 0.0) \\
& \texttt{#f}
\end{align*}
\]

\[(\text{negative? } x) \rightarrow \text{boolean?}\]
\[x : \text{real?}\]

Returns \(< x 0\).

Examples:
\[
\begin{align*}
> & (\text{negative? } 10) \\
& \texttt{#f}
\end{align*}
\]
\[
\begin{align*}
> & (\text{negative? } -10) \\
& \texttt{#t}
\end{align*}
\]
\[
\begin{align*}
> & (\text{negative? } -0.0) \\
& \texttt{#f}
\end{align*}
\]

\[(\text{even? } n) \rightarrow \text{boolean?}\]
\[n : \text{integer?}\]

Returns \((\text{zero? } (\text{modulo } n 2))\).

Examples:
> (even? 10.0)
#t
> (even? 11)
#f
> (even? +inf.0)
  even?: contract violation
  expected: integer?
  given: +inf.0

> (odd? 10.0)
#f
> (odd? 11)
#t
> (odd? +inf.0)
  odd?: contract violation
  expected: integer?
  given: +inf.0

(odd? n) → boolean?
  n : integer?

Returns (not (even? n)).

Examples:

> (odd? 10.0)
#f
> (odd? 11)
#t
> (odd? +inf.0)
  odd?: contract violation
  expected: integer?
  given: +inf.0

(exact? z) → boolean?
  z : number?

Returns #t if z is an exact number, #f otherwise.

Examples:

> (exact? 1)
#t
> (exact? 1.0)
#f

(inexact? z) → boolean?
  z : number?

Returns #t if z is an inexact number, #f otherwise.

Examples:
> (inexact? 1)
#f
> (inexact? 1.0)
#t

(inexact->exact z) \rightarrow exact?
z : number?

Coerces z to an exact number. If z is already exact, it is returned. If z is +inf.0, -inf.0, +nan.0, +inf.f, -inf.f, or +nan.f, then the exn:fail:contract exception is raised.

Examples:
> (inexact->exact 1)
1
> (inexact->exact 1.0)
1

(exact->inexact z) \rightarrow inexact?
z : number?

Coerces z to an inexact number. If z is already inexact, it is returned.

Examples:
> (exact->inexact 1)
1.0
> (exact->inexact 1.0)
1.0

(real->single-flonum x) \rightarrow single-flonum?
x : real?

Coerces x to a single-precision floating-point number. If x is already a single-precision floating-point number, it is returned.

(real->double-flonum x) \rightarrow flonum?
x : real?

Coerces x to a double-precision floating-point number. If x is already a double-precision floating-point number, it is returned.
4.3.2 Generic Numerics

Most Racket numeric operations work on any kind of number.

**Arithmetic**

\[
(+ z ... ) \rightarrow \text{number?}
\]
\[
z : \text{number?}
\]

Returns the sum of the zs, adding pairwise from left to right. If no arguments are provided, the result is 0.

Examples:

\[
> (+ 1 2)
3
\]
\[
> (+ 1.0 2+3i 5)
8.0+3.0i
\]
\[
> (+)
0
\]

\[
(- z ) \rightarrow \text{number?}
\]
\[
z : \text{number?}
\]

\[
(- z w ... +) \rightarrow \text{number?}
\]
\[
z : \text{number?}
\]
\[
w : \text{number?}
\]

When no ws are supplied, returns (- 0 z). Otherwise, returns the subtraction of the ws from z working pairwise from left to right.

Examples:

\[
> (- 5 3.0)
2.0
\]
\[
> (- 1)
-1
\]
\[
> (- 2+7i 1 3)
-2+7i
\]

\[
(*) z ... ) \rightarrow \text{number?}
\]
\[
z : \text{number?}
\]

Returns the product of the zs, multiplying pairwise from left to right. If no arguments are provided, the result is 1. Multiplying any number by exact 0 produces exact 0.

Examples:
> (\(2 \cdot 3\))
6
> (\(8.0 \cdot 9\))
72.0
> (\(1+2i \cdot 3+4i\))
-5+10i

\((/ z) \rightarrow \text{number}\
 z : \text{number}\
(/ z w ...+) \rightarrow \text{number}\
 z : \text{number}\
 w : \text{number}\

When no \(w\)s are supplied, returns \((/ 1 z)\). Otherwise, returns the division of \(z\) by the \(w\)s working pairwise from left to right.

If \(z\) is exact \(0\) and no \(w\) is exact \(0\), then the result is exact \(0\). If any \(w\) is exact \(0\), the \texttt{exn:fail:contract:divide-by-zero} exception is raised.

Examples:

> (/ 3 4)
3/4
> (/ 81 3 3)
9
> (/ 10.0)
0.1
> (/ 1+2i 3+4i)
11/25+2/25i

\((\text{quotient } n m) \rightarrow \text{integer}\
 n : \text{integer}\
 m : \text{integer}\

Returns \((\text{truncate } (/ n m))\).

Examples:

> (quotient 10 3)
3
> (quotient -10.0 3)
-3.0
> (quotient +inf.0 3)
quotient: contract violation
expected: integer
given: +inf.0
(remainder n m) → integer?
n : integer?
m : integer?

Returns q with the same sign as n such that

- \((\text{abs } q)\) is between 0 (inclusive) and \((\text{abs } m)\) (exclusive), and
- \((+ q (* m (quotient n m)))\) equals n.

If m is exact 0, the exn:fail:contract:divide-by-zero exception is raised.

Examples:

> (remainder 10 3)
1
> (remainder -10.0 3)
-1.0
> (remainder 10.0 -3)
1.0
> (remainder -10 -3)
-1
> (remainder +inf.0 3)
remainder: contract violation
   expected: integer?
given: +inf.0

(quotient/remainder n m) → integer? integer?
n : integer?
m : integer?

Returns \((\text{values (quotient n m) (remainder n m)})\), but the combination may be computed more efficiently than separate calls to quotient and remainder.

Example:

> (quotient/remainder 10 3)
3
1

(modulo n m) → integer?
n : integer?
m : integer?
Returns $q$ with the same sign as $m$ where

- $(\text{abs } q)$ is between 0 (inclusive) and $(\text{abs } m)$ (exclusive), and
- the difference between $q$ and $(- n \times m \div \text{quotient } n m))$ is a multiple of $m$.

If $m$ is exact 0, the `exn:fail:contract:divide-by-zero` exception is raised.

Examples:

```lisp
> (modulo 10 3)
1
> (modulo -10.0 3)
2.0
> (modulo 10.0 -3)
-2.0
> (modulo -10 -3)
-1
> (modulo +inf.0 3)
modulo: contract violation
   expected: integer?
   given: +inf.0
```

`(add1 z) → number?
  z : number?

Returns $(+ z 1)$.

`(sub1 z) → number?
  z : number?

Returns $(- z 1)$.

`(abs x) → number?
  x : real?

Returns the absolute value of $x$.

Examples:

```lisp
> (abs 1.0)
1.0
> (abs -1)
1
```
\[(\text{max } x \ldots ) \to \text{real}?
\quad x : \text{real}\]

Returns the largest of the \(x\)s, or \(+\text{nan.}0\) if any \(x\) is \(+\text{nan.}0\). If any \(x\) is inexact, the result is coerced to inexact. See also \(\text{argmax}\).

Examples:
\[
> (\text{max } 1 3 2)
3
> (\text{max } 1 3 2.0)
3.0
\]

\[(\text{min } x \ldots ) \to \text{real}?
\quad x : \text{real}\]

Returns the smallest of the \(x\)s, or \(+\text{nan.}0\) if any \(x\) is \(+\text{nan.}0\). If any \(x\) is inexact, the result is coerced to inexact. See also \(\text{argmin}\).

Examples:
\[
> (\text{min } 1 3 2)
1
> (\text{min } 1 3 2.0)
1.0
\]

\[(\text{gcd } n \ldots ) \to \text{rational}?
\quad n : \text{rational}\]

Returns the greatest common divisor (a non-negative number) of the \(n\)s; for non-integer \(n\)s, the result is the \(\text{gcd}\) of the numerators divided by the \(\text{lcm}\) of the denominators. If no arguments are provided, the result is 0. If all arguments are zero, the result is zero.

Examples:
\[
> (\text{gcd } 10)
10
> (\text{gcd } 12 81.0)
3.0
> (\text{gcd } 1/2 1/3)
1/6
\]

\[(\text{lcm } n \ldots ) \to \text{rational}?
\quad n : \text{rational}\]
Returns the least common multiple (a non-negative number) of the ns; non-integer ns, the result is the absolute value of the product divided by the gcd. If no arguments are provided, the result is 1. If any argument is zero, the result is zero; furthermore, if any argument is exact 0, the result is exact 0.

Examples:

```lisp
> (lcm 10)
10
> (lcm 3 4.0)
12.0
> (lcm 1/2 2/3)
2
```

`(round x) → (or/c integer? +inf.0 -inf.0 +nan.0)

x : real?

Returns the integer closest to x, resolving ties in favor of an even number, but +inf.0, -inf.0, and +nan.0 round to themselves.

Examples:

```lisp
> (round 17/4)
4
> (round -17/4)
-4
> (round 2.5)
2.0
> (round -2.5)
-2.0
> (round +inf.0)
+inf.0
```

`(floor x) → (or/c integer? +inf.0 -inf.0 +nan.0)

x : real?

Returns the largest integer that is no more than x, but +inf.0, -inf.0, and +nan.0 floor to themselves.

Examples:

```lisp
> (floor 17/4)
4
> (floor -17/4)
-5
```
\[
\begin{align*}
> \text{(floor 2.5)} & \quad 2.0 \\
> \text{(floor -2.5)} & \quad -3.0 \\
> \text{(floor +inf.0)} & \quad +\text{inf.0}
\end{align*}
\]

\[
(ceiling \ x) \rightarrow (or/c \ integer? \ +\text{inf.0} \ -\text{inf.0} \ +\text{nan.0}) \\
x : \text{real?}
\]

Returns the smallest integer that is at least as large as \( x \), but \( +\text{inf.0}, -\text{inf.0}, \) and \( +\text{nan.0} \) ceiling to themselves.

Examples:
\[
\begin{align*}
> \text{(ceiling 17/4)} & \quad 5 \\
> \text{(ceiling -17/4)} & \quad -4 \\
> \text{(ceiling 2.5)} & \quad 3.0 \\
> \text{(ceiling -2.5)} & \quad -2.0 \\
> \text{(ceiling +inf.0)} & \quad +\text{inf.0}
\end{align*}
\]

\[
(truncate \ x) \rightarrow (or/c \ integer? \ +\text{inf.0} \ -\text{inf.0} \ +\text{nan.0}) \\
x : \text{real?}
\]

Returns the integer farthest from \( 0 \) that is not farther from \( 0 \) than \( x \), but \( +\text{inf.0}, -\text{inf.0}, \) and \( +\text{nan.0} \) truncate to themselves.

Examples:
\[
\begin{align*}
> \text{(truncate 17/4)} & \quad 4 \\
> \text{(truncate -17/4)} & \quad -4 \\
> \text{(truncate 2.5)} & \quad 2.0 \\
> \text{(truncate -2.5)} & \quad -2.0 \\
> \text{(truncate +inf.0)} & \quad +\text{inf.0}
\end{align*}
\]
(numerator q) → integer?
q : rational?

Coerces q to an exact number, finds the numerator of the number expressed in its simplest fractional form, and returns this number coerced to the exactness of q.

Examples:

> (numerator 5)
5
> (numerator 17/4)
17
> (numerator 2.3)
2589569785738035.0

(denominator q) → (and/c integer? positive?)
q : rational?

Coerces q to an exact number, finds the denominator of the number expressed in its simplest fractional form, and returns this number coerced to the exactness of q.

Examples:

> (denominator 5)
1
> (denominator 17/4)
4
> (denominator 2.3)
1125899906842624.0

(rationalize x tolerance) → real?
x : real?
tolerance : real?

Among the real numbers within (abs tolerance) of x, returns the one corresponding to an exact number whose denominator is the smallest. If multiple integers are within tolerance of x, the one closest to 0 is used.

Examples:

> (rationalize 1/4 1/10)
1/3
> (rationalize -1/4 1/10)
-1/3
> (rationalize 1/4 1/4)
0
> (rationalize 11/40 1/4)
1/2

Number Comparison

\[
(= z w ...) \rightarrow \text{boolean}?
\]
\[
z : \text{number}?
\]
\[
w : \text{number}?
\]

Returns #t if all of the arguments are numerically equal, #f otherwise. An inexact number is numerically equal to an exact number when the exact coercion of the inexact number is the exact number. Also, 0.0 and -0.0 are numerically equal, but +nan.0 is not numerically equal to itself.

Examples:

> (= 1 1.0)
#t
> (= 1 2)
#f
> (= 2+3i 2+3i 2+3i)
#t
> (= 1)
#t

Changed in version 7.0.0.13 of package base: Allow one argument, in addition to allowing two or more.

\[
(< x y ...) \rightarrow \text{boolean}?
\]
\[
x : \text{real}?
\]
\[
y : \text{real}?
\]

Returns #t if the arguments in the given order are strictly increasing, #f otherwise.

Examples:

> (< 1 1)
#f
> (< 1 2 3)
#t
> (< 1)
#t
> (< 1 +inf.0)
#t
> (< 1 +nan.0)
#f
Changed in version 7.0.0.13 of package base: Allow one argument, in addition to allowing two or more.

\[ (\leq x y \ldots) \rightarrow \text{boolean?} \]
\[ x : \text{real?} \]
\[ y : \text{real?} \]

Returns \#t if the arguments in the given order are non-decreasing, \#f otherwise.

Examples:
> (\leq 1 1)  
\#t  
> (\leq 1 2 1)  
\#f

Changed in version 7.0.0.13 of package base: Allow one argument, in addition to allowing two or more.

\[ (> x y \ldots+) \rightarrow \text{boolean?} \]
\[ x : \text{real?} \]
\[ y : \text{real?} \]

Returns \#t if the arguments in the given order are strictly decreasing, \#f otherwise.

Examples:
> (> 1 1)  
\#f  
> (> 3 2 1)  
\#t  
> (> +inf.0 1)  
\#t  
> (> +nan.0 1)  
\#f

Changed in version 7.0.0.13 of package base: Allow one argument, in addition to allowing two or more.

\[ (\geq x y \ldots) \rightarrow \text{boolean?} \]
\[ x : \text{real?} \]
\[ y : \text{real?} \]

Returns \#t if the arguments in the given order are non-increasing, \#f otherwise.

Examples:
> (\geq 1 1)  
\#t  
> (\geq 1 2 1)  
\#f
Changed in version 7.0.0.13 of package base: Allow one argument, in addition to allowing two or more.

**Powers and Roots**

\[(\text{sqrt } z) \rightarrow \text{number?}\]
\[z : \text{number?}\]

Returns the principal square root of \(z\). The result is exact if \(z\) is exact and \(z\)’s square root is rational. See also \texttt{integer-sqrt}.

Examples:

\[
> (\text{sqrt } 4/9) \\
2/3
\]

\[
> (\text{sqrt } 2) \\
1.4142135623730951
\]

\[
> (\text{sqrt } -1) \\
0+1i
\]

\[(\text{integer-sqrt } n) \rightarrow \text{complex?}\]
\[n : \text{integer?}\]

Returns \((\text{floor } (\text{sqrt } n))\) for positive \(n\). The result is exact if \(n\) is exact. For negative \(n\), the result is \((* (\text{integer-sqrt } (- n)) 0+1i)\).

Examples:

\[
> (\text{integer-sqrt } 4.0) \\
2.0
\]

\[
> (\text{integer-sqrt } 5) \\
2
\]

\[
> (\text{integer-sqrt } -4.0) \\
0.0+2.0i
\]

\[
> (\text{integer-sqrt } -4) \\
0+2i
\]

\[(\text{integer-sqrt/remainder } n) \rightarrow \text{complex? integer?}\]
\[n : \text{integer?}\]

Returns \((\text{integer-sqrt } n)\) and \((- n (\text{expt } (\text{integer-sqrt } n) 2))\).

Examples:

\[
> (\text{integer-sqrt/remainder } 4.0) \\
2.0
\]

\[
0.0
\]
> (integer-sqrt/remainder 5)
  2
  1

(expt z w) → number?
  z : number?
  w : number?

Returns z raised to the power of w.

If w is exact 0, the result is exact 1. If w is 0.0 or -0.0 and z is a real number other than exact 1 or 0, the result is 1.0 (even if z is +nan.0).

If z is exact 1, the result is exact 1. If z is 1.0 and w is a real number, the result is 1.0 (even if w is +nan.0).

If z is exact 0, the result is as follows:

• w is exact 0 — result is 1
• w is 0.0 or -0.0 — result is 1.0
• real part of w is negative — the exn:fail:contract:divide-by-zero exception is raised
• w is nonreal with a nonpositive real part — the exn:fail:contract:divide-by-zero exception is raised
• w is +nan.0 — result is +nan.0
• otherwise — result is 0

If w is exact 1/2, the result is the same as (sqrt z), which can be exact. Other fractional powers are not treated specially in this manner.

Examples:

> (expt 9 1/2)
  3
> (expt 9 0.5)
  3.0
> (expt 16 1/4)
  2.0
> (expt 16 0.25)
  2.0

Further special cases when w is a real number:

These special cases correspond to pow in C99 [C99], except when z is negative and w is not an integer.
• \((\text{expt } 0.0 \ w)\):
  - \(w\) is negative — result is \(+inf.0\)
  - \(w\) is positive — result is \(0.0\)

• \((\text{expt } -0.0 \ w)\):
  - \(w\) is negative:
    * \(w\) is an odd integer — result is \(-inf.0\)
    * \(w\) otherwise rational — result is \(+inf.0\)
  - \(w\) is positive:
    * \(w\) is an odd integer — result is \(-0.0\)
    * \(w\) otherwise rational — result is \(0.0\)

• \((\text{expt } z \ -inf.0)\) for positive \(z\):
  - \(z\) is less than \(1.0\) — result is \(+inf.0\)
  - \(z\) is greater than \(1.0\) — result is \(0.0\)

• \((\text{expt } z \ +inf.0)\) for positive \(z\):
  - \(z\) is less than \(1.0\) — result is \(0.0\)
  - \(z\) is greater than \(1.0\) — result is \(+inf.0\)

• \((\text{expt } -inf.0 \ w)\) for integer \(w\):
  - \(w\) is negative:
    * \(w\) is odd — result is \(-0.0\)
    * \(w\) is even — result is \(0.0\)
  - \(w\) is positive:
    * \(w\) is odd — result is \(-inf.0\)
    * \(w\) is even — result is \(+inf.0\)

• \((\text{expt } +inf.0 \ w)\):
  - \(w\) is negative — result is \(0.0\)
  - \(w\) is positive — result is \(+inf.0\)

Examples:

> (expt 2 3)
8
> (expt 4 0.5)
2.0
> (expt +inf.0 0)
1
(exp z) → number?
  z : number?

Returns Euler’s number raised to the power of z. The result is normally inexact, but it is exact 1 when z is an exact 0. See also expt.

Examples:
> (exp 1)
2.718281828459045
> (exp 2+3i)
-7.315110094901103+1.0427436562359045i
> (exp 0)
1

(log z [b]) → number?
  z : number?
  b : number? = (exp 1)

Returns the natural logarithm of z. The result is normally inexact, but it is exact 0 when z is an exact 1. When z is exact 0, exn:fail:contract:divide-by-zero exception is raised.

If b is provided, it serves as an alternative base. It is equivalent to (/ (log z) (log b)), but can potentially run faster. If b is exact 1, exn:fail:contract:divide-by-zero exception is raised.

Consider using fllogb instead when accuracy is important.

Examples:
> (log (exp 1))
1.0
> (log 2+3i)
1.2824746787307684+0.982793723247329i
> (log 0)
0
> (log 100 10)
2.0
> (log 8 2)
3.0
> (log 5 5)
1.0

Changed in version 6.9.0.1 of package base: Added second argument for arbitrary bases.
Trigonometric Functions

\( (\sin z) \rightarrow \text{number?} \)
\[ z : \text{number?} \]

Returns the sine of \( z \), where \( z \) is in radians. The result is normally inexact, but it is exact 0 if \( z \) is exact 0.

Examples:

\[
> (\sin 3.14159) \\
2.6535897933527304e-6 \\
> (\sin 1.0+5.0i) \\
62.44551846769654+40.0921657779984i
\]

\( (\cos z) \rightarrow \text{number?} \)
\[ z : \text{number?} \]

Returns the cosine of \( z \), where \( z \) is in radians.

Examples:

\[
> (\cos 3.14159) \\
-0.9999999999964793 \\
> (\cos 1.0+5.0i) \\
40.09580630629883-62.43984868079963i
\]

\( (\tan z) \rightarrow \text{number?} \)
\[ z : \text{number?} \]

Returns the tangent of \( z \), where \( z \) is in radians. The result is normally inexact, but it is exact 0 if \( z \) is exact 0.

Examples:

\[
> (\tan 0.7854) \\
1.0000036732118494 \\
> (\tan 1.0+5.0i) \\
8.2567198342296e-5+1.0000377833796008i
\]

\( (\text{asin} z) \rightarrow \text{number?} \)
\[ z : \text{number?} \]

Returns the arcsine in radians of \( z \). The result is normally inexact, but it is exact 0 if \( z \) is exact 0.

Examples:
\[
> (\text{asin} \ 0.25) \\
0.25268025514207865 \\
> (\text{asin} \ 1.0+5.0i) \\
0.1937931365549322+2.3309746530493123i
\]

\[
(\text{acos} \ z) \to \text{number} \\
\quad z: \text{number}
\]

Returns the arccosine in radians of \( z \).

Examples:

\[
> (\text{acos} \ 0.25) \\
1.318116071652818 \\
> (\text{acos} \ 1.0+5.0i) \\
1.3770031902399644-2.3309746530493123i
\]

\[
(\text{atan} \ z) \to \text{number} \\
\quad z: \text{number} \\
(\text{atan} \ y \ x) \to \text{number} \\
\quad y: \text{real} \\
\quad x: \text{real}
\]

In the one-argument case, returns the arctangent of the inexact approximation of \( z \), except that the result is an exact 0 for \( z \) as 0, and the \text{exn:fail:contract:divide-by-zero} exception is raised for \( z \) as exact 0+1i or exact 0-1i.

In the two-argument case, the result is roughly the same as \((\text{atan} \ (/ (\text{exact->inexact} \ y)) \ (\text{exact->inexact} \ x))\) but the signs of \( y \) and \( x \) determine the quadrant of the result. Moreover, a suitable angle is returned when \( y \) divided by \( x \) produces \text{+nan.0} in the case that neither \( y \) nor \( x \) is \text{+nan.0}. Finally, if \( y \) is exact 0 and \( x \) is a positive number, the result is exact 0. If both \( x \) and \( y \) are exact 0, the \text{exn:fail:contract:divide-by-zero} exception is raised.

Examples:

\[
> (\text{atan} \ 0.5) \\
0.46364760900080615 \\
> (\text{atan} \ 2 \ 1) \\
1.0714871779409040 \\
> (\text{atan} \ -2 \ -1) \\
-2.0344439357957027 \\
> (\text{atan} \ 1.0+5.0i) \\
1.530881333938778+0.19442614214700213i \\
> (\text{atan} \ +\text{inf.0} \ -\text{inf.0}) \\
2.356194490192345
\]
Complex Numbers

\[(\text{make-rectangular } x \ y) \to \text{number?}\]
\[x : \text{real?}\]
\[y : \text{real?}\]

Creates a complex number with \(x\) as the real part and \(y\) as the imaginary part. That is, returns \((+ x (* y 0+1i))\).

Example:

\[\texttt{> (make-rectangular 3 4.0)}\]
\[3.0+4.0i\]

\[(\text{make-polar } \text{magnitude} \ \text{angle}) \to \text{number?}\]
\[\text{magnitude} : \text{real?}\]
\[\text{angle} : \text{real?}\]

Creates a complex number which, if thought of as a point, is \(\text{magnitude}\) away from the origin and is rotated \(\text{angle}\) radians counter clockwise from the positive x-axis. That is, returns \((+ (* \text{magnitude} (\cos \text{angle})) (* \text{magnitude} (\sin \text{angle}) 0+1i))\).

Examples:

\[\texttt{> (make-polar 10 (* pi 1/2))}\]
\[6.123233995736766e-16+10.0i\]
\[\texttt{> (make-polar 10 (* pi 1/4))}\]
\[7.0710678118654755+7.071067811865475i\]

\[(\text{real-part } z) \to \text{real?}\]
\[z : \text{number?}\]

Returns the real part of the complex number \(z\) in rectangle coordinates.

Examples:

\[\texttt{> (real-part 3+4i)}\]
\[3\]
\[\texttt{> (real-part 5.0)}\]
\[5.0\]

\[(\text{imag-part } z) \to \text{real?}\]
\[z : \text{number?}\]
Returns the imaginary part of the complex number \( z \) in rectangle coordinates.

Examples:

\[
\begin{align*}
& \texttt{> (imag-part 3+i)} \\
& 4 \\
& \texttt{> (imag-part 5.0)} \\
& 0 \\
& \texttt{> (imag-part 5.0+0.0i)} \\
& 0.0 \\
\end{align*}
\]

\[
\begin{align*}
\text{(magnitude } z \text{)} & \to (\text{and/c real? (not/c negative?)}) \\
\text{z : number?}
\end{align*}
\]

Returns the magnitude of the complex number \( z \) in polar coordinates. A complex number with \(+\infty.0\) or \(-\infty.0\) as a component has magnitude \(+\infty.0\), even if the other component is \(+\text{nan}.0\).

Examples:

\[
\begin{align*}
& \texttt{> (magnitude -3)} \\
& 3 \\
& \texttt{> (magnitude 3.0)} \\
& 3.0 \\
& \texttt{> (magnitude 3+i)} \\
& 5 \\
\end{align*}
\]

Changed in version 7.2.0.2 of package base: Changed to always return \(+\infty.0\) for a complex number with a \(+\infty.0\) or \(-\infty.0\) component.

\[
\begin{align*}
\text{(angle } z \text{)} & \to \text{real?} \\
\text{z : number?}
\end{align*}
\]

Returns the angle of the complex number \( z \) in polar coordinates.

The result is guaranteed to be between \(- \pi\) and \(\pi\), possibly equal to \(\pi\) (but never equal to \(- \pi\)).

Examples:

\[
\begin{align*}
& \texttt{> (angle -3)} \\
& 3.141592653589793 \\
& \texttt{> (angle 3.0)} \\
& 0 \\
& \texttt{> (angle 3+i)} \\
\end{align*}
\]
0.9272952180016122
> (angle +inf.0+inf.0i)
0.7853981633974483
> (angle -1)
3.141592653589793

Bitwise Operations

(bitwise-ior n ...) → exact-integer?
| n : exact-integer?

Returns the bitwise “inclusive or” of the n's in their (semi-infinite) two's complement representation. If no arguments are provided, the result is 0.

Examples:

> (bitwise-ior 1 2)
  3
> (bitwise-ior -32 1)
  -31

(bitwise-and n ...) → exact-integer?
| n : exact-integer?

Returns the bitwise “and” of the n's in their (semi-infinite) two's complement representation. If no arguments are provided, the result is -1.

Examples:

> (bitwise-and 1 2)
  0
> (bitwise-and -32 -1)
  -32

(bitwise-xor n ...) → exact-integer?
| n : exact-integer?

Returns the bitwise “exclusive or” of the n's in their (semi-infinite) two's complement representation. If no arguments are provided, the result is 0.

Examples:

> (bitwise-xor 1 5)
  4
> (bitwise-xor -32 -1)
  31

205
(bitwise-not n) → exact-integer?
  n : exact-integer?

Returns the bitwise “not” of $n$ in its (semi-infinite) two’s complement representation.

Examples:

> (bitwise-not 5)
-6
> (bitwise-not -1)
0

(bitwise-bit-set? n m) → boolean?
  n : exact-integer?
  m : exact-nonnegative-integer?

Returns #t when the $m$th bit of $n$ is set in $n$’s (semi-infinite) two’s complement representation.

This operation is equivalent to (not (zero? (bitwise-and n (arithmetic-shift 1 m)))), but it is faster and runs in constant time when $n$ is positive.

Examples:

> (bitwise-bit-set? 5 0)
#t
> (bitwise-bit-set? 5 2)
#t
> (bitwise-bit-set? -5 (expt 2 700))
#t

(bitwise-bit-field n start end) → exact-integer?
  n : exact-integer?
  start : exact-nonnegative-integer?
  end : (and/c exact-nonnegative-integer? (=/c start))

Extracts the bits between position $start$ and (- end 1) (inclusive) from $n$ and shifts them down to the least significant portion of the number.

This operation is equivalent to the computation

(bitwise-and (sub1 (arithmetic-shift 1 (- end start)))
  (arithmetic-shift n (- start)))

206
but it runs in constant time when \( n \) is positive, \( \text{start} \) and \( \text{end} \) are fixnums, and \((- \text{end} \text{start})\) is no more than the maximum width of a fixnum.

Each pair of examples below uses the same numbers, showing the result both in binary and as integers.

Examples:

\[
\begin{align*}
&> (\text{format "~b" (bitwise-bit-field (string->number "1101" 2) 1 1))}
  
  \text{"0"}
  
  > (\text{bitwise-bit-field 13 1 1})
  
  0
  
  > (\text{format "~b" (bitwise-bit-field (string->number "1101" 2) 1 3))}
  
  \text{"10"}
  
  > (\text{bitwise-bit-field 13 1 3})
  
  2
  
  > (\text{format "~b" (bitwise-bit-field (string->number "1101" 2) 1 4))}
  
  \text{"110"}
  
  > (\text{bitwise-bit-field 13 1 4})
  
  6
\end{align*}
\]

\[(\text{arithmetic-shift } n \ m) \rightarrow \text{exact-integer?}\]
\[
\begin{align*}
  n &: \text{exact-integer} \\
  m &: \text{exact-integer}
\end{align*}
\]

Returns the bitwise “shift” of \( n \) in its (semi-infinite) two’s complement representation. If \( m \) is non-negative, the integer \( n \) is shifted left by \( m \) bits; i.e., \( m \) new zeros are introduced as rightmost digits. If \( m \) is negative, \( n \) is shifted right by \((- m)\) bits; i.e., the rightmost \( m \) digits are dropped.

Examples:

\[
\begin{align*}
&> (\text{arithmetic-shift 1 10})
  
  1024
  
  > (\text{arithmetic-shift 255 -3})
  
  31
\end{align*}
\]

\[(\text{integer-length } n) \rightarrow \text{exact-integer?}\]
\[
\begin{align*}
  n &: \text{exact-integer}
\end{align*}
\]

Returns the number of bits in the (semi-infinite) two’s complement representation of \( n \) after removing all leading zeros (for non-negative \( n \)) or ones (for negative \( n \)).

Examples:
Random Numbers

\[ (\text{random } k \ [\text{rand-gen}]) \rightarrow \text{exact-nonnegative-integer}? \]
\[
\begin{align*}
k & : (\text{integer-in} \ 1 \ 4294967087) \\
\text{rand-gen} & : \text{pseudo-random-generator}? \\
& = (\text{current-pseudo-random-generator})
\end{align*}
\]

\[ (\text{random} \ \text{min} \ \text{max} \ [\text{rand-gen}]) \rightarrow \text{exact-nonnegative-integer}? \]
\[
\begin{align*}
\text{min} & : \text{exact-integer}? \\
\text{max} & : (\text{integer-in} \ (+ \ 1 \ \text{min}) \ (+ \ 4294967087 \ \text{min})) \\
\text{rand-gen} & : \text{pseudo-random-generator}? \\
& = (\text{current-pseudo-random-generator})
\end{align*}
\]

\[ (\text{random} \ [\text{rand-gen}]) \rightarrow (\text{and/c} \ \text{real?} \ \text{inexact?} \ (>/\text{c} \ 0) \ (</\text{c} \ 1)) \]
\[
\begin{align*}
\text{rand-gen} & : \text{pseudo-random-generator}? \\
& = (\text{current-pseudo-random-generator})
\end{align*}
\]

When called with an integer argument \( k \), returns a random exact integer in the range 0 to \( k-1 \).

When called with two integer arguments \( \text{min} \) and \( \text{max} \), returns a random exact integer in the range \( \text{min} \) to \( \text{max} \)-1.

When called with zero arguments, returns a random inexact number between 0 and 1, exclusive.

In each case, the number is provided by the given pseudo-random number generator (which defaults to the current one, as produced by \text{current-pseudo-random-generator}). The generator maintains an internal state for generating numbers. The random number generator uses L’Ecuyer’s MRG32k3a algorithm [L’Ecuyer02] that has a state space of practically 192 bits.

Changed in version 6.4 of package base: Added support for ranges.

\[ (\text{random-seed} \ k) \rightarrow \text{void}? \]
\[
\begin{align*}
k & : (\text{integer-in} \ 0 \ (\text{sub1} \ (\text{expt} \ 2 \ 31)))
\end{align*}
\]

Seeds the current pseudo-random number generator with \( k \). Seeding a generator sets its internal state deterministically; that is, seeding a generator with a particular number forces it to produce a sequence of pseudo-random numbers that is the same across runs and across platforms.

The \text{random-seed} function is convenient for some purposes, but note that the space of states for a pseudo-random number generator is much larger that the space of allowed values for
Use `vector->pseudo-random-generator!` to set a pseudo-random number generator to any of its possible states.

```scheme
(make-pseudo-random-generator) \rightarrow pseudo-random-generator?
```

Returns a new pseudo-random number generator. The new generator is seeded with a number derived from `(current-milliseconds)`.

```scheme
(pseudo-random-generator? v) \rightarrow boolean?
  v : any/c
```

Returns `#t` if `v` is a pseudo-random number generator, `#f` otherwise.

```scheme
(current-pseudo-random-generator) \rightarrow pseudo-random-generator?
(current-pseudo-random-generator rand-gen) \rightarrow void?
  rand-gen : pseudo-random-generator?
```

A parameter that determines the pseudo-random number generator used by `random`.

```scheme
(pseudo-random-generator->vector rand-gen) \rightarrow pseudo-random-generator-vector?
  rand-gen : pseudo-random-generator?
```

Produces a vector that represents the complete internal state of `rand-gen`. The vector is suitable as an argument to `vector->pseudo-random-generator` to recreate the generator in its current state (across runs and across platforms).

```scheme
(vector->pseudo-random-generator vec) \rightarrow pseudo-random-generator?
  vec : pseudo-random-generator-vector?
```

Produces a pseudo-random number generator whose internal state corresponds to `vec`.

```scheme
(vector->pseudo-random-generator! rand-gen vec) \rightarrow void?
  rand-gen : pseudo-random-generator?
  vec : pseudo-random-generator-vector?
```

Like `vector->pseudo-random-generator`, but changes `rand-gen` to the given state, instead of creating a new generator.

```scheme
(pseudo-random-generator-vector? v) \rightarrow boolean?
  v : any/c
```
Returns \#t if \( v \) is a vector of six exact integers, where the first three integers are in the range 0 to 4294967086, inclusive; the last three integers are in the range 0 to 4294944442, inclusive; at least one of the first three integers is non-zero; and at least one of the last three integers is non-zero. Otherwise, the result is \#f.

Other Randomness Utilities

\[
\text{(require racket/random)} \quad \text{package: base}
\]

\[
\text{(crypto-random-bytes } n) \rightarrow \text{bytes?}
\]
\[n : \text{exact-positive-integer?}\]

Provides an interface to randomness from the underlying operating system. Use \text{crypto-random-bytes} instead of \text{random} wherever security is a concern.

Returns \( n \) random bytes. On Unix systems, the bytes are obtained from "/dev/urandom", while Windows uses the \text{RtlGenRand} system function.

Example:

\[
\text{> (crypto-random-bytes 14)}
\]
\"\0\1\1\2\3\5\b\r\25"7Y\220\351"

Added in version 6.3 of package \text{base}.

\[
\text{(random-ref seq [rand-gen]) \rightarrow any/c}
\]
\[seq : \text{sequence?}
\]
\[rand-gen : \text{pseudo-random-generator} = (\text{current-pseudo-random-generator})\]

Returns a random element of the sequence. Like \text{sequence-length}, does not terminate on infinite sequences, and evaluates the entire sequence.

Added in version 6.4 of package \text{base}.

\[
\text{(random-sample seq } n \text{ [rand-gen] [replacement?]}) \rightarrow \text{(listof any/c)}
\]
\[seq : \text{sequence?}
\]
\[n : \text{exact-positive-integer?}
\]
\[rand-gen : \text{pseudo-random-generator} = (\text{current-pseudo-random-generator})\]
\[replacement? : \text{any/c }= \#t\]

Returns a list of \( n \) elements of \( \text{seq} \), picked at random, listed in any order. If \text{replacement?} is non-false, elements are drawn with replacement, which allows for duplicates.
Like `sequence-length`, does not terminate on infinite sequences, and evaluates the entire sequence.

Added in version 6.4 of package `base`.

**Number–String Conversions**

\[
\text{number→string } z \ [\text{radix}] \rightarrow \text{string?}
\]

\[
z : \text{number?}
\]

\[
\text{radix : (or/c 2 8 10 16) } = 10
\]

Returns a string that is the printed form of \( z \) (see §1.4.2 “Printing Numbers”) in the base specified by \( \text{radix} \). If \( z \) is inexact, \( \text{radix} \) must be \( 10 \), otherwise the `exn:fail:contract` exception is raised.

Examples:

\[
> \text{(number→string 3.0)} \\
"3.0"
\]

\[
> \text{(number→string 255 8)} \\
"377"
\]

\[
\text{string→number } s \\
\text{[radix} \\
\text{convert-mode} \\
\text{decimal-mode} \\
\text{single-mode}]\]

\[
\rightarrow \ (\text{or/c number? #f string? extflonum?})
\]

\[
s : \text{string?}
\]

\[
\text{radix} : \ (\text{integer-in 2 16}) = 10
\]

\[
\text{convert-mode} : \ (\text{or/c 'number-or-false 'read}) \\
\text{decimal-mode} : \ (\text{or/c 'decimal-as-inexact 'decimal-as-exact}) \\
\text{single-mode} : \ (\text{or/c 'single 'double})
\]

Reads and returns a number datum from \( s \) (see §1.3.3 “Reading Numbers”). The optional \( \text{radix} \) argument specifies the default base for the number, which can be overridden by \#b, \#o, \#d, or \#x in the string.

If \( \text{convert-mode} \) is \'number-or-false, the result is \#f if \( s \) does not parse exactly as a number datum (with no whitespace). If \( \text{convert-mode} \) is \'read, the result can be an
extflonum, and it can be a string that contains an error message if read of s would report a reader exception (but the result can still be #f if read would report a symbol).

The decimal-mode argument controls number parsing the same way that the read-decimal-as-inexact parameter affects read.

The single-mode argument controls number parsing the same way that the read-single-flonum parameter affects read.

Examples:

> (string->number "3.0+2.5i")
3.0+2.5i
> (string->number "hello")
#f
> (string->number "111" 7)
57
> (string->number "#b111" 7)
7
> (string->number "#e+inf.0" 10 'read)
"no exact representation for +inf.0"
> (string->number "10.3" 10 'read 'decimal-as-exact)
103/10

Changed in version 6.8.0.2 of package base: Added the convert-mode and decimal-mode arguments.
Changed in version 7.3.0.5: Added the single-mode argument.

(real->decimal-string n [decimal-digits]) → string?
  n : rational?
  decimal-digits : exact-nonnegative-integer? = 2

Prints n into a string and returns the string. The printed form of n shows exactly decimal-digits digits after the decimal point. The printed form uses a minus sign if n is negative, and it does not use a plus sign if n is positive.

Before printing, n is converted to an exact number, multiplied by (expt 10 decimal-digits), rounded, and then divided again by (expt 10 decimal-digits). The result of this process is an exact number whose decimal representation has no more than decimal-digits digits after the decimal (and it is padded with trailing zeros if necessary).

If n is a real number with no decimal representation (e.g. +nan.0, +inf.0), then the exn:fail:contract exception is raised. (Any real number that is convertible to decimal notation is rational, so n must be rational?, despite the name of the function.)

Examples:

> (real->decimal-string pi)
"3.14"
> (real->decimal-string pi 5)
"3.14159"

\[
\text{(integer-bytes->integer \ bstr \ signed? \ [\text{big-endian?} \ start \ end])} \rightarrow \text{exact-integer?}
\]

\[
bstr : \text{bytes?} \\
\text{signed?} : \text{any/c} \\
\text{big-endian?} : \text{any/c} = (\text{system-big-endian?}) \\
\text{start} : \text{exact-nonnegative-integer?} = 0 \\
\text{end} : \text{exact-nonnegative-integer?} = (\text{bytes-length}\ bstr)
\]

Converts the machine-format number encoded in \text{bstr} to an exact integer. The \text{start} and \text{end} arguments specify the substring to decode, where \((- \text{end start})\) must be 1, 2, 4, or 8. If \text{signed?} is true, then the bytes are decoded as a two's-complement number, otherwise it is decoded as an unsigned integer. If \text{big-endian?} is true, then the first byte's value provides the most significant eight bits of the number, otherwise the first byte provides the least-significant eight bits, and so on.

Changed in version 6.10.0.1 of package \text{base}: Added support for decoding a 1-byte string.

\[
\text{(integer->integer-bytes \ n \ size-n \ signed? \ [\text{big-endian?} \ dest-bstr \ start])} \rightarrow \text{bytes?}
\]

\[
n : \text{exact-integer?} \\
\text{size-n} : (\text{or/c} 1 2 4 8) \\
\text{signed?} : \text{any/c} \\
\text{big-endian?} : \text{any/c} = (\text{system-big-endian?}) \\
\text{dest-bstr} : (\text{and/c} \ \text{bytes?} \ (\text{not/c} \ \text{immutable?})) \\
\quad = (\text{make-bytes} \ \text{size-n}) \\
\text{start} : \text{exact-nonnegative-integer?} = 0
\]

Converts the exact integer \text{n} to a machine-format number encoded in a byte string of length \text{size-n}, which must be 1, 2, 4, or 8. If \text{signed?} is true, then the number is encoded as two's complement, otherwise it is encoded as an unsigned bit stream. If \text{big-endian?} is true, then the most significant eight bits of the number are encoded in the first byte of the resulting byte string, otherwise the least-significant bits are encoded in the first byte, and so on.
The `dest-bstr` argument must be a mutable byte string of length `size-n`. The encoding of `n` is written into `dest-bstr` starting at offset `start`, and `dest-bstr` is returned as the result.

If `n` cannot be encoded in a byte string of the requested size and format, the `exn:fail:contract` exception is raised. If `dest-bstr` is not of length `size-n`, the `exn:fail:contract` exception is raised.

changed in version 6.10.0.1 of package base: Added support for encoding a 1-byte value.

```
(floating-point-bytes->real bstr
 [big-endian? start end]) -> flonum?
```

Converts the IEEE floating-point number encoded in `bstr` from position `start` (inclusive) to `end` (exclusive) to an inexact real number. The difference between `start` and `end` must be either 4 or 8 bytes. If `big-endian?` is true, then the first byte’s ASCII value provides the most significant eight bits of the IEEE representation, otherwise the first byte provides the least-significant eight bits, and so on.

```
(real->floating-point-bytes x size-n
 [big-endian? dest-bstr start]) -> bytes?
```

Converts the real number `x` to its IEEE representation in a byte string of length `size-n`, which must be 4 or 8. If `big-endian?` is true, then the most significant eight bits of the number are encoded in the first byte of the resulting byte string, otherwise the least-significant bits are encoded in the first character, and so on.

The `dest-bstr` argument must be a mutable byte string of length `size-n`. The encoding of `n` is written into `dest-bstr` starting with byte `start`, and `dest-bstr` is returned as the result.
If `dest-bstr` is provided and it has less than `start` plus `size-n` bytes, the `exn:fail:contract` exception is raised.

\[(system-big-endian?) \rightarrow boolean?\]

Returns `#t` if the native encoding of numbers is big-endian for the machine running Racket, `#f` if the native encoding is little-endian.

**Extra Constants and Functions**

\{(require racket/math)\} package:b

The bindings documented in this section are provided by the `racket/math` and `racket` libraries, but not `racket/base`.

\[\text{pi} : flonum?\]

An approximation of π, the ratio of a circle’s circumference to its diameter.

Examples:

\[\text{> pi}\]
\[3.141592653589793\]
\[\text{> (cos pi)}\]
\[-1.0\]

\[\text{pi.f} : (or/c single-flonum? flonum?)\]

The same value as `pi`, but as a single-precision floating-point number if the current platform supports it.

Changed in version 7.3.0.5 of package base: Allow value to be a double-precision flonum.

\[(\text{degrees->radians } x) \rightarrow real?\]

\[x : real?\]

Converts an `x`-degree angle to radians.

Examples:

\[\text{> (degrees->radians 180)}\]
\[3.141592653589793\]
\[\text{> (sin (degrees->radians 45))}\]
\[0.7071067811865475\]
\[(\text{radians->degrees } x) \rightarrow \text{real?}\]
\[x : \text{real?}\]

Converts \(x\) radians to degrees.

Examples:

\[
\begin{align*}
> (\text{radians->degrees } \pi) \\
180.0 \\
> (\text{radians->degrees } (* 1/4 \pi)) \\
45.0
\end{align*}
\]

\[(\text{sqr } z) \rightarrow \text{number?}\]
\[z : \text{number?}\]

Returns \((z \cdot z)\).

Examples:

\[
\begin{align*}
> (\text{sqr } 10) \\
1 \\
> (\text{sqr } -10.0) \\
-1.0 \\
> (\text{sqr } 0) \\
0 \\
> (\text{sqr } -0.0) \\
-0.0 \\
> (\text{sqr } 0.0) \\
0.0 \\
> (\text{sqr } +\text{nan}.0) \\
+\text{nan}.0 \\
> (\text{sqr } +\text{inf}.0) \\
1.0 \\
> (\text{sqr } -\text{inf}.0) \\
-1.0
\end{align*}
\]

\[(\text{conjugate } z) \rightarrow \text{number?}\]
\[z : \text{number?}\]
Returns the complex conjugate of \( z \).

Examples:

\[
\begin{align*}
> \text{(conjugate 1)} \\
1 \\
> \text{(conjugate 3+4i)} \\
3-4i
\end{align*}
\]

\((\sinh z) \to \text{number?} \)
\[ z : \text{number?} \]

Returns the hyperbolic sine of \( z \).

\((\cosh z) \to \text{number?} \)
\[ z : \text{number?} \]

Returns the hyperbolic cosine of \( z \).

\((\tanh z) \to \text{number?} \)
\[ z : \text{number?} \]

Returns the hyperbolic tangent of \( z \).

\((\text{exact-round } x) \to \text{exact-integer?} \)
\[ x : \text{rational?} \]

Equivalent to \((\text{inexact->exact } (\text{round } x))\).

\((\text{exact-floor } x) \to \text{exact-integer?} \)
\[ x : \text{rational?} \]

Equivalent to \((\text{inexact->exact } (\text{floor } x))\).

\((\text{exact-ceiling } x) \to \text{exact-integer?} \)
\[ x : \text{rational?} \]

Equivalent to \((\text{inexact->exact } (\text{ceiling } x))\).

\((\text{exact-truncate } x) \to \text{exact-integer?} \)
\[ x : \text{rational?} \]

Equivalent to \((\text{inexact->exact } (\text{truncate } x))\).

217
(order-of-magnitude r) → (and/c exact? integer?)
r : (and/c real? positive?)

Computes the greatest exact integer m such that:

(<= (expt 10 m) (inexact->exact r))

Hence also:

(< (inexact->exact r) (expt 10 (add1 m)))

Examples:

> (order-of-magnitude 999)
2
> (order-of-magnitude 1000)
3
> (order-of-magnitude 1/100)
-2
> (order-of-magnitude 1/101)
-3

(nan? x) → boolean?
x : real?

Returns #t if x is eqv? to +nan.0 or +nan.f; otherwise #f.

(infinite? x) → boolean?
x : real?

Returns #t if x is +inf.0, -inf.0, +inf.f, -inf.f; otherwise #f.

(positive-integer? x) → boolean?
x : any/c

Like exact-positive-integer?, but also returns #t for positive inexact? integers.

Added in version 6.8.0.2 of package base.

(negative-integer? x) → boolean?
x : any/c
The same as \((\text{and} (\text{integer}\ x) (\text{negative}\ x))\).

Added in version 6.8.0.2 of package base.

\((\text{nonpositive-integer}\ x) \rightarrow \text{boolean}\)
\(x : \text{any/c}\)

The same as \((\text{and} (\text{integer}\ x) (\text{not} (\text{positive}\ x)))\).

Added in version 6.8.0.2 of package base.

\((\text{nonnegative-integer}\ x) \rightarrow \text{boolean}\)
\(x : \text{any/c}\)

Like \text{exact-nonnegative-integer?}, but also returns \#t for non-negative \text{inexact?} integers.

Added in version 6.8.0.2 of package base.

\((\text{natural}\ x) \rightarrow \text{boolean}\)
\(x : \text{any/c}\)

An alias for \text{exact-nonnegative-integer?}.

Added in version 6.8.0.2 of package base.

### 4.3.3 Flonums

\(\text{(require racket/flonum)}\)

The \text{racket/flonum} library provides operations like \text{fl+} that consume and produce only flonums. Flonum-specific operations can provide better performance when used consistently, and they are as safe as generic operations like +.

#### Flonum Arithmetic

\((\text{fl+} \ a \ldots) \rightarrow \text{flonum}\)
\(a : \text{flonum}\)

\((\text{fl-} \ a \ b \ldots) \rightarrow \text{flonum}\)
\(a : \text{flonum}\)
\(b : \text{flonum}\)

\((\text{fl*} \ a \ldots) \rightarrow \text{flonum}\)
\(a : \text{flonum}\)

\((\text{fl/} \ a \ b \ldots) \rightarrow \text{flonum}\)
\(a : \text{flonum}\)
\(b : \text{flonum}\)

See also §19.8 "Fixnum and Flonum Optimizations" in \textit{The Racket Guide}.
Like +, -, *, /, and abs, but constrained to consume flonums. The result is always a flonum.

Like =, <, >, <=, >=, min, and max, but constrained to consume flonums.

Like round, floor, ceiling, and truncate, but constrained to consume flonums.
Returns a value like \(a\), but potentially discards precision and range so that the result can be represented as a single-precision IEEE floating-point number (even if single-flonums are not supported).

Using \texttt{flsingle} on the arguments and results of \texttt{fl+}, \texttt{fl-}, \texttt{fl*}, \texttt{fl/}, and \texttt{flsqrt}—that is, performing double-precision operations on values representable in single precision and then rounding the result to single precision—is always the same as performing the corresponding single-precision operation [Roux14]. (For other operations, the IEEE floating-point specification does not make enough guarantees to say more about the interaction with \texttt{flsingle}.)

Added in version 7.8.0.7 of package \texttt{base}.

```
(flsin a) → flonum?
  a : flonum?
(flcos a) → flonum?
  a : flonum?
(fltan a) → flonum?
  a : flonum?
(flasin a) → flonum?
  a : flonum?
(flacos a) → flonum?
  a : flonum?
(flatan a) → flonum?
  a : flonum?
(fllog a) → flonum?
  a : flonum?
(flexp a) → flonum?
  a : flonum?
(flsqrt a) → flonum?
  a : flonum?
```

Like \texttt{sin}, \texttt{cos}, \texttt{tan}, \texttt{asin}, \texttt{acos}, \texttt{atan}, \texttt{log}, \texttt{exp}, and \texttt{sqrt}, but constrained to consume and produce flonums. The result is \texttt{+nan.0} when a number outside the range \(-1.0\) to \(1.0\) is given to \texttt{flasin} or \texttt{flacos}, or when a negative number is given to \texttt{fllog} or \texttt{flsqrt}.

```
(flexpt a b) → flonum?
  a : flonum?
  b : flonum?
```

Like \texttt{expt}, but constrained to consume and produce flonums.

Due to the result constraint, the results compared to \texttt{expt} differ in the following cases:

- \((\texttt{flexpt } -1.0 \ +\text{inf}.0) \rightarrow 1.0\)
- \((\texttt{flexpt } a \ +\text{inf}.0)\) where \(a\) is negative — \((\texttt{expt } (\texttt{abs } a) \ +\text{inf}.0)\)
• $(\text{flexpt } a \ -\infty.0)$ where $a$ is negative — $(\text{expt} \ (\text{abs } a) \ -\infty.0)$

• $(\text{expt} \ -\infty.0 \ b)$ where $b$ is a non-integer:
  - $b$ is negative — $0.0$
  - $b$ is positive — $+\infty.0$

• $(\text{flexpt } a \ b)$ where $a$ is negative and $b$ is not an integer — $+\text{nan}.0$

<table>
<thead>
<tr>
<th>(&lt;-fl a) → flonum?</th>
<th>a : exact-integer?</th>
</tr>
</thead>
</table>

Like $\text{exact->inexact}$, but constrained to consume exact integers, so the result is always a flonum.

<table>
<thead>
<tr>
<th>(fl-&gt;exact-integer a) → exact-integer?</th>
<th>a : flonum?</th>
</tr>
</thead>
</table>

Like $\text{inexact->exact}$, but constrained to consume an integer flonum, so the result is always an exact integer.

<table>
<thead>
<tr>
<th>(make-flrectangular a b)</th>
<th>→ (and/c complex?</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>(lambda (c) (flonum? (real-part c)))</td>
</tr>
<tr>
<td></td>
<td>(lambda (c) (flonum? (imag-part c))))</td>
</tr>
<tr>
<td>a : flonum?</td>
<td>b : flonum?</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>(flreal-part a) → flonum?</th>
<th>a : (and/c complex?</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>(lambda (c) (flonum? (real-part c)))</td>
</tr>
<tr>
<td></td>
<td>(lambda (c) (flonum? (imag-part c))))</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>(flimag-part a) → flonum?</th>
<th>a : (and/c complex?</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>(lambda (c) (flonum? (real-part c)))</td>
</tr>
<tr>
<td></td>
<td>(lambda (c) (flonum? (imag-part c))))</td>
</tr>
</tbody>
</table>

Like $\text{make-rectangular, real-part, and imag-part}$, but both parts of the complex number must be inexact.

| (flrandom rand-gen) → (and flonum? (>c 0) (<c 1)) |
|---------------------------------|-------------------|
| rand-gen : pseudo-random-generator? |

Equivalent to $(\text{random } rand-gen)$.

**Flonum Vectors**
A flvector is like a vector, but it holds only inexact real numbers. This representation can be more compact, and unsafe operations on flvectors (see racket/unsafe/ops) can execute more efficiently than unsafe operations on vectors of inexact reals.

An f64vector as provided by ffi/vector stores the same kinds of values as a flvector, but with extra indirections that make f64vectors more convenient for working with foreign libraries. The lack of indirections makes unsafe flvector access more efficient.

Two flvectors are equal? if they have the same length, and if the values in corresponding slots of the flvectors are equal?.

A printed flvector starts with #fl, optionally with a number between the #fl and (, See §1.3.10 "Reading Vectors" for information on reading flvectors and §1.4.7 "Printing Vectors" for information on printing flvectors.

(flvector? v) → boolean?
  v : any/c

Returns #t if v is a flvector, #f otherwise.

(flvector x ...) → flvector?
  x : flonum?

Creates a flvector containing the given inexact real numbers.

Example:

> (flvector 2.0 3.0 4.0 5.0)
(flvector 2.0 3.0 4.0 5.0)

(make-flvector size [x]) → flvector?
  size : exact-nonnegative-integer?
  x : flonum? = 0.0

 Creates a flvector with size elements, where every slot in the flvector is filled with x.

Example:

> (make-flvector 4 3.0)
(flvector 3.0 3.0 3.0 3.0)

(flvector-length vec) → exact-nonnegative-integer?
  vec : flvector?

Returns the length of vec (i.e., the number of slots in the flvector).
### (flvector-ref vec pos) → flonum?
vec : flvector?
pos : exact-nonnegative-integer?

Returns the inexact real number in slot pos of vec. The first slot is position 0, and the last slot is one less than (flvector-length vec).

### (flvector-set! vec pos x) → flonum?
vec : flvector?
pos : exact-nonnegative-integer?
x : flonum?

Sets the inexact real number in slot pos of vec. The first slot is position 0, and the last slot is one less than (flvector-length vec).

### (flvector-copy vec [start end]) → flvector?
vec : flvector?
start : exact-nonnegative-integer? = 0
end : exact-nonnegative-integer? = (vector-length v)

Creates a fresh flvector of size (- end start), with all of the elements of vec from start (inclusive) to end (exclusive).

### (in-flvector vec [start stop step]) → sequence?
vec : flvector?
start : exact-nonnegative-integer? = 0
stop : (or/c exact-integer? #f) = #f
step : (and/c exact-integer? (not/c zero?)) = 1

Returns a sequence equivalent to vec when no optional arguments are supplied.

The optional arguments start, stop, and step are as in in-vector.

A in-flvector application can provide better performance for flvector iteration when it appears directly in a for clause.

```scheme
(for/flvector maybe-length (for-clause ...) body ...)
(for*/flvector maybe-length (for-clause ...) body ...)
```

```
maybe-length =
  #:length length-expr
  #:length length-expr #:fill fill-expr
```

length-expr : exact-nonnegative-integer?
fill-expr : flonum?
Like for/vector or for*/vector, but for flvectors. The default fill-expr produces 0.0.

\[
\begin{align*}
\text{(shared-flvector } x\ldots) & \rightarrow \text{ flvector} \\
\text{ } x : \text{ flonum?}
\end{align*}
\]

Creates a flvector containing the given inexact real numbers. For communication among places, the new flvector is allocated in the shared memory space.

Example:

\[
\begin{align*}
\text{> (shared-flvector } 2.0\ 3.0\ 4.0\ 5.0) \\
\text{(flvector 2.0 3.0 4.0 5.0)}
\end{align*}
\]

\[
\begin{align*}
\text{(make-shared-flvector } \text{ size \[x\]} \rightarrow \text{ flvector} \\
\text{ size : exact-nonnegative-integer?} \\
\text{ x : flonum? = 0.0}
\end{align*}
\]

Creates a flvector with size elements, where every slot in the flvector is filled with x. For communication among places, the new flvector is allocated in the shared memory space.

Example:

\[
\begin{align*}
\text{> (make-shared-flvector } 4\ 3.0) \\
\text{(flvector 3.0 3.0 3.0 3.0)}
\end{align*}
\]

### 4.3.4 Fixnums

\[
\begin{align*}
\text{(require racket/fixnum)} \quad \text{package: base}
\end{align*}
\]

The racket/fixnum library provides operations like fx+ that consume and produce only fixnums. The operations in this library are meant to be safe versions of unsafe operations like unsafe-fx+. These safe operations are generally no faster than using generic primitives like +.

The expected use of the racket/fixnum library is for code where the require of racket/fixnum is replaced with

\[
\begin{align*}
\text{(require (filtered-in} \\
\quad \text{\(\lambda\) (name) \\
\quad \text{(and (regexp-match #rx"unsafe-fx" name)} \\
\quad \text{(regexp-replace #rx"unsafe-" name ")")}) \\
\quad \text{racket/unsafe/ops}))}
\end{align*}
\]
to drop in unsafe versions of the library. Alternately, when encountering crashes with code
that uses unsafe fixnum operations, use the racket/fixnum library to help debug the prob-
lems.

**Fixnum Arithmetic**

\[
\begin{align*}
(fx+ \ a \ldots) & \rightarrow \text{fixnum?} \\
 & a : \text{fixnum?} \\
(fx- \ a \ b \ldots) & \rightarrow \text{fixnum?} \\
 & a : \text{fixnum?} \\
 & b : \text{fixnum?} \\
(fx* \ a \ldots) & \rightarrow \text{fixnum?} \\
 & a : \text{fixnum?} \\
(fx\text{quotient} \ a \ b) & \rightarrow \text{fixnum?} \\
 & a : \text{fixnum?} \\
 & b : \text{fixnum?} \\
(fx\text{remainder} \ a \ b) & \rightarrow \text{fixnum?} \\
 & a : \text{fixnum?} \\
 & b : \text{fixnum?} \\
(fx\text{modulo} \ a \ b) & \rightarrow \text{fixnum?} \\
 & a : \text{fixnum?} \\
 & b : \text{fixnum?} \\
(fx\text{abs} \ a) & \rightarrow \text{fixnum?} \\
 & a : \text{fixnum?}
\end{align*}
\]

Safe versions of unsafe-fx+, unsafe-fx-, unsafe-fx*, unsafe-fxquotient, unsafe-
fxremainder, unsafe-fxmodulo, and unsafe-fxabs. The exn:fail:contract:non-
fixnum-result exception is raised if the arithmetic result would not be a fixnum.

Changed in version 7.0.0.13 of package base: Allow zero or more arguments for fx+ and fx* and one or more
arguments for fx-.

\[
\begin{align*}
(fx\text{and} \ a \ldots) & \rightarrow \text{fixnum?} \\
 & a : \text{fixnum?} \\
(fx\text{ior} \ a \ldots) & \rightarrow \text{fixnum?} \\
 & a : \text{fixnum?} \\
(fx\text{xor} \ a \ldots) & \rightarrow \text{fixnum?} \\
 & a : \text{fixnum?} \\
(fx\text{not} \ a) & \rightarrow \text{fixnum?} \\
 & a : \text{fixnum?} \\
(fx\text{lshift} \ a \ b) & \rightarrow \text{fixnum?} \\
 & a : \text{fixnum?} \\
 & b : \text{fixnum?} \\
(fx\text{rshift} \ a \ b) & \rightarrow \text{fixnum?} \\
 & a : \text{fixnum?} \\
 & b : \text{fixnum?}
\end{align*}
\]
Like bitwise-and, bitwise-ior, bitwise-xor, bitwise-not, and arithmetic-shift, but constrained to consume fixnums; the result is always a fixnum. The unsafe-fxlshift and unsafe-fxrshift operations correspond to arithmetic-shift, but require non-negative arguments; unsafe-fxlshift is a positive (i.e., left) shift, and unsafe-fxrshift is a negative (i.e., right) shift, where the number of bits to shift must be no more than the number of bits used to represent a fixnum. The exn:fail:contract:non-fixnum-result exception is raised if the arithmetic result would not be a fixnum.

Changed in version 7.0.0.13 of package base: Allow any number of arguments for fxand, fxior, and fxxor.

```
(fx+/wraparound a b) → fixnum?
a : fixnum?
b : fixnum?
(fx-/wraparound a b) → fixnum?
a : fixnum?
b : fixnum?
(fx*/wraparound a b) → fixnum?
a : fixnum?
b : fixnum?
(fxlshift/wraparound a b) → fixnum?
a : fixnum?
b : fixnum?
```

Like fx+, fx-, fx*, and fxlshift, but a fixnum result is produced for any allowed arguments (i.e., for any fixnum argument, except that the second fxlshift/wraparound argument must be between 0 and the number of bits in a fixnum, inclusive). The result is produced by simply discarding bits that do not fit in a fixnum representation. The result is negative if the highest of the retained bits is set—even, for example, if the value was produced by adding two positive fixnums.

Added in version 7.9.0.6 of package base.

```
(fx= a b ...) → boolean?
a : fixnum?
b : fixnum?
(fx< a b ...) → boolean?
a : fixnum?
b : fixnum?
(fx> a b ...) → boolean?
a : fixnum?
b : fixnum?
(fx<= a b ...) → boolean?
a : fixnum?
b : fixnum?
(fx>= a b ...) → boolean?
a : fixnum?
b : fixnum?
```
(fxmin a b ...) → fixnum?
  a : fixnum?
  b : fixnum?
(fxmax a b ...) → fixnum?
  a : fixnum?
  b : fixnum?

Like =, <, >, ≤, ≥, min, and max, but constrained to consume fixnums.

Changed in version 7.0.0.13 of package base: Allow one argument, in addition to allowing two or more.

(fx->fl a) → flonum?
  a : fixnum?
(fl->fx fl) → fixnum?
  fl : flonum?

Conversion between fixnums and flonums with truncation in the case of converting a flonum to a fixnum.

The fx->fl function is the same as exact->inexact or ->fl constrained to a fixnum argument.

The fl->fx function is the same as truncate followed by inexact->exact or fl->exact-integer constrained to returning a fixnum. If the truncated flonum does not fit into a fixnum, the exn:fail:contract exception is raised.

Changed in version 7.7.0.8 of package base: Changed fl->fx to truncate.

(fixnum-for-every-system? v) → boolean?
  v : any/c

Returns #t if v is a fixnum and is represented by fixnum by every Racket implementation, #f otherwise.

Added in version 7.3.0.11 of package base.

**Fixnum Vectors**

A fxvector is like a vector, but it holds only fixnums. The only advantage of a fxvector over a vector is that a shared version can be created with functions like shared-fxvector.

Two fxvectors are equal? if they have the same length, and if the values in corresponding slots of the fxvectors are equal?.

A printed fxvector starts with #fx, optionally with a number between the #fx and (. See §1.3.10 “Reading Vectors” for information on reading fxvectors and §1.4.7 “Printing Vectors” for information on printing fxvectors.
(fxvector? v) → boolean?
    v : any/c

Returns #t if v is a fxvector, #f otherwise.

(fxvector x ...) → fxvector?
    x : fixnum?

Creates a fxvector containing the given fixnums.

Example:

> (fxvector 2 3 4 5)
(fxvector 2 3 4 5)

(make-fxvector size [x]) → fxvector?
    size : exact-nonnegative-integer?
    x : fixnum? = 0

Creates a fxvector with size elements, where every slot in the fxvector is filled with x.

Example:

> (make-fxvector 4 3)
(fxvector 3 3 3 3)

(fxvector-length vec) → exact-nonnegative-integer?
    vec : fxvector?

Returns the length of vec (i.e., the number of slots in the fxvector).

(fxvector-ref vec pos) → fixnum?
    vec : fxvector?
    pos : exact-nonnegative-integer?

Returns the fixnum in slot pos of vec. The first slot is position 0, and the last slot is one less than (fxvector-length vec).

(fxvector-set! vec pos x) → fixnum?
    vec : fxvector?
    pos : exact-nonnegative-integer?
    x : fixnum?
Sets the fixnum in slot \text{pos} of \text{vec}. The first slot is position 0, and the last slot is one less than \text{(fxvector-length vec)}.

\[(\text{fxvector-copy vec [start end]) \rightarrow fxvector?}\]
vec : fxvector?
start : exact-nonnegative-integer? = 0
end : exact-nonnegative-integer? = (vector-length v)

Creates a fresh fxvector of size \((- end start\)), with all of the elements of \text{vec} from \text{start} (inclusive) to \text{end} (exclusive).

\[(\text{in-fxvector vec [start stop step]})) \rightarrow \text{sequence?}\]
vec : fxvector?
start : exact-nonnegative-integer? = 0
stop : (or/c exact-integer? #f) = #f
step : (and/c exact-integer? (not/c zero?)) = 1

Returns a sequence equivalent to \text{vec} when no optional arguments are supplied.

The optional arguments \text{start}, \text{stop}, and \text{step} are as in \text{in-vector}.

An \text{in-fxvector} application can provide better performance for fxvector iteration when it appears directly in a \text{for} clause.

\[(\text{for/fxvector maybe-length (for-clause ...)} \ text{body} ...)\]
\[(\text{for*/fxvector maybe-length (for-clause ...)} \ text{body} ...)]

\text{maybe-length} =
| #:length length-expr
| #:length length-expr #:fill fill-expr

length-expr : exact-nonnegative-integer?
fill-expr : fixnum?

Like \text{for/vector} or \text{for*/vector}, but for fxvectors. The default \text{fill-expr} produces 0.

\[(\text{shared-fxvector x ...}) \rightarrow fxvector?\]
x : fixnum?

Creates a fxvector containing the given fixnums. For communication among places, the new fxvector is allocated in the shared memory space.

Example:

\[
> (\text{shared-fxvector 2 3 4 5})
(fxvector 2 3 4 5)
\]
(make-shared-fxvector size [x]) → fxvector?
  size : exact-nonnegative-integer?
  x : fixnum? = 0

Creates a fxvector with size elements, where every slot in the fxvector is filled with x. For communication among places, the new fxvector is allocated in the shared memory space.

Example:

> (make-shared-fxvector 4 3)
(fxvector 3 3 3 3)

### 4.3.5 Extflonums

(require racket/extflonum)  package: base

An extflonum is an extended-precision (80-bit) floating-point number. Extflonum arithmetic is supported on platforms with extended-precision hardware and where the extflonum implementation does not conflict with normal double-precision arithmetic (i.e., on x86 and x86_64 platforms when Racket is compiled to use SSE instructions for floating-point operations, and on Windows when "longdouble.dll" is available).

A extflonum is not a number in the sense of number?. Only extflonum-specific operations such as extfl+ perform extflonum arithmetic.

A literal extflonum is written like an inexact number, but using an explicit t or T exponent marker (see §1.3.4 "Reading Extflonums"). For example, 3.5t0 is an extflonum. The extflonum values for infinity are +inf.t and -inf.t. The extflonum value for not-a-number is +nan.t.

If (extflonum-available?) produces #f, then all operations exported by racket/extflonum raise exn:fail:unsupported, except for extflonum?, extflonum-available?, and extflvector? (which always work). The reader (see §1.3 "The Reader") always accepts extflonum input; when extflonum operations are not supported, printing an extflonum from the reader uses its source notation (as opposed to normalizing the format).

Two extflonums are equal? if extfl= produces #t for the extflonums. If extflonums are not supported in a platform, extflonums are equal? only if they are eq?.

(extflonum? v) → boolean?
  v : any/c

Returns #t if v is an extflonum, #f otherwise.
Returns #t if extflonum operations are supported on the current platform, #f otherwise.

**Extflonum Arithmetic**

(\texttt{extfl+ a b} → \texttt{extflonum}
  
  \(a : \texttt{extflonum}\)
  
  \(b : \texttt{extflonum}\)

(\texttt{extfl- a b} → \texttt{extflonum}
  
  \(a : \texttt{extflonum}\)
  
  \(b : \texttt{extflonum}\)

(\texttt{extfl* a b} → \texttt{extflonum}
  
  \(a : \texttt{extflonum}\)
  
  \(b : \texttt{extflonum}\)

(\texttt{extfl/ a b} → \texttt{extflonum}
  
  \(a : \texttt{extflonum}\)
  
  \(b : \texttt{extflonum}\)

(\texttt{extflabs a} → \texttt{extflonum}
  
  \(a : \texttt{extflonum}\)

Like \texttt{fl+}, \texttt{fl-}, \texttt{fl*}, \texttt{fl/}, and \texttt{flabs}, but for extflonums.

(\texttt{extfl= a b} → \texttt{boolean}
  
  \(a : \texttt{extflonum}\)
  
  \(b : \texttt{extflonum}\)

(\texttt{extfl< a b} → \texttt{boolean}
  
  \(a : \texttt{extflonum}\)
  
  \(b : \texttt{extflonum}\)

(\texttt{extfl> a b} → \texttt{boolean}
  
  \(a : \texttt{extflonum}\)
  
  \(b : \texttt{extflonum}\)

(\texttt{extfl<= a b} → \texttt{boolean}
  
  \(a : \texttt{extflonum}\)
  
  \(b : \texttt{extflonum}\)

(\texttt{extfl>= a b} → \texttt{boolean}
  
  \(a : \texttt{extflonum}\)
  
  \(b : \texttt{extflonum}\)

(\texttt{extflmin a b} → \texttt{extflonum}
  
  \(a : \texttt{extflonum}\)
  
  \(b : \texttt{extflonum}\)

(\texttt{extflmax a b} → \texttt{extflonum}
  
  \(a : \texttt{extflonum}\)
  
  \(b : \texttt{extflonum}\)

Like \texttt{fl=}, \texttt{fl<}, \texttt{fl>}, \texttt{fl<=}, \texttt{fl>=}, \texttt{flmin}, and \texttt{flmax}, but for extflonums.
Like `flround`, `flfloor`, `fceiling`, and `fltruncate`, but for extflonums.

Like `flsin`, `flcos`, `fatan`, `flasin`, `flacos`, `flatan`, `fllog`, `flexp`, and `flsqrt`, and `flexpt`, but for extflonums.
The first six are like \textit{->fl}, \textit{fl->exact-integer}, \textit{real->double-flonum}, \textit{inexact->exact}, \textit{fl->fx}, and \textit{fx->fl}, but for extflonums. The \textit{extfl->inexact} function converts an extflonum to its closest flonum approximation.

Changed in version 7.7.0.8 of package \texttt{base}: Changed \texttt{extfl->fx} to truncate.

### Extflonum Constants

- \texttt{pi.t : extflonum?}

  Like \texttt{pi}, but with 80 bits precision.

### Extflonum Vectors

An \textit{extflvector} is like an flvector, but it holds only extflonums. See also §17.4 “Unsafe Extflonum Operations”.

Two extflvectors are \texttt{equal?} if they have the same length, and if the values in corresponding slots of the extflvectors are \texttt{equal??}.

Like \texttt{flvector?}, \texttt{flvector}, \texttt{make-flvector}, \texttt{flvector-length}, \texttt{flvector-ref}, \texttt{flvector-set!}, \texttt{flvector-copy}, \texttt{flvector-ref!}.
flvector-set, and flvector-copy, but for extflvectors.

**(in-extflvector vec [start stop step]) → sequence?**
vec : extflvector?
  start : exact-nonnegative-integer? = 0
  stop : (or/c exact-integer? #f) = #f
  step : (and/c exact-integer? (not/c zero?)) = 1
(for/extflvector maybe-length (for-clause ...) body ...)
(for*/extflvector maybe-length (for-clause ...) body ...)

maybe-length =
  | #:length length-expr
  | #:length length-expr #:fill fill-expr

length-expr : exact-nonnegative-integer?
fill-expr : extflonum?

Like in-flvector, for/flvector, and for*/flvector, but for extflvectors.

**(shared-extflvector x ...) → extflvector?**
x : extflonum?
(make-shared-extflvector size [x]) → extflvector?
  size : exact-nonnegative-integer?
x : extflonum? = 0.0t0

Like shared-flvector and make-shared-flvector, but for extflvectors.

**Extflonum Byte Strings**

**(floating-point-bytes->extfl bstr [big-endian? start end]) → extflonum?**
bstr : bytes?
  big-endian? : any/c = (system-big-endian?)
  start : exact-nonnegative-integer? = 0
  end : exact-nonnegative-integer? = (bytes-length bstr)

Like floating-point-bytes->real, but for extflonums: Converts the extended-precision floating-point number encoded in bstr from position start (inclusive) to end (exclusive) to an extflonum. The difference between start and end must be 10 bytes.

**(extfl->floating-point-bytes x dest-bstr [big-endian? start]) → bytes?**
Like `real->floating-point-bytes`, but for extflonums: Converts `x` to its representation in a byte string of length 10.

4.4 Strings

A string is a fixed-length array of characters.

A string can be mutable or immutable. When an immutable string is provided to a procedure like `string-set!`, the `exn:fail:contract` exception is raised. String constants generated by the default reader (see §1.3.7 “Reading Strings”) are immutable, and they are interned in `read-syntax` mode. Use `immutable?` to check whether a string is immutable.

Two strings are `equal?` when they have the same length and contain the same sequence of characters.

A string can be used as a single-valued sequence (see §4.15.1 “Sequences”). The characters of the string serve as elements of the sequence. See also `in-string`.

See §1.3.7 “Reading Strings” for information on reading strings and §1.4.6 “Printing Strings” for information on printing strings.

See also: `immutable?`, `symbol->string`, `bytes->string/utf-8`.

4.4.1 String Constructors, Selectors, and Mutators

\[
\text{(string? } v \text{)} \rightarrow \text{ boolean?}
\]

\[
v : \text{ any/c}
\]

Returns `#t` if `v` is a string, `#f` otherwise.

Examples:

> `(string? "Apple")`

`#t`

> `(string? 'apple)`

`#f`
(make-string k [char]) → string?
  k : exact-nonnegative-integer?
  char : char? = #\nul

Returns a new mutable string of length k where each position in the string is initialized with the character char.

Example:

> (make-string 5 #\z)
"zzzzz"

(string char ...) → string?
  char : char?

Returns a new mutable string whose length is the number of provided chars, and whose positions are initialized with the given chars.

Example:

> (string #\A #\p #\p #\l #\e)
"Apple"

(string->immutable-string str) → (and/c string? immutable?)
  str : string?

Returns an immutable string with the same content as str, returning str itself if str is immutable.

(string-length str) → exact-nonnegative-integer?
  str : string?

Returns the length of str.

Example:

> (string-length "Apple")
5

(string-ref str k) → char?
  str : string?
  k : exact-nonnegative-integer?
Returns the character at position \( k \) in \( str \). The first position in the string corresponds to 0, so the position \( k \) must be less than the length of the string, otherwise the \texttt{exn:fail:contract} exception is raised.

Example:

```scheme
> (string-ref "Apple" 0)
#\A
```

\texttt{(string-set! str k char)} \rightarrow \texttt{void}?

\texttt{str} : (and/c string? (not/c immutable?))

\texttt{k} : exact-nonnegative-integer?

\texttt{char} : char?

Changes the character position \( k \) in \( str \) to \texttt{char}. The first position in the string corresponds to 0, so the position \( k \) must be less than the length of the string, otherwise the \texttt{exn:fail:contract} exception is raised.

Examples:

```scheme
> (define s (string #\A #\p #\p #\l #\e))
> (string-set! s 4 #\y)
> s
"Apply"
```

\texttt{(substring str start [end])} \rightarrow \texttt{string}?

\texttt{str} : string?

\texttt{start} : exact-nonnegative-integer?

\texttt{end} : exact-nonnegative-integer? = (string-length \texttt{str})

Returns a new mutable string that is \((-\text{end} \text{start})\) characters long, and that contains the same characters as \texttt{str} from \texttt{start} inclusive to \texttt{end} exclusive. The first position in a string corresponds to 0, so the \texttt{start} and \texttt{end} arguments must be less than or equal to the length of \texttt{str}, and \texttt{end} must be greater than or equal to \texttt{start}, otherwise the \texttt{exn:fail:contract} exception is raised.

Examples:

```scheme
> (substring "Apple" 1 3)
"pp"
> (substring "Apple" 1)
"pple"
```

\texttt{(string-copy str)} \rightarrow \texttt{string}?

\texttt{str} : string?
Returns \( \text{substring \ str \ 0} \).

Examples:

\[
\begin{align*}
&> (\text{define \ s1 \ "Yui"}) \\
&> (\text{define \ pilot \ (string-copy \ s1)}) \\
&> (\text{list \ s1 \ pilot}) \\
&\text{'("Yui" \ "Yui")} \\
&> (\text{for \ ([i \ (in-naturals)] \ [ch \ '(\#\R \#\e \ #\i)])}) \\
&\quad (\text{string-set! \ pilot \ i \ ch}) \\
&> (\text{list \ s1 \ pilot}) \\
&\text{'("Yui" \ \"Rei\")}
\end{align*}
\]

\[
\begin{align*}
&\text{(string-copy! \ dest} \\
&\quad \text{dest-start} \\
&\quad \text{src} \\
&\quad [\text{src-start} \\
&\quad \text{src-end}]) \rightarrow \text{void?} \\
&\text{dest} : (\text{and/c \ string?} \ (\text{not/c \ immutable?)}) \\
&\text{dest-start} : \text{exact-nonnegative-integer?} \\
&\text{src} : \text{string?} \\
&\text{src-start} : \text{exact-nonnegative-integer?} = 0 \\
&\text{src-end} : \text{exact-nonnegative-integer?} = (\text{string-length \ src})
\end{align*}
\]

Changes the characters of \text{dest} starting at position \text{dest-start} to match the characters in \text{src} from \text{src-start} (inclusive) to \text{src-end} (exclusive), where the first position in a string corresponds to 0. The strings \text{dest} and \text{src} can be the same string, and in that case the destination region can overlap with the source region; the destination characters after the copy match the source characters from before the copy. If any of \text{dest-start, src-start, or src-end} are out of range (taking into account the sizes of the strings and the source and destination regions), the \text{exn:fail:contract} exception is raised.

Examples:

\[
\begin{align*}
&> (\text{define \ s \ (string \ \#\A \ \#\p \ \#\p \ \#\l \ \#\e))} \\
&> (\text{string-copy! \ s \ 4 \ "y")} \\
&> (\text{string-copy! \ s \ 0 \ 3 \ 4)} \\
&> s \\
&\text{"lply"}
\end{align*}
\]

\[
\begin{align*}
&\text{(string-fill! \ dest \ char) \rightarrow \text{void?} } \\
&\text{dest} : (\text{and/c \ string?} \ (\text{not/c \ immutable?)}) \\
&\text{char} : \text{char?}
\end{align*}
\]

Changes \text{dest} so that every position in the string is filled with \text{char}.

Examples:
> (define s (string #\A #\p #\p #\l #\e))
> (string-fill! s #\q)
> s
"qqqqq"

(string-append str ...) → string?
str : string?

Returns a new mutable string that is as long as the sum of the given strs’ lengths, and that contains the concatenated characters of the given strs. If no strs are provided, the result is a zero-length string.

Example:

> (string-append "Apple" "Banana")
"AppleBanana"

(string-append-immutable str ...) → (and/c string? immutable?)
str : string?

The same as string-append, but the result is an immutable string.

Examples:

> (string-append-immutable "Apple" "Banana")
"AppleBanana"
> (immutable? (string-append-immutable "A" "B"))
#t

Added in version 7.5.0.14 of package base.

(string->list str) → (listof char?)
str : string?

Returns a new list of characters corresponding to the content of str. That is, the length of the list is (string-length str), and the sequence of characters in str is the same sequence in the result list.

Example:

> (string->list "Apple")
'(#\A #\p #\p #\l #\e)
Returns a new mutable string whose content is the list of characters in `lst`. That is, the length of the string is `(length lst)`, and the sequence of characters in `lst` is the same sequence in the result string.

Example:

```scheme
> (list->string (list #\A #\p #\p #\l #\e))
"Apple"
```

```scheme
(build-string n proc) → string?
  n : exact-nonnegative-integer?
  proc : (exact-nonnegative-integer? . -> . char?)
```

Creates a string of `n` characters by applying `proc` to the integers from 0 to `(sub1 n)` in order. If `str` is the resulting string, then `(string-ref str i)` is the character produced by `(proc i)`.

Example:

```scheme
> (build-string 5 (lambda (i) (integer->char (+ i 97))))
"abcde"
```

### 4.4.2 String Comparisons

```scheme
(string=? str1 str2 ...) → boolean?
  str1 : string?
  str2 : string?
```

Returns `#t` if all of the arguments are `equal?`.

Examples:

```scheme
> (string=? "Apple" "apple")
#f
> (string=? "a" "as" "a")
#f
```

Changed in version 7.0.0.13 of package `base`: Allow one argument, in addition to allowing two or more.

```scheme
(string<? str1 str2 ...) → boolean?
  str1 : string?
  str2 : string?
```
Returns #t if the arguments are lexicographically sorted increasing, where individual characters are ordered by char<?, #f otherwise.

Examples:

```lisp
> (string<? "Apple" "apple")
#t
> (string<? "apple" "Apple")
#f
> (string<? "a" "b" "c")
#t
```

Changed in version 7.0.0.13 of package base: Allow one argument, in addition to allowing two or more.

```
(string<=? str1 str2 ...) → boolean?
str1 : string?
str2 : string?
```

Like string<?, but checks whether the arguments are nondecreasing.

Examples:

```lisp
> (string<=? "Apple" "apple")
#t
> (string<=? "apple" "Apple")
#f
> (string<=? "a" "b" "b")
#t
```

Changed in version 7.0.0.13 of package base: Allow one argument, in addition to allowing two or more.

```
(string>? str1 str2 ...) → boolean?
str1 : string?
str2 : string?
```

Like string<?, but checks whether the arguments are decreasing.

Examples:

```lisp
> (string>? "Apple" "apple")
#f
> (string>? "apple" "Apple")
#t
> (string>? "c" "b" "a")
#t
```
(string>=? str1 str2 ...) → boolean?
str1 : string?
str2 : string?

Like `string<?`, but checks whether the arguments are nonincreasing.

Examples:
> (string>=? "Apple" "apple")
#f
> (string>=? "apple" "Apple")
#t
> (string>=? "c" "b" "b")
#t

Changed in version 7.0.0.13 of package base: Allow one argument, in addition to allowing two or more.

(string-ci=? str1 str2 ...) → boolean?
str1 : string?
str2 : string?

Returns #t if all of the arguments are equal? after locale-insensitive case-folding via `string-foldcase`.

Examples:
> (string-ci=? "Apple" "apple")
#t
> (string-ci=? "a" "a" "a")
#t

Changed in version 7.0.0.13 of package base: Allow one argument, in addition to allowing two or more.

(string-ci<? str1 str2 ...) → boolean?
str1 : string?
str2 : string?

Like `string<?`, but checks whether the arguments would be in increasing order if each was first case-folded using `string-foldcase` (which is locale-insensitive).

Examples:
> (string-ci<? "Apple" "apple")
(string-ci<? "apple" "banana")
#t
(string-ci<? "a" "b" "c")
#t

Changed in version 7.0.0.13 of package base: Allow one argument, in addition to allowing two or more.

(string-ci<=? str1 str2 ...) \rightarrow boolean?
str1 : string?
str2 : string?

Like string-ci<?, but checks whether the arguments would be nondecreasing after case-folding.

Examples:
> (string-ci<=? "Apple" "apple")
#t
> (string-ci<=? "apple" "Apple")
#t
> (string-ci<=? "a" "b" "b")
#t

Changed in version 7.0.0.13 of package base: Allow one argument, in addition to allowing two or more.

(string-ci>? str1 str2 ...) \rightarrow boolean?
str1 : string?
str2 : string?

Like string-ci<?, but checks whether the arguments would be decreasing after case-folding.

Examples:
> (string-ci>? "Apple" "apple")
#f
> (string-ci>? "banana" "Apple")
#t
> (string-ci>? "c" "b" "a")
#t

Changed in version 7.0.0.13 of package base: Allow one argument, in addition to allowing two or more.

(string-ci>=? str1 str2 ...) \rightarrow boolean?
str1 : string?
str2 : string?
Like `string-ci<=?`, but checks whether the arguments would be nonincreasing after case-folding.

Examples:

```scheme
> (string-ci>=? "Apple" "apple")
#t
> (string-ci>=? "apple" "Apple")
#t
> (string-ci>=? "c" "b" "b")
#t
```

Changed in version 7.0.0.13 of package base: Allow one argument, in addition to allowing two or more.

### 4.4.3 String Conversions

```scheme
(string-upcase str) → string?
str : string?
```

Returns a string whose characters are the upcase conversion of the characters in `str`. The conversion uses Unicode’s locale-independent conversion rules that map code-point sequences to code-point sequences (instead of simply mapping a 1-to-1 function on code points over the string), so the string produced by the conversion can be longer than the input string.

Examples:

```scheme
> (string-upcase "abc!")
"ABC!"
> (string-upcase "Straße")
"STRASSE"
```

```scheme
(string-downcase string) → string?
string : string?
```

Like `string-upcase`, but the downcase conversion.

Examples:

```scheme
> (string-downcase "aBC!")
"abc!"
> (string-downcase "Straße")
"straße"
> (string-downcase "ΚΑΩΣ")
"ΚΑΩΣ"
> (string-downcase "Σ")
"Σ"
```
(string-titlecase string) → string?
  string : string?

Like string-upcase, but the titlecase conversion only for the first character in each sequence of cased characters in str (ignoring case-ignorable characters).

Examples:
> (string-titlecase "aBC twO")
"Abc Two"
> (string-titlecase "y2k")
"Y2k"
> (string-titlecase "main straße")
"Main Straße"
> (string-titlecase "stra ße")
"Stra Sse"

(string-foldcase string) → string?
string : string?

Like string-upcase, but the case-folding conversion.

Examples:
> (string-foldcase "aBC!")
"abc!"
> (string-foldcase "Straße")
"strasse"
> (string-foldcase "ΚΑΘ")
"καθ"

(string-normalize-nfd string) → string?
string : string?

Returns a string that is the Unicode normalized form D of string. If the given string is already in the corresponding Unicode normal form, the string may be returned directly as the result (instead of a newly allocated string).

(string-normalize-nfkd string) → string?
string : string?

Like string-normalize-nfd, but for normalized form KD.

(string-normalize-nfc string) → string?
string : string?
Like `string-normalize-nfd`, but for normalized form C.

```
(string-normalize-nfkc string) → string?
```

Like `string-normalize-nfd`, but for normalized form KC.

### 4.4.4 Locale-Specific String Operations

```
(string-locale=? str1 str2 ...) → boolean?
str1 : string?
str2 : string?
```

Like `string=？`, but the strings are compared in a locale-specific way, based on the value of `current-locale`. See §13.1.1 “Encodings and Locales” for more information on locales.

Changed in version 7.0.0.13 of package `base`: Allow one argument, in addition to allowing two or more.

```
(string-locale<? str1 str2 ...) → boolean?
str1 : string?
str2 : string?
```

Like `string<？`, but the sort order compares strings in a locale-specific way, based on the value of `current-locale`. In particular, the sort order may not be simply a lexicographic extension of character ordering.

Changed in version 7.0.0.13 of package `base`: Allow one argument, in addition to allowing two or more.

```
(string-locale>? str1 str2 ...) → boolean?
str1 : string?
str2 : string?
```

Like `string>？`, but locale-specific like `string-locale<？`.

Changed in version 7.0.0.13 of package `base`: Allow one argument, in addition to allowing two or more.

```
(string-locale-ci=? str1 str2 ...) → boolean?
str1 : string?
str2 : string?
```

Like `string-locale=?`, but strings are compared using rules that are both locale-specific and case-insensitive (depending on what “case-insensitive” means for the current locale).

Changed in version 7.0.0.13 of package `base`: Allow one argument, in addition to allowing two or more.
(string-locale-ci<? str1 str2 ...) → boolean?
  str1 : string?
  str2 : string?

Like string<?, but both locale-sensitive and case-insensitive like string-locale-ci=?.

Changed in version 7.0.0.13 of package base: Allow one argument, in addition to allowing two or more.

(string-locale-ci>? str1 str2 ...) → boolean?
  str1 : string?
  str2 : string?

Like string>?, but both locale-sensitive and case-insensitive like string-locale-ci=?.

Changed in version 7.0.0.13 of package base: Allow one argument, in addition to allowing two or more.

(string-locale-upcase string) → string?
  string : string?

Like string-upcase, but using locale-specific case-conversion rules based on the value of current-locale.

(string-locale-downcase string) → string?
  string : string?

Like string-downcase, but using locale-specific case-conversion rules based on the value of current-locale.

4.4.5 Additional String Functions

(require racket/string)          package: base

The bindings documented in this section are provided by the racket/string and racket libraries, but not racket/base.

(string-append* str ... strs) → string?
  str : string?
  strs : (listof string?)

Like string-append, but the last argument is used as a list of arguments for string-append, so (string-append* str ... strs) is the same as (apply string-append str ... strs). In other words, the relationship between string-append and string-append* is similar to the one between list and list*.

Examples:
> (string-append* "a" "b" '("c" "d")
"abcd"
> (string-append* (cdr (append* (map (lambda (x) (list "", " x))
 ('("Alpha" "Beta" "Gamma"))))
"Alpha, Beta, Gamma"

(string-join strs
  [sep
    #:before-first before-first
    #:before-last before-last
    #:after-last after-last]) → string?
strs : (listof string?)
sep : string? = " "
before-first : string? = ""
before-last : string? = sep
after-last : string? = ""

Appends the strings in strs, inserting sep between each pair of strings in strs. before-last, before-first, and after-last are analogous to the inputs of add-between: they specify an alternate separator between the last two strings, a prefix string, and a suffix string respectively.

Examples:
> (string-join '("one" "two" "three" "four")
"one two three four"
> (string-join '("one" "two" "three" "four")", "
"one, two, three, four"
> (string-join '("one" "two" "three" "four") potato)
"one potato two potato three potato four"
> (string-join '("x" "y" "z")", "
  #:before-first "Todo: "
  #:before-last " and "
  #:after-last ".")
"Todo: x, y and z."

(string-normalize-spaces str
  [sep
    space
    #:trim? trim?
    #:repeat? repeat?]) → string?
str : string?
sep : (or/c string? regexp?) = #px"\s+"
space : string? = " "
trim? : any/c = #t
repeat? : any/c = #f

249
Normalizes spaces in the input \texttt{str} by trimming it (using \texttt{string-trim} and \texttt{sep}) and replacing all whitespace sequences in the result with \texttt{space}, which defaults to a single space.

Example:

```scheme
> (string-normalize-spaces " foo bar baz \r\n\t")
"foo bar baz"
```

The result of \texttt{(string-normalize-spaces str sep space)} is the same as \texttt{(string-join (string-split str sep ...) space)}.

```scheme
(string-replace str from to [#:all? all?] → string?
str : string?
from : (or/c string? regexp?)
to : string?
all? : any/c = #t)
```

Returns \texttt{str} with all occurrences of \texttt{from} replaced with by \texttt{to}. If \texttt{from} is a string, it is matched literally (as opposed to being used as a regular expression).

By default, all occurrences are replaced, but only the first match is replaced if \texttt{all?} is \texttt{#f}.

Example:

```scheme
> (string-replace "foo bar baz" "bar" "blah")
"foo blah baz"
```

```scheme
(string-split str
[sep
 #:trim? trim?
 #:repeat? repeat?] → (listof string?)
str : string?
sep : (or/c string? regexp?) = #px"\s+"
trim? : any/c = #t
repeat? : any/c = #f)
```

Splits the input \texttt{str} on \texttt{sep}, returning a list of substrings of \texttt{str} that are separated by \texttt{sep}, defaulting to splitting the input on whitespaces. The input is first trimmed using \texttt{sep} (see \texttt{string-trim}), unless \texttt{trim?} is \texttt{#f}. Empty matches are handled in the same way as for \texttt{regexp-split}. As a special case, if \texttt{str} is the empty string after trimming, the result is \texttt{'}()\texttt{'} instead of \texttt{'}("").

Like \texttt{string-trim}, provide \texttt{sep} to use a different separator, and \texttt{repeat?} controls matching repeated sequences.

Examples:
> (string-split " foo bar baz \r\n\t")
'("foo" "bar" "baz")
> (string-split " ")
'()
> (string-split " " #:trim? #f)
'("")

(string-trim str
  [sep
    #:left? left?
    #:right? right?
    #:repeat? repeat?]) -> string?

str : string?
sep : (or/c string? regexp?) = #px"\s+"
left? : any/c = #t
right? : any/c = #t
repeat? : any/c = #f

Trims the input str by removing prefix and suffix sep, which defaults to whitespace. A string sep is matched literally (as opposed to being used as a regular expression).

Use #:left? #f or #:right? #f to suppress trimming the corresponding side. When repeat? is #f (the default), only one match is removed from each side; when repeat? is true, all initial or trailing matches are trimmed (which is an alternative to using a regular expression sep that contains *).

Examples:

> (string-trim " foo bar baz \r\n\t")
"foo bar baz"
> (string-trim " foo bar baz \r\n\t" " #:repeat? #t)
"foo bar baz \r\n\t"
> (string-trim "aaaxaayaa" "aa")
"axaay"

(non-empty-string? x) -> boolean?
x : any/c

Returns #t if x is a string and is not empty; returns #f otherwise.

Added in version 6.3 of package base.

(string-contains? s contained) -> boolean?
s : string?
contained : string?
(string-prefix? s prefix) → boolean?
s : string?
  prefix : string?
(string-suffix? s suffix) → boolean?
s : string?
  suffix : string?

Checks whether s includes at any location, start with, or ends with the second argument, respectively.

Examples:
> (string-prefix? "Racket" "R")
#t
> (string-prefix? "Jacket" "R")
#f
> (string-suffix? "Racket" "et")
#t
> (string-contains? "Racket" "ack")
#t

Added in version 6.3 of package base.

4.4.6 Converting Values to Strings

(require racket/format)  package: base

The bindings documented in this section are provided by the racket/format and racket libraries, but not racket/base.

The racket/format library provides functions for converting Racket values to strings. In addition to features like padding and numeric formatting, the functions have the virtue of being shorter than format (with format string), number->string, or string-append.

(~a
  ... [#:separator separator
  #:width width
  #:max-width max-width
  #:min-width min-width
  #:limit-marker limit-marker
  #:limit-prefix? limit-prefix?
  #:align align
  #:pad-string pad-string
  #:left-pad-string left-pad-string
  #:right-pad-string right-pad-string]) → string?

252
v : any/c
separator : string? = ""
width : (or/c exact-nonnegative-integer? #f) = #f
max-width : (or/c exact-nonnegative-integer? +inf.0)
  = (or width +inf.0)
min-width : exact-nonnegative-integer? = (or width 0)
limit-marker : string? = ""
limit-prefix? : boolean? = #f
align : (or/c 'left 'center 'right) = 'left
pad-string : non-empty-string? = " "
left-pad-string : non-empty-string? = pad-string
right-pad-string : non-empty-string? = pad-string

Converts each v to a string in display mode—that is, like (format "~a" v)—then concatenates the results with separator between consecutive items, and then pads or truncates the string to be at least min-width characters and at most max-width characters.

Examples:

> (~a "north")
"north"
> (~a 'south)
"south"
> (~a #"east")
"east"
> (~a \\w "e" 'st)
"west"
> (~a (list "red" 'green #"blue"))
"(red green blue)"
> (~a 17)
"17"
> (~a #e1e20)
"1000000000000000000000000"
> (~a pi)
"3.141592653589793"
> (~a (expt 6.1 87))
"2.107150386211452e+68"

The ~a function is primarily useful for strings, numbers, and other atomic data. The ~v and ~s functions are better suited to compound data.

Let s be the concatenated string forms of the vs plus separators. If s is longer than max-width characters, it is truncated to exactly max-width characters. If s is shorter than min-width characters, it is padded to exactly min-width characters. Otherwise s is returned unchanged. If min-width is greater than max-width, an exception is raised.
If $s$ is longer than $max-width$ characters, it is truncated and the end of the string is replaced with $limit-marker$. If $limit-marker$ is longer than $max-width$, an exception is raised. If $limit-prefix?$ is #t, the beginning of the string is truncated instead of the end.

Examples:

```scheme
> (~a "abcde" #:max-width 5)
"abcde"
> (~a "abcde" #:max-width 4)
"abcd"
> (~a "abcde" #:max-width 4 #:limit-marker ";")
"abc\n" > (~a "abcde" #:max-width 4 #:limit-marker ";...")
"a..."
> (~a "The quick brown fox" #:max-width 15 #:limit-marker ";")
"The quick brown"
> (~a "The quick brown fox" #:max-width 15 #:limit-marker ";...")
"The quick br..."
> (~a "The quick brown fox" #:max-width 15 #:limit-marker ";..." #:limit-prefix? #f)
"The quick br..."
```

If $s$ is shorter than $min-width$, it is padded to at least $min-width$ characters. If $align$ is 'left, then only right padding is added; if $align$ is 'right, then only left padding is added; and if $align$ is 'center, then roughly equal amounts of left padding and right padding are added.

Padding is specified as a non-empty string. Left padding consists of $left-pad-string$ repeated in its entirety as many times as possible followed by a prefix of $left-pad-string$ to fill the remaining space. In contrast, right padding consists of a suffix of $right-pad-string$ followed by a number of copies of $right-pad-string$ in its entirety. Thus left padding starts with the start of $left-pad-string$ and right padding ends with the end of $right-pad-string$.

Examples:

```scheme
> (~a "apple" #:min-width 20 #:align 'left)
"apple"
> (~a "pear" #:min-width 20 #:align 'left #:right-pad-string ".")
"pear ......"
> (~a "plum" #:min-width 20 #:align 'right #:left-pad-string ".")
".... plum"
> (~a "orange" #:min-width 20 #:align 'center
    #:left-pad-string "- " #:right-pad-string " -")
```

254
Use `width` to set both `max-width` and `min-width` simultaneously, ensuring that the resulting string is exactly `width` characters long:

\[
\begin{align*}
&\texttt{> (~a "terse" #:width 6)} \\
&\quad \text{"terse "} \\
&\texttt{> (~a "loquacious" #:width 6)} \\
&\quad \text{"loquac"}
\end{align*}
\]

\[
(\sim v \ v \\
\ldots \\
\#\text{separator separator} \\
\#\text{width width} \\
\#\text{max-width max-width} \\
\#\text{min-width min-width} \\
\#\text{limit-marker limit-marker} \\
\#\text{limit-prefix? limit-prefix?} \\
\#\text{align align} \\
\#\text{pad-string pad-string} \\
\#\text{left-pad-string left-pad-string} \\
\#\text{right-pad-string right-pad-string}) \to \text{string?}
\]

\[
\begin{align*}
&v : \text{any/c} \\
&\text{separator : string? = " "} \\
&\text{width : (or/c exact-nonnegative-integer? #f) = #f} \\
&\text{max-width : (or/c exact-nonnegative-integer? +inf.0)} \\
&\quad = (\text{or width +inf.0}) \\
&\text{min-width : exact-nonnegative-integer? = (or width 0)} \\
&\text{limit-marker : string? = "..."} \\
&\text{limit-prefix? : boolean? = #f} \\
&\text{align : (or/c 'left 'center 'right) = 'left} \\
&\text{pad-string : non-empty-string? = " "} \\
&\text{left-pad-string : non-empty-string? = pad-string} \\
&\text{right-pad-string : non-empty-string? = pad-string}
\end{align*}
\]

Like `~a`, but each value is converted like `(format "~v" v)`, the default separator is " ", and the default limit marker is "...".

Examples:

\[
\begin{align*}
&\texttt{> (~v "north")} \\
&\quad \text{"\"north\""} \\
&\texttt{> (~v 'south)} \\
&\quad \text{"'south"} \\
&\texttt{> (~v #"east")}
\end{align*}
\]
Use \texttt{~v} to produce text that talks about Racket values.

Example:

\begin{verbatim}
> (let ([nums (for/list ([i 10]) i)])
  (~a "The even numbers in " (~v nums)
    " are " (~v (filter even? nums)) ".")
"The even numbers in '(0 1 2 3 4 5 6 7 8 9) are '(0 2 4 6 8)."
\end{verbatim}

\begin{verbatim}
(~s v ...
  #:separator separator
  #:width width
  #:max-width max-width
  #:min-width min-width
  #:limit-marker limit-marker
  #:limit-prefix? limit-prefix?
  #:align align
  #:pad-string pad-string
  #:left-pad-string left-pad-string
  #:right-pad-string right-pad-string)) \to string?
\end{verbatim}

Like \texttt{~a}, but each value is converted like \texttt{(format "~s" v)}, the default separator is \\
" ", and the default limit marker is "...".

Examples:

\begin{verbatim}
> (~s "north")
\end{verbatim}
"\"north\""
> (~s 'south)
"south"
> (~s #"east")
"#\"east\""
> (~s #\w)
"#\w"
> (~s (list "red" 'green="#blue"))
"("red" green="#blue")"

(~ e v
  ...
  #:separator separator
  #:width width
  #:max-width max-width
  #:min-width min-width
  #:limit-marker limit-marker
  #:limit-prefix? limit-prefix?
  #:align align
  #:pad-string pad-string
  #:left-pad-string left-pad-string
  #:right-pad-string right-pad-string)

v : any/c
separator : string? = " "
width : (or/c exact-nonnegative-integer? #f) = #f
max-width : (or/c exact-nonnegative-integer? +inf.0)
  = (or width +inf.0)
min-width : exact-nonnegative-integer? = (or width 0)
limit-marker : string? = "...
limit-prefix? : boolean? = #f
align : (or/c 'left 'center 'right) = 'left
pad-string : non-empty-string? = " "
left-pad-string : non-empty-string? = pad-string
right-pad-string : non-empty-string? = pad-string

Like ~a, but each value is converted like (format "~e" v), the default separator is " ",
and the default limit marker is "...".

Examples:

> (~e "north")
"\"north\""
> (~e 'south)
"'south"
> (~e #"east")
Converts the rational number \( x \) to a string in either positional or exponential notation, depending on \( notation \). The exactness or inexactness of \( x \) does not affect its formatting.

The optional arguments control number formatting:

- **notation** — determines whether the number is printed in positional or exponential notation. If \( notation \) is a function, it is applied to \( x \) to get the notation to be used.

Examples:

\[
\begin{align*}
\texttt{> (\texttt{~r 12345})} & \quad \texttt{"12345"} \\
\texttt{> (\texttt{~r 12345 #:notation 'exponential})} & \quad \texttt{"1.2345e+04"}
\end{align*}
\]
> (let ([pick-notation
    (lambda (x)
      (if (or (< (abs x) 0.001) (> (abs x) 1000))
        'exponential
        'positional))])
  (for/list ([i (in-range 1 5)])
    (~r (expt 17 i) #:notation pick-notation)))
'("17" "289" "4.913e+03" "8.3521e+04")

- `precision` — controls the number of digits after the decimal point (or more accurately, the radix point). When `x` is formatted in exponential form, `precision` applies to the significand.

  If `precision` is a natural number, then up to `precision` digits are displayed, but trailing zeroes are dropped, and if all digits after the decimal point are dropped the decimal point is also dropped. If `precision` is `(list '=' digits)`, then exactly `digits` digits after the decimal point are used, and the decimal point is never dropped.

  Examples:

  > (~r pi)
  "3.141593"
  > (~r pi #:precision 4)
  "3.1416"
  > (~r pi #:precision 0)
  "3"
  > (~r 1.5 #:precision 4)
  "1.5"
  > (~r 1.5 #:precision '(= 4))
  "1.5000"
  > (~r 50 #:precision 2)
  "50"
  > (~r 50 #:precision '(= 2))
  "50.00"
  > (~r 50 #:precision '(= 0))
  "50."

- `min-width` — if `x` would normally be printed with fewer than `min-width` digits (including the decimal point but not including the sign indicator), the digits are left-padded using `pad-string`.

  Examples:

  > (~r 17)
  "17"
  > (~r 17 #:min-width 4)
  " 17"
  > (~r -42 #:min-width 4)
"- 42"
> (r 1.5 #:min-width 4)
" 1.5"
> (r 1.5 #:precision 4 #:min-width 10)
" 1.5"
> (r 1.5 #:precision (= 4) #:min-width 10)
" 1.5000"
> (r #e1e10 #:min-width 6)
"10000000000"

• **pad-string** — specifies the string used to pad the number to at least `min-width` characters (not including the sign indicator). The padding is placed between the sign and the normal digits of `x`.

Examples:

> (r 17 #:min-width 4 #:pad-string "0")
"0017"
> (r -42 #:min-width 4 #:pad-string "0")
"-0042"

• **sign** — controls how the sign of the number is indicated:

  - If `sign` is `#f` (the default), no sign output is generated if `x` is either positive or zero, and a minus sign is prefixed if `x` is negative.
    
    Example:

    > (for/list ([x '(17 0 -42)]) (r x))
    '("17" "0" "-42")

  - If `sign` is `'+`, no sign output is generated if `x` is zero, a plus sign is prefixed if `x` is positive, and a minus sign is prefixed if `x` is negative.
    
    Example:

    > (for/list ([x '(17 0 -42)]) (r x #:sign '+))
    '("+17" "0" "-42")

  - If `sign` is `'++`, a plus sign is prefixed if `x` is zero or positive, and a minus sign is prefixed if `x` is negative.
    
    Example:

    > (for/list ([x '(17 0 -42)]) (r x #:sign '++))
    '("+17" "+0" "-42")

  - If `sign` is `parens`, no sign output is generated if `x` is zero or positive, and the number is enclosed in parentheses if `x` is negative.
    
    Example:
> (for/list ([x '(17 0 -42)]) (~r x #:sign 'parens)
  "("17" "0" ")")

– If sign is (list pos-ind zero-ind neg-ind), then pos-ind, zero-ind, and neg-ind are used to indicate positive, zero, and negative numbers, respectively. Each indicator is either a string to be used as a prefix or a list containing two strings: a prefix and a suffix.

Example:

> (let ([sign-table '(("" " up") "an even " ("" " down"))]]
  (for/list ([x '(17 0 -42)]) (~r x #:sign sign-table)))
  "("17 up" "an even 0" ")")

The default behavior is equivalent to '("" " "")'; the 'parens mode is equivalent to '("" " ("" ")")'.

• base — controls the base that x is formatted in. If base is a number greater than 10, then lower-case letters are used. If base is (list 'up base*) and base* is greater than 10, then upper-case letters are used.

Examples:

> (~r 100 #:base 7)
"202"
> (~r 4.5 #:base 2)
"100.1"
> (~r 3735928559 #:base 16)
"deadbeef"
> (~r 3735928559 #:base 'up 16)
"DEADBEEF"
> (~r 3735928559 #:base 'up 16) #:notation 'exponential)
"D.EADBEF+16^+07"

• format-exponent — determines how the exponent is displayed.

If format-exponent is a string, the exponent is displayed with an explicit sign (as with a sign of '++') and at least two digits, separated from the significand by the “exponent marker” format-exponent:

> (~r 1234 #:notation 'exponential #:format-exponent "E")
"1.234E+03"

If format-exponent is #f, the “exponent marker” is "e" if base is 10 and a string involving base otherwise:
> (r 1234 #:notation 'exponential)
"1.234e+03"
> (r 1234 #:notation 'exponential #:base 8)
"2.322*8^-03"

If \textit{format-exponent} is a procedure, it is applied to the exponent and the resulting string is appended to the significand:

> (r 1234 #:notation 'exponential
  #:format-exponent (lambda (e) (format "E\(^{-}\)a" e)))
"1.234E3"

\begin{verbatim}
(\texttt{\textbackslash .a } \texttt{v
  ...\textbackslash #:separator separator
  #:width width
  #:max-width max-width
  #:min-width min-width
  #:limit-marker limit-marker
  #:limit-prefix? limit-prefix?
  #:align align
  #:pad-string pad-string
  #:left-pad-string left-pad-string
  #:right-pad-string right-pad-string}) \rightarrow \texttt{string}?
\end{verbatim}

\begin{verbatim}
\texttt{v : any/c}
\texttt{separator : string? = ""}
\texttt{width : (or/c exact-nonnegative-integer? #f) = #f}
\texttt{max-width : (or/c exact-nonnegative-integer? +inf.0)
  = (or width +inf.0)}
\texttt{min-width : exact-nonnegative-integer? = (or width 0)}
\texttt{limit-marker : string? = ""}
\texttt{limit-prefix? : boolean? = #f}
\texttt{align : (or/c 'left 'center 'right) = 'left}
\texttt{pad-string : non-empty-string? = " "}
\texttt{left-pad-string : non-empty-string? = pad-string}
\texttt{right-pad-string : non-empty-string? = pad-string}
\end{verbatim}
v : any/c
separator : string? = " "
width : (or/c exact-nonnegative-integer? #f) = #f
max-width : (or/c exact-nonnegative-integer? +inf.0)
            = (or width +inf.0)
min-width : exact-nonnegative-integer? = (or width 0)
limit-marker : string? = "...
limit-prefix? : boolean? = #f
align : (or/c 'left 'center 'right) = 'left
pad-string : non-empty-string? = " "
left-pad-string : non-empty-string? = pad-string
right-pad-string : non-empty-string? = pad-string
Like ~a, ~v, and ~s, but each v is formatted like (format "~a" v), (format "~v" v), and (format "~s" v), respectively.

### 4.5 Byte Strings

A *byte string* is a fixed-length array of bytes. A *byte* is an exact integer between 0 and 255 inclusive.

A byte string can be *mutable* or *immutable*. When an immutable byte string is provided to a procedure like bytes-set!, the exn:fail:contract exception is raised. Byte-string constants generated by the default reader (see §1.3.7 “Reading Strings”) are immutable, and they are interned in read-syntax mode. Use immutable? to check whether a byte string is immutable.

Two byte strings are equal? when they have the same length and contain the same sequence of bytes.

A byte string can be used as a single-valued sequence (see §4.15.1 “Sequences”). The bytes of the string serve as elements of the sequence. See also in-bytes.
See §1.3.7 “Reading Strings” for information on reading byte strings and §1.4.6 “Printing Strings” for information on printing byte strings.

See also: immutable?.

### 4.5.1 Byte String Constructors, Selectors, and Mutators

**(bytes? v)** → boolean?

v : any/c

Returns #t if v is a byte string, #f otherwise.

Examples:

> (bytes? #“Apple”)
#t
> (bytes? "Apple")
#f

**(make-bytes k [b])** → bytes?

k : exact-nonnegative-integer?

b : byte? = 0

Returns a new mutable byte string of length k where each position in the byte string is initialized with the byte b.

Example:

> (make-bytes 5 65)
#"AAAAA"

**(bytes b ...)** → bytes?

b : byte?

Returns a new mutable byte string whose length is the number of provided bs, and whose positions are initialized with the given bs.

Example:

> (bytes 65 112 112 108 101)
#"Apple"

**(bytes->immutable-bytes bstr)** → (and/c bytes? immutable?)

bstr : bytes?
Returns an immutable byte string with the same content as \texttt{bstr}, returning \texttt{bstr} itself if \texttt{bstr} is immutable.

Examples:

\begin{verbatim}
> (bytes->immutable-bytes (bytes 65 65 65))
#"AAA"
> (define b (bytes->immutable-bytes (make-bytes 5 65)))
> (bytes->immutable-bytes b)
#"AAAAA"
> (eq? (bytes->immutable-bytes b) b)
#t
\end{verbatim}

\texttt{(byte? v)} \to \texttt{boolean?}\\
\hspace{1cm} v : \texttt{any/c}

Returns \texttt{#t} if \texttt{v} is a byte (i.e., an exact integer between 0 and 255 inclusive), \texttt{#f} otherwise.

Examples:

\begin{verbatim}
> (byte? 65)
#t
> (byte? 0)
#t
> (byte? 256)
#f
> (byte? -1)
#f
\end{verbatim}

\texttt{(bytes-length bstr)} \to \texttt{exact-nonnegative-integer?}\\
\hspace{1cm} bstr : \texttt{bytes?}

Returns the length of \texttt{bstr}.

Example:

\begin{verbatim}
> (bytes-length #"Apple")
5
\end{verbatim}

\texttt{(bytes-ref bstr k)} \to \texttt{byte?}\\
\hspace{1cm} bstr : \texttt{bytes?}\\
\hspace{1cm} k : \texttt{exact-nonnegative-integer?}

Returns the character at position \texttt{k} in \texttt{bstr}. The first position in the bytes corresponds to 0, so the position \texttt{k} must be less than the length of the bytes, otherwise the \texttt{exn:fail:contract} exception is raised.

Example:
(bytes-ref #'Apple 0)
65

(bytes-set! bstr k b) → void?
  bstr : (and/c bytes? (not/c immutable?))
  k : exact-nonnegative-integer?
  b : byte?

Changes the character position k in bstr to b. The first position in the byte string corresponds to 0, so the position k must be less than the length of the bytes, otherwise the exn:fail:contract exception is raised.

Examples:
> (define s (bytes 65 112 112 108 101))
> (bytes-set! s 4 121)
> s
"Apply"

(subbytes bstr start [end]) → bytes?
  bstr : bytes?
  start : exact-nonnegative-integer?
  end : exact-nonnegative-integer? = (bytes-length str)

Returns a new mutable byte string that is (end start) bytes long, and that contains the same bytes as bstr from start inclusive to end exclusive. The start and end arguments must be less than or equal to the length of bstr, and end must be greater than or equal to start, otherwise the exn:fail:contract exception is raised.

Examples:
> (subbytes #'Apple 1 3)
"pp"
> (subbytes #'Apple 1)
"pple"

(bytes-copy bstr) → bytes?
  bstr : bytes?

Returns (subbytes str 0).

(bytes-copy! dest
  dest-start
  src
  [src-start
  src-end]) → void?

267
Changes the bytes of `dest` starting at position `dest-start` to match the bytes in `src` from `src-start` (inclusive) to `src-end` (exclusive). The byte strings `dest` and `src` can be the same byte string, and in that case the destination region can overlap with the source region; the destination bytes after the copy match the source bytes from before the copy. If any of `dest-start`, `src-start`, or `src-end` are out of range (taking into account the sizes of the byte strings and the source and destination regions), the `exn:fail:contract` exception is raised.

Examples:

```racket
> (define s (bytes 65 112 112 108 101))
> (bytes-copy! s 4 #"y")
> (bytes-copy! s 0 s 3 4)
> s
"lpply"
```

Changes `dest` so that every position in the bytes is filled with `b`.

Examples:

```racket
> (define s (bytes 65 112 112 108 101))
> (bytes-fill! s 113)
> s
"qqqqq"
```

Returns a new mutable byte string that is as long as the sum of the given `bstrs'` lengths, and that contains the concatenated bytes of the given `bstrs`. If no `bstrs` are provided, the result is a zero-length byte string.

Example:
> (bytes-append #"Apple" #"Banana")
#"AppleBanana"

(bytes->list bstr) → (listof byte?)
bstr : bytes?

Returns a new list of bytes corresponding to the content of bstr. That is, the length of the list is (bytes-length bstr), and the sequence of bytes in bstr is the same sequence in the result list.

Example:
> (bytes->list #"Apple")
'(65 112 112 108 101)

(list->bytes lst) → bytes?
lst : (listof byte?)

Returns a new mutable byte string whose content is the list of bytes in lst. That is, the length of the byte string is (length lst), and the sequence of bytes in lst is the same sequence in the result byte string.

Example:
> (list->bytes (list 65 112 112 108 101))
#"Apple"

(make-shared-bytes k [b]) → bytes?
k : exact-nonnegative-integer?
b : byte? = 0

Returns a new mutable byte string of length k where each position in the byte string is initialized with the byte b. For communication among places, the new byte string is allocated in the shared memory space.

Example:
> (make-shared-bytes 5 65)
#"AAAAA"

(shared-bytes b ...) → bytes?
b : byte?
Returns a new mutable byte string whose length is the number of provided `bs`, and whose positions are initialized with the given `bs`. For communication among places, the new byte string is allocated in the shared memory space.

Example:

```
> (shared-bytes 65 112 112 108 101)
#"Apple"
```

### 4.5.2 Byte String Comparisons

```
(bytes=? bstr1 bstr2 ...) -> boolean?
bstr1 : bytes?
bstr2 : bytes?
```

Returns `#t` if all of the arguments are `eqv?`.

Examples:

```
> (bytes=? #"Apple" #"apple")
#f
> (bytes=? #"a" #"as" #"a")
#f
```

Changed in version 7.0.0.13 of package `base`: Allow one argument, in addition to allowing two or more.

```
(bytes<? bstr1 bstr2 ...) -> boolean?
bstr1 : bytes?
bstr2 : bytes?
```

Returns `#t` if the arguments are lexicographically sorted increasing, where individual bytes are ordered by `<`, `#f` otherwise.

Examples:

```
> (bytes<? #"Apple" #"apple")
#t
> (bytes<? #"apple" #"Apple")
#f
> (bytes<? #"a" #"b" #"c")
#t
```

Changed in version 7.0.0.13 of package `base`: Allow one argument, in addition to allowing two or more.
Like `bytes<?`, but checks whether the arguments are decreasing.

Examples:

```scheme
> (bytes>? #"Apple" #"apple")
#f
> (bytes>? #"apple" #"Apple")
#t
> (bytes>? #"c" #"b" #"a")
#t
```

Changed in version 7.0.0.13 of package base: Allow one argument, in addition to allowing two or more.

### 4.5.3 Bytes to/from Characters, Decoding and Encoding

```scheme
(bytes->string/utf-8 bstr [err-char start end]) -> string?
  bstr : bytes?
  err-char : (or/c #f char?) = #f
  start : exact-nonnegative-integer? = 0
  end : exact-nonnegative-integer? = (bytes-length bstr)
```

Produces a string by decoding the `start` to `end` substring of `bstr` as a UTF-8 encoding of Unicode code points. If `err-char` is not `#f`, then it is used for bytes that fall in the range 128 to 255 but are not part of a valid encoding sequence. (This rule is consistent with reading characters from a port; see §13.1.1 “Encodings and Locales” for more details.) If `err-char` is `#f`, and if the `start` to `end` substring of `bstr` is not a valid UTF-8 encoding overall, then the `exn:fail:contract` exception is raised.

Example:

```scheme
> (bytes->string/utf-8 (bytes 195 167 195 176 195 182 194 163))
"çðö£"
```

```scheme
(bytes->string/locale bstr [err-char start end]) -> string?
  bstr : bytes?
```

271
err-char : (or/c #f char?) = #f
start : exact-nonnegative-integer? = 0
end : exact-nonnegative-integer? = (bytes-length bstr)

Produces a string by decoding the start to end substring of bstr using the current locale’s encoding (see also §13.1.1 “Encodings and Locales”). If err-char is not #f, it is used for each byte in bstr that is not part of a valid encoding; if err-char is #f, and if the start to end substring of bstr is not a valid encoding overall, then the exn:fail:contract exception is raised.

(bytes->string/latin-1 bstr [err-char start end]) → string?

bstr : bytes?
err-char : (or/c #f char?) = #f
start : exact-nonnegative-integer? = 0
end : exact-nonnegative-integer? = (bytes-length bstr)

Produces a string by decoding the start to end substring of bstr as a Latin-1 encoding of Unicode code points; i.e., each byte is translated directly to a character using integer->char, so the decoding always succeeds. The err-char argument is ignored, but present for consistency with the other operations.

Example:

> (bytes->string/latin-1 (bytes 254 211 209 165))
"ÞÓ¥"

(string->bytes/utf-8 str [err-byte start end]) → bytes?

str : string?
err-byte : (or/c #f byte?) = #f
start : exact-nonnegative-integer? = 0
end : exact-nonnegative-integer? = (string-length str)

Produces a byte string by encoding the start to end substring of str via UTF-8 (always succeeding). The err-byte argument is ignored, but included for consistency with the other operations.

Examples:

> (define b
  (bytes->string/utf-8
   (bytes 195 167 195 176 195 182 194 163)))
> (string->bytes/utf-8 b)
(string->bytes/locale str [err-byte start end]) → bytes?
  str : string?
  err-byte : (or/c #f byte?) = #f
  start : exact-nonnegative-integer? = 0
  end : exact-nonnegative-integer? = (string-length str)

 Produces a string by encoding the start to end substring of str using the current locale’s encoding (see also §13.1.1 “Encodings and Locales”). If err-byte is not #f, it is used for each character in str that cannot be encoded for the current locale; if err-byte is #f, and if the start to end substring of str cannot be encoded, then the exn:fail:contract exception is raised.

(string->bytes/latin-1 str [err-byte start end]) → bytes?
  str : string?
  err-byte : (or/c #f byte?) = #f
  start : exact-nonnegative-integer? = 0
  end : exact-nonnegative-integer? = (string-length str)

 Produces a string by encoding the start to end substring of str using Latin-1; i.e., each character is translated directly to a byte using char->integer. If err-byte is not #f, it is used for each character in str whose value is greater than 255. If err-byte is #f, and if the start to end substring of str has a character with a value greater than 255, then the exn:fail:contract exception is raised.

Examples:

> (define b
  (bytes->string/latin-1 (bytes 254 211 209 165)))
'(\376\323\321\245
> (string->bytes/latin-1 b)
"\376\323\321\245"
> (bytes->string/latin-1 (string->bytes/latin-1 b))
"pðיה"

(string-utf-8-length str [start end]) → exact-nonnegative-integer?
  str : string?
  start : exact-nonnegative-integer? = 0
  end : exact-nonnegative-integer? = (string-length str)
>Returns the length in bytes of the UTF-8 encoding of str’s substring from start to end, but without actually generating the encoded bytes.

Examples:

```scheme
> (string-utf-8-length
   (bytes->string/utf-8 (bytes 195 167 195 176 195 182 194 163)))
8
> (string-utf-8-length "hello")
5
```

(Returns the length in characters of the UTF-8 decoding of bstr’s substring from start to end, but without actually generating the decoded characters. If err-char is #f and the substring is not a UTF-8 encoding overall, the result is #f. Otherwise, err-char is used to resolve decoding errors as in bytes->string/utf-8.

Examples:

```scheme
> (bytes-utf-8-length (bytes 195 167 195 176 195 182 194 163))
4
> (bytes-utf-8-length (make-bytes 5 65))
5
```

(Returns the skipth character in the UTF-8 decoding of bstr’s substring from start to end, but without actually generating the other decoded characters. If the substring is not a UTF-8 encoding up to the skipth character (when err-char is #f), or if the substring decoding produces fewer than skipt characters, the result is #f. If err-char is not #f, it is used to resolve decoding errors as in bytes->string/utf-8.

Examples:
Returns the offset in bytes into `bstr` at which the `skip`th character’s encoding starts in the UTF-8 decoding of `bstr`’s substring from `start` to `end` (but without actually generating the other decoded characters). The result is relative to the start of `bstr`, not to `start`. If the substring is not a UTF-8 encoding up to the `skip`th character (when `err-char` is `#f`), or if the substring decoding produces fewer than `skip` characters, the result is `#f`. If `err-char` is not `#f`, it is used to resolve decoding errors as in `bytes->string/utf-8`.

Examples:

```scheme
> (bytes-utf-8-index (bytes 195 167 195 176 195 182 194 163) 0)
0
> (bytes-utf-8-index (bytes 195 167 195 176 195 182 194 163) 1)
2
> (bytes-utf-8-index (bytes 195 167 195 176 195 182 194 163) 2)
4
> (bytes-utf-8-index (bytes 65 66 67 68) 0)
0
> (bytes-utf-8-index (bytes 65 66 67 68) 1)
1
> (bytes-utf-8-index (bytes 65 66 67 68) 2)
2
```

275
4.5.4 Bytes to Bytes Encoding Conversion

```scheme
(bytes-open-converter from-name to-name)
→ (or/c bytes-converter? #f)
from-name : string?
to-name : string?
```

Produces a byte converter to go from the encoding named by `from-name` to the encoding named by `to-name`. If the requested conversion pair is not available, `#f` is returned instead of a converter.

Certain encoding combinations are always available:

- `(bytes-open-converter "UTF-8" "UTF-8")` — the identity conversion, except that encoding errors in the input lead to a decoding failure.
- `(bytes-open-converter "UTF-8-permissive" "UTF-8")` — the identity conversion, except that any input byte that is not part of a valid encoding sequence is effectively replaced by the UTF-8 encoding sequence for `\uFFFD`. (This handling of invalid sequences is consistent with the interpretation of port bytes streams into characters; see §13.1 “Ports”)
- `(bytes-open-converter "" "UTF-8")` — converts from the current locale’s default encoding (see §13.1.1 “Encodings and Locales”) to UTF-8.
- `(bytes-open-converter "UTF-8" "")` — converts from UTF-8 to the current locale’s default encoding (see §13.1.1 “Encodings and Locales”).
- `(bytes-open-converter "platform-UTF-8" "platform-UTF-16")` — converts UTF-8 to UTF-16 on Unix and Mac OS, where each UTF-16 code unit is a sequence of two bytes ordered by the current platform’s endianness. On Windows, the conversion is the same as `(bytes-open-converter "WTF-8" "WTF-16")` to support unpaired surrogate code units.
- `(bytes-open-converter "platform-UTF-8-permissive" "platform-UTF-16")` — like `(bytes-open-converter "platform-UTF-8" "platform-UTF-16")`, but an input byte that is not part of a valid UTF-8 encoding sequence (or valid for the unpaired-surrogate extension on Windows) is effectively replaced with `\uFFFD`.
- `(bytes-open-converter "platform-UTF-16" "platform-UTF-8")` — converts UTF-16 (bytes ordered by the current platform’s endianness) to UTF-8 on Unix and Mac OS. On Windows, the conversion is the same as `(bytes-open-converter "WTF-16" "WTF-8")` to support unpaired surrogates. On Unix and Mac OS, surrogates are assumed to be paired: a pair of bytes with the bits `\x80` starts a surrogate pair, and the `\x03FF` bits are used from the pair and following pair (independent of the value of the `\xDC80` bits). On all platforms, performance may be poor when decoding from an odd offset within an input byte string.
• `(bytes-open-converter "WTF-8" "WTF-16")` — converts the WTF-8 [Sapin18] superset of UTF-8 to a superset of UTF-16 to support unpaired surrogate code units, where each UTF-16 code unit is a sequence of two bytes ordered by the current platform’s endianness.

• `(bytes-open-converter "WTF-8-permissive" "WTF-16")` — like `(bytes-open-converter "WTF-8" "WTF-16")`, but an input byte that is not part of a valid WTF-8 encoding sequence is effectively replaced with `\uFFFD`.

• `(bytes-open-converter "WTF-16" "WTF-8")` — converts the WTF-16 [Sapin18] superset of UTF-16 to the WTF-8 superset of UTF-8. The input can include UTF-16 code units that are unpaired surrogates, and the corresponding output includes an encoding of each surrogate in a natural extension of UTF-8.

A newly opened byte converter is registered with the current custodian (see §14.7 "Custodians"), so that the converter is closed when the custodian is shut down. A converter is not registered with a custodian (and does not need to be closed) if it is one of the guaranteed combinations not involving "" on Unix, or if it is any of the guaranteed combinations (including "") on Windows and Mac OS.

The set of available encodings and combinations varies by platform, depending on the `iconv` library that is installed; the `from-name` and `to-name` arguments are passed on to `iconv_open`. On Windows, "iconv.dll" or "libiconv.dll" must be in the same directory as "libmzschVERS.dll" (where `VERS` is a version number), in the user’s path, in the system directory, or in the current executable’s directory at run time, and the DLL must either supply `_errno` or link to "msvcrt.dll" for `_errno`; otherwise, only the guaranteed combinations are available.

Use `bytes-convert` with the result to convert byte strings.

Use `bytes-convert-end`.

Use `bytes-close-converter converter` to close the given converter, so that it can no longer be used with `bytes-convert` or `bytes-convert-end`.

In the Racket software distributions for Windows, a suitable "iconv.dll" is included with "libmzschVERS.dll".
(or/c bytes? exact-nonnegative-integer?)
  exact-nonnegative-integer?
  (or/c 'complete 'continues 'aborts 'error)
converter : bytes-converter?
src-bstr : bytes?
src-start-pos : exact-nonnegative-integer? = 0
src-end-pos : exact-nonnegative-integer?
  = (bytes-length src-bstr)
dest-bstr : (or/c bytes? #f) = #f
dest-start-pos : (or/c exact-nonnegative-integer? #f)
  = (and dest-bstr
      (bytes-length dest-bstr))

Converts the bytes from src-start-pos to src-end-pos in src-bstr.

If dest-bstr is not #f, the converted bytes are written into dest-bstr from dest-start-pos to dest-end-pos. If dest-bstr is #f, then a newly allocated byte string holds the conversion results, and if dest-end-pos is not #f, the size of the result byte string is no more than (- dest-end-pos dest-start-pos).

The result of bytes-convert is three values:

- result-bstr or dest-wrote-amt — a byte string if dest-bstr is #f or not provided, or the number of bytes written into dest-bstr otherwise.
- src-read-amt — the number of bytes successfully converted from src-bstr.
- 'complete, 'continues, 'aborts, or 'error — indicates how conversion terminated:
  - 'complete: The entire input was processed, and src-read-amt will be equal to (- src-end-pos src-start-pos).
  - 'continues: Conversion stopped due to the limit on the result size or the space in dest-bstr; in this case, fewer than (- dest-end-pos dest-start-pos) bytes may be returned if more space is needed to process the next complete encoding sequence in src-bstr.
  - 'aborts: The input stopped part-way through an encoding sequence, and more input bytes are necessary to continue. For example, if the last byte of input is 195 for a "UTF-8-permissive" decoding, the result is 'aborts, because another byte is needed to determine how to use the 195 byte.
  - 'error: The bytes starting at (+ src-start-pos src-read-amt) bytes in src-bstr do not form a legal encoding sequence. This result is never produced for some encodings, where all byte sequences are valid encodings. For example, since "UTF-8-permissive" handles an invalid UTF-8 sequence by dropping characters or generating "?", every byte sequence is effectively valid.
Applying a converter accumulates state in the converter (even when the third result of `bytes-convert` is `'complete`). This state can affect both further processing of input and further generation of output, but only for conversions that involve “shift sequences” to change modes within a stream. To terminate an input sequence and reset the converter, use `bytes-convert-end`.

Examples:

```scheme
> (define convert (bytes-open-converter "UTF-8" "UTF-16"))
> (bytes-convert convert (bytes 65 66 67 68))
#"\376\377\0A\0B\0C\0D"
4
'complete
> (bytes convert (bytes 195 167 195 176 195 182 194 163))
#"\303\347\303\260\303\266\302\243"
> (bytes-convert-end convert)

BYTES-CONVERT-END

### (bytes-convert-end converter
 [dest-bstr
  dest-start-pos
  dest-end-pos])

→ (or/c bytes? exact-nonnegative-integer?)
  (or/c 'complete 'continues)
converter : bytes-converter?
dest-bstr : (or/c bytes? #f) = #f
dest-start-pos : exact-nonnegative-integer? = 0
dest-end-pos : (or/c exact-nonnegative-integer? #f)
  = (and dest-bstr
       (bytes-length dest-bstr))

Like `bytes-convert`, but instead of converting bytes, this procedure generates an ending sequence for the conversion (sometimes called a “shift sequence”), if any. Few encodings use shift sequences, so this function will succeed with no output for most encodings. In any case, successful output of a (possibly empty) shift sequence resets the converter to its initial state.

The result of `bytes-convert-end` is two values:

- `result-bstr` or `dest-wrote-amt` — a byte string if `dest-bstr` is `#f` or not provided, or the number of bytes written into `dest-bstr` otherwise.
• 'complete' or 'continues' — indicates whether conversion completed. If 'complete', then an entire ending sequence was produced. If 'continues', then the conversion could not complete due to the limit on the result size or the space in dest-bstr, and the first result is either an empty byte string or 0.

(bytes-converter? v) → boolean?

v : any/c

Returns #t if v is a byte converter produced by bytes-open-converter, #f otherwise.

Examples:

> (bytes-converter? (bytes-open-converter "UTF-8" "UTF-16"))
#t
> (bytes-converter? (bytes-open-converter "whacky" "not likely"))
#f
> (define b (bytes-open-converter "UTF-8" "UTF-16"))
> (bytes-close-converter b)
> (bytes-converter? b)
#t

(locale-string-encoding) → any

Returns a string for the current locale’s encoding (i.e., the encoding normally identified by ""). See also system-language+country.

4.5.5 Additional Byte String Functions

(require racket/bytes) package: base

The bindings documented in this section are provided by the racket/bytes and racket libraries, but not racket/base.

(bytes-append* str ... strs) → bytes?
str : bytes?
strs : (listof bytes?)

Like bytes-append, but the last argument is used as a list of arguments for bytes-append, so (bytes-append* str ... strs) is the same as (apply bytes-append str ... strs). In other words, the relationship between bytes-append and bytes-append* is similar to the one between list and list*.

Examples:
> (bytes-append* #"a" #"b" '(#"c" #"d"))
#"abcd"
> (bytes-append* (cdr (append* (map (lambda (x) (list #"" x))
(lambda (x) (list #"" x))
'( #"Alpha" #"Beta" #"Gamma"))))))
#"Alpha, Beta, Gamma"

(bytes-join strs sep) → bytes?
strs : (listof bytes?)
sep : bytes?

Appends the byte strings in strs, inserting sep between each pair of bytes in strs.

Example:

> (bytes-join '( #"one" #"two" #"three" #"four" ) #" potato ")
#"one potato two potato three potato four"

4.6 Characters

Characters range over Unicode scalar values, which includes characters whose values range from \#x0 to \#x10FFFF, but not including \#xD800 to \#xDFFF. The scalar values are a subset of the Unicode code points.

Two characters are eqv? if they correspond to the same scalar value. For each scalar value less than 256, character values that are eqv? are also eq?. Characters produced by the default reader are interned in read-syntax mode.

See §1.3.14 “Reading Characters” for information on reading characters and §1.4.11 “Printing Characters” for information on printing characters.

Changed in version 6.1.1.8 of package base: Updated from Unicode 5.0.1 to Unicode 7.0.0.

4.6.1 Characters and Scalar Values

(char? v) → boolean?

v : any/c

Return #t if v is a character, #f otherwise.

(char->integer char) → exact-integer?

char : char?
Returns a character’s code-point number.

Example:

> (char->integer #\A)
65

\[
\text{(integer->char } k \text{) → char?}
\]
\[
k : \text{(and/c exact-integer?}
\]
\[
\text{(or/c (integer-in 0 55295)}
\]
\[
\text{(integer-in 57344 1114111))}
\]

Return the character whose code-point number is \( k \). For \( k \) less than 256, the result is the same object for the same \( k \).

Example:

> (integer->char 65)
#\A

\[
\text{(char-utf-8-length char) → (integer-in 1 6)}
\]
\[
\text{char : char?}
\]

Produce the same result as \( \text{(bytes-length (string->bytes/utf-8 (string char)))} \).

### 4.6.2 Character Comparisons

\[
\text{(char=? char1 char2 ...)} \rightarrow \text{boolean?}
\]
\[
\text{char1 : char?}
\]
\[
\text{char2 : char?}
\]

Returns \#t if all of the arguments are eqv?.

Examples:

> (char=? #\a #\a)
#t
> (char=? #\a #\A #\a)
#f

Changed in version 7.0.0.13 of package base: Allow one argument, in addition to allowing two or more.
(char<? char1 char2 ...) → boolean?
  char1 : char?
  char2 : char?

Returns #t if the arguments are sorted increasing, where two characters are ordered by their scalar values, #f otherwise.

Examples:
  > (char<? #\A #\a)
     #t
  > (char<? #\a #\A)
     #f
  > (char<? #\a #\b #\c)
     #t

Changed in version 7.0.0.13 of package base: Allow one argument, in addition to allowing two or more.

(char<=? char1 char2 ...) → boolean?
  char1 : char?
  char2 : char?

Like char<?, but checks whether the arguments are nondecreasing.

Examples:
  > (char<=? #\A #\a)
     #t
  > (char<=? #\a #\A)
     #f
  > (char<=? #\a #\b #\b)
     #t

Changed in version 7.0.0.13 of package base: Allow one argument, in addition to allowing two or more.

(char>? char1 char2 ...) → boolean?
  char1 : char?
  char2 : char?

Like char<?, but checks whether the arguments are decreasing.

Examples:
  > (char>? #\A #\a)
\[ (\texttt{char} > ? \ #\a \ #\A) \]
\[ \texttt{#f} \]
\[ (\texttt{char} > ? \ #\c \ #\b \ #\a) \]
\[ \texttt{#t} \]

Changed in version 7.0.0.13 of package base: Allow one argument, in addition to allowing two or more.

\[ (\texttt{char} > = ? \ char1 \ char2 \ \ldots) \rightarrow \texttt{boolean}? \]
\[ char1 : \texttt{char}? \]
\[ char2 : \texttt{char}? \]

Like \texttt{char} < ?, but checks whether the arguments are nonincreasing.

Examples:

\[ (\texttt{char} > = ? \ #\A \ #\a) \]
\[ \texttt{#f} \]
\[ (\texttt{char} > = ? \ #\a \ #\A) \]
\[ \texttt{#t} \]
\[ (\texttt{char} > = ? \ #\c \ #\b \ #\b) \]
\[ \texttt{#t} \]

Changed in version 7.0.0.13 of package base: Allow one argument, in addition to allowing two or more.

\[ (\texttt{char-ci=} ? \ char1 \ char2 \ \ldots) \rightarrow \texttt{boolean}? \]
\[ char1 : \texttt{char}? \]
\[ char2 : \texttt{char}? \]

Returns \texttt{#t} if all of the arguments are \texttt{eqv} after locale-insensitive case-folding via \texttt{char-foldcase}.

Examples:

\[ (\texttt{char-ci=} ? \ #\A \ #\a) \]
\[ \texttt{#t} \]
\[ (\texttt{char-ci=} ? \ #\a \ #\a \ #\a) \]
\[ \texttt{#t} \]

Changed in version 7.0.0.13 of package base: Allow one argument, in addition to allowing two or more.

\[ (\texttt{char-ci<} ? \ char1 \ char2 \ \ldots) \rightarrow \texttt{boolean}? \]
\[ char1 : \texttt{char}? \]
\[ char2 : \texttt{char}? \]

284
Like `char-ci?`, but checks whether the arguments would be in increasing order if each was first case-folded using `char-foldcase` (which is locale-insensitive).

Examples:

```lisp
> (char-ci? #\A #\a)
#f
> (char-ci? #\a #\b)
#t
> (char-ci? #\a #\b #\c)
#t
```

Changed in version 7.0.0.13 of package `base`: Allow one argument, in addition to allowing two or more.

```lisp
(char-ci<=? char1 char2 ...) → boolean?
char1 : char?
char2 : char?
```

Like `char-ci<=?`, but checks whether the arguments would be nondecreasing after case-folding.

Examples:

```lisp
> (char-ci<=? #\A #\a)
#t
> (char-ci<=? #\a #\A)
#t
> (char-ci<=? #\a #\b #\b)
#t
```

Changed in version 7.0.0.13 of package `base`: Allow one argument, in addition to allowing two or more.

```lisp
(char-ci>? char1 char2 ...) → boolean?
char1 : char?
char2 : char?
```

Like `char-ci>?`, but checks whether the arguments would be decreasing after case-folding.

Examples:

```lisp
> (char-ci>? #\A #\a)
#f
> (char-ci>? #\b #\A)
#t
> (char-ci>? #\c #\b #\a)
#t
```
Like `char-ci<=?`, but checks whether the arguments would be nonincreasing after case-folding.

Examples:

```scheme
> (char-ci>=? #\A #\a)
#t
> (char-ci>=? #\a #\A)
#t
> (char-ci>=? #\c #\b #\b)
#t
```

4.6.3 Classifications

```
(char-alphabetic? char) → boolean?
char : char?
```

Returns `#t` if `char` has the Unicode “Alphabetic” property.

```
(char-lower-case? char) → boolean?
char : char?
```

Returns `#t` if `char` has the Unicode “Lowercase” property.

```
(char-upper-case? char) → boolean?
char : char?
```

Returns `#t` if `char` has the Unicode “Uppercase” property.

```
(char-title-case? char) → boolean?
char : char?
```

Returns `#t` if `char`’s Unicode general category is Lt, `#f` otherwise.

```
(char-numeric? char) → boolean?
char : char?
```
Returns #t if char has a Unicode “Numeric_Type” property value that is not None.

(char-symbolic? char) → boolean?
char : char

Returns #t if char’s Unicode general category is Sm, Sc, Sk, or So, #f otherwise.

(char-punctuation? char) → boolean?
char : char

Returns #t if char’s Unicode general category is Pc, Pd, Pe, Pi, Pf, or Po, #f otherwise.

(char-graphic? char) → boolean?
char : char

Returns #t if char’s Unicode general category is Ll, Lm, Lo, Lt, Lu, Nd, Nl, No, Mn, Mc, or Me, or if one of the following produces #t when applied to char: char-alphabetic?, char-numeric?, char-symbolic?, or char-punctuation?.

(char-whitespace? char) → boolean?
char : char

Returns #t if char has the Unicode “White_Space” property.

(char-blank? char) → boolean?
char : char

Returns #t if char’s Unicode general category is Zs or if char is #\tab. (These correspond to horizontal whitespace.)

(char-iso-control? char) → boolean?
char : char

Return #t if char is between #\null and #\u001F inclusive or #\rubout and #\u009F inclusive.

(char-general-category char) → symbol?
char : char

Returns a symbol representing the character’s Unicode general category, which is 'lu, 'll, 'lt, 'lm, 'lo, 'mn, 'mc, 'me, 'nd, 'nl, 'no, 'ps, 'pe, 'pi, 'pf, 'pd, 'pc, 'po, 'sc, 'sm, 'sk, 'so, 'zs, 'zp, 'zl, 'cc, 'cf, 'cs, 'co, or 'cn.

(make-known-char-range-list)
→ (listof (list/c exact-nonnegative-integer? exact-nonnegative-integer? boolean?))
Produces a list of three-element lists, where each three-element list represents a set of consecutive code points for which the Unicode standard specifies character properties. Each three-element list contains two integers and a boolean; the first integer is a starting code-point value (inclusive), the second integer is an ending code-point value (inclusive), and the boolean is \#t when all characters in the code-point range have identical results for all of the character predicates above, have analogous transformations (shifting by the same amount, if any, in code-point space) for char-downcase, char-upcase, and char-titlecase, and have the same decomposition–normalization behavior. The three-element lists are ordered in the overall result list such that later lists represent larger code-point values, and all three-element lists are separated from every other by at least one code-point value that is not specified by Unicode.

4.6.4 Character Conversions

<table>
<thead>
<tr>
<th>(char-upcase char) → char?</th>
<th>char : char?</th>
</tr>
</thead>
</table>

Produces a character consistent with the 1-to-1 code point mapping defined by Unicode. If char has no upcase mapping, char-upcase produces char.

Examples:

> (char-upcase #\a)  #\A
> (char-upcase #\l)   #\l
> (char-upcase #\space) #\space

<table>
<thead>
<tr>
<th>(char-downcase char) → char?</th>
<th>char : char?</th>
</tr>
</thead>
</table>

Like char-upcase, but for the Unicode downcase mapping.

Examples:

> (char-downcase #\A)  #\a
> (char-downcase #\l)   #\l
> (char-downcase #\space) #\space

String procedures, such as string-upcase, handle the case where Unicode defines a locale-independent mapping from the code point to a code-point sequence (in addition to the 1-1 mapping on scalar values).
Like `char-upcase`, but for the Unicode titlecase mapping.

Examples:

```scheme
> (char-upcase #\a)
#\A
> (char-upcase #\l)
#\Λ
> (char-upcase #\space)
#\space
```

Like `char-upcase`, but for the Unicode case-folding mapping.

Examples:

```scheme
> (char-foldcase #\A)
#\a
> (char-foldcase #\Σ)
#\σ
> (char-foldcase #\ς)
#\σ
> (char-foldcase #\space)
#\space
```

4.7 Symbols

A `symbol` is like an immutable string, but symbols are normally interned, so that two symbols with the same character content are normally `eq?`. All symbols produced by the default reader (see §1.3.2 “Reading Symbols”) are interned.

The two procedures `string->uninterned-symbol` and `gensym` generate `uninterned` symbols, i.e., symbols that are not `eq?`, `eqv?`, or `equal?` to any other symbol, although they may print the same as other symbols.

The procedure `string->unreadable-symbol` returns an `unreadable symbol` that is partially interned. The default reader (see §1.3.2 “Reading Symbols”) never produces a unreadable symbol, but two calls to `string->unreadable-symbol` with `equal?` strings produce
eq? results. An unreadable symbol can print the same as an interned or uninterned symbol. Unreadable symbols are useful in expansion and compilation to avoid collisions with symbols that appear in the source; they are usually not generated directly, but they can appear in the result of functions like identifier-binding.

Interned and unreadable symbols are only weakly held by the internal symbol table. This weakness can never affect the result of an eq?, eqv?, or equal? test, but a symbol may disappear when placed into a weak box (see §16.1 “Weak Boxes”) used as the key in a weak hash table (see §4.14 “Hash Tables”), or used as an ephemeron key (see §16.2 “Ephemerons”).

See §1.3.2 “Reading Symbols” for information on reading symbols and §1.4.1 “Printing Symbols” for information on printing symbols.

(symbol? v) → boolean?
  v : any/c

Returns #t if v is a symbol, #f otherwise.

Examples:
  > (symbol? 'Apple)
  #t
  > (symbol? 10)
  #f

(symbol-interned? sym) → boolean?
  sym : symbol?

Returns #t if sym is interned, #f otherwise.

Examples:
  > (symbol-interned? 'Apple)
  #t
  > (symbol-interned? (gensym))
  #f
  > (symbol-interned? (string->unreadable-symbol "Apple"))
  #f

(symbol-unreadable? sym) → boolean?
  sym : symbol?

Returns #t if sym is an unreadable symbol, #f otherwise.

Examples:

(symbol-unreadable? 'Apple)
#f
(s symbol-unreadable? (gensym))
#f
(symbol-unreadable? (string->unreadable-symbol "Apple"))
#t

(symbol->string sym) → string?
sym : symbol?

Returns a freshly allocated mutable string whose characters are the same as in sym.
See also symbol->immutable-string from racket/symbol.

Example:
> (symbol->string 'Apple)
"Apple"

(string->symbol str) → symbol?
str : string?

Returns an interned symbol whose characters are the same as in str.

Examples:
> (string->symbol "Apple")
'Apple
> (string->symbol "1")
'|1|

(string->uninterned-symbol str) → symbol?
str : string?

Like (string->symbol str), but the resulting symbol is a new uninterned symbol. Calling string->uninterned-symbol twice with the same str returns two distinct symbols.

Examples:
> (string->uninterned-symbol "Apple")
'Apple
> (eq? 'a (string->uninterned-symbol "a"))
#f
> (eq? (string->uninterned-symbol "a")
(string->uninterned-symbol "a"))
Like `(string->symbol str)`, but the resulting symbol is a new unreadable symbol. Calling `string->unreadable-symbol` twice with equivalent `strs` returns the same symbol, but `read` never produces the symbol.

Examples:

```scheme
> (string->unreadable-symbol "Apple")
'Apple
> (eq? 'a (string->unreadable-symbol "a"))
#f
> (eq? (string->unreadable-symbol "a")
  (string->unreadable-symbol "a"))
#t
```

Returns a new uninterned symbol with an automatically-generated name. The optional `base` argument is a prefix symbol or string.

Example:

```scheme
> (gensym "apple")
'apple1116096
```

Returns `#t` if the arguments are sorted, where the comparison for each pair of symbols is the same as using `symbol->string` with `string->bytes/utf-8` and `bytes<?`.

Changed in version 7.0.0.13 of package `base`: Allow one argument, in addition to allowing two or more.

### 4.7.1 Additional Symbol Functions

```
(require racket/symbol)  ; package: base
```
The bindings documented in this section are provided by the `racket/symbol` and `racket` libraries, but not `racket/base`. Added in version 7.6 of package `base`.

```
(symbol->immutable-string sym) → (and/c string? immutable?)
sym : symbol?
```

Like `symbol->string`, but the result is an immutable string, not necessarily freshly allocated.

Examples:

```
> (symbol->immutable-string 'Apple)
"Apple"
> (immutable? (symbol->immutable-string 'Apple))
#t
```

Added in version 7.6 of package `base`.

### 4.8 Regular Expressions

Regular expressions are specified as strings or byte strings, using the same pattern language as either the Unix utility `egrep` or Perl. A string-specified pattern produces a character regexp matcher, and a byte-string pattern produces a byte regexp matcher. If a character regexp is used with a byte string or input port, it matches UTF-8 encodings (see §13.1.1 “Encodings and Locales”) of matching character streams; if a byte regexp is used with a character string, it matches bytes in the UTF-8 encoding of the string.

A regular expression that is represented as a string or byte string can be compiled to a regexp value, which can be used more efficiently by functions such as `regexp-match` compared to the string or byte string form. The `regexp` and `byte-regexp` procedures convert a string or byte string (respectively) into a regexp value using a syntax of regular expressions that is most compatible to `egrep`. The `pregexp` and `byte-pregexp` procedures produce a regexp value using a slightly different syntax of regular expressions that is more compatible with Perl.

Two regexp values are equal? if they have the same source, use the same pattern language, and are both character regexps or both byte regexps.

A literal or printed regexp value starts with #rx or #px. See §1.3.16 “Reading Regular Expressions” for information on reading regular expressions and §1.4.13 “Printing Regular Expressions” for information on printing regular expressions. Regexp values produced by the default reader are interned in `read-syntax` mode.
On the BC variant of Racket, the internal size of a regexp value is limited to 32 kilobytes; this limit roughly corresponds to a source string with 32,000 literal characters or 5,000 operators.

4.8.1 Regexp Syntax

The following syntax specifications describe the content of a string that represents a regular expression. The syntax of the corresponding string may involve extra escape characters. For example, the regular expression `(.*\1)` can be represented with the string "\".*\"\1" or the regexp constant `#rx".*\"\1"`; the \ in the regular expression must be escaped to include it in a string or regexp constant.

The regexp and pregexp syntaxes share a common core:

```
<regexp> ::= <pces>
  | <regex>\<regex>
  | <pce>\<regex>
  | <atom>\*
  | <atom>+\*
  | <atom>?

<regex> ::= <pces>

<pces> ::= <pce>\<pces>
  | <regex>\<pces>
  | <regex>\<regex>
  | <regex>\<regex>

<pce> ::= <repeat>
  | <atom>

<repeat> ::= <atom>*
  | <atom>+
  | <atom>?

<atom> ::= (\<regex>)*
  | [(\<rng>)]
  | [^\<rng>]
  | .
  | *
  | $.
  | \<look>
  | (?<\<regex>)[\pces]

<look> ::= (?<\<mode>)(\<regex>)

<rnge> ::= |
  | \<rnge>
  | \<rnge>\=\<rnge>

<rnge> ::= |
  | \<rnge>
  | \<rnge>\=\<rnge>

<lirng> ::= \<lirng>\<lirng>

<lirng> ::= \<lirng>\=\<lirng>
```

294
The following completes the grammar for \texttt{regexp}, which treats \$ and \# as literals, \% as a literal within ranges, and \texttt{\}} as a literal producer outside of ranges.

\begin{verbatim}
\langle literal \rangle ::= \text{Any character except } \langle \langle \rangle \rangle, *, _, ? \rangle, \langle \rangle, , ., \%, \\texttt{\}} \text{, or } \\texttt{\}}
| \langle aliteral \rangle Match \langle aliteral \rangle ex21
\end{verbatim}

\begin{verbatim}
\langle aliteral \rangle ::= \text{Any character}
\langle riliteral \rangle ::= \text{Any character except } \langle \langle \rangle \rangle, *, _, ? \rangle, \langle \rangle, , ., \%, \texttt{\}}
\langle rliteral \rangle ::= \text{Any character except } \langle \rangle, *, _ \rangle, or \texttt{\}}
\end{verbatim}

The following completes the grammar for \texttt{pregexp}, which uses \$ and \# bounded repetition and uses \\texttt{\}} for meta-characters both inside and outside of ranges.

\begin{verbatim}
\langle repeat \rangle ::= \ldots
| \langle atom \rangle \langle \langle n \rangle \rangle Match \langle atom \rangle \text{ exactly } \langle n \rangle \text{ times } ex7
| \langle atom \rangle \langle \langle n \rangle \rangle Match \langle atom \rangle \langle n \rangle \text{ or more times } ex8
| \langle atom \rangle \langle \langle m \rangle \rangle Match \langle atom \rangle \text{ between } 0 \text{ and } \langle m \rangle \text{ times } ex9
| \langle atom \rangle \langle \langle n \rangle \rangle \langle \langle m \rangle \rangle Match \langle atom \rangle \text{ between } \langle n \rangle \text{ and } \langle m \rangle \text{ times } ex10
| \langle atom \rangle \langle \langle > \rangle \rangle Match \langle atom \rangle 0 \text{ or more times }
\end{verbatim}

\begin{verbatim}
\langle atom \rangle ::= \ldots
| \langle n \rangle Match latest reported match for \langle n \rangle th \ \langle \rangle ex16
| \langle class \rangle Match any character in \langle class \rangle
| \langle b \rangle Match \texttt{\}} boundary ex17
| \langle B \rangle Match where \langle b \rangle does not ex18
| \langle p \langle property \rangle \rangle Match (UTF-8 encoded) in \langle property \rangle ex19
| \langle P \langle property \rangle \rangle Match (UTF-8 encoded) not in \langle property \rangle ex20
\end{verbatim}

\begin{verbatim}
\langle literal \rangle ::= \text{Any character except } \langle \langle \rangle \rangle, *, _, ? \rangle, \langle \rangle, , ., \%, \\texttt{\}} \text{, or } \\texttt{\}}
| \langle aliteral \rangle Match \langle aliteral \rangle ex21
\end{verbatim}
\langle \textit{aliteral} \rangle \ ::= \text{Any character except } a-z, A-Z, 0-9

\langle \textit{liring} \rangle \ ::= \ldots \ldots \\
| \langle \textit{class} \rangle & \langle \textit{liring} \rangle \text{ contains all characters in } \langle \textit{class} \rangle \\
| \langle \textit{posix} \rangle & \langle \textit{liring} \rangle \text{ contains all characters in } \langle \textit{posix} \rangle \text{ ex26} \\
| \text{\} } \langle \textit{eliteral} \rangle & \langle \textit{liring} \rangle \text{ contains } \langle \textit{eliteral} \rangle \\

\langle \textit{riliteral} \rangle \ ::= \text{Any character except } [ ] , \_ , \text{ or } ^ \\

\langle \textit{eliteral} \rangle \ ::= \text{Any character except } a-z, A-Z \\

\langle \textit{class} \rangle \ ::= \langle d \rangle \text{ Contains 0-9 ex23} \\
| \langle D \rangle & \text{Contains characters not in } \langle d \rangle \\
| \langle w \rangle & \text{Contains } a-z, k-Z, 0-9, _ \text{ ex24} \\
| \langle W \rangle & \text{Contains characters not in } \langle w \rangle \\
| \langle s \rangle & \text{Contains space, tab, newline, formfeed, return ex25} \\
| \langle S \rangle & \text{Contains characters not in } \langle s \rangle \\

\langle \textit{posix} \rangle \ ::= \langle :alpha: \rangle \text{ Contains } a-z, k-Z \\
| \langle :upper: \rangle & \text{Contains } A-Z \\
| \langle :lower: \rangle & \text{Contains } a-z \text{ ex26} \\
| \langle :digit: \rangle & \text{Contains } 0-9 \\
| \langle :xdigit: \rangle & \text{Contains } 0-9, a-f, A-F \\
| \langle :alnum: \rangle & \text{Contains } a-z, k-Z, 0-9 \\
| \langle :word: \rangle & \text{Contains } a-z, k-Z, 0-9, _ \\
| \langle :blank: \rangle & \text{Contains space and tab} \\
| \langle :space: \rangle & \text{Contains space, tab, newline, formfeed, return} \\
| \langle :graph: \rangle & \text{Contains all ASCII characters that use ink} \\
| \langle :print: \rangle & \text{Contains space, tab, and ASCII ink users} \\
| \langle :cntrl: \rangle & \text{Contains all characters with scalar value } < 32 \\
| \langle :ascii: \rangle & \text{Contains all ASCII characters} \\

\langle \textit{property} \rangle \ ::= \langle \textit{category} \rangle \text{ Includes all characters in } \langle \textit{category} \rangle \\
| \text{\} } \langle \textit{category} \rangle & \text{Includes all characters not in } \langle \textit{category} \rangle \\

In case-insensitive mode, a backreference of the form \textbackslash\langle n \rangle matches case-insensitively only with respect to ASCII characters.

The Unicode categories follow.

\langle \textit{category} \rangle \ ::= \langle \texttt{Li} \rangle \text{ Letter, lowercase ex19} \\
| \langle \texttt{Lu} \rangle & \text{Letter, uppercase} \\
| \langle \texttt{Lt} \rangle & \text{Letter, titlecase} \\
| \langle \texttt{Lm} \rangle & \text{Letter, modifier} \\
| \langle \texttt{Ll} \rangle & \text{Union of } \langle \texttt{Li} \rangle, \texttt{Lu}, \texttt{Lt}, \texttt{Lm} \\
| \langle \texttt{Lo} \rangle & \text{Letter, other} \\
| \langle \texttt{L} \rangle & \text{Union of } \langle \texttt{Ll} \rangle \text{ and } \texttt{Lo} \\
| \langle \texttt{Nd} \rangle & \text{Number, decimal digit} \\
| \langle \texttt{Nl} \rangle & \text{Number, letter} \\
| \langle \texttt{No} \rangle & \text{Number, other} \\
| \langle \texttt{N} \rangle & \text{Union of } \langle \texttt{Nd} \rangle, \texttt{Nl}, \texttt{No} \\

296
Examples:

> (regexp-match #rx"a\|b" "cat") ; ex1
  'a"
> (regexp-match #rx"[at]\" "cat") ; ex2
  'a"
> (regexp-match #rx"ca\[at\]" "caaat") ; ex3
  "caaat"
> (regexp-match #rx"ca\[at\]" "caaat") ; ex4
  "caaat"
> (regexp-match #rx"ca?t?" "ct") ; ex5
  "ct"
> (regexp-match #rx"ca\?[at]\" "caaat") ; ex6
  "ca"
> (regexp-match #px"ca\{2\}" "caaat") ; ex7, uses #px
  "caa"
> (regexp-match #px"ca\{2,\}t" "catcaat") ; ex8, uses #px
  "catcaat"
('"caat"
 ) (regexp-match #px"ca{,2}t" "caaatcat") ; ex9, uses #px
('"cat"
 ) (regexp-match #px"ca{1,2}t" "caaatcat") ; ex10, uses #px
('"cat"
 ) (regexp-match #rx"(c*)(a*)" "caat") ; ex11
('"caa" "c" "aa"
 ) (regexp-match #rx"[-ca]" "caaat") ; ex12
('"t"
 ) (regexp-match #rx".\b." "cat in hat") ; ex13
('"cat" "a"
 ) (regexp-match #rx"-a\-c" "cat") ; ex14
('"c"
 ) (regexp-match #rx"a\$|t\$" "cat") ; ex15
('"t"
 ) (regexp-match #px"c(.)\1t" "caat") ; ex16, uses #px
('"caat" "a"
 ) (regexp-match #px".\b." "cat in hat") ; ex17, uses #px
('"t "
 ) (regexp-match #px".\B." "cat in hat") ; ex18, uses #px
('"ca"
 ) (regexp-match #px"\p{Ll}" "Cat") ; ex19, uses #px
('"a"
 ) (regexp-match #px"\P{Ll}" "cat!") ; ex20, uses #px
('"!
 ) (regexp-match #rx"\|" "c\|t") ; ex21
('"|
 ) (regexp-match #rx"[a-f]*" "cat") ; ex22
('"ca"
 ) (regexp-match #px"[a-f\d]*" "1cat") ; ex23, uses #px
('"1ca"
 ) (regexp-match #px" \p{Ll}" "Cat") ; ex24, uses #px
('" h"
 ) (regexp-match #px"t\[\s\]" "cat\hat") ; ex25, uses #px
('"t\n"
 ) (regexp-match #px"\[[:lower:]\]+" "Cat") ; ex26, uses #px
('"at"
 ) (regexp-match #rx"\]\" "c\]t") ; ex27
('"]"
 ) (regexp-match #rx"[-]" "c-t") ; ex28
('"-"
 ) (regexp-match #rx"[a][]+" "c[a]t") ; ex29
('"[a]\"
 ) (regexp-match #rx"\[a\]+" "ca\-t") ; ex30
('"a\-"
 ) (regexp-match #rx".a(?=p)" "cat nap") ; ex31
298
4.8.2 Additional Syntactic Constraints

In addition to matching a grammar, regular expressions must meet two syntactic restrictions:

- In a \(\langle\text{repeat}\rangle\) other than \(\langle\text{atom}\rangle\), the \(\langle\text{atom}\rangle\) must not match an empty sequence.
- In a \(\langle\text{(?=\text{regexp})}\rangle\) or \(\langle\text{(?<\text{regexp})}\rangle\), the \(\langle\text{regexp}\rangle\) must match a bounded sequence only.

These constraints are checked syntactically by the following type system. A type \(\langle n, m \rangle\) corresponds to an expression that matches between \(n\) and \(m\) characters. In the rule for \(\langle\text{Regexp}\rangle\), \(N\) means the number such that the opening parenthesis is the \(N\)th opening parenthesis for collecting match reports. Non-emptiness is inferred for a backreference pattern, \(\langle N\rangle\), so that a backreference can be used for repetition patterns; in the case of mutual dependencies among backreferences, the inference chooses the fixpoint that maximizes non-emptiness. Finiteness is not inferred for backreferences (i.e., a backreference is assumed to match an arbitrarily large sequence).

\[
\begin{align*}
\langle\text{regexp}\rangle_1 & : [n_1, m_1] & \langle\text{regexp}\rangle_2 & : [n_2, m_2] \\
\langle\text{regexp}\rangle_1 \langle\text{regexp}\rangle_2 & : [\min(n_1, n_2), \max(m_1, m_2)] \\
\langle\text{pce}\rangle & : [n_1, m_1] & \langle\text{pces}\rangle & : [n_2, m_2] \\
\langle\text{pce}\rangle \langle\text{pces}\rangle & : [n_1+n_2, m_1+m_2] \\
\langle\text{repeat}\rangle & : [n, m] & \langle\text{atom}\rangle & : [n, m] & n > 0 \\
\langle\text{atom}\rangle^? & : [0, m] \\
\langle\text{atom}\rangle^+ & : [1, \infty] \\
\langle\text{atom}\rangle^* & : [n, m] & \langle\text{atom}\rangle^* & : [n^*, m^*] \\
\end{align*}
\]
\[
\langle \text{atom} \rangle : [n, m] \quad n > 0
\]
\[
\langle \text{atom} \rangle \langle \text{n} \rangle : [n^*(n), \infty]
\]
\[
\langle \text{atom} \rangle : [n, m] \quad n > 0
\]
\[
\langle \text{atom} \rangle \langle \text{m} \rangle : [0, m^*(m)]
\]
\[
\langle \text{atom} \rangle \langle \text{n}, \text{m} \rangle : [n^*(n), m^*(m)]
\]
\[
\langle \text{regexp} \rangle : [n, m]
\]
\[
\langle ?\langle \text{mode} \rangle \rangle \langle \text{regexp} \rangle : [n, m]
\]
\[
\langle \text{regexp} \rangle : [n, m]
\]
\[
\langle ?=\langle \text{regexp} \rangle \rangle : [0, 0]
\]
\[
\langle ?<\langle \text{regexp} \rangle \rangle : [0, 0]
\]
\[
\langle ?<\langle \text{regexp} \rangle \rangle : [0, 0]
\]
\[
\langle ?>\langle \text{regexp} \rangle \rangle : [n, m]
\]
\[
\langle \text{tst} \rangle : [n_0, m_0] \quad \langle \text{pces} \rangle_1 : [n_1, m_1] \quad \langle \text{pces} \rangle_2 : [n_2, m_2]
\]
\[
\langle ?\langle \text{tst} \rangle \langle \text{pces} \rangle_1 \langle \text{pces} \rangle_2 \rangle : [\min(n_1, n_2), \max(m_1, m_2)]
\]
\[
\langle \text{tst} \rangle : [n_0, m_0] \quad \langle \text{pces} \rangle : [n_1, m_1]
\]
\[
\langle ?\langle \text{tst} \rangle \langle \text{pces} \rangle \rangle : [0, m_1]
\]
\[
\langle \text{n} \rangle : [\alpha_N, \infty] \quad \langle \text{rng} \rangle : [1, 1] \quad \langle ?\langle \text{rng} \rangle \rangle : [1, 1]
\]
\[
\cdot : [1, 1] \quad \text{□} : [0, 0] \quad $ : [0, 0]
\]
\[
\langle \text{literal} \rangle : [1, 1] \quad \langle \text{n} \rangle : [\alpha_N, \infty] \quad \langle \text{class} \rangle : [1, 1]
\]
\[
\\text{\textbackslash b} : [0, 0] \quad \text{\textbackslash B} : [0, 0]
\]
\[
\text{\textbackslash p\{}\langle \text{property} \rangle\}\ : [1, 6] \quad \text{\textbackslash P\{}\langle \text{property} \rangle\}\ : [1, 6]
\]

### 4.8.3 Regexp Constructors

\[
\langle \text{regexp} \rangle \langle \text{v} \rangle \to \text{boolean?}
\]
(regexp? v) → boolean?
  v : any/c

Returns #t if v is a regexp value created by regexp or pregexp, #f otherwise.

(byte-regexp? v) → boolean?
  v : any/c

Returns #t if v is a regexp value created by byte-regexp or byte-pregexp, #f otherwise.

(byte-pregexp? v) → boolean?
  v : any/c

Returns #t if v is a regexp value created by byte-pregexp (not byte-regexp), #f otherwise.

(regexp str) → regexp?
  str : string?
(regexp str handler) → any
  str : string?
  handler : (or/c #f (string? -> any))

Takes a string representation of a regular expression (using the syntax in §4.8.1 "Regexp Syntax") and compiles it into a regexp value. Other regular expression procedures accept either a string or a regexp value as the matching pattern. If a regular expression string is used multiple times, it is faster to compile the string once to a regexp value and use it for repeated matches instead of using the string each time.

If handler is provided and not #f, it is called and its result is returned when str is not a valid representation of a regular expression; the argument to handler is a string that describes the problem with str. If handler is #f or not provided, then exn:fail:contract exception is raised.

The object-name procedure returns the source string for a regexp value.

Examples:

> (regexp "ap*le")
#rx"ap*le"
> (object-name #rx"ap*le")
"ap*le"
> (regexp "+" (λ (s) (list s)))
'("\+" follows nothing in pattern)
Changed in version 6.5.0.1 of package base: Added the handler argument.

<table>
<thead>
<tr>
<th>(pregexp str) → pregexp?</th>
</tr>
</thead>
<tbody>
<tr>
<td>str : string?</td>
</tr>
<tr>
<td>(pregexp str handler) → any</td>
</tr>
<tr>
<td>str : string?</td>
</tr>
<tr>
<td>handler : (or/c #f (string? -&gt; any))</td>
</tr>
</tbody>
</table>

Like regexp, except that it uses a slightly different syntax (see §4.8.1 “Regexp Syntax”). The result can be used with regexp-match, etc., just like the result from regexp.

Examples:

```scheme
> (pregexp "ap*le")
#px"ap*le"
> (regexp? #px"ap*le")
#t
> (pregexp "+" (λ (s) (vector s)))
"#("+" follows nothing in pattern")
```

Changed in version 6.5.0.1 of package base: Added the handler argument.

<table>
<thead>
<tr>
<th>(byte-regexp bstr) → byte-regexp?</th>
</tr>
</thead>
<tbody>
<tr>
<td>bstr : bytes?</td>
</tr>
<tr>
<td>(byte-regexp bstr handler) → any</td>
</tr>
<tr>
<td>bstr : bytes?</td>
</tr>
<tr>
<td>handler : (or/c #f (bytes? -&gt; any))</td>
</tr>
</tbody>
</table>

Takes a byte-string representation of a regular expression (using the syntax in §4.8.1 “Regexp Syntax”) and compiles it into a byte-regexp value.

If handler is provided, it is called and its result is returned if str is not a valid representation of a regular expression.

The object-name procedure returns the source byte string for a regexp value.

Examples:

```scheme
> (byte-regexp "ap*le")
#rx"ap*le"
> (object-name #rx"ap*le")
"ap*le"
> (byte-regexp "ap*le")
byte-regexp: contract violation
expected: bytes?
given: "ap*le"
```

302
> (byte-regexp "+" (λ (s) (list s)))
'("+" follows nothing in pattern")

Changed in version 6.5.0.1 of package base: Added the handler argument.

(byte-pregexp bstr) → byte-pregexp?
bstr : bytes?
(byte-pregexp bstr handler) → any
bstr : bytes?
handler : (or/c #f (bytes? -> any))

Like byte-regexp, except that it uses a slightly different syntax (see §4.8.1 "Regexp Syntax"). The result can be used with regexp-match, etc., just like the result from byte-regexp.

Examples:
> (byte-pregexp "ap*le")
#px#"ap*le"
> (byte-pregexp "+" (λ (s) (vector s)))
'("+" follows nothing in pattern")

Changed in version 6.5.0.1 of package base: Added the handler argument.

(regexp-quote str [case-sensitive?]) → string?
str : string?
  case-sensitive? : any/c = #t
(regexp-quote bstr [case-sensitive?]) → bytes?
bstr : bytes?
  case-sensitive? : any/c = #t

Produces a string or byte string suitable for use with regexp to match the literal sequence of characters in str or sequence of bytes in bstr. If case-sensitive? is true (the default), the resulting regexp matches letters in str or bytes case-sensitively, otherwise it matches case-insensitively.

Examples:
> (regexp-match "." "apple.scm")
'("a")
> (regexp-match (regexp-quote ".") "apple.scm")
'(".")

(regexp-max-lookbehind pattern) → exact-nonnegative-integer?
pattern : (or/c regexp? byte-regexp?)
Returns the maximum number of bytes that pattern may consult before the starting position of a match to determine the match. For example, the pattern `(⇐abc)d` consults three bytes preceding a matching d, while `e(⇐a...)d` consults two bytes before a matching ed. A ^ pattern may consult a preceding byte to determine whether the current position is the start of the input or of a line.

### 4.8.4 Regexp Matching

```
(regex-match pattern
  input
  [start-pos
    end-pos
    output-port
    input-prefix])
→ (if (and (or (string? pattern) (regexp? pattern))
  (or (string? input) (path? input)))
  (or/c #f (cons/c string? (listof (or/c string? #f))))
  (or/c #f (cons/c bytes? (listof (or/c bytes? #f)))))

pattern : (or/c string? bytes? regexp? byte-regexp?)
input : (or/c string? bytes? path? input-port?)
start-pos : exact-nonnegative-integer? = 0
end-pos : (or/c exact-nonnegative-integer? #f) = #f
output-port : (or/c output-port? #f) = #f
input-prefix : bytes? = #""
```

Attempts to match pattern (a string, byte string, regexp value, or byte-regexp value) once to a portion of input. The matcher finds a portion of input that matches and is closest to the start of the input (after start-pos).

If input is a path, it is converted to a byte string with path->bytes if pattern is a byte string or a byte-based regexp. Otherwise, input is converted to a string with path->string.

The optional start-pos and end-pos arguments select a portion of input for matching; the default is the entire string or the stream up to an end-of-file. When input is a string, start-pos is a character position; when input is a byte string, then start-pos is a byte position; and when input is an input port, start-pos is the number of bytes to skip before starting to match. The end-pos argument can be #f, which corresponds to the end of the string or an end-of-file in the stream; otherwise, it is a character or byte position, like start-pos. If input is an input port, and if an end-of-file is reached before start-pos bytes are skipped, then the match fails.

In pattern, a start-of-string ^ refers to the first position of input after start-pos, assuming that input-prefix is #"". The end-of-input $ refers to the end-pos{th} position or (in the case of an input port) an end-of-file, whichever comes first.
The *input-prefix* specifies bytes that effectively precede *input* for the purposes of \(^\wedge\) and other look-behind matching. For example, a \('#''\ prefix means that \(^\wedge\) matches at the beginning of the stream, while a \('#''\n''\ input-prefix means that a start-of-line \(^\wedge\) can match the beginning of the input, while a start-of-file \(^\wedge\) cannot.

If the match fails, \(#f\) is returned. If the match succeeds, a list containing strings or byte string, and possibly \(#f\), is returned. The list contains strings only if *input* is a string and *pattern* is not a byte regexp. Otherwise, the list contains byte strings (substrings of the UTF-8 encoding of *input*, if *input* is a string).

The first [byte] string in a result list is the portion of *input* that matched *pattern*. If two portions of *input* can match *pattern*, then the match that starts earliest is found.

Additional [byte] strings are returned in the list if *pattern* contains parenthesized sub-expressions (but not when the opening parenthesis is followed by ?.) Matches for the sub-expressions are provided in the order of the opening parentheses in *pattern*. When sub-expressions occur in branches of an \| “or” pattern, in a \* “zero or more” pattern, or other places where the overall pattern can succeed without a match for the sub-expression, then a \(#f\) is returned for the sub-expression if it did not contribute to the final match. When a single sub-expression occurs within a \* “zero or more” pattern or other multiple-match positions, then the rightmost match associated with the sub-expression is returned in the list.

If the optional *output-port* is provided as an output port, the part of *input* from its beginning (not *start-pos*) that precedes the match is written to the port. All of *input* up to *end-pos* is written to the port if no match is found. This functionality is most useful when *input* is an input port.

When matching an input port, a match failure reads up to *end-pos* bytes (or end-of-file), even if *pattern* begins with a start-of-string \(^\wedge\); see also regexp-try-match. On success, all bytes up to and including the match are eventually read from the port, but matching proceeds by first peeking bytes from the port (using peek-bytes-avail!), and then (re-)reading matching bytes to discard them after the match result is determined. Non-matching bytes may be read and discarded before the match is determined. The matcher peeks in blocking mode only as far as necessary to determine a match, but it may peek extra bytes to fill an internal buffer if immediately available (i.e., without blocking). Greedy repeat operators in *pattern*, such as \* or \+, tend to force reading the entire content of the port (up to *end-pos*) to determine a match.

If the input port is read simultaneously by another thread, or if the port is a custom port with inconsistent reading and peeking procedures (see §13.1.9 “Custom Ports”), then the bytes that are peeked and used for matching may be different than the bytes read and discarded after the match completes; the matcher inspects only the peeked bytes. To avoid such interleaving, use regexp-match-peek (with a progress-evt argument) followed by port-commit-peeked.

Examples:
Like `regexp-match`, but the result is a list of strings or byte strings corresponding to a sequence of matches of `pattern` in `input`.

The `pattern` is used in order to find matches, where each match attempt starts at the end of the last match, and `^` is allowed to match the beginning of the input (if `input-prefix` is `#""`) only for the first match. Empty matches are handled like other matches, returning a zero-length string or byte sequence (they are more useful in making this a complement of `regexp-split`), but `pattern` is restricted from matching an empty sequence immediately after an empty match.
If `input` contains no matches (in the range `start-pos` to `end-pos`), `null` is returned. Otherwise, each item in the resulting list is a distinct substring or byte sequence from `input` that matches `pattern`. The `end-pos` argument can be `#f` to match to the end of `input` (which corresponds to an end-of-file if `input` is an input port).

Examples:

```scheme
> (regexp-match* #rx"x." "12x4x6")
'("x4" "x6")
> (regexp-match* #rx"x*" "12x4x6")
'("" "x" "x" "x" "")
```

`match-select` specifies the collected results. The default of `car` means that the result is the list of matches without returning parenthesized sub-patterns. It can be given as a `selector` function which chooses an item from a list, or it can choose a list of items. For example, you can use `cdr` to get a list of lists of parenthesized sub-patterns matches, or `values` (as an identity function) to get the full matches as well. (Note that the selector must choose an element of its input list or a list of elements, but it must not inspect its input as they can be either a list of strings or a list of position pairs. Furthermore, the selector must be consistent in its choice(s).)

Examples:

```scheme
> (regexp-match* #rx"x(.)" "12x4x6" #:match-select cadr)
'("4" "6")
> (regexp-match* #rx"x(.)" "12x4x6" #:match-select values)
'(("x4" "4") ("x6" "6"))
```

In addition, specifying `gap-select` as a non-`#f` value will make the result an interleaved list of the matches as well as the separators between them matches, starting and ending with a separator. In this case, `match-select` can be given as `#f` to return only the separators, making such uses equivalent to `regexp-split`.

Examples:

```scheme
> (regexp-match* #rx"x(.)" "12x4x6" #:match-select cadr #:gap-select? #t)
'("12" "4" "6" "")
> (regexp-match* #rx"x(.)" "12x4x6" #:match-select #f #:gap-select? #t)
'("12" "")
```

```scheme
(regexp-try-match pattern
  input
  [start-pos
    end-pos
    output-port
    input-prefix])
```
Like `regexp-match` on input ports, except that if the match fails, no characters are read and discarded from `in`.

This procedure is especially useful with a `pattern` that begins with a start-of-string `^` or with a non-`#f` `end-pos`, since each limits the amount of peeking into the port. Otherwise, beware that a large portion of the stream may be peeked (and therefore pulled into memory) before the match succeeds or fails.

```
(regexp-match-positions pattern input [start-pos end-pos output-port input-prefix])
```

Like `regexp-match`, but returns a list of number pairs (and `#f`) instead of a list of strings. Each pair of numbers refers to a range of characters or bytes in `input`. If the result for the same arguments with `regexp-match` would be a list of byte strings, the resulting ranges correspond to byte ranges; in that case, if `input` is a character string, the byte ranges correspond to bytes in the UTF-8 encoding of the string.

Range results are returned in a `substring`- and `subbytes`-compatible manner, independent of `start-pos`. In the case of an input port, the returned positions indicate the number of bytes that were read, including `start-pos`, before the first matching byte.

Examples:
Like `regexp-match-positions`, but returns multiple matches like `regexp-match*`.

Examples:

```lisp
> (regexp-match-positions* #rx"x." "12x4x6")
'((2 . 4))
> (regexp-match-positions #rx"x." "12x4x6" 3)
'((4 . 6))
> (regexp-match-positions #rx"([0-9]+)" "a--12--345b")
'((1 . 9) (5 . 9))
```

Note that unlike `regexp-match*`, there is no `#:gap-select?` input keyword, as this information can be easily inferred from the resulting matches.
Like `regexp-match`, but returns merely `#t` when the match succeeds, `#f` otherwise.

Examples:

```
> (regexp-match? #rx"x." "12x4x6")  
#t
> (regexp-match? #rx"y." "12x4x6")  
#f
```

Like `regexp-match?`, but `#t` is only returned when the entire content of `input` matches `pattern`.

Examples:

```
> (regexp-match-exact? #rx"x." "12x4x6")  
#f
> (regexp-match-exact? #rx"1.*x." "12x4x6")  
#t
```

Like `regexp-match` on input ports, but only peeks bytes from `input` instead of reading them. Furthermore, instead of an output port, the last optional argument is a progress event.
for *input* (see *port-progress-evt*). If *progress* becomes ready, then the match stops peeking from *input* and returns #f. The *progress* argument can be #f, in which case the peek may continue with inconsistent information if another process meanwhile reads from *input*.

Examples:

```scheme
> (define p (open-input-string "a abcd"))
> (regexp-match-peek ".*bc" p)
'("a abc")
> (regexp-match-peek ".*bc" p 2)
'("abc")
> (regexp-match ".*bc" p 2)
'("abc")
> (peek-char p)
 #'\d
> (regexp-match ".*bc" p)
#f
> (peek-char p)
#'\eof
```

```scheme
(regexp-match-peek-positions pattern input
  [start-pos
   end-pos
   progress
   input-prefix])
→ (or/c (cons/c (cons/c exact-nonnegative-integer?
                  exact-nonnegative-integer?)
                  (listof (or/c (cons/c exact-nonnegative-integer?
                                 exact-nonnegative-integer?)
                                 #f)))
   #f)
pattern : (or/c string? bytes? regexp? byte-regexp?)
input : input-port?
start-pos : exact-nonnegative-integer? = 0
end-pos : (or/c exact-nonnegative-integer? #f) = #f
progress : (or/c evt #f) = #f
input-prefix : bytes? = #\"
```

Like *regexp-match-positions* on input ports, but only peeks bytes from *input* instead of reading them, and with a *progress* argument like *regexp-match-peek*.  

311
Like `regexp-match-peek`, but it attempts to match only bytes that are available from `input` without blocking. The match fails if not-yet-available characters might be used to match `pattern`.

```
(regex-match-peek-immediate  pattern
   input
   [start-pos
    end-pos
    progress
    input-prefix])
→ (or/c (cons/c bytes? (listof (or/c bytes? #f)))
    #f)

pattern : (or/c string? bytes? regexp? byte-regexp?)
input : input-port?
start-pos : exact-nonnegative-integer? = 0
end-pos : (or/c exact-nonnegative-integer? #f) = #f
progress : (or/c evt #f) = #f
input-prefix : bytes? = #""
```

Like `regexp-match-peek-positions`, but it attempts to match only bytes that are available from `input` without blocking. The match fails if not-yet-available characters might be used to match `pattern`.

```
(regex-match-peek-positions-immediate  pattern
   input
   [start-pos
    end-pos
    progress
    input-prefix])
→ (or/c (cons/c (cons/c exact-nonnegative-integer?
                 exact-nonnegative-integer?)
           (listof (or/c (cons/c exact-nonnegative-integer?
                        exact-nonnegative-integer?)
                    #f)))
    #f)

pattern : (or/c string? bytes? regexp? byte-regexp?)
input : input-port?
start-pos : exact-nonnegative-integer? = 0
end-pos : (or/c exact-nonnegative-integer? #f) = #f
progress : (or/c evt #f) = #f
input-prefix : bytes? = #""
```

Like `regexp-match-peek-positions`, but it attempts to match only bytes that are available from `input` without blocking. The match fails if not-yet-available characters might be used to match `pattern`. 
Like `regexp-match-peek-positions`, but returns multiple matches like `regexp-match-positions*`.

Like `regexp-match`, but with a second result: a byte string of up to `count` bytes that correspond to the input (possibly including the `input-prefix`) leading to the end of the match; the second result is `#f` if no match is found.

The second result can be useful as an `input-prefix` for attempting a second match on
input starting from the end of the first match. In that case, use \texttt{regexp-max-lookbehind}
to determine an appropriate value for \texttt{count}.

\begin{verbatim}
(regex-matches-end \pattern\ input
\begin{itemize}
\item \texttt{start-pos}\n\item \texttt{end-pos}\n\item \texttt{input-prefix}\n\item \texttt{count}\n\end{itemize})
\end{verbatim}

\texttt{(listof \(\text{cons/c exact-nonnegative-integer?}
\text{exact-nonnegative-integer?}\))}

\begin{verbatim}
\text{pattern : (or/c string? bytes? \texttt{regexp}? \texttt{byte-regexp}?)
\text{input : (or/c string? bytes? path? \texttt{input-port}?)
\text{start-pos : exact-nonnegative-integer? = 0
\text{end-pos : (or/c exact-nonnegative-integer? \texttt{#f}) = \texttt{#f}
\text{input-prefix : bytes? = \texttt{""}
\text{count : exact-nonnegative-integer? = 1
\end{verbatim}

\begin{verbatim}
(regex-match-peek-positions-end \pattern\ input
\begin{itemize}
\item \texttt{start-pos}\n\item \texttt{end-pos}\n\item \texttt{progress}\n\item \texttt{input-prefix}\n\item \texttt{count}\n\end{itemize})
\end{verbatim}

\texttt{(listof \(\text{cons/c (cons/c exact-nonnegative-integer?}
\text{exact-nonnegative-integer?)}\))}

\begin{verbatim}
\text{pattern : (or/c string? bytes? \texttt{regexp}? \texttt{byte-regexp}?)
\text{input : \texttt{input-port?}
\text{start-pos : exact-nonnegative-integer? = 0
\text{end-pos : (or/c exact-nonnegative-integer? \texttt{#f}) = \texttt{#f}
\text{progress : (or/c \texttt{evt \texttt{#f}) = \texttt{#f}
\text{input-prefix : bytes? = \texttt{""}
\text{count : exact-nonnegative-integer? = 1
\end{verbatim}
Like `regexp-match-positions`, etc., but with a second result like `regexp-match/end`.

### 4.8.5 Regexp Splitting

The complement of `regexp-match*`: the result is a list of strings (if `pattern` is a string or character regexp and `input` is a string) or byte strings (otherwise) from `input` that are
separated by matches to \texttt{pattern}. Adjacent matches are separated with "" or "#". Zero-length matches are treated the same as for \texttt{regexp-match*}.

If \texttt{input} contains no matches (in the range \texttt{start-pos} to \texttt{end-pos}), the result is a list containing \texttt{input}'s content (from \texttt{start-pos} to \texttt{end-pos}) as a single element. If a match occurs at the beginning of \texttt{input} (at \texttt{start-pos}), the resulting list will start with an empty string or byte string, and if a match occurs at the end (at \texttt{end-pos}), the list will end with an empty string or byte string. The \texttt{end-pos} argument can be \texttt{#f}, in which case splitting goes to the end of \texttt{input} (which corresponds to an end-of-file if \texttt{input} is an input port).

Examples:

\begin{verbatim}
> (regexp-split #rx" +" "12 34")
'("12" "34")
> (regexp-split #rx"." "12 34")
'("" "" "" "" "" "")
> (regexp-split #rx"" "" "" "")
'("" "" "" "")
> (regexp-split #rx" *" "12 34")
'("" "1" "2" "" "" "3" "4" "")
> (regexp-split #px"\b" "12, 13 and 14.")
'("" "12" ", " "13" " "and" " "14" ".")
> (regexp-split #rx" +" ")
'(""
\end{verbatim}

4.8.6 Regexp Substitution

\begin{verbatim}
(regexp-replace pattern
ingput
insert
[input-prefix])
→ (if (and (or (string? pattern) (regexp? pattern))
          (string? input))
    (string? insert)
    bytes?)

pattern : (or/c string? bytes? regexp? byte-regexp?)
ingput : (or/c string? bytes?)
insert : (or/c string? bytes?)
[input-prefix] : bytes? = """"
The `insert` argument can be either a (byte) string, or a function that returns a (byte) string. In the latter case, the function is applied on the list of values that `regexp-match` would return (i.e., the first argument is the complete match, and then one argument for each parenthesized sub-expression) to obtain a replacement (byte) string.

If `pattern` is a string or character regexp and `input` is a string, then `insert` must be a string or a procedure that accept strings, and the result is a string. If `pattern` is a byte string or byte regexp, or if `input` is a byte string, then `insert` as a string is converted to a byte string. `insert` as a procedure is called with a byte string, and the result is a byte string.

If `insert` contains &, then & is replaced with the matching portion of `input` before it is substituted into the match’s place. If `insert` contains \(n\) for some integer \(n\), then it is replaced with the \(n\)th matching sub-expression from `input`. A & and \0 are aliases. If the \(n\)th sub-expression was not used in the match, or if \(n\) is greater than the number of sub-expressions in `pattern`, then \(n\) is replaced with the empty string.

To substitute a literal & or \, use \& and \, respectively, in `insert`. A \$ in `insert` is equivalent to an empty sequence; this can be used to terminate a number \(n\) following \. If a \ in `insert` is followed by anything other than a digit, & or \$, then the \ by itself is treated as \0.

Note that the \ described in the previous paragraphs is a character or byte of `input`. To write such an `input` as a Racket string literal, an escaping \ is needed before the \. For example, the Racket constant "\1" is \1.

Examples:

```
> (regexp-replace #rx"mi" "mi casa" "su")
"su casa"
> (regexp-replace #rx"mi" "mi casa" string-upcase)
"MI casa"
> (regexp-replace #rx"([Mm])i ([a-zA-Z]*)" "Mi Casa" "\\iy \2")
"My Casa"
> (regexp-replace #rx"([Mm])i ([a-zA-Z]*)" "mi cerveza Mi Mi Mi"
"\\iy \2")
"my cerveza Mi Mi Mi"
> (regexp-replace #rx"x" "12x4x6" "\\\\")
"12\4x6"
> (display (regexp-replace #rx"x" "12x4x6" "\\\\"))
12\4x6
```

```
(regexp-replace*  pattern
    input
    insert
    [start-pos
     end-pos
     input-prefix])  →  (or/c string? bytes?)
```
Like `regexp-replace`, except that every instance of `pattern` in `input` is replaced with `insert`, instead of just the first match. Only non-overlapping instances of `pattern` in `input` are replaced, so instances of `pattern` within inserted strings are not replaced recursively. Zero-length matches are treated the same as in `regexp-match*`.

The optional `start-pos` and `end-pos` arguments select a portion of `input` for matching; the default is the entire string or the stream up to an end-of-file.

Examples:

```lisp
> (regexp-replace* #rx"([Mm])i ([a-zA-Z]*)" "mi cerveza Mi Mi Mi"
  "\1y \2")
"my cerveza My Mi Mi"
```

```lisp
> (regexp-replace* #rx"([Mm])i ([a-zA-Z]*)" "mi cerveza Mi Mi Mi"
  (lambda (all one two)
    (string-append (string-downcase one) "y"
      (string-upcase two))))
"myCERVEZA myMI Mi"
```

```lisp
> (regexp-replace* #px\w" "hello world" string-upcase 0 5)
"HELLO world"
```

```lisp
> (display (regexp-replace* #rx"x" "12x4x6" "\\\"))
12\4\6
```

(regexp-replaces input replacements) → (or/c string? bytes?)

```lisp
(input : (or/c string? bytes?)

(replacements : (listof

  (list/c (or/c string? bytes? regexp? byte-regexp?)

  (or/c string? bytes?

    ((string?) () #:rest (listof string?) . ->* . string?)

    ((bytes?) () #:rest (listof bytes?) . ->* . bytes?)))))
```

Performs a chain of `regexp-replace*` operations, where each element in `replacements` specifies a replacement as a `(list pattern replacement)`. The replacements are done in order, so later replacements can apply to previous insertions.

Examples:
> (regexp-replaces "zero-or-more?"
  '([#rx"-" "] [#rx"(.*)\?\$" "is_\1"]))
"is_zero_or_more"
> (regexp-replaces "zero-or-more?"
  '([#rx"e" "o"] [#rx"o" "oo"]))
"zooroo-oor-mooroo"

| (regexp-replace-quote str) → string? |
| str : string? |
| (regexp-replace-quote bstr) → bytes? |
| bstr : bytes? |

Produces a string suitable for use as the third argument to `regexp-replace` to insert the literal sequence of characters in `str` or `bstr` as a replacement. Concretely, every `\` and `&` in `str` or `bstr` is protected by a quoting `\`.

Examples:

> (regexp-replace #rx"UT" "Go UT!" "A&M")
"Go AUTM!"
> (regexp-replace #rx"UT" "Go UT!" (regexp-replace-quote "A&M"))
"Go A&M!"

### 4.9 Keywords

A **keyword** is like an interned symbol, but its printed form starts with `#`, and a keyword cannot be used as an identifier. Furthermore, a keyword by itself is not a valid expression, though a keyword can be quoted to form an expression that produces the symbol.

Two keywords are `eq?` if and only if they print the same (i.e., keywords are always interned).

Like symbols, keywords are only weakly held by the internal keyword table; see §4.7 “Symbols” for more information.

See §1.3.15 “Reading Keywords” for information on `reading` keywords and §1.4.12 “Printing Keywords” for information on `printing` keywords.

| (keyword? v) → boolean? |
| v : any/c |

Returns `#t` if `v` is a keyword, `#f` otherwise.

Examples:
> (keyword? '#:apple)
 #t
> (keyword? 'define)
 #f
> (keyword? '#:define)
 #t

(keyword->string keyword) → string?
keyboard : keyword?

Returns a string for the displayed form of keyword, not including the leading #\#:.

See also keyword->immutable-string from racket/keyword.

Example:

> (keyword->string '#:apple)
"apple"

(string->keyword str) → keyword?
str : string?

Returns a keyword whose displayed form is the same as that of str, but with a leading #\#.

Example:

> (string->keyword "apple")
'#:apple

(keyword<? a-keyword b-keyword ...) → boolean?
a-keyword : keyword?
b-keyword : keyword?

Returns #t if the arguments are sorted, where the comparison for each pair of keywords is the same as using keyword->string with string->bytes/utf-8 and bytes<?.

Example:

> (keyword<? '#:apple '#:banana)
 #t

Changed in version 7.0.0.13 of package base: Allow one argument, in addition to allowing two or more.
4.9.1 Additional Keyword Functions

(require racket/keyword) package: base

The bindings documented in this section are provided by the racket/keyword and racket libraries, but not racket/base.

Added in version 7.6 of package base.

(keyword->immutable-string sym) → (and/c string? immutable?)
  sym : keyword?

Like keyword->string, but the result is an immutable string, not necessarily freshly allocated.

Examples:
> (keyword->immutable-string '#:apple)
"apple"
> (immutable? (keyword->immutable-string '#:apple))
#t

Added in version 7.6 of package base.

4.10 Pairs and Lists

A pair combines exactly two values. The first value is accessed with the car procedure, and the second value is accessed with the cdr procedure. Pairs are not mutable (but see §4.11 “Mutable Pairs and Lists”).

A list is recursively defined: it is either the constant null, or it is a pair whose second value is a list.

A list can be used as a single-valued sequence (see §4.15.1 “Sequences”). The elements of the list serve as elements of the sequence. See also in-list.

Cyclic data structures can be created using only immutable pairs via read or make-reader-graph. If starting with a pair and using some number of cdrs returns to the starting pair, then the pair is not a list.

See §1.3.6 “Reading Pairs and Lists” for information on reading pairs and lists and §1.4.5 “Printing Pairs and Lists” for information on printing pairs and lists.

4.10.1 Pair Constructors and Selectors
(pair? v) → boolean?
  v : any/c

Returns #t if v is a pair, #f otherwise.

Examples:
  > (pair? 1)
  #f
  > (pair? (cons 1 2))
  #t
  > (pair? (list 1 2))
  #t
  > (pair? '(1 2))
  #t
  > (pair? '())
  #f

(null? v) → boolean?
  v : any/c

Returns #t if v is the empty list, #f otherwise.

Examples:
  > (null? 1)
  #f
  > (null? '(1 2))
  #f
  > (null? '())
  #t
  > (null? (cdr (list 1)))
  #t

(cons a d) → pair?
  a : any/c
  d : any/c

Returns a newly allocated pair whose first element is a and second element is d.

Examples:
  > (cons 1 2)
  '(1 . 2)
  > (cons 1 '())
  '(1)
(car p) → any/c
p : pair?

Returns the first element of the pair p.

Examples:
> (car '(1 2))
1
> (car (cons 2 3))
2

(cdr p) → any/c
p : pair?

Returns the second element of the pair p.

Examples:
> (cdr '(1 2))
'(2)
> (cdr '(1))
'()

null : null?

The empty list.

Examples:
> null
'()
> '()
'()
> (eq? '() null)
#t

(list? v) → boolean?
v : any/c

Returns #t if v is a list: either the empty list, or a pair whose second element is a list. This procedure effectively takes constant time due to internal caching (so that any necessary traversals of pairs can in principle count as an extra cost of allocating the pairs).

Examples:
> (list? '(1 2))
#t
> (list? (cons 1 (cons 2 '())))
#t
> (list? (cons 1 2))
#f

(list v ...) → list?
v : any/c

Returns a newly allocated list containing the vs as its elements.

Examples:
> (list 1 2 3 4)
'(1 2 3 4)
> (list (list 1 2) (list 3 4))
'((1 2) (3 4))

(list* v ... tail) → any/c
v : any/c
tail : any/c

Like list, but the last argument is used as the tail of the result, instead of the final element. The result is a list only if the last argument is a list.

Examples:
> (list* 1 2)
'(1 . 2)
> (list* 1 2 (list 3 4))
'(1 2 3 4)

(build-list n proc) → list?
n : exact-nonnegative-integer?
proc : (exact-nonnegative-integer? . -> . any)

Creates a list of n elements by applying proc to the integers from 0 to (sub1 n) in order. If lst is the resulting list, then (list-ref lst i) is the value produced by (proc i).

Examples:
> (build-list 10 values)
'(0 1 2 3 4 5 6 7 8 9)
> (build-list 5 (lambda (x) (* x x)))
'(0 1 4 9 16)
4.10.2 List Operations

(length lst) → exact-nonnegative-integer?
  lst : list?

Returns the number of elements in lst. This function takes time proportional to that length.

Examples:
> (length (list 1 2 3 4))
4
> (length '())
0

(list-ref lst pos) → any/c
  lst : pair?
  pos : exact-nonnegative-integer?

Returns the element of lst at position pos, where the list’s first element is position 0. If the list has pos or fewer elements, then the exn:fail:contract exception is raised.

The lst argument need not actually be a list; lst must merely start with a chain of at least (add1 pos) pairs.

This function takes time proportional to pos.

Examples:
> (list-ref (list 'a 'b 'c) 0)
'a
> (list-ref (list 'a 'b 'c) 1)
'b
> (list-ref (list 'a 'b 'c) 2)
'c
> (list-ref (cons 1 2) 0)
1
> (list-ref (cons 1 2) 1)
list-ref: index reaches a non-pair
  index: 1
  in: '(1 . 2)

(list-tail lst pos) → any/c
  lst : any/c
  pos : exact-nonnegative-integer?
Returns the list after the first \(pos\) elements of \(lst\). If the list has fewer than \(pos\) elements, then the \texttt{exn:fail:contract} exception is raised.

The \(lst\) argument need not actually be a list; \(lst\) must merely start with a chain of at least \(pos\) pairs.

This function takes time proportional to \(pos\).

Examples:

\[
> (\texttt{list-tail} (\texttt{list} 1 2 3 4 5) 2) \\
'(3 4 5)
\]

\[
> (\texttt{list-tail} (\texttt{cons} 1 2) 1) \\
2
\]

\[
> (\texttt{list-tail} (\texttt{cons} 1 2) 2) \\
\texttt{list-tail: index reaches a non-pair} \\
\texttt{index: 2} \\
in: '(1 2)
\]

\[
> (\texttt{list-tail} 'not-a-pair 0) \\
'not-a-pair
\]

\[
(\texttt{append} \ \texttt{lst} \ldots) \rightarrow \texttt{list?} \\
\texttt{lst: list?}
\]

\[
(\texttt{append} \ \texttt{lst} \ldots \ \texttt{v}) \rightarrow \texttt{any/c} \\
\texttt{lst: list?} \\
\texttt{v: any/c}
\]

When given all list arguments, the result is a list that contains all of the elements of the given lists in order. The last argument is used directly in the tail of the result.

The last argument need not be a list, in which case the result is an “improper list.”

This function takes time proportional to the length of all arguments (added together) except the last argument.

Examples:

\[
> (\texttt{append} (\texttt{list} 1 2) (\texttt{list} 3 4)) \\
'(1 2 3 4)
\]

\[
> (\texttt{append} (\texttt{list} 1 2) (\texttt{list} 3 4) (\texttt{list} 5 6) (\texttt{list} 7 8)) \\
'(1 2 3 4 5 6 7 8)
\]

\[
(\texttt{reverse} \ \texttt{lst}) \rightarrow \texttt{list?} \\
\texttt{lst: list?}
\]

Returns a list that has the same elements as \(lst\), but in reverse order.
This function takes time proportional to the length of lst.

Example:

> (reverse (list 1 2 3 4))
'(4 3 2 1)

### 4.10.3 List Iteration

\[
\text{map} \quad \textproc{proc} \quad \textlst{...+} \rightarrow \textlist?
\]

\[
\begin{array}{l}
\textproc{proc} : \text{procedure?} \\
\textlst{ : } \text{list?}
\end{array}
\]

Applies proc to the elements of the lsts from the first elements to the last. The proc argument must accept the same number of arguments as the number of supplied lsts, and all lsts must have the same number of elements. The result is a list containing each result of proc in order.

Examples:

> (map (lambda (number) (+ 1 number))
   '(1 2 3 4))
'(2 3 4 5)

> (map (lambda (number1 number2) (+ number1 number2))
   '(1 2 3 4)
   '(10 100 1000 10000))
'(11 102 1003 10004)

\[
\text{andmap} \quad \textproc{proc} \quad \textlst{...+} \rightarrow \text{any}
\]

\[
\begin{array}{l}
\textproc{proc} : \text{procedure?} \\
\textlst{ : } \text{list?}
\end{array}
\]

Similar to map in the sense that proc is applied to each element of lst, but

- the result is #f if any application of proc produces #f, in which case proc is not applied to later elements of the lsts; and

- the result is that of proc applied to the last elements of the lsts; more specifically, the application of proc to the last elements in the lsts is in tail position with respect to the andmap call.

If the lsts are empty, then #t is returned.

Examples:
> (andmap positive? '(1 2 3))
#t
> (andmap positive? '(1 2 a))
positive?: contract violation
  expected: real?
given: 'a
> (andmap positive? '(1 -2 a))
#f
> (andmap + '(1 2 3) '(4 5 6))
9

(ormap proc lst ...+) → any
  proc : procedure?
  lst : list?

Similar to map in the sense that proc is applied to each element of lst, but

  • the result is #f if every application of proc produces #f; and

  • the result is that of the first application of proc producing a value other than #f, in
    which case proc is not applied to later elements of the lsts; the application of proc
to the last elements of the lsts is in tail position with respect to the ormap call.

If the lsts are empty, then #f is returned.

Examples:

> (ormap eq? '(a b c) '(a b c))
#t
> (ormap positive? '(1 2 a))
#t
> (ormap + '(1 2 3) '(4 5 6))
5

(for-each proc lst ...+) → void?
  proc : procedure?
  lst : list?

Similar to map, but proc is called only for its effect, and its result (which can be any number
of values) is ignored.

Example:

> (for-each (lambda (arg)
          (printf "Got ~a\n" arg)
          23)
          '(1 2 3 4))
Like `map`, `foldl` applies a procedure to the elements of one or more lists. Whereas `map` combines the return values into a list, `foldl` combines the return values in an arbitrary way that is determined by `proc`.

If `foldl` is called with `n` lists, then `proc` must take `n+1` arguments. The extra argument is the combined return values so far. The `proc` is initially invoked with the first item of each list, and the final argument is `init`. In subsequent invocations of `proc`, the last argument is the return value from the previous invocation of `proc`. The input `lst`s are traversed from left to right, and the result of the whole `foldl` application is the result of the last application of `proc`. If the `lst`s are empty, the result is `init`.

Unlike `foldr`, `foldl` processes the `lst`s in constant space (plus the space for each call to `proc`).

Examples:

```scheme
> (foldl cons '()' '(1 2 3 4))
'(4 3 2 1)
> (foldl + 0 '(1 2 3 4))
10
> (foldl (lambda (a b result)
          (* result (- a b)))
    1
  '(1 2 3)
  '(4 5 6))
-27
```

Like `foldl`, but the lists are traversed from right to left. Unlike `foldl`, `foldr` processes the `lst`s in space proportional to the length of `lst`s (plus the space for each call to `proc`).

Examples:
4.10.4 List Filtering

(filter pred lst) → list?
  pred : procedure?
  lst : list?

Returns a list with the elements of \textit{lst} for which \textit{pred} produces a true value. The \textit{pred} procedure is applied to each element from first to last.

Example:

\[
> \text{(filter positive? '}(1 -2 3 4 -5))
\]
\[
'\(1 3 4)\n\]

(remove \textit{v} \textit{lst} \textit{[proc]}) → list?
  \textit{v} : any/c
  \textit{lst} : list?
  \textit{proc} : procedure? = equal?

Returns a list that is like \textit{lst}, omitting the first element of \textit{lst} that is equal to \textit{v} using the comparison procedure \textit{proc} (which must accept two arguments).

Examples:

\[
> \text{(remove 2 (list 1 2 3 2 4))}
\]
\[
'\(1 3 2 4)\n\]
\[
> \text{(remove 2 (list 1 2 3 2 4) =)}
\]
\[
'\(1 3 2 4)\n\]
\[
> \text{(remove '(2) (list '(1) '(2) '(3)))}
\]
\[
'\((1) (3))\n\]
\[
> \text{(remove "2" (list "1" "2" "3"))}
\]
\[
'\("1" "3")\n\]
\[
> \text{(remove \#\c (list \#\a \#\b \#\c))}
\]
\[
'\(#\a #\b)\n\]

(remq \textit{v} \textit{lst}) → list?
  \textit{v} : any/c
  \textit{lst} : list?
Returns `(remove v lst eq?)`.

Examples:

```lisp
> (remq 2 (list 1 2 3 4 5))
'(1 3 4 5)
> (remq '(2) (list '(1) '(2) '(3)))
'((1) (2) (3))
> (remq "2" (list "1" "2" "3"))
'("1" "3")
> (remq #\c (list #\a #\b #\c))
'(#\a #\b)
```

`(remv v lst) → list?`

`v : any/c`

`lst : list?`

Returns `(remove v lst eqv?)`.

Examples:

```lisp
> (remv 2 (list 1 2 3 4 5))
'(1 3 4 5)
> (remv '(2) (list '(1) '(2) '(3)))
'((1) (2) (3))
> (remv "2" (list "1" "2" "3"))
'("1" "3")
> (remv #\c (list #\a #\b #\c))
'(#\a #\b)
```

`(remove* v-lst lst [proc]) → list?`

`v-lst : list?`

`lst : list?`

`proc : procedure? = equal?`

Like `remove`, but removes from `lst` every instance of every element of `v-lst`.

Example:

```lisp
> (remove* (list 1 2) (list 1 2 3 2 4 5 2))
'(3 4 5)
```

`(remq* v-lst lst) → list?`

`v-lst : list?`

`lst : list?`
Returns \( \text{remove* v\-lst lst eq?} \).

Example:

\[
> (\text{remq* (list 1 2) (list 1 2 3 2 4 5 2)}) \\
'(3 4 5)
\]

\( \text{remv* v\-lst lst eqv?} \)

Example:

\[
> (\text{remv* (list 1 2) (list 1 2 3 2 4 5 2)}) \\
'(3 4 5)
\]

\( \text{sort lst less-than?} \)

\[
(\text{sort lst less-than?}) \\
[\#:key extract-key \#:cache-keys? cache-keys?]) \rightarrow \text{list?} \\
lst : \text{list?} \\
less-than? : (\text{any/c any/c . -> . any/c}) \\
extract-key : (\text{any/c . -> . any/c}) = (\text{lambda (x) x}) \\
cache-keys? : \text{boolean?} = \#f
\]

Returns a list sorted according to the \( \text{less-than?} \) procedure, which takes two elements of \( \text{lst} \) and returns a true value if the first is less (i.e., should be sorted earlier) than the second.

The sort is stable; if two elements of \( \text{lst} \) are “equal” (i.e., \( \text{less-than?} \) does not return a true value when given the pair in either order), then the elements preserve their relative order from \( \text{lst} \) in the output list. To preserve this guarantee, use \( \text{sort} \) with a strict comparison functions (e.g., \(<\) or \(\text{string<?}\); not \(<=\) or \(\text{string<=?}\)).

The \#:key argument \( \text{extract-key} \) is used to extract a key value for comparison from each list element. That is, the full comparison procedure is essentially

\[
(\text{lambda (x y) (less-than? (extract-key x) (extract-key y)))}
\]

By default, \( \text{extract-key} \) is applied to two list elements for every comparison, but if \( \text{cache-keys?} \) is true, then the \( \text{extract-key} \) function is used exactly once for each list item. Supply a true value for \( \text{cache-keys?} \) when \( \text{extract-key} \) is an expensive operation; for example, if \( \text{file-or-directory-modify-seconds} \) is used to extract a timestamp for

Because of the peculiar fact that the IEEE-754 number system specifies that +nan.0 is neither greater nor less than nor equal to any other number, sorting lists containing this value may produce a surprising result.
every file in a list, then `cache-keys?` should be #t to minimize file-system calls, but if `extract-key` is car, then `cache-keys?` should be #f. As another example, providing `extract-key` as `(lambda (x) (random))` and #t for `cache-keys?` effectively shuffles the list.

Examples:

```lisp
> (sort '(1 3 4 2) <)
'(1 2 3 4)
> (sort '("aardvark" "dingo" "cow" "bear") string<?)
'("aardvark" "bear" "cow" "dingo")
> (sort '(("aardvark") ("dingo") ("cow") ("bear"))
    #:key car string<?)
'(("aardvark") ("bear") ("cow") ("dingo"))
```

4.10.5 List Searching

```
(member v lst [is-equal?]) → (or/c #f list? any/c)
  v : any/c
  lst : (or/c list? any/c)
  is-equal? : (any/c any/c -> any/c) = equal?
```

Locates the first element of `lst` that is `equal?` to `v`. If such an element exists, the tail of `lst` starting with that element is returned. Otherwise, the result is #f.

The `lst` argument need not actually be a list; `lst` must merely start with a chain of pairs until a matching element is found. If no matching element is found, then `lst` must be a list (and not a cyclic list). The result can be a non-list in the case that an element is found and the returned tail of `lst` is a non-list.

Examples:

```lisp
> (member 2 (list 1 2 3 4))
'(2 3 4)
> (member 9 (list 1 2 3 4))
#f
> (member #'x (list #'x #'y) free-identifier=?)
'(#<syntax:eval:539:0 x> #<syntax:eval:539:0 y>)
> (member #'a (list #'x #'y) free-identifier=?)
#f
> (member 'b '(a b . etc))
'(b . etc)
```

```
(memv v lst) → (or/c #f list? any/c)
  v : any/c
  lst : (or/c list? any/c)
```

333
Like `member`, but finds an element using `eqv?`.

Examples:

```scheme
> (memv 2 (list 1 2 3 4))
'(2 3 4)
> (memv 9 (list 1 2 3 4))
#f
```

```
(memq v lst) → (or/c #f list? any/c)
  v : any/c
  lst : (or/c list? any/c)
```

Like `member`, but finds an element using `eq?`.

Examples:

```scheme
> (memq 2 (list 1 2 3 4))
'(2 3 4)
> (memq 9 (list 1 2 3 4))
#f
```

```
(memf proc lst) → (or/c #f list? any/c)
  proc : procedure?
  lst : list?
```

Like `member`, but finds an element using the predicate `proc`; an element is found when `proc` applied to the element returns a true value.

Example:

```scheme
> (memf (lambda (arg)
          (> arg 9))
       '(7 8 9 10 11))
'(10 11)
```

```
(findf proc lst) → any/c
  proc : procedure?
  lst : list?
```

Like `memf`, but returns the element or `#f` instead of a tail of `lst` or `#f`.

Example:
> (findf (lambda (arg)
   (> arg 9))
   '(7 8 9 10 11))
10

(assoc v lst [is-equal?]) → (or/c pair? #f)
   v : any/c
   lst : (or/c (listof pair?) any/c)
   is-equal? : (any/c any/c -> any/c) = equal?

Locates the first element of lst whose car is equal to v according to is-equal?. If such an element exists, the pair (i.e., an element of lst) is returned. Otherwise, the result is #f.

The lst argument need not actually be a list of pairs; lst must merely start with a chain of pairs contains pairs until a matching element is found. If no matching element is found, then lst must be a list of pairs (and not a cyclic list).

Examples:
> (assoc 3 (list (list 1 2) (list 3 4) (list 5 6)))
'(3 4)
> (assoc 9 (list (list 1 2) (list 3 4) (list 5 6)))
#f
> (assoc 3.5
   (list (list 1 2) (list 3 4) (list 5 6))
   (lambda (a b) (< (abs (- a b)) 1)))
'(3 4)

(assv v lst) → (or/c pair? #f)
   v : any/c
   lst : (or/c (listof pair?) any/c)

Like assoc, but finds an element using eqv?.

Example:
> (assv 3 (list (list 1 2) (list 3 4) (list 5 6)))
'(3 4)

(assq v lst) → (or/c pair? #f)
   v : any/c
   lst : (or/c (listof pair?) any/c)

Like assoc, but finds an element using eq?.

Example:
> (assq 'c (list (list 'a 'b) (list 'c 'd) (list 'e 'f)))

'(c d)

(assf proc lst) → (or/c pair? #f)
proc : procedure?
lst : (or/c (listof pair?) any/c)

Like assoc, but finds an element using the predicate proc; an element is found when proc applied to the car of an lst element returns a true value.

Example:

> (assf (lambda (arg)
            (> arg 2))
       (list (list 1 2) (list 3 4) (list 5 6)))

'(3 4)

4.10.6 Pair Accessor Shorthands

(caar v) → any/c
v : (cons/c pair? any/c)

Returns (car (car v)).

Example:

> (caar '((1 2) 3 4))
1

(cadr v) → any/c
v : (cons/c any/c pair?)

Returns (car (cdr v)).

Example:

> (cadr '((1 2) 3 4))
3

(cdar v) → any/c
v : (cons/c pair? any/c)

336
Returns \( \text{cdr (car v)} \).  

Example:  
\[
> (\text{cdar '((7 6 5 4 3 2 1) 8 9))}
\]
\[
'(6 5 4 3 2 1)
\]

\( \text{(cddr v)} \rightarrow \text{any/c} \)  
\( \text{v : (cons/c any/c pair?)} \)  

Returns \( \text{cdr (cdr v)} \).  

Example:  
\[
> (\text{cddr '(2 1))}
\]
\[
'()
\]

\( \text{(caaar v)} \rightarrow \text{any/c} \)  
\( \text{v : (cons/c (cons/c pair? any/c) any/c)} \)  

Returns \( \text{car (car (car v))} \).  

Example:  
\[
> (\text{caaar '((((6 5 4 3 2 1) 7) 8 9))})
\]
\[
6
\]

\( \text{(caadr v)} \rightarrow \text{any/c} \)  
\( \text{v : (cons/c any/c (cons/c pair? any/c))} \)  

Returns \( \text{car (car (cdr v))} \).  

Example:  
\[
> (\text{caadr '(9 (7 6 5 4 3 2 1) 8))}
\]
\[
7
\]

\( \text{(cadar v)} \rightarrow \text{any/c} \)  
\( \text{v : (cons/c (cons/c any/c pair? any/c))} \)  

Returns \( \text{car (cdr (car v))} \).  

Example:
> (cadar '((7 6 5 4 3 2 1) 8 9))
6

(caddr v) → any/c
v : (cons/c any/c (cons/c any/c pair?))

Returns (car (cdr (cdr v))).

Example:
> (caddr '(3 2 1))
1

(cdaar v) → any/c
v : (cons/c (cons/c pair? any/c) any/c)

Returns (cdr (car (car v))).

Example:
> (cdaar '(((6 5 4 3 2 1) 7) 8 9))
'(5 4 3 2 1)

(cdadr v) → any/c
v : (cons/c any/c (cons/c pair? any/c))

Returns (cdr (car (cdr v))).

Example:
> (cdadr '(9 (7 6 5 4 3 2 1) 8))
'(6 5 4 3 2 1)

(cddar v) → any/c
v : (cons/c (cons/c any/c pair?) any/c)

Returns (cdr (cdr (car v))).

Example:
> (cddar '((7 6 5 4 3 2 1) 8 9))
'(5 4 3 2 1)

(cdddr v) → any/c
v : (cons/c any/c (cons/c any/c pair?))

Returns (cdr (cdr (cdr v))).
Example:
> (cdddr '(3 2 1))
'()

(caaar v) → any/c
v : (cons/c (cons/c (cons/c pair? any/c) any/c) any/c)

Returns (car (car (car (car v)))).
Example:
> (caaar (((((5 4 3 2 1) 6) 7) 8 9)) 5

(caadr v) → any/c
v : (cons/c any/c (cons/c (cons/c pair? any/c) any/c))

Returns (car (car (car (cdr v)))).
Example:
> (caadr '((9 ((6 5 4 3 2 1) 7) 8)) 6

(caadar v) → any/c
v : (cons/c (cons/c any/c (cons/c pair? any/c)) any/c)

Returns (car (car (cdr (car v)))).
Example:
> (caadar '((7 5 4 3 2 1) 6) 8 9))
5

\[(\text{caaddr } v) \rightarrow \text{any/c} \\\n\text{v} : (\text{cons/c any/c (cons/c any/c (cons/c pair? any/c)))}\]

Returns \((\text{car (car (cdr (cdr v))))}\).
Example:

> (caaddr '((9 8 (6 5 4 3 2 1) 7))
6

\[(\text{cadaar } v) \rightarrow \text{any/c} \\\n\text{v} : (\text{cons/c (cons/c (cons/c any/c pair?) any/c) any/c})\]

Returns \((\text{car (cdr (car (car v))))}\).
Example:

> (cadaar '(((6 5 4 3 2 1) 7) 8 9))
5

\[(\text{cadadr } v) \rightarrow \text{any/c} \\\n\text{v} : (\text{cons/c any/c (cons/c (cons/c any/c pair?) any/c)})\]

Returns \((\text{car (cdr (car (cdr v))))}\).
Example:

> (cadadr '((7 6 5 4 3 2 1) 8))
6

\[(\text{caddar } v) \rightarrow \text{any/c} \\\n\text{v} : (\text{cons/c (cons/c any/c (cons/c any/c pair?)) any/c})\]

Returns \((\text{car (cdr (car (cdr v))))}\).
Example:
\[
(caddar \ '((7 \ 6 \ 5 \ 4 \ 3 \ 2 \ 1) \ 8 \ 9))
\]

\[
5
\]

\[
(caddr \ v) \rightarrow \ any/c
\]

\[
v : (cons/c \ any/c \ (cons/c \ any/c \ (cons/c \ any/c \ (cons/c \ any/c \ pair??))))
\]

Returns \((\text{car} \ (\text{cdr} \ (\text{cdr} \ (\text{cdr} \ \ v))))\).

Example:

\[
> \ (\text{caddr} \ ' (4 \ 3 \ 2 \ 1))
\]

\[
1
\]

\[
(cdaaar \ v) \rightarrow \ any/c
\]

\[
v : (cons/c \ (cons/c \ (cons/c \ (pair? \ any/c) \ any/c) \ any/c) \ any/c)
\]

Returns \((\text{cdr} \ (\text{car} \ (\text{car} \ (\text{car} \ \ v))))\).

Example:

\[
> \ (\text{cdaaar} \ '(((5 \ 4 \ 3 \ 2 \ 1) \ 6) \ 7) \ 8 \ 9))
\]

\[
'(4 \ 3 \ 2 \ 1)
\]

\[
(cdaadr \ v) \rightarrow \ any/c
\]

\[
v : (cons/c \ any/c \ (cons/c \ (cons/c \ (pair? \ any/c) \ any/c) \ any/c))
\]

Returns \((\text{cdr} \ (\text{car} \ (\text{car} \ (\text{cdr} \ \ v))))\).

Example:

\[
> \ (\text{cdaadr} \ ' (9 ((6 \ 5 \ 4 \ 3 \ 2 \ 1) \ 7) \ 8))
\]

\[
'(5 \ 4 \ 3 \ 2 \ 1)
\]

\[
(cdadar \ v) \rightarrow \ any/c
\]

\[
v : (cons/c \ (cons/c \ any/c \ (cons/c \ (pair? \ any/c)) \ any/c))
\]

Returns \((\text{cdr} \ (\text{car} \ (\text{car} \ (\text{car} \ \ v))))\).

Example:
(cdadar '(7 (5 4 3 2 1) 6) 8 9)  
'(4 3 2 1)

(cdaddr v) → any/c  
v : (cons/c any/c (cons/c any/c (cons/c pair? any/c)))

Returns (cdr (car (cdr (cdr v)))).
Example:
> (cdaddr '((9 8 (6 5 4 3 2 1) 7))  
'(5 4 3 2 1)

(cddaar v) → any/c  
v : (cons/c (cons/c (cons/c any/c pair?) any/c) any/c)

Returns (cdr (car (car (cdr v))))).
Example:
> (cddaar '(((6 5 4 3 2 1) 7) 8 9))  
'(4 3 2 1)

(cddadr v) → any/c  
v : (cons/c any/c (cons/c (cons/c any/c pair?) any/c))

Returns (cdr (car (cdr (cdr v))))).
Example:
> (cddadr '(9 (7 6 5 4 3 2 1) 8))  
'(5 4 3 2 1)

(cdddar v) → any/c  
v : (cons/c (cons/c any/c (cons/c any/c pair?) any/c))

Returns (cdr (cdr (car (cdr v))))).
Example:
> (cdddar '((7 6 5 4 3 2 1) 8 9))
'(4 3 2 1)

(cdddr v) → any/c
v : (cons/c any/c (cons/c any/c (cons/c any/c (cons/c any/c pair?)))

Returns (cdr (cdr (cdr (cdr v)))).

Example:
> (cddddr '((4 3 2 1))
'()

4.10.7 Additional List Functions and Synonyms

(require racket/list) package:base

The bindings documented in this section are provided by the racket/list and racket libraries, but not racket/base.

empty : null?

The empty list.

Examples:
> empty
'()
> (eq? empty null)
#t

(cons? v) → boolean?
v : any/c

The same as (pair? v).

Example:
> (cons? '(1 2))
#t

(empty? v) → boolean?
v : any/c
The same as (null? v).

Examples:

> (empty? '(1 2))
#f
> (empty? '())
#t

(first lst) → any/c
lst : list?

The same as (car lst), but only for lists (that are not empty).

Example:

> (first '(1 2 3 4 5 6 7 8 9 10))
1

(rest lst) → list?
lst : list?

The same as (cdr lst), but only for lists (that are not empty).

Example:

> (rest '(1 2 3 4 5 6 7 8 9 10))
'(2 3 4 5 6 7 8 9 10)

(second lst) → any
lst : list?

Returns the second element of the list.

Example:

> (second '(1 2 3 4 5 6 7 8 9 10))
2

(third lst) → any
lst : list?

Returns the third element of the list.

Example:
> (third '(1 2 3 4 5 6 7 8 9 10))
3

(fourth lst) → any
lst : list?

Returns the fourth element of the list.

Example:
> (fourth '(1 2 3 4 5 6 7 8 9 10))
4

(fifth lst) → any
lst : list?

Returns the fifth element of the list.

Example:
> (fifth '(1 2 3 4 5 6 7 8 9 10))
5

(sixth lst) → any
lst : list?

Returns the sixth element of the list.

Example:
> (sixth '(1 2 3 4 5 6 7 8 9 10))
6

(seventh lst) → any
lst : list?

Returns the seventh element of the list.

Example:
> (seventh '(1 2 3 4 5 6 7 8 9 10))
7
\[(\text{eighth} \ \text{lst}) \rightarrow \text{any} \]
\[\text{lst : list?}\]

Returns the eighth element of the list.

Example:

\[
> \ (\text{eighth} \ '1 \ 2 \ 3 \ 4 \ 5 \ 6 \ 7 \ 8 \ 9 \ 10') \\
8
\]

\[(\text{ninth} \ \text{lst}) \rightarrow \text{any} \]
\[\text{lst : list?}\]

Returns the ninth element of the list.

Example:

\[
> \ (\text{ninth} \ '1 \ 2 \ 3 \ 4 \ 5 \ 6 \ 7 \ 8 \ 9 \ 10') \\
9
\]

\[(\text{tenth} \ \text{lst}) \rightarrow \text{any} \]
\[\text{lst : list?}\]

Returns the tenth element of the list.

Example:

\[
> \ (\text{tenth} \ '1 \ 2 \ 3 \ 4 \ 5 \ 6 \ 7 \ 8 \ 9 \ 10') \\
10
\]

\[(\text{last} \ \text{lst}) \rightarrow \text{any} \]
\[\text{lst : list?}\]

Returns the last element of the list.

This function takes time proportional to the length of \text{lst}.

Example:

\[
> \ (\text{last} \ '1 \ 2 \ 3 \ 4 \ 5 \ 6 \ 7 \ 8 \ 9 \ 10') \\
10
\]

\[(\text{last-pair} \ p) \rightarrow \text{pair?} \]
\[p : \text{pair?}\]

346
Returns the last pair of a (possibly improper) list.

This function takes time proportional to the “length” of p.

Example:
> (last-pair '(1 2 3 4))
'(4)

(make-list k v) → list?
  k : exact-nonnegative-integer?
  v : any/c

Returns a newly constructed list of length k, holding v in all positions.

Example:
> (make-list 7 'foo)
'(foo foo foo foo foo foo foo)

(list-update lst pos updater) → list?
  lst : list?
  pos : (and/c (>=/c 0) (</c (length lst)))
  updater : (-> any/c any/c)

Returns a list that is the same as lst except at the specified index. The element at the specified index is (updater (list-ref lst pos)).

This function takes time proportional to pos.

Example:
> (list-update '(zero one two) 1 symbol->string)
'(zero "one" two)

Added in version 6.3 of package base.

(list-set lst pos value) → list?
  lst : list?
  pos : (and/c (>=/c 0) (</c (length lst)))
  value : any/c

Returns a list that is the same as lst except at the specified index. The element at the specified index is value.

This function takes time proportional to pos.

Example:
> (list-set '(zero one two) 2 "two")
'(zero one "two")

Added in version 6.3 of package `base`.

(index-of lst v [is-equal?]) → (or/c exact-nonnegative-integer? #f)
 lst : list?
v : any/c
is-equal? : (any/c any/c . -> . any/c) = equal?

Like `member`, but returns the index of the first element found instead of the tail of the list.

Example:
```scheme
> (index-of '(1 2 3 4) 3)
2
```

Added in version 6.7.0.3 of package `base`.

(index-where lst proc) → (or/c exact-nonnegative-integer? #f)
 lst : list?
proc : (any/c . -> . any/c)

Like `index-of` but with the predicate-searching behavior of `memf`.

Example:
```scheme
> (index-where '(1 2 3 4) even?)
1
```

Added in version 6.7.0.3 of package `base`.

(indexes-of lst v [is-equal?])
 → (listof exact-nonnegative-integer?)
 lst : list?
v : any/c
is-equal? : (any/c any/c . -> . any/c) = equal?

Like `index-of`, but returns the a list of all the indexes where the element occurs in the list instead of just the first one.

Example:
```scheme
> (indexes-of '(1 2 1 2 1) 2)
'(1 3)
```
(indexes-where lst proc) → (listof exact-nonnegative-integer?)
  lst : list?
  proc : (any/c . -> . any/c)

Like `indexes-of` but with the predicate-searching behavior of `index-where`.

Example:

```
> (indexes-where '(1 2 3 4) even?)
'(1 3)
```

(take lst pos) → list?
  lst : any/c
  pos : exact-nonnegative-integer?

Returns a fresh list whose elements are the first pos elements of lst. If lst has fewer than pos elements, the exn:fail:contract exception is raised.

The lst argument need not actually be a list; lst must merely start with a chain of at least pos pairs.

This function takes time proportional to pos.

Examples:

```
> (take '(1 2 3 4 5) 2)
'(1 2)
> (take 'non-list 0)
'()
```

(drop lst pos) → any/c
  lst : any/c
  pos : exact-nonnegative-integer?

Just like `list-tail`.

(split-at lst pos) → list? any/c
  lst : any/c
  pos : exact-nonnegative-integer?

Returns the same result as
except that it can be faster, but it will still take time proportional to \( pos \).

\[
(takef \ lst \ pred) \rightarrow \ list?
\]
\[
\begin{array}{ll}
lst : \ any/c \\
pred : \ procedure?
\end{array}
\]

Returns a fresh list whose elements are taken successively from \( lst \) as long as they satisfy \( pred \). The returned list includes up to, but not including, the first element in \( lst \) for which \( pred \) returns \#f.

The \( lst \) argument need not actually be a list; the chain of pairs in \( lst \) will be traversed until a non-pair is encountered.

Examples:

```lisp
> (takef '(2 4 5 8) even?)
'(2 4)
> (takef '(2 4 6 8) odd?)
'()
> (takef '(2 4 . 6) even?)
'(2 4)
```

\[
(dropf \ lst \ pred) \rightarrow \ any/c
\]
\[
\begin{array}{ll}
lst : \ any/c \\
pred : \ procedure?
\end{array}
\]

Drops elements from the front of \( lst \) as long as they satisfy \( pred \).

Examples:

```lisp
> (dropf '(2 4 5 8) even?)
'(5 8)
> (dropf '(2 4 6 8) odd?)
'(2 4 6 8)
```

\[
(splitf-at \ lst \ pred) \rightarrow \ list? \ any/c
\]
\[
\begin{array}{ll}
lst : \ any/c \\
pred : \ procedure?
\end{array}
\]

Returns the same result as

\[
(values \ (takef \ lst \ pred) \ (dropf \ lst \ pred))
\]
except that it can be faster.

\[
\text{(take-right } \mathit{lst} \ \mathit{pos}) \rightarrow \text{any/c} \\
\mathit{lst} : \text{any/c} \\
\mathit{pos} : \text{exact-nonnegative-integer?}
\]

Returns the \textit{list}'s \textit{pos}-length tail. If \textit{lst} has fewer than \textit{pos} elements, then the \texttt{exn:fail:contract} exception is raised.

The \textit{lst} argument need not actually be a list; \textit{lst} must merely end with a chain of at least \textit{pos} pairs.

This function takes time proportional to the length of \textit{lst}.

Examples:

\begin{verbatim}
> (take-right '(1 2 3 4 5) 2)
'(4 5)
> (take-right 'non-list 0)
'non-list
\end{verbatim}

\[
\text{(drop-right } \mathit{lst} \ \mathit{pos}) \rightarrow \text{list?} \\
\mathit{lst} : \text{any/c} \\
\mathit{pos} : \text{exact-nonnegative-integer?}
\]

Returns a fresh list whose elements are the prefix of \textit{lst}, dropping its \textit{pos}-length tail. If \textit{lst} has fewer than \textit{pos} elements, then the \texttt{exn:fail:contract} exception is raised.

The \textit{lst} argument need not actually be a list; \textit{lst} must merely end with a chain of at least \textit{pos} pairs.

This function takes time proportional to the length of \textit{lst}.

Examples:

\begin{verbatim}
> (drop-right '(1 2 3 4 5) 2)
'(1 2 3)
> (drop-right 'non-list 0)
'()
\end{verbatim}

\[
\text{(split-at-right } \mathit{lst} \ \mathit{pos}) \rightarrow \text{list? any/c} \\
\mathit{lst} : \text{any/c} \\
\mathit{pos} : \text{exact-nonnegative-integer?}
\]

Returns the same result as
(values (drop-right lst pos) (take-right lst pos))

except that it can be faster, but it will still take time proportional to the length of lst.

Examples:

> (split-at-right '(1 2 3 4 5 6) 3)
'(1 2 3)
'(4 5 6)
> (split-at-right '(1 2 3 4 5 6) 4)
'(1 2)
'(3 4 5 6)

(takef-right lst pred) → any/c
  lst : any/c
  pred : procedure?
(dropf-right lst pred) → list?
  lst : any/c
  pred : procedure?
(splitf-at-right lst pred) → list? any/c
  lst : any/c
  pred : procedure?

Like takef, dropf, and splitf-at, but combined with the from-right functionality of take-right, drop-right, and split-at-right.

(list-prefix? l r [same?]) → boolean?
  l : list?
  r : list?
  same? : (any/c any/c . -> . any/c) = equal?

True if l is a prefix of r.

Example:

> (list-prefix? '(1 2) '(1 2 3 4 5))
#t

Added in version 6.3 of package base.

(take-common-prefix l r [same?]) → list?
  l : list?
  r : list?
  same? : (any/c any/c . -> . any/c) = equal?
Returns the longest common prefix of \( l \) and \( r \).

Example:

\[
> \ (\text{take-common-prefix} \ \ '(a \ b \ c \ d) \ \ '(a \ b \ x \ y \ z)) \\
\ ' (a \ b)
\]

Added in version 6.3 of package base.

\[
\text{(drop-common-prefix} \ l \ r \ [\text{same}?]) \to \ \text{list?} \ \ \text{list?} \\
\ l : \ \text{list?} \\
\ r : \ \text{list?} \\
\ \text{same}? : \ (\text{any/c} \ \text{any/c} \ . \to \ . \ \text{any/c}) = \ \text{equal?}
\]

Returns the tails of \( l \) and \( r \) with the common prefix removed.

Example:

\[
> \ (\text{drop-common-prefix} \ \ '(a \ b \ c \ d) \ \ '(a \ b \ x \ y \ z)) \\
\ ' (c \ d) \\
\ ' (x \ y \ z)
\]

Added in version 6.3 of package base.

\[
\text{(split-common-prefix} \ l \ r \ [\text{same}?]) \to \ \text{list?} \ \text{list?} \ \text{list?} \\
\ l : \ \text{list?} \\
\ r : \ \text{list?} \\
\ \text{same}? : \ (\text{any/c} \ \text{any/c} \ . \to \ . \ \text{any/c}) = \ \text{equal?}
\]

Returns the longest common prefix together with the tails of \( l \) and \( r \) with the common prefix removed.

Example:

\[
> \ (\text{split-common-prefix} \ \ '(a \ b \ c \ d) \ \ '(a \ b \ x \ y \ z)) \\
\ ' (a \ b) \\
\ ' (c \ d) \\
\ ' (x \ y \ z)
\]

Added in version 6.3 of package base.

\[
\text{(add-between} \ \ l s t \\
\ v \ [\ #:\text{before-first} \ \text{before-first} \\
\ #:\text{before-last} \ \text{before-last} \\
\ #:\text{after-last} \ \text{after-last} \\
\ #:\text{splice?} \ \text{splice?}]) \to \ \text{list?}
\]

353
Returns a list with the same elements as `lst`, but with `v` between each pair of elements in `lst`; the last pair of elements will have `before-last` between them, instead of `v` (but `before-last` defaults to `v`).

If `splice?` is true, then `v` and `before-last` should be lists, and the list elements are spliced into the result. In addition, when `splice?` is true, `before-first` and `after-last` are inserted before the first element and after the last element respectively.

Examples:

```
> (add-between '(x y z) 'and)
'(x and y and z)
> (add-between '(x) 'and)
'(x)
> (add-between '("a" "b" "c" "d") "," #:before-last "and")
'("a" "," "b" "," "c" "and" "d")
> (add-between '(x y z) '(-) #:before-last '(- -) #:before-first '(begin) #:after-last '(end LF) #:splice? #t)
'(begin x - y - - z end LF)
```

Like `append`, but the last argument is used as a list of arguments for `append`, so `(append* lst ... lsts)` is the same as `(apply append lst ... lsts)`. In other words, the relationship between `append` and `append*` is similar to the one between `list` and `list*`.

Examples:

```
> (append* '(a) '(b) '((c) (d)))
'(a b c d)
> (cdr (append* (map (lambda (x) (list "," x)) 
                   '("Alpha" "Beta" "Gamma"))))
```
(flatten v) → list?
  v : any/c

Flattens an arbitrary S-expression structure of pairs into a list. More precisely, \( v \) is treated as a binary tree where pairs are interior nodes, and the resulting list contains all of the non-null leaves of the tree in the same order as an inorder traversal.

Examples:
> (flatten '((a) b (c (d) . e) ()))
'(a b c d e)
> (flatten 'a)
'(a)

(check-duplicates lst
  [same?
    #:key extract-key
    #:default failure-result]) → any

lst : list?
same? : (any/c any/c . -> . any/c) = equal?
extract-key : (-> any/c any/c) = (lambda (x) x)
failure-result : failure-result/c = (lambda () #f)

Returns the first duplicate item in \( lst \). More precisely, it returns the first \( x \) such that there was a previous \( y \) where \( \text{same?} (\text{extract-key } x) (\text{extract-key } y) \).

If no duplicate is found, then \( \text{failure-result} \) determines the result:

- If \( \text{failure-result} \) is a procedure, it is called (through a tail call) with no arguments to produce the result.
- Otherwise, \( \text{failure-result} \) is returned as the result.

The \( \text{same?} \) argument should be an equivalence predicate such as \( \text{equal?} \) or \( \text{eqv?} \) or a dictionary. The procedures \( \text{equal?}, \text{eqv?}, \text{and eq?} \) automatically use a dictionary for speed.

Examples:
> (check-duplicates '(1 2 3 4))
#f
> (check-duplicates '(1 2 3 2 1))
2
> (check-duplicates '(((a 1) (b 2) (a 3)) #:key car)
'(a 3)
> (check-duplicates '(1 2 3 4 5 6)
  (lambda (x y) (equal? (modulo x 3) (modulo y 3))))
4
> (check-duplicates '(1 2 3 4) #:default "no duplicates")
"no duplicates"

Added in version 6.3 of package base.
Changed in version 6.11.0.2: Added the #:default optional argument.

(remove-duplicates lst
  [same?
    #:key extract-key]) → list?

lst : list?
same? : (any/c any/c . -> . any/c) = equal?
extract-key : (any/c . -> . any/c) = (lambda (x) x)

Returns a list that has all items in lst, but without duplicate items, where same? determines whether two elements of the list are equivalent. The resulting list is in the same order as lst, and for any item that occurs multiple times, the first one is kept.

The #:key argument extract-key is used to extract a key value from each list element, so two items are considered equal if (same? (extract-key x) (extract-key y)) is true.

Examples:

> (remove-duplicates '(a b b a))
'(a b)
> (remove-duplicates '(1 2 1.0 0))
'(1 2 1.0 0)
> (remove-duplicates '(1 2 1.0 0) =)
'(1 2 0)

(filter-map proc lst ...) → list?
proc : procedure?
lst : list?

Like (map proc lst ...), except that, if proc returns #false, that element is omitted from the resulting list. In other words, filter-map is equivalent to (filter (lambda (x) x) (map proc lst ...)), but more efficient, because filter-map avoids building the intermediate list.

Example:

> (filter-map (lambda (x) (and (negative? x) (abs x))) '(1 2 -3 -4 8))
'(3 4)
(count proc lst ...+) -> exact-nonnegative-integer?
  proc : procedure?
  lst : list?

Returns (length (filter-map proc lst ...)), but without building the intermediate list.

Example:

> (count positive? '(1 -1 2 3 -2 5))
4

(partition pred lst) -> list? list?
  pred : procedure?
  lst : list?

Similar to filter, except that two values are returned: the items for which pred returns a true value, and the items for which pred returns #f.

The result is the same as

(values (filter pred lst) (filter (negate pred) lst))

but pred is applied to each item in lst only once.

Example:

> (partition even? '(1 2 3 4 5 6))
'(2 4 6)
'(1 3 5)

(range end) -> list?
  end : real?

(range start end [step]) -> list?
  start : real?
  end : real?
  step : real? = 1

Similar to in-range, but returns lists.

The resulting list holds numbers starting at start and whose successive elements are computed by adding step to their predecessor until end (excluded) is reached. If no starting point is provided, 0 is used. If no step argument is provided, 1 is used.
Like **in-range**, a **range** application can provide better performance when it appears directly in a `for` clause.

Examples:

```lisp
> (range 10)
'(0 1 2 3 4 5 6 7 8 9)
> (range 10 20)
'(10 11 12 13 14 15 16 17 18 19)
> (range 20 40 2)
'(20 22 24 26 28 30 32 34 36 38)
> (range 20 10 -1)
'(20 19 18 17 16 15 14 13 12 11)
> (range 10 15 1.5)
'(10 11.5 13.0 14.5)
```

Changed in version 6.7.0.4 of package base: Adjusted to cooperate with `for` in the same way that **in-range** does.

```lisp
(inclusive-range start end [step]) → list?

  start : real?
  end : real?
  step : real? = 1
```

Similar to **in-inclusive-range**, but returns lists.

The resulting list holds numbers starting at `start` and whose successive elements are computed by adding `step` to their predecessor until `end` (included) is reached. If no `step` argument is provided, 1 is used.

Like **in-inclusive-range**, an **inclusive-range** application can provide better performance when it appears directly in a `for` clause.

Examples:

```lisp
> (inclusive-range 10 20)
'(10 11 12 13 14 15 16 17 18 19 20)
> (inclusive-range 20 40 2)
'(20 22 24 26 28 30 32 34 36 38)
> (inclusive-range 20 10 -1)
'(20 19 18 17 16 15 14 13 12 11)
> (inclusive-range 10 15 1.5)
'(10 11.5 13.0 14.5)
```

Added in version 8.0.0.13 of package base.

```lisp
(append-map proc lst ...) → list?
```
proc : procedure?
lst : list?

Returns (append* (map proc lst ...)).

Example:

> (append-map vector->list '(#(1) #(2 3) #(4)))
'(1 2 3 4)

(filter-not pred lst) → list?
pred : (any/c . -> . any/c)
lst : list?

Like filter, but the meaning of the pred predicate is reversed: the result is a list of all items for which pred returns #f.

Example:

> (filter-not even? '(1 2 3 4 5 6))
'(1 3 5)

(shuffle lst) → list?
lst : list?

Returns a list with all elements from lst, randomly shuffled.

Examples:

> (shuffle '(1 2 3 4 5 6))
'(2 6 1 5 3 4)
> (shuffle '(1 2 3 4 5 6))
'(1 5 3 4 2 6)
> (shuffle '(1 2 3 4 5 6))
'(6 1 3 5 2 4)

(combinations lst) → list?
lst : list?
(combinations lst size) → list?
lst : list?
size : exact-nonnegative-integer?

Return a list of all combinations of elements in the input list (aka the powerset of lst). If size is given, limit results to combinations of size elements.

Examples:
(in-combinations lst size) → sequence?
  lst : list?
  size : exact-nonnegative-integer?

Returns a sequence of all combinations of elements in the input list, or all combinations of length size if size is given. Builds combinations one-by-one instead of all at once.

Examples:

> (time (begin (combinations (range 15)) (void)))
  cpu time: 27 real time: 4 gc time: 0
> (time (begin (in-combinations (range 15)) (void)))
  cpu time: 0 real time: 0 gc time: 0

(in-permutations lst) → sequence?
  lst : list?

Returns a sequence of all permutations of the input list. It is equivalent to (in-list (permutations lst)) but much faster since it builds the permutations one-by-one on each iteration. Raises an error if the input list contains more than 256 elements.

Examples:

> (in-permutations '(1 2 3))
'((1 2 3) (2 1 3) (1 3 2) (3 1 2) (2 3 1) (3 2 1))
> (in-permutations '(x x))
'((x x) (x x))
Returns the first element in the list \texttt{lst} that minimizes the result of \texttt{proc}. Signals an error on an empty list. See also \texttt{min}.

Examples:

\begin{verbatim}
> (argmin car '(((3 pears) (1 banana) (2 apples)))
  '(1 banana)
> (argmin car '(((1 banana) (1 orange)))
  '(1 banana)
\end{verbatim}

\begin{verbatim}
(argmax proc lst) \rightarrow \texttt{any/c}
    proc : (\rightarrow \texttt{any/c real?})
    lst : (\texttt{and/c pair? list?})
\end{verbatim}

Returns the first element in the list \texttt{lst} that maximizes the result of \texttt{proc}. Signals an error on an empty list. See also \texttt{max}.

Examples:

\begin{verbatim}
> (argmax car '(((3 pears) (1 banana) (2 apples)))
  '(3 pears)
> (argmax car '(((3 pears) (3 oranges)))
  '(3 pears)
\end{verbatim}

\begin{verbatim}
(group-by key lst [same?]) \rightarrow (list of list?)
    key : (\rightarrow \texttt{any/c any/c})
    lst : list?
    same? : (\texttt{any/c any/c} \rightarrow \texttt{any/c}) = \texttt{equal?}
\end{verbatim}

Groups the given list into equivalence classes, with equivalence being determined by \texttt{same?}. Within each equivalence class, \texttt{group-by} preserves the ordering of the original list. Equivalence classes themselves are in order of first appearance in the input.

Example:

\begin{verbatim}
> (group-by (lambda (x) (modulo x 3)) '(1 2 1 2 54 2 5 43 7 2 643 1 2 0))
'((1 1 43 7 643 1) (2 2 2 5 2 2) (54 0))
\end{verbatim}

Added in version 6.3 of package base.

\begin{verbatim}
(cartesian-product lst ...) \rightarrow (list of list?)
    lst : list?
\end{verbatim}

Computes the n-ary cartesian product of the given lists.

Examples:
\(\text{cartesian-product}\ (1\ 2\ 3)\ (a\ b\ c)\)
\[\{(1\ a)\ (1\ b)\ (1\ c)\ (2\ a)\ (2\ b)\ (2\ c)\ (3\ a)\ (3\ b)\ (3\ c)\}\]
\(\text{cartesian-product}\ (4\ 5\ 6)\ (d\ e\ f)\ (\#t\ \#f)\)
\[\{(4\ d\ \#t)\ (4\ e\ \#t)\ (4\ e\ \#f)\ (4\ f\ \#t)\ (4\ f\ \#f)\ (5\ d\ \#t)\ (5\ d\ \#f)\ (5\ e\ \#t)\ (5\ e\ \#f)\ (5\ f\ \#t)\ (5\ f\ \#f)\ (6\ d\ \#t)\ (6\ d\ \#f)\ (6\ e\ \#t)\ (6\ e\ \#f)\ (6\ f\ \#t)\ (6\ f\ \#f)\}\]

Added in version 6.3 of package base.

\begin{verbatim}
(remf pred lst) -> list?
  pred : procedure?
  lst : list?
\end{verbatim}

Returns a list that is like \textit{lst}, omitting the first element of \textit{lst} for which \textit{pred} produces a true value.

Example:
\begin{verbatim}
> (remf negative? '(1 -2 3 4 -5))
'(1 3 4 -5)
\end{verbatim}

Added in version 6.3 of package base.

\begin{verbatim}
(remf* pred lst) -> list?
  pred : procedure?
  lst : list?
\end{verbatim}

Like \textit{remf}, but removes all the elements for which \textit{pred} produces a true value.

Example:
> (remf* negative? '(1 -2 3 4 -5))
'(1 3 4)

Added in version 6.3 of package base.

4.10.8 Immutable Cyclic Data

\[\text{(make-reader-graph } v) \rightarrow \text{ any/c}\]
\[v : \text{ any/c}\]

Returns a value like \(v\), with placeholders created by \text{make-placeholder} replaced with the values that they contain, and with placeholders created by \text{make-hash-placeholder} with an immutable hash table. No part of \(v\) is mutated; instead, parts of \(v\) are copied as necessary to construct the resulting graph, where at most one copy is created for any given value.

Since the copied values can be immutable, and since the copy is also immutable, \text{make-reader-graph} can create cycles involving only immutable pairs, vectors, boxes, and hash tables.

Only the following kinds of values are copied and traversed to detect placeholders:

- pairs
- vectors, both mutable and immutable
- boxes, both mutable and immutable
- hash tables, both mutable and immutable
- instances of a prefab structure type
- placeholders created by \text{make-placeholder} and \text{make-hash-placeholder}

Due to these restrictions, \text{make-reader-graph} creates exactly the same sort of cyclic values as \text{read}.

Example:

> (let* ([ph (make-placeholder #f)]
[\text{x (cons 1 ph)}])
\text{placeholder-set! ph x)}
\text{(make-reader-graph x))}

#0=\'(1 . #0#)
(placeholder? v) → boolean?
 v : any/c

Returns #t if v is a placeholder created by make-placeholder, #f otherwise.

(make-placeholder v) → placeholder?
 v : any/c

Returns a placeholder for use with placeholder-set! and make-reader-graph. The v argument supplies the initial value for the placeholder.

(placeholder-set! ph datum) → void?
 ph : placeholder?
 datum : any/c

Changes the value of ph to v.

(placeholder-get ph) → any/c
 ph : placeholder?

Returns the value of ph.

(hash-placeholder? v) → boolean?
 v : any/c

Returns #t if v is a placeholder created by make-hash-placeholder, #f otherwise.

(make-hash-placeholder assocs) → hash-placeholder?
 assocs : (listof pair?)

Like make-immutable-hash, but produces a table placeholder for use with make-reader-graph.

(make-hasheq-placeholder assocs) → hash-placeholder?
 assocs : (listof pair?)

Like make-immutable-hasheq, but produces a table placeholder for use with make-reader-graph.

(make-hasheqv-placeholder assocs) → hash-placeholder?
 assocs : (listof pair?)

Like make-immutable-hasheqv, but produces a table placeholder for use with make-reader-graph.
4.11 Mutable Pairs and Lists

A mutable pair is like a pair created by cons, but it supports set-mcar! and set-mcdr! mutation operations to change the parts of the mutable pair (like traditional Lisp and Scheme pairs).

A mutable list is analogous to a list created with pairs, but instead created with mutable pairs.

A mutable pair is not a pair; they are completely separate datatypes. Similarly, a mutable list is not a list, except that the empty list is also the empty mutable list. Instead of programming with mutable pairs and mutable lists, data structures such as pairs, lists, and hash tables are practically always better choices.

A mutable list can be used as a single-valued sequence (see §4.15.1 “Sequences”). The elements of the mutable list serve as elements of the sequence. See also in-mlist.

4.11.1 Mutable Pair Constructors and Selectors

\[(\text{mpair? } v) \rightarrow \text{boolean?}\]
\[v : \text{any/c}\]

Returns #t if \(v\) is a mutable pair, #f otherwise.

\[(\text{mcons } a \ d) \rightarrow \text{mpair?}\]
\[a : \text{any/c}\]
\[d : \text{any/c}\]

Returns a newly allocated mutable pair whose first element is \(a\) and second element is \(d\).

\[(\text{mcar } p) \rightarrow \text{any/c}\]
\[p : \text{mpair?}\]

Returns the first element of the mutable pair \(p\).

\[(\text{mcdr } p) \rightarrow \text{any/c}\]
\[p : \text{mpair?}\]

Returns the second element of the mutable pair \(p\).

\[(\text{set-mcar! } p \ v) \rightarrow \text{void}\]
\[p : \text{mpair?}\]
\[v : \text{any/c}\]

Changes the mutable pair \(p\) so that its first element is \(v\).
(set-mcdr! p v) → void?
  p : mpair?
  v : any/c

Changes the mutable pair p so that its second element is v.

4.12 Vectors

A vector is a fixed-length array with constant-time access and update of the vector slots, which are numbered from 0 to one less than the number of slots in the vector.

Two vectors are equal? if they have the same length, and if the values in corresponding slots of the vectors are equal?.

A vector can be mutable or immutable. When an immutable vector is provided to a procedure like vector-set!, the exn:fail:contract exception is raised. Vectors generated by the default reader (see §1.3.7 "Reading Strings") are immutable. Use immutable? to check whether a vector is immutable.

A vector can be used as a single-valued sequence (see §4.15.1 "Sequences"). The elements of the vector serve as elements of the sequence. See also in-vector.

A literal or printed vector starts with #\(\), optionally with a number between the # and \(\). See §1.3.10 “Reading Vectors” for information on reading vectors and §1.4.7 “Printing Vectors” for information on printing vectors.

(vector? v) → boolean?
  v : any/c

Returns #t if v is a vector, #f otherwise.

(make-vector size [v]) → vector?
  size : exact-nonnegative-integer?
  v : any/c = 0

Returns a mutable vector with size slots, where all slots are initialized to contain v.

This function takes time proportional to size.

(vector v ...) → vector?
  v : any/c

Returns a newly allocated mutable vector with as many slots as provided vs, where the slots are initialized to contain the given vs in order.
(vector-immutable v ...) \rightarrow \(\text{and/c vector?}\) \n\text{immutable?}\)
\n\(v : \text{any/c}\)

Returns a newly allocated immutable vector with as many slots as provided vs, where the slots are contain the given vs in order.

(vector-length vec) \rightarrow \text{exact-nonnegative-integer?}
\nvec : \text{vector?}

Returns the length of \(vec\) (i.e., the number of slots in the vector).

This function takes constant time.

(vector-ref vec pos) \rightarrow \text{any/c}
\nvec : \text{vector?}
\npos : \text{exact-nonnegative-integer?}

Returns the element in slot \(pos\) of \(vec\). The first slot is position 0, and the last slot is one less than (vector-length vec).

This function takes constant time.

(vector-set! vec pos v) \rightarrow \text{void?}
\nvec : (\text{and/c vector?} \ (\text{not/c immutable?}))
\npos : \text{exact-nonnegative-integer?}
\nv : \text{any/c}

Updates the slot \(pos\) of \(vec\) to contain \(v\).

This function takes constant time.

(vector*-length vec) \rightarrow \text{exact-nonnegative-integer?}
\nvec : (\text{and/c vector?} \ (\text{not/c impersonator?}))
\n(vector*-ref vec pos) \rightarrow \text{any/c}
\nvec : (\text{and/c vector?} \ (\text{not/c impersonator?}))
\npos : \text{exact-nonnegative-integer?}
\n(vector*-set! vec pos v) \rightarrow \text{void?}
\nvec : (\text{and/c vector?} \ (\text{not/c immutable?}) \ (\text{not/c impersonator?}))
\npos : \text{exact-nonnegative-integer?}
\nv : \text{any/c}

Like \text{vector-length}, \text{vector-ref}, and \text{vector-set!}, but constrained to work on vectors that are not impersonators.

Added in version 6.90.0.15 of package base.
(vector-cas! vec pos old-v new-v) → boolean?
   vec : (and/c vector? (not/c immutable?) (not/c impersonator?))
   pos : exact-nonnegative-integer?
   old-v : any/c
   new-v : any/c

Compare and set operation for vectors. See box-cas!.

Added in version 6.11.0.2 of package base.

(vector->list vec) → list?
   vec : vector?

Returns a list with the same length and elements as vec.

This function takes time proportional to the size of vec.

(list->vector lst) → vector?
   lst : list?

Returns a mutable vector with the same length and elements as lst.

This function takes time proportional to the length of lst.

(vector->immutable-vector vec) → (and/c vector? immutable?)
   vec : vector?

Returns an immutable vector with the same length and elements as vec. If vec is itself immutable, then it is returned as the result.

This function takes time proportional to the size of vec when vec is mutable.

(vector-fill! vec v) → void?
   vec : (and/c vector? (not/c immutable?))
   v : any/c

Changes all slots of vec to contain v.

This function takes time proportional to the size of vec.

(vector-copy! dest dest-start src [src-start src-end]) → void?

368
dest : (and/c vector? (not/c immutable?))
derest-start : exact-nonnegative-integer?
src : vector?
src-start : exact-nonnegative-integer? = 0
src-end : exact-nonnegative-integer? = (vector-length src)

Changes the elements of dest starting at position dest-start to match the elements in src from src-start (inclusive) to src-end (exclusive). The vectors dest and src can be the same vector, and in that case the destination region can overlap with the source region; the destination elements after the copy match the source elements from before the copy. If any of dest-start, src-start, or src-end are out of range (taking into account the sizes of the vectors and the source and destination regions), the exn:fail:contract exception is raised.

This function takes time proportional to (- src-end src-start).

Examples:
> (define v (vector 'A 'p 'p 'l 'e))
> (vector-copy! v 4 #(y))
> (vector-copy! v 0 v 3 4)
> v
'#(l p p l y)

(vector->values vec [start-pos end-pos]) → any
vec : vector?
start-pos : exact-nonnegative-integer? = 0
end-pos : exact-nonnegative-integer? = (vector-length vec)

Returns end-pos - start-pos values, which are the elements of vec from start-pos (inclusive) to end-pos (exclusive). If start-pos or end-pos are greater than (vector-length vec), or if end-pos is less than start-pos, the exn:fail:contract exception is raised.

This function takes time proportional to the size of vec.
(build-vector n proc) → vector?
n : exact-nonnegative-integer?
proc : (exact-nonnegative-integer? . -> . any/c)

Creates a vector of n elements by applying proc to the integers from 0 to (sub1 n) in order. If vec is the resulting vector, then (vector-ref vec i) is the value produced by (proc i).

Example:
> (build-vector 5 add1)
'#(1 2 3 4 5)
4.12.1 Additional Vector Functions

(require racket/vector)  package: base

The bindings documented in this section are provided by the racket/vector and racket libraries, but not racket/base.

(vector-empty? v) → boolean?

v : vector?

Returns #t if v is empty (i.e. its length is 0), #f otherwise.

Added in version 7.4.0.4 of package base.

(vector-set! vec pos v ... ...) → void?

vec : (and/c vector? (not/c immutable?))
pos : exact-nonnegative-integer?
v : any/c

Updates each slot pos of vec to contain each v. The update takes place from the left so later updates overwrite earlier updates.

(vector-map proc vec ...) → vector?

proc : procedure?
vec : vector?

Applies proc to the elements of the vecs from the first elements to the last. The proc argument must accept the same number of arguments as the number of supplied vecs, and all vecs must have the same number of elements. The result is a fresh vector containing each result of proc in order.

Example:

> (vector-map + #(1 2) #(3 4))
'(4 6)

(vector-map! proc vec ...) → vector?

proc : procedure?
vec : (and/c vector? (not/c immutable?))

Like vector-map, but result of proc is inserted into the first vec at the index that the arguments to proc were taken from. The result is the first vec.

Examples:
> (define v (vector 1 2 3 4))
> (vector-map! add1 v)
'(#2 3 4 5)
> v
'(2 3 4 5)

(vector-append vec ...) → vector?
vec : vector?

Creates a fresh vector that contains all of the elements of the given vectors in order.

Example:

> (vector-append #(1 2) #(3 4))
'(1 2 3 4)

(vector-take vec pos) → vector?
vec : vector?
pos : exact-nonnegative-integer?

Returns a fresh vector whose elements are the first pos elements of vec. If vec has fewer than pos elements, then the exn:fail:contract exception is raised.

Example:

> (vector-take #(1 2 3 4) 2)
'(1 2)

(vector-take-right vec pos) → vector?
vec : vector?
pos : exact-nonnegative-integer?

Returns a fresh vector whose elements are the last pos elements of vec. If vec has fewer than pos elements, then the exn:fail:contract exception is raised.

Example:

> (vector-take-right #(1 2 3 4) 2)
'(3 4)

(vector-drop vec pos) → vector?
vec : vector?
pos : exact-nonnegative-integer?
Returns a fresh vector whose elements are the elements of \( \text{vec} \) after the first \( \text{pos} \) elements. If \( \text{vec} \) has fewer than \( \text{pos} \) elements, then the \texttt{exn:fail:contract} exception is raised.

Example:

\[
> (\text{vector-drop} \ (1 \ 2 \ 3 \ 4) \ 2) \\
\quad #'(3 \ 4)
\]

\[
(\text{vector-drop-right} \ \text{vec} \ \text{pos}) \rightarrow \text{vector}?
vec : \text{vector}?
\quad pos : \text{exact-nonnegative-integer}?
\]

Returns a fresh vector whose elements are the prefix of \( \text{vec} \), dropping its \( \text{pos} \)-length tail. If \( \text{vec} \) has fewer than \( \text{pos} \) elements, then the \texttt{exn:fail:contract} exception is raised.

Examples:

\[
> (\text{vector-drop-right} \ (1 \ 2 \ 3 \ 4) \ 1) \\
\quad #'(1 \ 2 \ 3) \\
> (\text{vector-drop-right} \ (1 \ 2 \ 3 \ 4) \ 3) \\
\quad #'(1)
\]

\[
(\text{vector-split-at} \ \text{vec} \ \text{pos}) \rightarrow \text{vector} \ \text{vector}?
vec : \text{vector}?
\quad pos : \text{exact-nonnegative-integer}?
\]

Returns the same result as

\[
(\text{values} \ (\text{vector-take} \ \text{vec} \ \text{pos}) \ (\text{vector-drop} \ \text{vec} \ \text{pos}))
\]

except that it can be faster.

Example:

\[
> (\text{vector-split-at} \ (1 \ 2 \ 3 \ 4 \ 5) \ 2) \\
\quad #'(1 \ 2) \\
\quad #'(3 \ 4 \ 5)
\]

\[
(\text{vector-split-at-right} \ \text{vec} \ \text{pos}) \rightarrow \text{vector} \ \text{vector}?
vec : \text{vector}?
\quad pos : \text{exact-nonnegative-integer}?
\]

Returns the same result as
(values (vector-take-right vec pos) (vector-drop-right vec pos))

except that it can be faster.

Example:

> (vector-split-at-right #(1 2 3 4 5) 2)
  '#(1 2 3)
  '#(4 5)

(vector-copy vec [start end]) -> vector?
vec : vector?
start : exact-nonnegative-integer? = 0
end : exact-nonnegative-integer? = (vector-length v)

Creates a fresh vector of size (- end start), with all of the elements of vec from start (inclusive) to end (exclusive).

Examples:

> (vector-copy #(1 2 3 4))
  '#(1 2 3 4)
> (vector-copy #(1 2 3 4) 3)
  '#(4)
> (vector-copy #(1 2 3 4) 2 3)
  '#(3)

(vector-filter pred vec) -> vector?
pred : procedure?
vec : vector?

Returns a fresh vector with the elements of vec for which pred produces a true value. The pred procedure is applied to each element from first to last.

Example:

> (vector-filter even? #(1 2 3 4 5 6))
  '#(2 4 6)

(vector-filter-not pred vec) -> vector?
pred : procedure?
vec : vector?

Like vector-filter, but the meaning of the pred predicate is reversed: the result is a vector of all items for which pred returns #f.

Example:
> (vector-filter-not even? #'(1 2 3 4 5 6))
'(1 3 5)

(vector-count proc vec ...) → exact-nonnegative-integer?
proc : procedure?
vec : vector?

Returns the number of elements of the vec ... (taken in parallel) on which proc does not evaluate to #f.

Examples:
> (vector-count even? #'(1 2 3 4 5))
2
> (vector-count = #'(1 2 3 4 5) #'(5 4 3 2 1))
1

(vector-argmin proc vec) → any/c
proc : (-> any/c real?)
vec : vector?

This returns the first element in the non-empty vector vec that minimizes the result of proc.

Examples:
> (vector-argmin car #((3 pears) (1 banana) (2 apples)))
'(1 banana)
> (vector-argmin car #((1 banana) (1 orange)))
'(1 banana)

(vector-argmax proc vec) → any/c
proc : (-> any/c real?)
vec : vector?

This returns the first element in the non-empty vector vec that maximizes the result of proc.

Examples:
> (vector-argmax car #((3 pears) (1 banana) (2 apples)))
'(3 pears)
> (vector-argmax car #((3 pears) (3 oranges)))
'(3 pears)
(vector-member v vec) \to (or/c natural-number/c #f)
  v : any/c
  vec : vector?

Locates the first element of vec that is equal? to v. If such an element exists, the index of that element in vec is returned. Otherwise, the result is #f.

Examples:
  > (vector-member 2 (vector 1 2 3 4))
  1
  > (vector-member 9 (vector 1 2 3 4))
  #f

(vector-memv v vec) \to (or/c natural-number/c #f)
  v : any/c
  vec : vector?

Like vector-member, but finds an element using eqv?.

Examples:
  > (vector-memv 2 (vector 1 2 3 4))
  1
  > (vector-memv 9 (vector 1 2 3 4))
  #f

(vector-memq v vec) \to (or/c natural-number/c #f)
  v : any/c
  vec : vector?

Like vector-member, but finds an element using eq?.

Examples:
  > (vector-memq 2 (vector 1 2 3 4))
  1
  > (vector-memq 9 (vector 1 2 3 4))
  #f

(vector-sort vec
  less-than?
  [start
  end
  #:key key
  #:cache-keys? cache-keys?]) \to vector?
Like `sort`, but operates on vectors; a fresh vector of length \((- \text{end} \text{start})\) is returned containing the elements from indices `start` (inclusive) through `end` (exclusive) of `vec`, but in sorted order (i.e., `vec` is not modified). This sort is stable (i.e., the order of “equal” elements is preserved).

Examples:

```scheme
> (define v1 (vector 4 3 2 1))
> (vector-sort v1 <)
'(1 2 3 4)
> v1
'(4 3 2 1)
> (define v2 (vector '(4) '(3) '(2) '(1)))
> (vector-sort v2 < 1 3 #:key car)
'(2 (3))
> v2
'(4 (3) (2) (1))
```

Added in version 6.6.0.5 of package `base`.

Like `vector-sort`, but updates indices `start` (inclusive) through `end` (exclusive) of `vec` by sorting them according to the `less-than?` procedure.

Examples:

```scheme
> (define v1 (vector 4 3 2 1))
```
> `(vector-sort! v1 `<)
> v1
> '#(1 2 3 4)
> `(define v2 (vector '(4) '(3) '(2) '(1)))
> `(vector-sort! v2 `< 1 3 #:key car)
> v2
> '#((4) (2) (3) (1))

Added in version 6.6.0.5 of package base.

### 4.13 Boxes

A box is like a single-element vector, normally used as minimal mutable storage.

A box can be mutable or immutable. When an immutable box is provided to a procedure like `set-box!`, the `exn:fail:contract` exception is raised. Box constants generated by the default reader (see §1.3.7 “Reading Strings”) are immutable. Use `immutable?` to check whether a box is immutable.

A literal or printed box starts with `#&`. See §1.3.13 “Reading Boxes” for information on reading boxes and §1.4.10 “Printing Boxes” for information on printing boxes.

```
(box? v) → boolean?
| v : any/c
```

Returns `#t` if `v` is a box, `#f` otherwise.

```
(box v) → box?
| v : any/c
```

Returns a new mutable box that contains `v`.

```
(box-immutable v) → (and/c box? immutable?)
| v : any/c
```

Returns a new immutable box that contains `v`.

```
(unbox box) → any/c
| box : box?
```

Returns the content of `box`.

For any `v`, `(unbox (box v))` and `(unbox (box-immutable v))` returns `v`. 
(set-box! box v) → void?
box : (and/c box? (not/c immutable?))
v : any/c

Sets the content of box to v.

(unbox* box) → any/c
box : (and box? (not/c impersonator?))

(set-box*! box v) → void?
box : (and/c box? (not/c immutable?) (not/c impersonator?))
v : any/c

Like unbox and set-box!, but constrained to work on boxes that are not impersonators.

Added in version 6.90.0.15 of package base.

(box-cas! box old new) → boolean?
box : (and/c box? (not/c immutable?) (not/c impersonator?))
old : any/c
new : any/c

Atomically updates the contents of box to new, provided that box currently contains a value that is eq? to old, and returns #t in that case. If box does not contain old, then the result is #f.

If no other threads or futures attempt to access box, the operation is equivalent to

(and (eq? old (unbox box)) (set-box! box new) #t)

except that box-cas! can spuriously fail on some platforms. That is, with low probability, the result can be #f with the value in box left unchanged, even if box contains old.

When Racket is compiled with support for futures, box-cas! is guaranteed to use a hard-ware compare and set operation. Uses of box-cas! be performed safely in a future (i.e., allowing the future thunk to continue in parallel). See also §11.7 “Machine Memory Order”

4.14 Hash Tables

A hash table (or simply hash) maps each of its keys to a single value. For a given hash table, keys are equivalent via equal?, eqv?, or eq?, and keys are retained either strongly, weakly (see §16.1 “Weak Boxes”), or like ephemerons. A hash table is also either mutable or immutable. Immutable hash tables support effectively constant-time access and update, just

§3.10 “Hash Tables” in The Racket Guide introduces hash tables.
like mutable hash tables; the constant on immutable operations is usually larger, but the func-
tional nature of immutable hash tables can pay off in certain algorithms. Use immutable? to check whether a hash table is immutable.

For equal?-based hashing, the built-in hash functions on strings, pairs, lists, vectors, prefab or transparent structures, etc., take time proportional to the size of the value. The hash code for a compound data structure, such as a list or vector, depends on hashing each item of the container, but the depth of such recursive hashing is limited (to avoid potential problems with cyclic data). For a non-list pair, both car and cdr hashing is treated as a deeper hash, but the cdr of a list is treated as having the same hashing depth as the list.

A hash table can be used as a two-valued sequence (see §4.15.1 “Sequences”). The keys and values of the hash table serve as elements of the sequence (i.e., each element is a key and its associated value). If a mapping is added to or removed from the hash table during iteration, then an iteration step may fail with exn:fail:contract, or the iteration may skip or duplicate keys and values. See also in-hash, in-hash-keys, in-hash-values, and in-hash-pairs.

Two hash tables cannot be equal? unless they have the same mutability, use the same key-comparison procedure (equal?, eqv?, or eq?), both hold keys strongly, weakly, or like ephemeron. Empty immutable hash tables are eq? when they are equal?.

Changed in version 7.2.0.9 of package base: Made empty immutable hash tables eq? when they are equal?.

Caveats concerning concurrent modification: A mutable hash table can be manipulated with hash-ref, hash-set!, and hash-remove! concurrently by multiple threads, and the operations are protected by a table-specific semaphore as needed. Several caveats apply, however:

• If a thread is terminated while applying hash-ref, hash-ref-key, hash-set!, hash-remove!, hash-ref!, hash-update!, or hash-clear! to a hash table that uses equal? or eqv? key comparisons, all current and future operations on the hash table may block indefinitely.

• The hash-map, hash-for-each, and hash-clear! procedures do not use the table’s semaphore to guard the traversal as a whole (if a traversal is needed, in the case of hash-clear!). Changes by one thread to a hash table can affect the keys and values seen by another thread part-way through its traversal of the same hash table.

• The hash-update! and hash-ref! functions use a table’s semaphore independently for the hash-ref and hash-set! parts of their functionality, which means that the update as a whole is not “atomic.”

• Adding a mutable hash table as a key in itself is trouble on the grounds that the key is being mutated (see the caveat below), but it is also a kind of concurrent use of the hash table: computing a hash table’s hash code may require waiting on the table’s semaphore, but the semaphore is already held for modifying the hash table, so the hash-table addition can block indefinitely.
Caveat concerning mutable keys: If a key in an `equal?-`-based hash table is mutated (e.g., a key string is modified with `string-set!`), then the hash table’s behavior for insertion and lookup operations becomes unpredictable.

A literal or printed hash table starts with `#hash`, `#hasheqv`, or `#hasheq`. See §1.3.12 “Reading Hash Tables” for information on reading hash tables and §1.4.9 “Printing Hash Tables” for information on printing hash tables.

```
(hash? v) -> boolean?
  v : any/c
```

Returns `#t` if `v` is a hash table, `#f` otherwise.

```
(hash-equal? hash) -> boolean?
  hash : hash?
```

Returns `#t` if `hash` compares keys with `equal?`, `#f` if it compares with `eq?` or `eqv?`.

```
(hash-eqv? hash) -> boolean?
  hash : hash?
```

Returns `#t` if `hash` compares keys with `eqv?`, `#f` if it compares with `equal?` or `eq?`.

```
(hash-eq? hash) -> boolean?
  hash : hash?
```

Returns `#t` if `hash` compares keys with `eq?`, `#f` if it compares with `equal?` or `eqv?`.

```
(hash-strong? hash) -> boolean?
  hash : hash?
```

Returns `#t` if `hash` retains its keys strongly, `#f` if it retains keys strongly or like ephemerons.

Added in version 8.0.0.10 of package `base`.

```
(hash-weak? hash) -> boolean?
  hash : hash?
```

Returns `#t` if `hash` retains its keys weakly, `#f` if it retains keys strongly or like ephemerons.

```
(hash-ephemeron? hash) -> boolean?
  hash : hash?
```

Returns `#t` if `hash` retains its keys like ephemerons, `#f` if it retains keys strongly or merely weakly.

Added in version 8.0.0.10 of package `base`.
(hash key val ... ...) → (and/c hash? hash-equal? immutable?)
  key : any/c
  val : any/c
(hasheq key val ... ...) → (and/c hash? hash-eq? immutable?)
  key : any/c
  val : any/c
(hasheqv key val ... ...) → (and/c hash? hash-eqv? immutable?)
  key : any/c
  val : any/c

Creates an immutable hash table with each given key mapped to the following val; each key must have a val, so the total number of arguments to hash must be even.

The hash procedure creates a table where keys are compared with equal?, hasheq procedure creates a table where keys are compared with eq?, and hasheqv procedure creates a table where keys are compared with eqv?.

The key to val mappings are added to the table in the order that they appear in the argument list, so later mappings can hide earlier mappings if the keys are equal.

(make-hash [assocs]) → (and/c hash? hash-equal?)
  assocs : (listof pair?) = null
(make-hasheq [assocs]) → (and/c hash? hash-eq?)
  assocs : (listof pair?) = null
(make-hasheqv [assocs]) → (and/c hash? hash-eqv?)
  assocs : (listof pair?) = null

Creates a mutable hash table that holds keys strongly.

The make-hash procedure creates a table where keys are compared with equal?, make-hasheq procedure creates a table where keys are compared with eq?, and make-hasheqv procedure creates a table where keys are compared with eqv?.

The table is initialized with the content of assocs. In each element of assocs, the car is a key, and the cdr is the corresponding value. The mappings are added to the table in the order that they appear in assocs, so later mappings can hide earlier mappings.

See also make-custom-hash.

(make-weak-hash [assocs]) → (and/c hash? hash-equal? hash-weak?)
  assocs : (listof pair?) = null
(make-weak-hasheq [assocs]) → (and/c hash? hash-eq? hash-weak?)
  assocs : (listof pair?) = null
(make-weak-hasheqv [assocs]) → (and/c hash? hash-eqv? hash-weak?)
  assocs : (listof pair?) = null
Like `make-hash`, `make-hasheq`, and `make-hasheqv`, but creates a mutable hash table that holds keys weakly.

Beware that values in a weak hash table are retained normally. If a value in the table refers back to its key, then the table will retain the value and therefore the key; the mapping will never be removed from the table even if the key becomes otherwise inaccessible. To avoid that problem, use an ephemeron hash table as created by `make-ephemeron-hash`, `make-ephemeron-hasheqv`, or `make-ephemeron-hasheq`. For values that do not refer to keys, there is a modest extra cost to using an ephemeron hash table instead of a weak hash table, but prefer an ephemeron hash table when in doubt.

```
(make-ephemeron-hash [assocs])
  -> (and/c hash? hash-equal? hash-ephemeron?)
  assocs : (listof pair?) = null
(make-ephemeron-hasheqv [assocs])
  -> (and/c hash? hash-eqv? hash-ephemeron?)
  assocs : (listof pair?) = null
(make-ephemeron-hasheq [assocs])
  -> (and/c hash? hash-eq? hash-ephemeron?)
  assocs : (listof pair?) = null
```

Like `make-hash`, `make-hasheq`, and `make-hasheqv`, but creates a mutable hash table that holds keys-value combinations in the same way as an ephemeron.

Using an ephemeron hash table is like using a weak hash table and mapping each key to an ephemeron that pairs the key and value. An advantage of an ephemeron hash table is that the value need not be extracted with `ephemeron-value` from the result of functions like `hash-ref`. An ephemeron hash table might also be represented more compactly than a weak hash table with explicit ephemeron values.

Added in version 8.0.0.10 of package `base`.

```
(make-immutable-hash [assocs])
  -> (and/c hash? hash-equal? immutable?)
  assocs : (listof pair?) = null
(make-immutable-hasheqv [assocs])
  -> (and/c hash? hash-eqv? immutable?)
  assocs : (listof pair?) = null
(make-immutable-hasheq [assocs])
  -> (and/c hash? hash-eq? immutable?)
  assocs : (listof pair?) = null
```

Like `hash`, `hasheq`, and `hasheqv`, but accepts the key–value mapping in association-list form like `make-hash`, `make-hasheq`, and `make-hasheqv`.

```
(hash-set! hash key v) -> void?
```
hash : (and/c hash? (not/c immutable?))
key : any/c
v : any/c

Maps key to v in hash, overwriting any existing mapping for key.

See also the caveats concerning concurrent modification and the caveat concerning mutable keys above.

(hash-set*! hash key v ... ...) → void?
hash : (and/c hash? (not/c immutable?))
key : any/c
v : any/c

Maps each key to each v in hash, overwriting any existing mapping for each key. Mappings are added from the left, so later mappings overwrite earlier mappings.

See also the caveats concerning concurrent modification and the caveat concerning mutable keys above.

(hash-set hash key v) → (and/c hash? immutable?)
hash : (and/c hash? immutable?)
key : any/c
v : any/c

Functionally extends hash by mapping key to v, overwriting any existing mapping for key, and returning the extended hash table.

See also the caveat concerning mutable keys above.

(hash-set* hash key v ... ...) → (and/c hash? immutable?)
hash : (and/c hash? immutable?)
key : any/c
v : any/c

Functionally extends hash by mapping each key to v, overwriting any existing mapping for each key, and returning the extended hash table. Mappings are added from the left, so later mappings overwrite earlier mappings.

See also the caveat concerning mutable keys above.

(hash-ref hash key [failure-result]) → any
hash : hash?
key : any/c
failure-result : failure-result/c
= (lambda ()
  (raise (make-exn:fail:contract ....)))
Returns the value for key in hash. If no value is found for key, then failure-result determines the result:

- If failure-result is a procedure, it is called (through a tail call) with no arguments to produce the result.
- Otherwise, failure-result is returned as the result.

See also the caveats concerning concurrent modification and the caveat concerning mutable keys above.

```
(hash-ref-key hash key [failure-result]) ➞ any
  hash : hash?
  key : any/c
  failure-result : failure-result/c
    = (lambda ()
        (raise (make-exn:fail:contract ....)))
```

Returns the key held by hash that is equivalent to key according to hash’s key-comparison function. If no key is found, then failure-result is used as in hash-ref to determine the result.

If hash is not an impersonator, then the returned key, assuming it is found, will be eq?-equivalent to the one actually retained by hash:

Examples:

```
> (define original-key "hello")
> (define key-copy (string-copy original-key))
> (equal? original-key key-copy)
#t
> (eq? original-key key-copy)
#f
> (define table (make-hash))
> (hash-set! table original-key 'value)
> (eq? (hash-ref-key table "hello") original-key)
#t
> (eq? (hash-ref-key table "hello") key-copy)
#f
```

If a mutable hash is updated multiple times using keys that are not eq?-equivalent but are equivalent according to the hash’s key-comparison procedure, the hash retains the first one:

Examples:
> (define original-key "hello")
> (define key-copy (string-copy original-key))
> (define table (make-hash))
> (hash-set! table original-key 'one)
> (hash-set! table key-copy 'two)
> (eq? (hash-ref-key table "hello") original-key)
  #t
> (eq? (hash-ref-key table "hello") key-copy)
  #f

Conversely, an immutable hash retains the key that was most-recently used to update it:

Examples:

> (define original-key "hello")
> (define key-copy (string-copy original-key))
> (define table0 (hash))
> (define table1 (hash-set table0 original-key 'one))
> (define table2 (hash-set table1 key-copy 'two))
> (eq? (hash-ref-key table2 "hello") original-key)
  #f
> (eq? (hash-ref-key table2 "hello") key-copy)
  #t

If hash is an impersonator, then the returned key will be determined as described in the documentation to impersonate-hash.

See also the caveats concerning concurrent modification and the caveat concerning mutable keys above.

Added in version 7.4.0.3 of package base.

<table>
<thead>
<tr>
<th>(hash-ref! hash key to-set) → any</th>
</tr>
</thead>
<tbody>
<tr>
<td>hash : hash?</td>
</tr>
<tr>
<td>key : any/c</td>
</tr>
<tr>
<td>to-set : failure-result/c</td>
</tr>
</tbody>
</table>

Returns the value for key in hash. If no value is found for key, then to-set determines the result as in hash-ref (i.e., it is either a thunk that computes a value or a plain value), and this result is stored in hash for the key. (Note that if to-set is a thunk, it is not invoked in tail position.)

See also the caveats concerning concurrent modification and the caveat concerning mutable keys above.
(hash-has-key? hash key)  →  boolean?
  hash : hash?
  key : any/c

Returns #t if hash contains a value for the given key, #f otherwise.

[hash-update! hash
  key
  updater
  [failure-result])  →  void?
hash : (and/c hash? (not/c immutable?))
key : any/c
updater : (any/c . -> . any/c)
failure-result : failure-result/c
 = (lambda ()
  (raise (make-exn:fail:contract ....)))

Updates the value mapped by key in hash by applying updater to the value. The value returned by updater becomes the new mapping for key, overwriting the original value in hash.

Examples:

(define h (make-hash))
(hash-set! h 'a 5)

> (hash-update! h 'a add1)
> h
'hash((a . 6))

The optional failure-result argument is used when no mapping exists for key already, in the same manner as in hash-ref.

Examples:

(define h (make-hash))

> (hash-update! h 'b add1)
hash-update!: no value found for key: 'b
> (hash-update! h 'b add1 0)
> h
'hash((b . 1))

See also the caveats concerning concurrent modification and the caveat concerning mutable keys above.
(hash-update hash key updater [failure-result])
   \rightarrow (and/c hash? immutable?)
   hash : (and/c hash? immutable?)
   key : any/c
   updater : (any/c . \rightarrow . any/c)
   failure-result : failure-result/c
   = (lambda ()
       (raise (make-exn:fail:contract ....)))

Functionally updates the value mapped by key in hash by applying updater to the value and returning a new hash table. The value returned by updater becomes the new mapping for key in the returned hash table.

Examples:
(define h (hash 'a 5))
> (hash-update h 'a add1)
'hash((a . 6))

The optional failure-result argument is used when no mapping exists for key already, in the same manner as in hash-ref.

Examples:
(define h (hash))
> (hash-update h 'b add1)
hash-update: no value found for key: 'b
> (hash-update h 'b add1 0)
'hash((b . 1))

See also the caveat concerning mutable keys above.

(hash-remove! hash key) \rightarrow void?
   hash : (and/c hash? (not/c immutable?))
   key : any/c

Removes any existing mapping for key in hash.

See also the caveats concerning concurrent modification and the caveat concerning mutable keys above.
Functionally removes any existing mapping for key in hash, returning the fresh hash table.

See also the caveat concerning mutable keys above.

`(hash-clear! hash) → void?
  hash : (and/c hash? (not/c immutable?))`

Removes all mappings from hash.

If hash is not an impersonator, then all mappings are removed in constant time. If hash is an impersonator, then each key is removed one-by-one using `hash-remove!`.

See also the caveats concerning concurrent modification and the caveat concerning mutable keys above.

`(hash-clear hash) → (and/c hash? immutable?)
  hash : (and/c hash? immutable?)`

Functionally removes all mappings from hash.

If hash is not a chaperone, then clearing is equivalent to creating a new hash table, and the operation is performed in constant time. If hash is a chaperone, then each key is removed one-by-one using `hash-remove`.

`(hash-copy-clear hash) → hash?
  hash : hash?

Produces an empty hash table with the same key-comparison procedure and mutability of hash.

`(hash-map hash proc [try-order?]) → (listof any/c)
  hash : hash?
  proc : (any/c any/c . -> . any/c)
  try-order? : any/c = #f`

Applies the procedure proc to each element in hash in an unspecified order, accumulating the results into a list. The procedure proc is called each time with a key and its value, and the procedure’s individual results appear in order in the result list.

If a hash table is extended with new keys (either through proc or by another thread) while a hash-map or hash-for-each traversal is in process, arbitrary key–value pairs can be dropped or duplicated in the traversal. Key mappings can be deleted or remapped (by any thread) with no adverse affects; the change does not affect a traversal if the key has been seen already, otherwise the traversal skips a deleted key or uses the remapped key’s new value.

See also the caveats concerning concurrent modification above.

388
If `try-order?` is true, then the order of keys and values passed to `proc` is normalized under certain circumstances—including when every key is one of the following and with the following order (earlier bullets before later):

- booleans sorted `#f` before `#t`;
- characters sorted by `char<?`;
- real numbers sorted by `<`;
- symbols sorted with uninterned symbols before unreadable symbols before interned symbols, then sorted by `symbol<?`;
- keywords sorted by `keyword<?`;
- strings sorted by `string<?`;
- byte strings sorted by `bytes<?`;
- `null`;
- `<void>`; and
- `eof`.

Changed in version 6.3 of package `base`: Added the `try-order?` argument.

Changed in version 7.1.0.7: Added guarantees for `try-order?`.

```scheme
(hash-keys hash) → (listof any/c)
hash : hash?
```

Returns a list of the keys of `hash` in an unspecified order.

See `hash-map` for information about modifying `hash` during `hash-keys`.

See also the caveats concerning concurrent modification above.

```scheme
(hash-values hash) → (listof any/c)
hash : hash?
```

Returns a list of the values of `hash` in an unspecified order.

See `hash-map` for information about modifying `hash` during `hash-values`.

See also the caveats concerning concurrent modification above.

```scheme
(hash->list hash) → (listof (cons/c any/c any/c))
hash : hash?
```
Returns a list of the key–value pairs of hash in an unspecified order.

See hash-map for information about modifying hash during hash->list.

See also the caveats concerning concurrent modification above.

```
(hash-keys-subset? hash1 hash2) → boolean?
  hash1 : hash?
  hash2 : hash?
```

Returns #t if the keys of hash1 are a subset of or the same as the keys of hash2. The hash tables must both use the same key-comparison function (equal?, eqv?, or eq?), otherwise the exn:fail:contract exception is raised.

Using hash-keys-subset? on immutable hash tables can be much faster than iterating through the keys of hash1 to make sure that each is in hash2.

Added in version 6.5.0.8 of package base.

```
(hash-for-each hash proc [try-order?]) → void?
  hash : hash?
  proc : (any/c any/c . -> . any)
  try-order? : any/c = #f
```

Applies proc to each element in hash (for the side-effects of proc) in an unspecified order. The procedure proc is called each time with a key and its value.

See hash-map for information about try-order? and about modifying hash within proc.

See also the caveats concerning concurrent modification above.

Changed in version 6.3 of package base: Added the try-order? argument.

Changed in version 7.1.0.7: Added guarantees for try-order?.

```
(hash-count hash) → exact-nonnegative-integer?
  hash : hash?
```

Returns the number of keys mapped by hash.

For the CS implementation of Racket, the result is always computed in time and atomically. For the BC implementation of Racket, the result is computed in constant time and atomically only if hash does not retain keys weakly or like an ephemeron, otherwise, a traversal is required to count the keys.

```
(hash-empty? hash) → boolean?
  hash : hash?
```

390
Equivalent to \((\text{zero? } (\text{hash-count } \text{hash}))\).

\[
\begin{align*}
(\text{hash-iterate-first } \text{hash}) \\
\quad &\rightarrow (\text{or/c #f exact-nonnegative-integer?}) \\
\quad &\text{hash : hash?}
\end{align*}
\]

Returns #f if \text{hash} contains no elements, otherwise it returns an integer that is an index to the first element in the hash table; “first” refers to an unspecified ordering of the table elements, and the index values are not necessarily consecutive integers.

For a mutable \text{hash}, this index is guaranteed to refer to the first item only as long as no items are added to or removed from \text{hash}. More generally, an index is guaranteed to be a valid hash index for a given hash table only as long it comes from \text{hash-iterate-first} or \text{hash-iterate-next}, and only as long as the hash table is not modified. In the case of a hash table with weakly held keys or keys held like ephemerons, the hash table can be implicitly modified by the garbage collector (see §1.1.6 “Garbage Collection”) when it discovers that the key is not reachable.

\[
\begin{align*}
(\text{hash-iterate-next } \text{hash} \text{ pos}) \\
\quad &\rightarrow (\text{or/c #f exact-nonnegative-integer?}) \\
\quad &\text{hash : hash?} \\
\quad &\text{pos : exact-nonnegative-integer?}
\end{align*}
\]

Returns either an integer that is an index to the element in \text{hash} after the element indexed by \text{pos} (which is not necessarily one more than \text{pos}) or #f if \text{pos} refers to the last element in \text{hash}.

If \text{pos} is not a valid hash index of \text{hash}, then the result may be #f or it may be the next later index that remains valid. The latter result is guaranteed if a hash table has been modified only by the removal of keys.

Changed in version 7.0.0.10 of package base: Handle an invalid index by returning #f instead of raising exn:fail:contract.

\[
\begin{align*}
(\text{hash-iterate-key } \text{hash} \text{ pos}) &\rightarrow \text{any/c} \\
\quad &\text{hash : hash?} \\
\quad &\text{pos : exact-nonnegative-integer?} \\
(\text{hash-iterate-key } \text{hash} \text{ pos} \text{ bad-index-v}) &\rightarrow \text{any/c} \\
\quad &\text{hash : hash?} \\
\quad &\text{pos : exact-nonnegative-integer?} \\
\quad &\text{bad-index-v : any/c}
\end{align*}
\]

Returns the key for the element in \text{hash} at index \text{pos}.

If \text{pos} is not a valid hash index for \text{hash}, the result is \text{bad-index-v} if provided, otherwise the exn:fail:contract exception is raised.
Changed in version 7.0.0.10 of package base: Added the optional `bad-index-v` argument.

\[
\text{(hash-iterate-value hash pos)} \rightarrow \text{any}
\]

- `hash`: hash?
- `pos`: exact-nonnegative-integer?

\[
\text{(hash-iterate-value hash pos bad-index-v)} \rightarrow \text{any}
\]

- `hash`: hash?
- `pos`: exact-nonnegative-integer?
- `bad-index-v`: any/c

Returns the value for the element in `hash` at index `pos`.

If `pos` is not a valid hash index for `hash`, the result is `bad-index-v` if provided, otherwise the `exn:fail:contract` exception is raised.

Changed in version 7.0.0.10 of package base: Added the optional `bad-index-v` argument.

\[
\text{(hash-iterate-pair hash pos)} \rightarrow \text{(cons any/c any/c)}
\]

- `hash`: hash?
- `pos`: exact-nonnegative-integer?

\[
\text{(hash-iterate-pair hash pos bad-index-v)} \rightarrow \text{(cons any/c any/c)}
\]

- `hash`: hash?
- `pos`: exact-nonnegative-integer?
- `bad-index-v`: any/c

Returns a pair containing the key and value for the element in `hash` at index `pos`.

If `pos` is not a valid hash index for `hash`, the result is `(cons bad-index-v bad-index-v)` if `bad-index-v` is provided, otherwise the `exn:fail:contract` exception is raised.

Added in version 6.4.0.5 of package base.

Changed in version 7.0.0.10: Added the optional `bad-index-v` argument.

\[
\text{(hash-iterate-key+value hash pos)} \rightarrow \text{any/c any/c}
\]

- `hash`: hash?
- `pos`: exact-nonnegative-integer?

\[
\text{(hash-iterate-key+value hash pos bad-index-v)} \rightarrow \text{any/c any/c}
\]

- `hash`: hash?
- `pos`: exact-nonnegative-integer?
- `bad-index-v`: any/c

Returns the key and value for the element in `hash` at index `pos`.

If `pos` is not a valid hash index for `hash`, the result is `(values bad-index-v bad-index-v)` if `bad-index-v` is provided, otherwise the `exn:fail:contract` exception is raised.
Added in version 6.4.0.5 of package base.
Changed in version 7.0.0.10: Added the optional bad-index-v argument.

```
(hash-copy hash) → (and/c hash? (not/c immutable?))
  hash : hash?
```

Returns a mutable hash table with the same mappings, same key-comparison mode, and
same key-holding strength as hash.

4.14.1 Additional Hash Table Functions

```
(require racket/hash) package: base
```

The bindings documented in this section are provided by the racket/hash library, not
racket/base or racket.

```
(hash-union h0)
  h ... 
  [:combine combine
   #:combine/key combine/key]]
→ (and/c hash? immutable?)
  h0 : (and/c hash? immutable?)
  h : hash?
  combine : (-> any/c any/c any/c)
          = (lambda _ (error 'hash-union ....))
  combine/key : (-> any/c any/c any/c any/c any/c)
          = (lambda (k a b) (combine a b))
```

Computes the union of h0 with each hash table h by functional update, adding each element
of each h to h0 in turn. For each key k and value v, if a mapping from k to some value v0
already exists, it is replaced with a mapping from k to (combine/key k v0 v).

Examples:

```
> (hash-union (make-immutable-hash '([1 . one]))
  (make-immutable-hash '([2 . two]))
  (make-immutable-hash '([3 . three])))
'hash((1 . one) (2 . two) (3 . three))
> (hash-union (make-immutable-hash '([1 one uno] [2 two dos]))
  (make-immutable-hash '([1 eins un] [2 zwei deux]))
  #:combine/key (lambda (k v1 v2) (append v1 v2)))
'hash(((1 . (one uno eins un)) (2 . (two dos zwei deux)))
```

393
### hash-union!

Maps each hash table `h` to each hash table `h0` by mutable update, adding each element of each `h` to `h0` in turn. For each key `k` and value `v`, if a mapping from `k` to some value `v0` already exists, it is replaced with a mapping from `k` to `(combine/key k v0 v)

**Examples:**

```scheme
> (define h (make-hash))
> h
"#hash"
> (hash-union! h (make-immutable-hash ([(1 . one uno)] [(2 . two dos)])))
> h
"#hash((1 . (one uno)) (2 . (two dos)))"
> (hash-union! h

(make-immutable-hash ([(1 . eins un)] [(2 . zwei deux)]))
    #:combine/key (lambda (k v1 v2) (append v1 v2)))
> h
"#hash((1 . (one uno eins un)) (2 . (two dos zwei deux)))"
```

### hash-intersect

Constructs the hash table which is the intersection of `h0` with every hash table `h`. In the resulting hash table, a key `k` is mapped to a combination of the values to which `k` is mapped in each of the hash tables. The final values are computed by stepwise combination of the values appearing in each of the hash tables by applying `(combine/key k v v0 v)

**Examples:**

```scheme
(hash-intersect h0 h ...
    #:combine combine
    #:combine/key combine/key])
→ (and/c hash? immutable?)

(h0 : (and/c hash? immutable?)
  h : hash?)
  combine : (-> any/c any/c any/c)
       = (lambda _ (error 'hash-intersect ...))
  combine/key : (-> any/c any/c any/c any/c)
       = (lambda (k a b) (combine a b)))
```
\(v_i\), where \(v_i\) is the value to which \(k\) is mapped in the \(i\)-th hash table \(h\), and \(v\) is the accumulation of the values from the previous steps. The comparison predicate of the first argument (eq?, eqv?, equal?) determines the one for the result.

Examples:

\[
> \text{(hash-intersect (make-immutable-hash '(('a . 1) ('b . 2) ('c . 3)))
  (make-immutable-hash '(('a . 4) ('b . 5)))
  #:combine +)}
\]

'#hash((a . 5) (b . 7))

\[
> \text{(hash-intersect (make-immutable-hash '(('a . 1) ('b . 2) ('c . 3)))
  (make-immutable-hash '(('a . 4) ('b . 5)))
  #:combine/key
  (lambda (k v1 v2) (if (eq? k 'a) (+ v1 v2) (- v1 v2))))}
\]

'#hash((a . 5) (b . -3))

Added in version 7.9.0.1 of package base.

### 4.15 Sequences and Streams

Sequences and streams abstract over iteration of elements in a collection. Sequences allow iteration with for macros or with sequence operations such as sequence-map. Streams are functional sequences that can be used either in a generic way or a stream-specific way. Generators are closely related stateful objects that can be converted to a sequence and vice-versa.

#### 4.15.1 Sequences

A sequence encapsulates an ordered collection of values. The elements of a sequence can be extracted with one of the for syntactic forms, with the procedures returned by sequence-generate, or by converting the sequence into a stream.

The sequence datatype overlaps with many other datatypes. Among built-in datatypes, the sequence datatype includes the following:

- exact nonnegative integers (see below)
- strings (see §4.4 “Strings”)
- byte strings (see §4.5 “Byte Strings”)
- lists (see §4.10 “Pairs and Lists”)
- mutable lists (see §4.11 “Mutable Pairs and Lists”)
An exact number \( k \) that is a non-negative integer acts as a sequence similar to \((\text{in-range } k)\), except that \( k \) by itself is not a stream.

Custom sequences can be defined using structure type properties. The easiest method to define a custom sequence is to use the \texttt{gen:stream} generic interface. Streams are a suitable abstraction for data structures that are directly iterable. For example, a list is directly iterable with \texttt{first} and \texttt{rest}. On the other hand, vectors are not directly iterable: iteration has to go through an index. For data structures that are not directly iterable, the \texttt{iterator} for the data structure can be defined to be a stream (e.g., a structure containing the index of a vector).

For example, unrolled linked lists (represented as a list of vectors) themselves do not fit the stream abstraction, but have index-based iterators that can be represented as streams:

Examples:

```scheme
> (struct unrolled-list-iterator (idx lst))
#:methods gen:stream
[(define (stream-empty? iter)
    (define lst (unrolled-list-iterator-lst iter))
    (or (null? lst)
        (and (> idx (vector-length (first lst)))
            (null? (rest lst)))))

(define (stream-first iter)
    (vector-ref (first (unrolled-list-iterator-lst iter))
        (unrolled-list-iterator-idx iter)))

(define (stream-rest iter)
    (define idx (unrolled-list-iterator-idx iter))
    (define lst (unrolled-list-iterator-lst iter))
    (if (> idx (sub1 (vector-length (first lst))))
        (unrolled-list-iterator 0 (rest lst))
        (unrolled-list-iterator (add1 idx) lst)))]
```
The prop:sequence property provides more flexibility in specifying iteration, such as when a pre-processing step is needed to prepare the data for iteration. The make-do-sequence function creates a sequence given a thunk that returns procedures to implement a sequence, and the prop:sequence property can be associated with a structure type to implement its implicit conversion to a sequence.

For most sequence types, extracting elements from a sequence has no side-effect on the original sequence value; for example, extracting the sequence of elements from a list does not change the list. For other sequence types, each extraction implies a side-effect; for example, extracting the sequence of bytes from a port causes the bytes to be read from the port. A sequence’s state may either span all uses of the sequence, as for a port, or it may be confined to each distinct time that a sequence is initiated by a for form, sequence->stream, sequence-generate, or sequence-generate*. Concretely, the thunk passed to make-do-sequence is called to initiate the sequence each time the sequence is used.

Individual elements of a sequence typically correspond to single values, but an element may also correspond to multiple values. For example, a hash table generates two values—a key and its value—for each element in the sequence.

Sequence Predicate and Constructors

(sequence? v) → boolean?

v : any/c

Returns #t if v can be used as a sequence, #f otherwise.

Examples:

> (sequence? 42)
#t
> (sequence? '(a b c))
#t
> (sequence? "word")
#t
> (sequence? \x)
#f
Returns a sequence (that is also a stream) whose elements are numbers. The single-argument case \((\text{in-range } \text{end})\) is equivalent to \((\text{in-range } 0 \text{ end } 1)\). The first number in the sequence is \text{start}, and each successive element is generated by adding \text{step} to the previous element. The sequence stops before an element that would be greater or equal to \text{end} if \text{step} is non-negative, or less or equal to \text{end} if \text{step} is negative.

An \text{in-range} application can provide better performance for number iteration when it appears directly in a \text{for} clause.

Example: gaussian sum

\[
> (\text{for/sum} ([x (\text{in-range } 10)]) x)
\]

45

Example: sum of even numbers

\[
> (\text{for/sum} ([x (\text{in-range } 0 100 2)]) x)
\]

2450

When given zero as \text{step}, \text{in-range} returns an infinite sequence. It may also return infinite sequences when \text{step} is a very small number, and either \text{step} or the sequence elements are floating-point numbers.

\[(\text{in-inclusive-range } \text{start} \text{ end} \text{[step]}) \rightarrow \text{stream}\]

\[
\begin{align*}
\text{start} : \text{real} \\
\text{end} : \text{real} \\
\text{step} : \text{real} = 1
\end{align*}
\]

Similar to \text{in-range}, but the sequence stopping condition is changed so that the last element is allowed to be equal to \text{end}.

An \text{in-inclusive-range} application can provide better performance for number iteration when it appears directly in a \text{for} clause.

Examples:

\[
> (\text{sequence->list} (\text{in-inclusive-range } 7 11))
\]
'(7 8 9 10 11)
> (sequence->list (in-inclusive-range 7 11 2))
'(7 9 11)
> (sequence->list (in-inclusive-range 7 10 2))
'(7 9)

Added in version 8.0.0.13 of package base.

(in-naturals [start]) → stream?
  start : exact-nonnegative-integer? = 0

Returns an infinite sequence (that is also a stream) of exact integers starting with \textit{start},
where each element is one more than the preceding element.

An \texttt{in-naturals} application can provide better performance for integer iteration when it
appears directly in a \texttt{for} clause.

Example:

> (for/list ([k (in-naturals)]
> [x (in-range 10)])
> (list k x))
'(0 0) (1 1) (2 2) (3 3) (4 4) (5 5) (6 6) (7 7) (8 8) (9 9))

(in-list lst) → stream?
  lst : list?

Returns a sequence (that is also a stream) that is equivalent to using \textit{lst} directly as a se-
quence.

An \texttt{in-list} application can provide better performance for list iteration when it appears
directly in a \texttt{for} clause.

See \texttt{for} for information on the reachability of list elements during an iteration.

Example:

> (for/list ([x (in-list '(3 1 4))])
> '(,x ,(* x x)))
'(3 9) (1 1) (4 16)

Changed in version 6.7.0.4 of package base: Improved element-reachability guarantee for lists in \texttt{for}.

(in-mlist mlst) → sequence?
  mlst : mlist?
Returns a sequence equivalent to `mlst`.

An `in-mlist` application can provide better performance for mutable list iteration when it appears directly in a `for` clause.

Example:

```scheme
> (for/list ([x (in-mlist (mcons "RACKET" (mcons "LANG" '())))])
   (string-length x))
'(6 4)
```

Returns a sequence equivalent to `vec` when no optional arguments are supplied.

The optional arguments `start`, `stop`, and `step` are analogous to `in-range`, except that a `#f` value for `stop` is equivalent to `(vector-length vec)`. That is, the first element in the sequence is `(vector-ref vec start)`, and each successive element is generated by adding `step` to index of the previous element. The sequence stops before an index that would be greater or equal to `end` if `step` is non-negative, or less or equal to `end` if `step` is negative.

If `start` is not a valid index, then the `exn:fail:contract` exception is raised, except when `start`, `stop`, and `(vector-length vec)` are equal, in which case the result is an empty sequence.

Examples:

```scheme
> (for ([x (in-vector (vector 1) 1)]) x)
> (for ([x (in-vector (vector 1) 2)]) x)
in-vector: starting index is out of range
   starting index: 2
   valid range: [0, 0]
   vector: #(1)
> (for ([x (in-vector (vector 0 0)]) x)
> (for ([x (in-vector (vector 1 1)]) x)
```

If `stop` is not in `-1, (vector-length vec)`, then the `exn:fail:contract` exception is raised.

If `start` is less than `stop` and `step` is negative, then the `exn:fail:contract` exception is raised. Similarly, if `start` is more than `stop` and `step` is positive, then the `exn:fail:contract` exception is raised.
An **in-vector** application can provide better performance for vector iteration when it appears directly in a `for` clause.

Examples:

```scheme
> (define (histogram vector-of-words)
  (define a-hash (make-hash))
  (for ([word (in-vector vector-of-words)])
    (hash-set! a-hash word (add1 (hash-ref a-hash word 0))))
  a-hash)
> (histogram #("hello" "world" "hello" "sunshine"))
"#hash(("hello" . 2) ("sunshine" . 1) ("world" . 1))"
```

An **in-string** application can provide better performance for string iteration when it appears directly in a `for` clause.

Examples:

```scheme
> (define (line-count str)
  (for/sum ([ch (in-string str)])
    (if (char=? #\newline ch) 1 0)))
> (line-count "this string\nhas\nthree \nnewlines")
3
```

An **in-bytes** application can provide better performance for byte string iteration when it appears directly in a `for` clause.

Examples:

```scheme
> (define (byte-line-count bstr)
  (for/sum ([ch (in-bytes bstr)])
    (if (char=? #\newline ch) 1 0)))
> (byte-line-count "this string\nhas\nthree \nnewlines")
3
```

---

**Returns a sequence equivalent to `str` when no optional arguments are supplied.**

The optional arguments `start`, `stop`, and `step` are as in **in-vector**.

An **in-string** application can provide better performance for string iteration when it appears directly in a `for` clause.

Examples:

```scheme
> (define (line-count str)
  (for/sum ([ch (in-string str)])
    (if (char=? #\newline ch) 1 0)))
> (line-count "this string\nhas\nthree \nnewlines")
3
```

An **in-bytes** application can provide better performance for byte string iteration when it appears directly in a `for` clause.

Examples:
> (define (has-eof? bs)
  (for/or ([ch (in-bytes bs)])
    (= ch 0)))

> (has-eof? #"this byte string has an \0embedded zero byte")
#t

> (has-eof? #"this byte string does not")
#f

(in-port [r in]) → sequence?
  r : (input-port? . -> . any/c) = read
  in : input-port? = (current-input-port)

Returns a sequence whose elements are produced by calling r on in until it produces eof.

(in-input-port-bytes in) → sequence?
  in : input-port?

Returns a sequence equivalent to (in-port read-byte in).

(in-input-port-chars in) → sequence?
  in : input-port?

Returns a sequence whose elements are read as characters from in (equivalent to (in-port read-char in)).

(in-lines [in mode]) → sequence?
  in : input-port? = (current-input-port)
  mode : (or/c 'linefeed 'return 'return-linefeed 'any 'any-one) = 'any

Returns a sequence equivalent to (in-port (lambda (p) (read-line p mode)) in). Note that the default mode is 'any, whereas the default mode of read-line is 'linefeed.

(in-bytes-lines [in mode]) → sequence?
  in : input-port? = (current-input-port)
  mode : (or/c 'linefeed 'return 'return-linefeed 'any 'any-one) = 'any

Returns a sequence equivalent to (in-port (lambda (p) (read-bytes-line p mode)) in). Note that the default mode is 'any, whereas the default mode of read-bytes-line is 'linefeed.

(in-hash hash) → sequence?
  hash : hash?

(in-hash hash bad-index-v) → sequence?
  hash : hash?
  bad-index-v : any/c
Returns a sequence equivalent to `hash`, except when `bad-index-v` is supplied.

If `bad-index-v` is supplied, then `bad-index-v` is returned as both the key and the value in the case that the `hash` is modified concurrently so that iteration does not have a valid hash index. Providing `bad-index-v` is particularly useful when iterating through a hash table with weakly held keys, since entries can be removed asynchronously (i.e., after `in-hash` has committed to another iteration, but before it can access the entry for the next iteration).

Examples:

```scheme
> (define table (hash 'a 1 'b 2))
> (for ([key value] (in-hash table))
  (printf "key: ~a value: ~a\n" key value))
key: b value: 2
key: a value: 1
```

Changed in version 7.0.0.10 of package base: Added the optional `bad-index-v` argument.

```
(in-hash-keys hash) -> sequence?
  hash : hash?
(in-hash-keys hash bad-index-v) -> sequence?
  hash : hash?
  bad-index-v : any/c
```

Returns a sequence whose elements are the keys of `hash`, using `bad-index-v` in the same way as `in-hash`.

Examples:

```scheme
> (define table (hash 'a 1 'b 2))
> (for ([key (in-hash-keys table)])
  (printf "key: ~a\n" key))
key: b
key: a
```

Changed in version 7.0.0.10 of package base: Added the optional `bad-index-v` argument.

```
(in-hash-values hash) -> sequence?
  hash : hash?
(in-hash-values hash bad-index-v) -> sequence?
  hash : hash?
  bad-index-v : any/c
```

Returns a sequence whose elements are the values of `hash`, using `bad-index-v` in the same way as `in-hash`.

Examples:
> (define table (hash 'a 1 'b 2))
> (for ([value (in-hash-values table)])
    (printf "value: ~a\n" value))
  value: 2
  value: 1

Changed in version 7.0.0.10 of package base: Added the optional bad-index-v argument.

(in-hash-pairs hash) → sequence?
  hash : hash?
(in-hash-pairs hash bad-index-v) → sequence?
  hash : hash?
  bad-index-v : any/c

Returns a sequence whose elements are pairs, each containing a key and its value from hash (as opposed to using hash directly as a sequence to get the key and value as separate values for each element).

The bad-index-v argument, if supplied, is used in the same way as by in-hash. When an invalid index is encountered, the pair in the sequence with have bad-index-v as both its car and cdr.

Examples:

> (define table (hash 'a 1 'b 2))
> (for ([key+value (in-hash-pairs table)])
    (printf "key and value: ~a\n" key+value))
  key and value: (b . 2)
  key and value: (a . 1)

Changed in version 7.0.0.10 of package base: Added the optional bad-index-v argument.

(in-mutable-hash hash) → sequence?
  hash : (and/c hash? (not/c immutable?) (not/c hash-weak?))
(in-mutable-hash hash bad-index-v) → sequence?
  hash : (and/c hash? (not/c immutable?) (not/c hash-weak?))
  bad-index-v : any/c
(in-mutable-hash-keys hash) → sequence?
  hash : (and/c hash? (not/c immutable?) (not/c hash-weak?))
(in-mutable-hash-keys hash bad-index-v) → sequence?
  hash : (and/c hash? (not/c immutable?) (not/c hash-weak?))
  bad-index-v : any/c
(in-mutable-hash-values hash) → sequence?
  hash : (and/c hash? (not/c immutable?) (not/c hash-weak?))
(in-mutable-hash-values hash bad-index-v) → sequence?
  hash : (and/c hash? (not/c immutable?) (not/c hash-weak?))
  bad-index-v : any/c
(in-mutable-hash-pairs hash) → sequence?
  hash : (and/c hash? (not/c immutable?) (not/c hash-weak?))
(in-mutable-hash-pairs hash bad-index-v) → sequence?
  hash : (and/c hash? (not/c immutable?) (not/c hash-weak?))
    bad-index-v : any/c
(in-immutable-hash hash) → sequence?
  hash : (and/c hash? immutable?)
(in-immutable-hash hash bad-index-v) → sequence?
  hash : (and/c hash? immutable?)
    bad-index-v : any/c
(in-immutable-hash-keys hash) → sequence?
  hash : (and/c hash? immutable?)
(in-immutable-hash-keys hash bad-index-v) → sequence?
  hash : (and/c hash? immutable?)
    bad-index-v : any/c
(in-immutable-hash-values hash) → sequence?
  hash : (and/c hash? immutable?)
(in-immutable-hash-values hash bad-index-v) → sequence?
  hash : (and/c hash? immutable?)
    bad-index-v : any/c
(in-immutable-hash-pairs hash) → sequence?
  hash : (and/c hash? immutable?)
(in-immutable-hash-pairs hash bad-index-v) → sequence?
  hash : (and/c hash? immutable?)
    bad-index-v : any/c
(in-weak-hash hash) → sequence?
  hash : (and/c hash? hash-weak?)
(in-weak-hash hash bad-index-v) → sequence?
  hash : (and/c hash? hash-weak?)
    bad-index-v : any/c
(in-weak-hash-keys hash) → sequence?
  hash : (and/c hash? hash-weak?)
(in-weak-hash-keys hash bad-index-v) → sequence?
  hash : (and/c hash? hash-weak?)
    bad-index-v : any/c
(in-weak-hash-values hash) → sequence?
  hash : (and/c hash? hash-weak?)
(in-weak-hash-values hash bad-index-v) → sequence?
  hash : (and/c hash? hash-weak?)
    bad-index-v : any/c
(in-weak-hash-pairs hash) → sequence?
  hash : (and/c hash? hash-weak?)
(in-weak-hash-pairs hash bad-index-v) → sequence?
  hash : (and/c hash? hash-weak?)
    bad-index-v : any/c
Sequence constructors for specific kinds of hash tables. These may perform better than the analogous `in-hash` forms.

Added in version 6.4.0.6 of package `base`.

Changed in version 7.0.0.10: Added the optional `bad-index-v` argument.

Changed in version 8.0.0.10: Added `ephemeron` variants.

```scheme
(in-directory [dir use-dir?] \rightarrow sequence?
  dir : (or/c #f path-string?) = #f
  use-dir? : ((and/c path? complete-path?) . \rightarrow . any/c)
     = (lambda (dir-path) #t)
```

Returns a sequence that produces all of the paths for files, directories, and links within `dir`, except for the contents of any directory for which `use-dir?` returns `#f`. If `dir` is not `#f`, then every produced path starts with `dir` as its prefix. If `dir` is `#f`, then paths in and relative to the current directory are produced.

An `in-directory` sequence traverses nested subdirectories recursively (filtered by `use-dir?`). To generate a sequence that includes only the immediate content of a directory, use the result of `directory-list` as a sequence.

The immediate content of each directory is reported as sorted by `path<`, and the content of a subdirectory is reported before subsequent paths within the directory.

Examples:
> (current-directory (collection-path "info"))
> (for/list ([f (in-directory)])
   f)
'(#/path:compiled>
  #/path:compiled/main_rkt.dep>
  #/path:compiled/main_rkt.zo>
  #/path:main.rkt>)
> (for/list ([f (in-directory "compiled")])
   f)
'(#/path:main_rkt.dep> #/path:main_rkt.zo>)
> (for/list ([f (in-directory "compiled")])
   f)
'(#/path:compiled/main_rkt.dep> #/path:compiled/main_rkt.zo>)
> (for/list ([f (in-directory #f (lambda (p)
      (not (regexp-
        match? #rx"compiled" p))))])
   f)
'(#/path:main.rkt> #/path:compiled>)

Changed in version 6.0.0.1 of package base: Added use-dir? argument.
Changed in version 6.6.0.4: Added guarantee of sorted results.

(in-producer producer) → sequence?
  producer : procedure?
(in-producer producer stop arg ...) → sequence?
  producer : procedure?
  stop : any/c
  arg : any/c

Returns a sequence that contains values from sequential calls to producer, which would usually use some state to do its work.

If a stop value is not given, the sequence goes on infinitely, and therefore it common to use it with a finite sequence or using #:break etc. If a stop value is given, it is used to identify a value that marks the end of the sequence (and the stop value is not included in the sequence); stop can be a predicate that is applied to the results of producer, or it can be a value that is tested against the result of with eq?. (The stop argument must be a predicate if the stop value is itself a function or if producer returns multiple values.)

If additional args are specified, they are passed to every call to producer.

Examples:

> (define (counter)
   (define n 0)
   (lambda ([d 1]) (set! n (+ d n)) n))
> (for/list ([x (in-producer (counter))]) [y (in-range 4)]) x)
'(1 2 3 4)
> (for/list ([x (in-producer (counter))]) #:break (= x 5)) x)
'(1 2 3 4)
> (for/list ([x (in-producer (counter) 5)]) x)
'(1 2 3 4)
> (for/list ([x (in-producer (counter) 5 1/2)]) x)
'(1/2 1 3/2 2 5/2 3 7/2 4 9/2)
> (for/list ([x (in-producer read eof (open-input-string "1 2
3")]) x))
'(1 2 3)

(in-value v) → sequence?
v : any/c

Returns a sequence that produces a single value: v. This form is mostly useful for let-like bindings in forms such as for*/list.

(in-indexed seq) → sequence?
seq : sequence?

Returns a sequence where each element has two values: the value produced by seq, and a non-negative exact integer starting with 0. The elements of seq must be single-valued.

Example:

> (for [(ch i) (in-indexed "hello")])
   (printf "The char at position ~a is: ~a\n" i ch))
The char at position 0 is: h
The char at position 1 is: e
The char at position 2 is: l
The char at position 3 is: l
The char at position 4 is: o

(in-sequences seq ...) → sequence?
seq : sequence?

Returns a sequence that is made of all input sequences, one after the other. Each seq is initiated only after the preceding seq is exhausted. If a single seq is provided, then seq is returned; otherwise, the elements of each seq must all have the same number of values.

(in-cycle seq ...) → sequence?
seq : sequence?
Similar to in-sequences, but the sequences are repeated in an infinite cycle, where each seq is initiated afresh in each iteration. Beware that if no seqs are provided or if all seqs become empty, then the sequence produced by in-cycle never returns when an element is demanded—or even when the sequence is initiated, if all seqs are initially empty.

\[(in\text{-parallel} \ seq \ \ldots) \rightarrow \text{sequence?}\]

\[seq : \text{sequence?}\]

Returns a sequence where each element has as many values as the number of supplied seqs; the values, in order, are the values of each seq. The elements of each seq must be single-valued.

\[(in\text{-values-sequence} \ seq) \rightarrow \text{sequence?}\]

\[seq : \text{sequence?}\]

Returns a sequence that is like seq, but it combines multiple values for each element from seq as a list of elements.

\[(in\text{-values*-sequence} \ seq) \rightarrow \text{sequence?}\]

\[seq : \text{sequence?}\]

Returns a sequence that is like seq, but when an element of seq has multiple values or a single list value, then the values are combined in a list. In other words, in-values*-sequence is like in-values-sequence, except that non-list, single-valued elements are not wrapped in a list.

\[(stop\text{-before} \ seq \ pred) \rightarrow \text{sequence?}\]

\[seq : \text{sequence?}\]

\[pred : (\text{any/c} . \rightarrow . \text{any})\]

Returns a sequence that contains the elements of seq (which must be single-valued), but only until the last element for which applying pred to the element produces #t, after which the sequence ends.

\[(stop\text{-after} \ seq \ pred) \rightarrow \text{sequence?}\]

\[seq : \text{sequence?}\]

\[pred : (\text{any/c} . \rightarrow . \text{any})\]

Returns a sequence that contains the elements of seq (which must be single-valued), but only until the element (inclusive) for which applying pred to the element produces #t, after which the sequence ends.

\[(make\text{-do-sequence} \ thunk) \rightarrow \text{sequence?}\]
thunk : (or/c (-> (values (any/c . -> . any) (any/c . -> . any/c) any/c (or/c (any/c . -> . any/c) #f) (or/c () () #:rest list? . ->* . any/c) #f) (or/c ((any/c) () #:rest list? . ->* . any/c) #f)))

Returns a sequence whose elements are generated by the procedures and initial value returned by the thunk, which is called to initiate the sequence. The initiated sequence is defined in terms of a position, which is initialized to the third result of the thunk, and the element, which may consist of multiple values.

The thunk results define the generated elements as follows:

- The first result is a pos->element procedure that takes the current position and returns the value(s) for the current element.
- The optional second result is an early-next-pos procedure that is described further below. Alternatively, the optional second result can be #f, which is equivalent to the identity function.
- The third (or second) result is a next-pos procedure that takes the current position and returns the next position.
- The fourth (or third) result is the initial position.
- The fifth (or fourth) result is a continue-with-pos? function that takes the current position and returns a true result if the sequence includes the value(s) for the current position, and false if the sequence should end instead of including the value(s). Alternatively, the fifth (or fourth) result can be #f to indicate that the sequence should always include the current value(s). This function is checked on each position before pos->element is used.
- The sixth (or fifth) result is a continue-with-val? function that is like the fourth result, but it takes the current element value(s) instead of the current position. Alternatively, the sixth (or fifth) result can be #f to indicate that the sequence should always include the value(s) at the current position.
- The seventh (or sixth) result is a continue-after-pos+val? procedure that takes both the current position and the current element value(s) and determines whether the
sequence ends after the current element is already included in the sequence. Alternatively, the seventh (or sixth) result can be `#f` to indicate that the sequence can always continue after the current value(s).

The `early-next-pos` procedure, which is the optional second result, takes the current position and returns an updated position. This updated position is used for `next-pos` and `continue-after-pos+val?`, but not with `continue-with-pos?` (which uses the original current position). The intent of `early-next-pos` is to support a sequence where the position must be incremented to avoid keeping a value reachable while a loop processes the sequence value, so `early-next-pos` is applied just after `pos->element`.

Each of the procedures listed above is called only once per position. Among the last three procedures, as soon as one of the procedures returns `#f`, the sequence ends, and none are called again. Typically, one of the functions determines the end condition, and `#f` is used in place of the other two functions.

Changed in version 6.7.0.4 of package base: Added support for the optional second result.

### prop:sequence : struct-type-property?

Associates a procedure to a structure type that takes an instance of the structure and returns a sequence. If `v` is an instance of a structure type with this property, then `(sequence? v)` produces `#t`.

Using a pre-existing sequence:

Examples:

```
> (struct my-set (table)
   #:property prop:sequence
   (lambda (s)
     (in-hash-keys (my-set-table s))))
> (define (make-set . xs)
   (my-set (for/hash ([x (in-list xs)]
     (values x #t))))
> (for/list ([c (make-set 'celeriac 'carrot 'potato)])
  c)
'(potato celeriac carrot)
```

Using `make-do-sequence`:

Examples:

```
> (struct train (car next)
   #:property prop:sequence
   (lambda (t)
   ```
(make-do-sequence
  (lambda ()
    (values train-car train-next t
      (lambda (t) t)
      (lambda (v) #t)
      (lambda (t v) #t))))
)

> (for/list ([c (train 'engine
     (train 'boxcar
       (train 'caboose
         #f))))])
  c)
'(engine boxcar caboose)

Sequence Conversion

(sequence->stream seq) → stream?
seq : sequence?

Coverts a sequence to a stream, which supports the stream-first and stream-rest operations. Creation of the stream eagerly initiates the sequence, but the stream lazily draws elements from the sequence, caching each element so that stream-first produces the same result each time is applied to a stream.

If extracting an element from seq involves a side-effect, then the effect is performed each time that either stream-first or stream-rest is first used to access or skip an element.

(sequence-generate seq) → (-> boolean?) (-> any)
seq : sequence?

Initiates a sequence and returns two thunks to extract elements from the sequence. The first returns #t if more values are available for the sequence. The second returns the next element (which may be multiple values) from the sequence; if no more elements are available, the exn:fail:contract exception is raised.

(sequence-generate* seq) → (or/c list? #f)
  (-> (values (or/c list? #f) procedure?))
seq : sequence?

Like sequence-generate, but avoids state (aside from any inherent in the sequence) by returning a list of values for the sequence’s first element—or #f if the sequence is empty—and a thunk to continue with the sequence; the result of the thunk is the same as the result of sequence-generate*, but for the second element of the sequence, and so on. If the thunk is called when the element result is #f (indicating no further values in the sequence), the exn:fail:contract exception is raised.
Additional Sequence Operations

(require racket/sequence)  package: base

The bindings documented in this section are provided by the racket/sequence and racket libraries, but not racket/base.

empty-sequence : sequence?

A sequence with no elements.

(sequence->list s) → list?
  s : sequence?

Returns a list whose elements are the elements of s, each of which must be a single value. If s is infinite, this function does not terminate.

(sequence-length s) → exact-nonnegative-integer?
  s : sequence?

Returns the number of elements of s by extracting and discarding all of them. If s is infinite, this function does not terminate.

(sequence-ref s i) → any
  s : sequence?
  i : exact-nonnegative-integer?

Returns the i th element of s (which may be multiple values).

(sequence-tail s i) → sequence?
  s : sequence?
  i : exact-nonnegative-integer?

Returns a sequence equivalent to s, except that the first i elements are omitted.

In case initiating s involves a side effect, the sequence s is not initiated until the resulting sequence is initiated, at which point the first i elements are extracted from the sequence.

(sequence-append s ...) → sequence?
  s : sequence?

Returns a sequence that contains all elements of each sequence in the order they appear in the original sequences. The new sequence is constructed lazily.

If all given s s are streams, the result is also a stream.
(sequence-map f s) → sequence?
  f : procedure?
  s : sequence?

Returns a sequence that contains f applied to each element of s. The new sequence is constructed lazily.

If s is a stream, then the result is also a stream.

(sequence-andmap f s) → boolean?
  f : (-> any/c ... boolean?)
  s : sequence?

Returns #t if f returns a true result on every element of s. If s is infinite and f never returns a false result, this function does not terminate.

(sequence-ormap f s) → boolean?
  f : (-> any/c ... boolean?)
  s : sequence?

Returns #t if f returns a true result on some element of s. If s is infinite and f never returns a true result, this function does not terminate.

(sequence-for-each f s) → void?
  f : (-> any/c ... any)
  s : sequence?

Applies f to each element of s. If s is infinite, this function does not terminate.

(sequence-fold f i s) → any/c
  f : (-> any/c any/c ... any/c)
  i : any/c
  s : sequence?

Folds f over each element of s with i as the initial accumulator. If s is infinite, this function does not terminate. The f function takes the accumulator as its first argument and the next sequence element as its second.

(sequence-count f s) → exact-nonnegative-integer?
  f : procedure?
  s : sequence?

Returns the number of elements in s for which f returns a true result. If s is infinite, this function does not terminate.
(sequence-filter f s) → sequence?
f : (→ any/c ... boolean?)
s : sequence?

Returns a sequence whose elements are the elements of s for which f returns a true result. Although the new sequence is constructed lazily, if s has an infinite number of elements where f returns a false result in between two elements where f returns a true result, then operations on this sequence will not terminate during the infinite sub-sequence.

If s is a stream, then the result is also a stream.

(sequence-add-between s e) → sequence?
s : sequence?
e : any/c

Returns a sequence whose elements are the elements of s, but with e between each pair of elements in s. The new sequence is constructed lazily.

If s is a stream, then the result is also a stream.

Examples:

> (let* ([all-reds (in-cycle '("red"))] [red-and-blues (sequence-add-between all-reds "blue")]) (for/list ([n (in-range 10)] [elt red-and-blues]) elt))
'("red" "blue" "red" "blue" "red" "blue" "red" "blue" "red" "blue")

> (for ([text (sequence-add-between '("veni" "vidi" "duci") ", ")])
(display text))
veni, vidi, duci

(sequence/c [#:min-count min-count]
  elem/c ...)
  → contract?
min-count : (or/c #f exact-nonnegative-integer?) = #f
elem/c : contract?

Wraps a sequence, obligating it to produce elements with as many values as there are elem/c contracts, and obligating each value to satisfy the corresponding elem/c. The result is not guaranteed to be the same kind of sequence as the original value; for instance, a wrapped list is not guaranteed to satisfy list?.

If min-count is a number, the stream is required to have at least that many elements in it.
Examples:

```lisp
> (define/contract predicates
    (sequence/c (-> any/c boolean?))
    (in-list (list integer?
               string->symbol)))
> (for ([P predicates])
    (printf "P: ~a\n" P)
#f
predicates: broke its own contract
  promised: boolean?
  produced: 'cat
  in: an element of
    (sequence/c (-> any/c boolean?))
  contract from: (definition predicates)
  blaming: (definition predicates)
  (assuming the contract is correct)
  at: eval:33:0
> (define/contract numbers&strings
    (sequence/c number? string?)
    (in-dict (list (cons 1 "one")
                (cons 2 "two")
                (cons 3 'three))))
> (for ([N S numbers&strings])
    (printf "N: ~a\n" N S)
1: one
2: two
numbers&strings: broke its own contract
  promised: string?
  produced: 'three
  in: an element of
    (sequence/c number? string?)
  contract from: (definition numbers&strings)
  blaming: (definition numbers&strings)
  (assuming the contract is correct)
  at: eval:35:0
> (define/contract a-sequence
    (sequence/c #:min-count 2 char?)
    "x")
> (for ([x a-sequence]
       [i (in-naturals)])
    (printf "i: ~a is ~a\n" i x))
0 is x
a-sequence: broke its own contract
  promised: a sequence that contains at least 2 values
  produced: "x"
```

416
Additional Sequence Constructors

\[
\text{(in-syntax stx)} \to \text{sequence?}
\]
\[
stx : \text{syntax?}
\]

Produces a sequence whose elements are the successive subparts of \text{stx}. Equivalent to \text{(stx->list lst)}.

An \text{in-syntax} application can provide better performance for syntax iteration when it appears directly in a \text{for} clause.

Example:
\[
\text{> (for/list ([x (in-syntax #'(1 2 3))]) x)}
\]
\[
\]

Added in version 6.3 of package base.

\[
\text{(in-slice length seq)} \to \text{sequence?}
\]
\[
\text{length : exact-positive-integer?}
\]
\[
\text{seq : sequence?}
\]

Returns a sequence whose elements are lists with the first \text{length} elements of \text{seq}, then the next \text{length} and so on.

Example:
\[
\text{> (for/list ([e (in-slice 3 (in-range 8))]) e)}
\]
\[
'((0 1 2) (3 4 5) (6 7))
\]

Added in version 6.3 of package base.

4.15.2 Streams

A \text{stream} is a kind of sequence that supports functional iteration via \text{stream-first} and \text{stream-rest}. The \text{stream-cons} form constructs a lazy stream, but plain lists can be used as streams, and functions such as \text{in-range} and \text{in-naturals} also create streams.
(require racket/stream)   package: base

The bindings documented in this section are provided by the racket/stream and racket libraries, but not racket/base.

(stream? v) → boolean?
  v : any/c

Returns #t if v can be used as a stream, #f otherwise.

(stream-empty? s) → boolean?
  s : stream?

Returns #t if s has no elements, #f otherwise.

(stream-first s) → any
  s : (and/c stream? (not/c stream-empty?))

Returns the value(s) of the first element in s.

(stream-rest s) → stream?
  s : (and/c stream? (not/c stream-empty?))

Returns a stream that is equivalent to s without its first element.

(stream-cons first-expr rest-expr)
(stream-cons #:eager first-expr rest-expr)
(stream-cons first-expr #:eager rest-expr)
(stream-cons #:eager first-expr #:eager rest-expr)

Produces a stream whose first element is determined by first-expr and whose rest is determined by rest-expr.

If first-expr is not preceded by #:eager, then first-expr is not evaluated immediately. Instead, stream-first on the result stream forces the evaluation of first-expr (once) to produce the first element of the stream. If evaluating first-expr raises an exception or tries to force itself, then an exn:fail:contract exception is raised, and future attempts to force evaluation will trigger another exception.

If rest-expr is not preceded by #:eager, then rest-expr is not evaluated immediately. Instead, stream-rest on the result stream produces another stream that is like the one produced by (stream-lazy rest-expr).

The first element of the stream as produced by first-expr must be a single value. The rest-expr must produce a stream when it is evaluated, otherwise the exn:fail:contract? exception is raised.
(stream-lazy stream-expr)
(stream-lazy #:who who-expr stream-expr)

Similar to (delay stream-expr), but the result is a stream instead of a promise, and stream-expr must produce a stream when it is eventually forced. The stream produced by stream-lazy has the same content as the stream produced by stream-expr; that is, operations like stream-first on the result stream will force stream-expr and retry on its result.

If evaluating stream-expr raises an exception or tries to force itself, then an exn:fail:contract exception is raised, and future attempts to force evaluation will trigger another exception.

If who-expr is provided, it is evaluated when constructing the delayed stream. If stream-expr later produces a value that is not a stream, and if who-expr produced a symbol value, then the symbol is used for the error message.

Added in version 8.0.0.12 of package base.

(stream-force s) → stream?
  s : stream?

Forces the evaluation of a delayed stream from stream-lazy, from the stream-rest of a stream-cons, etc., returning the forced stream. If s is not a delayed stream, then s is returned.

Normally, stream-force is not needed, because operations like stream-first, stream-rest, and stream-empty? force a delayed stream as needed. In rare cases, stream-force can be useful to reveal the underlying implementation of a stream (e.g., a stream that is an instance of a structure type that has the prop:stream property).

Added in version 8.0.0.12 of package base.

(stream expr ...)

A shorthand for nested stream-conses ending with empty-stream.

(stream* expr ... rest-expr)

A shorthand for nested stream-conses, but the rest-expr must produce a stream when it is forced, and that stream is used as the rest of the stream instead of empty-stream. Similar to list* but for streams.

Added in version 6.3 of package base.

Changed in version 8.0.0.12: Changed to delay rest-expr even if zero exprs are provided.
(in-stream s) → sequence?
  s : stream?

Returns a sequence that is equivalent to s.

An in-stream application can provide better performance for streams iteration when it appears directly in a for clause.

See for for information on the reachability of stream elements during an iteration.

Changed in version 6.7.0.4 of package base: Improved element-reachability guarantee for streams in for.

empty-stream : stream?

A stream with no elements.

(stream->list s) → list?
  s : stream?

Returns a list whose elements are the elements of s, each of which must be a single value. If s is infinite, this function does not terminate.

(stream-length s) → exact-nonnegative-integer?
  s : stream?

Returns the number of elements of s. If s is infinite, this function does not terminate.

In the case of lazy streams, this function forces evaluation only of the sub-streams, and not the stream’s elements.

(stream-ref s i) → any
  s : stream?
  i : exact-nonnegative-integer?

Returns the i th element of s (which may be multiple values).

(stream-tail s i) → stream?
  s : stream?
  i : exact-nonnegative-integer?

Returns a stream equivalent to s, except that the first i elements are omitted.

In case extracting elements from s involves a side effect, they will not be extracted until the first element is extracted from the resulting stream.
(stream-take s i) → stream?
  s : stream?
  i : exact-nonnegative-integer?

Returns a stream of the first i elements of s.

(stream-append s ...) → stream?
  s : stream?

Returns a stream that contains all elements of each stream in the order they appear in the original streams. The new stream is constructed lazily, while the last given stream is used in the tail of the result.

(stream-map f s) → stream?
  f : procedure?
  s : stream?

Returns a stream that contains f applied to each element of s. The new stream is constructed lazily.

(stream-andmap f s) → boolean?
  f : (-> any/c ... boolean?)
  s : stream?

Returns #t if f returns a true result on every element of s. If s is infinite and f never returns a false result, this function does not terminate.

(stream-ormap f s) → boolean?
  f : (-> any/c ... boolean?)
  s : stream?

Returns #t if f returns a true result on some element of s. If s is infinite and f never returns a true result, this function does not terminate.

(stream-for-each f s) → void?
  f : (-> any/c ... any)
  s : stream?

Applies f to each element of s. If s is infinite, this function does not terminate.

(stream-fold f i s) → any/c
  f : (-> any/c any/c ... any/c)
  i : any/c
  s : stream?
Folds \( f \) over each element of \( s \) with \( i \) as the initial accumulator. If \( s \) is infinite, this function does not terminate. The \( f \) function takes the accumulator as its first argument and the next stream element as its second.

\[
\text{(stream-count } f \ s) \to \text{exact-nonnegative-integer}?
\]
\[
f : \text{procedure}?
\]
\[
s : \text{stream}?
\]

Returns the number of elements in \( s \) for which \( f \) returns a true result. If \( s \) is infinite, this function does not terminate.

\[
\text{(stream-filter } f \ s) \to \text{stream}?
\]
\[
f : (-> \text{any/c} \ldots \text{boolean}?)
\]
\[
s : \text{stream}?
\]

Returns a stream whose elements are the elements of \( s \) for which \( f \) returns a true result. Although the new stream is constructed lazily, if \( s \) has an infinite number of elements where \( f \) returns a false result in between two elements where \( f \) returns a true result, then operations on this stream will not terminate during the infinite sub-stream.

\[
\text{(stream-add-between } s \ e) \to \text{stream}?
\]
\[
s : \text{stream}?
\]
\[
e : \text{any/c}
\]

Returns a stream whose elements are the elements of \( s \), but with \( e \) between each pair of elements in \( s \). The new stream is constructed lazily.

\[
\text{(for/stream } (\text{for-clause} \ldots) \text{ body-or-break} \ldots \text{ body})
\]
\[
\text{(for*/stream } (\text{for-clause} \ldots) \text{ body-or-break} \ldots \text{ body})
\]

Iterates like \text{for/list} and \text{for*/list}, respectively, but the results are lazily collected into a stream instead of a list.

Unlike most \text{for} forms, these forms are evaluated lazily, so each \text{body} will not be evaluated until the resulting stream is forced. This allows \text{for/stream} and \text{for*/stream} to iterate over infinite sequences, unlike their finite counterparts.

Please note that these forms do not support returning multiple values.

Examples:

\[
> \text{(for/stream ([i '(1 2 3)]) (* i i))}
\]
\[
#<\text{stream}>
\]
\[
> \text{(stream->list (for/stream ([i '(1 2 3)]) (* i i)))}
\]
\[
'(1 4 9)
\]
\[
> \text{(stream-ref (for/stream ([i '(1 2 3)]) (displayln i) (* i i))) 1}
\]

422
2
4
> (stream-ref (for/stream ([i (in-naturals)]) (* i i)) 25) 625

Added in version 6.3.0.9 of package base.

||gen:stream : any/c

Associates three methods to a structure type to implement the generic interface (see §5.4 “Generic Interfaces”) for streams.

To supply method implementations, the #:methods keyword should be used in a structure type definition. The following three methods should be implemented:

- stream-empty?: accepts one argument
- stream-first: accepts one argument
- stream-rest : accepts one argument

Examples:

> (define-struct list-stream (v)
   #:methods gen:stream
   [(define (stream-empty? stream)
     (empty? (list-stream-v stream)))
    (define (stream-first stream)
     (first (list-stream-v stream)))
    (define (stream-rest stream)
     (list-stream (rest (list-stream-v stream))))]
> (define l1 (list-stream '(1 2)))
> (stream? l1) #t
> (stream-first l1) 1

||prop:stream : struct-type-property?

A structure type property used to define custom extensions to the stream API. Using the prop:stream property is discouraged; use the gen:stream generic interface instead. Accepts a vector of three procedures taking the same arguments as the methods in gen:stream.

| (stream/c c) -> contract? |
| c : contract? |
Returns a contract that recognizes streams. All elements of the stream must match \( c \).

If the \( c \) argument is a flat contract or a chaperone contract, then the result will be a chaperone contract. Otherwise, the result will be an impersonator contract.

When an `stream/c` contract is applied to a stream, the result is not `eq?` to the input. The result will be either a chaperone or impersonator of the input depending on the type of contract.

Contracts on streams are evaluated lazily by necessity (since streams may be infinite). Contract violations will not be raised until the value in violation is retrieved from the stream. As an exception to this rule, streams that are lists are checked immediately, as if \( c \) had been used with `listof`.

If a contract is applied to a stream, and that stream is subsequently used as the tail of another stream (as the second parameter to `stream-cons`), the new elements will not be checked with the contract, but the tail’s elements will still be enforced.

Added in version 6.1.1.8 of package `base`.

### 4.15.3 Generators

A **generator** is a procedure that returns a sequence of values, incrementing the sequence each time that the generator is called. In particular, the `generator` form implements a generator by evaluating a body that calls `yield` to return values from the generator.

```racket
(require racket/generator)  package: base

(generator? v) → boolean?
   v : any/c

Return `#t` if \( v \) is a generator, `#f` otherwise.

(generator formals body ...+)

```

Creates a generator, where `formals` specify the arguments. Keyword and optional arguments are not supported. This is the same as the `formals` of a single `case-lambda` clause.

For the first call to a generator, the arguments are bound to the `formals` and evaluation of `body` starts. During the dynamic extent of `body`, the generator can return immediately using the `yield` function. A second call to the generator resumes at the `yield` call, producing the
arguments of the second call as the results of the `yield`, and so on. The eventual results of `body` are supplied to an implicit final `yield`; after that final `yield`, calling the generator again returns the same values, but all such calls must provide 0 arguments to the generator.

Examples:

```scheme
> (define g (generator ()
   (let loop ([x '(a b c)])
     (if (null? x)
       0
       (begin
         (yield (car x))
         (loop (cdr x)))))))

> (g)
'a
> (g)
'b
> (g)
'c
> (g)
0
> (g)
0

(yield v ...) → any
v : any/c
```

Returns `vs` from a generator, saving the point of execution inside a generator (i.e., within the dynamic extent of a `generator` body) to be resumed by the next call to the generator. The results of `yield` are the arguments that are provided to the next call of the generator.

When not in the dynamic extent of a `generator`, infinite-generator, or `in-generator` body, `yield` raises `exn:fail` after evaluating its `exprs`.

Examples:

```scheme
> (define my-generator (generator () (yield 1) (yield 2 3 4)))
> (my-generator)
1
> (my-generator)
2
3
4

Examples:
```

425
> (define pass-values-generator
  (generator ()
    (let* ([from-user (yield 2)]
           [from-user-again (yield (add1 from-user))])
      (yield from-user-again)))
> (pass-values-generator)
2
> (pass-values-generator 5)
6
> (pass-values-generator 12)
12

(infinite-generator body ...+)

Like generator, but repeats evaluation of the bodies when the last body completes without implicitly yielding.

Examples:

> (define welcome
  (infinite-generator
    (yield 'hello)
    (yield 'goodbye)))
> (welcome)
'hello
> (welcome)
'goodbye
> (welcome)
'hello
> (welcome)
'goodbye

(in-generator maybe-arity body ...+)

maybe-arity =
   #:arity arity-k

Produces a sequence that encapsulates the generator formed by (generator () body ...+). The values produced by the generator form the elements of the sequence, except for the last value produced by the generator (i.e., the values produced by returning).

Example:

> (for/list ([i (in-generator
                  (let loop ([x '(a b c)])
                  (yield x))])
(when (not (null? x))
  (yield (car x))
  (loop (cdr x))))

'(a b c)

If `in-generator` is used immediately with a for (or for/list, etc.) binding’s right-hand side, then its result arity (i.e., the number of values in each element of the sequence) can be inferred. Otherwise, if the generator produces multiple values for each element, its arity should be declared with an `#:arity arity-k` clause; the `arity-k` must be a literal, exact, non-negative integer.

Examples:

> (let ([g (in-generator
(let loop ([n 3])
  (unless (zero? n) (yield n (add1 n)) (loop (sub1 n)))))
  (let-values ([(not-empty? next) (sequence-generate g)])
    (let loop () (when (not-empty?) (next) (loop)))) 'done))

stop?: arity mismatch;
the expected number of arguments does not match the given
number
expected: 1
given: 2

> (let ([g (in-generator #:arity 2
(let loop ([n 3])
  (unless (zero? n) (yield n (add1 n)) (loop (sub1 n)))))
  (let-values ([(not-empty? next) (sequence-generate g)])
    (let loop () (when (not-empty?) (next) (loop)))) 'done))

'done

To use an existing generator as a sequence, use `in-producer` with a stop-value known for the generator:

> (define abc-generator (generator ()
  (for ([x '(a b c)])
    (yield x))))

> (for/list ([i (in-producer abc-generator (void))])
  i)

'(a b c)

> (define my-stop-value (gensym))
> (define my-generator (generator ()
  (let loop ([x (list 'a (void) 'c)])
    (if (null? x)
      my-stop-value
      (yield x))))

427
(begin
  (yield (car x))
  (loop (cdr x)))))))

> (for/list ([i (in-producer my-generator my-stop-value)])
   i)
'(a #<void> c)

(generator-state g) → symbol?
g : generator?

Returns a symbol that describes the state of the generator.

- 'fresh — The generator has been freshly created and has not been called yet.
- 'suspended — Control within the generator has been suspended due to a call to yield. The generator can be called.
- 'running — The generator is currently executing.
- 'done — The generator has executed its entire body and will continue to produce the same result as from the last call.

Examples:

> (define my-generator (generator () (yield 1) (yield 2)))
> (generator-state my-generator)
'fresh
> (my-generator)
1
> (generator-state my-generator)
'suspended
> (my-generator)
2
> (generator-state my-generator)
'suspended
> (my-generator)
> (generator-state my-generator)
'done
> (define introspective-generator (generator () ((yield 1))))
> (introspective-generator)
1
> (introspective-generator
   (lambda () (generator-state introspective-generator)))
'running
> (generator-state introspective-generator)
sequence→generator \( s \) → (-> any)

\( s : \text{sequence?} \)

Converts a sequence to a generator. The generator returns the next element of the sequence each time the generator is invoked, where each element of the sequence must be a single value. When the sequence ends, the generator returns `#<void>` as its final result.

sequence→repeated-generator \( s \) → (-> any)

\( s : \text{sequence?} \)

Like sequence→generator, but when \( s \) has no further values, the generator starts the sequence again (so that the generator never stops producing values).

## 4.16 Dictionaries

A dictionary is an instance of a datatype that maps keys to values. The following datatypes are all dictionaries:

- hash tables;
- vectors (using only exact integers as keys);
- lists of pairs as an association list using `equal?` to compare keys, which must be distinct; and
- structures whose types implement the `gen:dict` generic interface.

When list of pairs is used as association list but does not have distinct keys (so it’s not an association list), operations like `dict-ref` and `dict-remove` operate on the first instance of the key, while operations like `dict-map` and `dict-keys` produce an element for every instance of the key.

(require racket/dict)  

package: base

The bindings documented in this section are provided by the `racket/dict` and `racket` libraries, but not `racket/base`. 
4.16.1 Dictionary Predicates and Contracts

\[(\text{dict? } v) \rightarrow \text{boolean?}\]
\[
v : \text{any/c}
\]

Returns \#t if \(v\) is a dictionary, \#f otherwise.

Beware that \text{dict?} is not a constant-time test on pairs, since checking that \(v\) is an association list may require traversing the list.

Examples:

\[
> (\text{dict? } \#\text{hash}((\text{a} . \text{"apple"})))
\]
\#

\[
> (\text{dict? } '\#(\text{"apple" } \text{"banana"}))
\]
\#

\[
> (\text{dict? } '(%\text{\"apple\" }\%\text{\"banana\"}))
\]
\#f

\[
> (\text{dict? } '((\text{a} . \text{"apple"}) (\text{b} . \text{"banana"})))
\]
\#t

\[(\text{dict-implements? } d \text{ sym ...}) \rightarrow \text{boolean?}\]
\[
d : \text{dict?}
\]
\[
sym : \text{symbol?}
\]

Returns \#t if \(d\) implements all of the methods from \text{gen:dict} named by the \text{sym}s; returns \#f otherwise. Fallback implementations do not affect the result; \(d\) may support the given methods via fallback implementations yet produce \#f.

Examples:

\[
> (\text{dict-implements? } (\text{hash } \text{\'a } \text{"apple"}) \text{\'dict-set!})
\]
\#f

\[
> (\text{dict-implements? } (\text{make-hash } '((\text{a} . \text{"apple"}) (\text{b} . \text{"banana"}))) \text{\'dict-set!})
\]
\#t

\[
> (\text{dict-implements? } (\text{make-hash } '((\text{b} . \text{"banana"}) (\text{a} . \text{"apple"}))) \text{\'dict-remove!})
\]
\#t

\[
> (\text{dict-implements? } (\text{vector } \text{"apple" } \text{"banana"}) \text{\'dict-set!})
\]
\#t

\[
> (\text{dict-implements? } (\text{vector } \text{\'a } \text{\'b}) \text{\'dict-remove!})
\]
\#f

\[
> (\text{dict-implements? } (\text{vector } \text{\'a } \text{"apple"}) \text{\'dict-set!} \text{\'dict-remove!})
\]
\#f

430
(dict-implements/c sym ...) → flat-contract?

sym : symbol?

Recognizes dictionaries that support all of the methods from gen:dict named by the sym.
Note that the generated contract is not similar to hash/c, but closer to dict-implements?.

Examples:

> (struct deformed-dict ()
   #:methods gen:dict [])
> (define/contract good-dict
dict-implements/c)
(deformed-dict))
> (define/contract bad-dict
dict-implements/c 'dict-ref)
(deformed-dict))

bad-dict: broke its own contract
promised: (dict-implements/c dict-ref)
produced: #<deformed-dict>
in: (dict-implements/c dict-ref)
contract from: (definition bad-dict)
blaming: (definition bad-dict)
(assuming the contract is correct)
at: eval:14:0

(dict-mutable? d) → boolean?
d : dict?

Returns #t if d is mutable via dict-set!, #f otherwise.

Equivalent to (dict-implements? d 'dict-set!).

Examples:

> (dict-mutable? #hash((a . "apple")))
#f
> (dict-mutable? (make-hash))
#t
> (dict-mutable? '('#("apple" "banana"))
#f
> (dict-mutable? (vector "apple" "banana")
#t
> (dict-mutable? '((a . "apple") (b . "banana")))
#f
(dict-can-remove-keys? d) → boolean?
   d : dict?

Returns #t if d supports removing mappings via dict-remove! and/or dict-remove, #f otherwise.

Equivalent to (or (dict-implements? d 'dict-remove!) (dict-implements? d 'dict-remove)).

Examples:
   > (dict-can-remove-keys? #hash((a . "apple")))
      #t
   > (dict-can-remove-keys? '('#("apple" "banana")))
      #f
   > (dict-can-remove-keys? '((a . "apple") (b . "banana")))
      #t

(dict-can-functional-set? d) → boolean?
   d : dict?

Returns #t if d supports functional update via dict-set, #f otherwise.

Equivalent to (dict-implements? d 'dict-set).

Examples:
   > (dict-can-functional-set? #hash((a . "apple")))
      #t
   > (dict-can-functional-set? (make-hash))
      #f
   > (dict-can-functional-set? '('#("apple" "banana")))
      #f
   > (dict-can-functional-set? '((a . "apple") (b . "banana")))
      #t

4.16.2 Generic Dictionary Interface

gen:dict

A generic interface (see §5.4 “Generic Interfaces”) that supplies dictionary method implementations for a structure type via the #:methods option of struct definitions. This interface can be used to implement any of the methods documented as §4.16.2.1 “Primitive Dictionary Methods” and §4.16.2.2 “Derived Dictionary Methods”
Examples:

```scheme
> (struct alist (v)
   #:methods gen:dict
   [(define (dict-ref dict key
        [default (lambda () (error "key not found" key))]
      (cond [(assoc key (alist-v dict)) => cdr]
           [else (if (procedure? default) (default) default)]))]
    (define (dict-set dict key val)
      (alist (cons (cons key val) (alist-v dict))))
    (define (dict-remove dict key)
      (define al (alist-v dict))
      (alist (remove* (filter (λ (p) (equal? (car p) key)) al) al)))
    (define (dict-count dict)
      (length (remove-duplicates (alist-v dict) #:key car))))
    ; etc. other methods
> (define d1 (alist '((1 . a) (2 . b))))
> (dict? d1)
#t
> (dict-ref d1 1)
'a
> (dict-remove d1 1)
#<alist

prop:dict : struct-type-property?

A structure type property used to define custom extensions to the dictionary API. Using the prop:dict property is discouraged; use the gen:dict generic interface instead. Accepts a vector of 10 method implementations:

- dict-ref
- dict-set!, or #f if unsupported
- dict-set, or #f if unsupported
- dict-remove!, or #f if unsupported
- dict-remove, or #f if unsupported
- dict-count
- dict-iterate-first
- dict-iterate-next
- dict-iterate-key
• dict-iterate-value

Primitive Dictionary Methods

These methods of gen:dict have no fallback implementations; they are only supported for dictionary types that directly implement them.

\[
\text{(dict-ref dict key [failure-result]) \to any}
\]
\[
\begin{aligned}
\text{dict} &: \text{dict} \\
\text{key} &: \text{any/c} \\
\text{failure-result} &: \text{failure-result/c}
\end{aligned}
= (\lambda () (\text{raise (make-exn:fail ...))))
\]

Returns the value for \( \text{key} \) in \( \text{dict} \). If no value is found for \( \text{key} \), then \( \text{failure-result} \) determines the result:

- If \( \text{failure-result} \) is a procedure, it is called (through a tail call) with no arguments to produce the result.
- Otherwise, \( \text{failure-result} \) is returned as the result.

Examples:

\[
\begin{align*}
> & \text{(dict-ref #hash((a . "apple") (b . "beer")) 'a)} \\
& \text{"apple"} \\
> & \text{(dict-ref #hash((a . "apple") (b . "beer")) 'c)} \\
& \text{hash-ref: no value found for key}\\n& \quad \text{key: 'c} \\
> & \text{(dict-ref #hash((a . "apple") (b . "beer")) 'c #f)} \\
& \text{#f} \\
> & \text{(dict-ref '((a . "apple") (b . "banana")) 'b)} \\
& \text{"banana"} \\
> & \text{(dict-ref "apple" "banana" 1)} \\
& \text{"banana"} \\
> & \text{(dict-ref "apple" "banana" 3 #f)} \\
& \text{#f} \\
> & \text{(dict-ref "apple" "banana" -3 #f)} \\
& \text{dict-ref: contract violation}\\n& \text{expected: natural?} \\
& \text{given: -3} \\
& \text{in: the k argument of}\\n& \quad (\rightarrow i) \\
& \quad \text{(((d dict?) (k (d) (dict-key-contract d)))}} \\
& \quad ((\text{default any/c})) \\
& \quad \text{any}
\end{align*}
\]
contract from: <collects>/racket/dict.rkt
blaming: top-level
(assuming the contract is correct)
at: <collects>/racket/dict.rkt:182:2

(dict-set! dict key v) → void?
dict : (and/c dict? (not/c immutable?))
key : any/c
v : any/c

Maps key to v in dict, overwriting any existing mapping for key. The update can fail with a exn:fail:contract exception if dict is not mutable or if key is not an allowed key for the dictionary (e.g., not an exact integer in the appropriate range when dict is a vector).

Examples:

> (define h (make-hash))
> (dict-set! h 'a "apple")
> h
'#{(a . "apple")}
> (define v (vector #f #f #f))
> (dict-set! v 0 "apple")
> v
'#{"apple" #f #f}

(dict-set dict key v) → (and/c dict? immutable?)
dict : (and/c dict? immutable?)
key : any/c
v : any/c

Functionally extends dict by mapping key to v, overwriting any existing mapping for key, and returning an extended dictionary. The update can fail with a exn:fail:contract exception if dict does not support functional extension or if key is not an allowed key for the dictionary.

Examples:

> (dict-set #hash() 'a "apple")
'#{(a . "apple")}
> (dict-set #hash((a . "apple") (b . "beer")) 'b "banana")
'#{(a . "apple") (b . "banana")}
> (dict-set '() 'a "apple")
'#{(a . "apple")}
> (dict-set '((a . "apple") (b . "beer")) 'b "banana")
'#{(a . "apple") (b . "banana")}
(dict-remove! dict key) → void?
dict : (and/c dict? (not/c immutable?))
key : any/c

Removes any existing mapping for key in dict. The update can fail if dict is not mutable or does not support removing keys (as is the case for vectors, for example).

Examples:
> (define h (make-hash))
> (dict-set! h 'a "apple")
> h
'hash((a . "apple"))
> (dict-remove! h 'a)
> h
'hash()

(dict-remove dict key) → (and/c dict? immutable?)
dict : (and/c dict? immutable?)
key : any/c

Functionally removes any existing mapping for key in dict, returning the fresh dictionary. The update can fail if dict does not support functional update or does not support removing keys.

Examples:
> (define h hash)
> (define h (dict-set h 'a "apple")
> h
'hash((a . "apple")
> (dict-remove h 'a)
'hash()
> h
'hash((a . "apple")
> (dict-remove h 'z)
'hash((a . "apple")
> (dict-remove '((a . "apple") (b . "banana") 'a)
'((b . "banana")

(dict-iterate-first dict) → any/c
dict : dict?

Returns #f if dict contains no elements, otherwise it returns a non-#f value that is an index to the first element in the dict table; “first” refers to an unspecified ordering of the dictionary
elements. For a mutable `dict`, this index is guaranteed to refer to the first item only as long as no mappings are added to or removed from `dict`.

Examples:

```lisp
> (dict-iterate-first #hash((a . "apple") (b . "banana")))
0
> (dict-iterate-first #hash())
#f
> (dict-iterate-first #("apple" "banana"))
0
> (dict-iterate-first '((a . "apple") (b . "banana")))
#<assoc-iter>
```

```
(dict-iterate-next dict pos) → any/c
  dict : dict?
  pos : any/c
```

Returns either a non-`#f` that is an index to the element in `dict` after the element indexed by `pos` or `#f` if `pos` refers to the last element in `dict`. If `pos` is not a valid index, then the `exn:fail:contract` exception is raised. For a mutable `dict`, the result index is guaranteed to refer to its item only as long as no items are added to or removed from `dict`. The `dict-iterate-next` operation should take constant time.

Examples:

```lisp
> (define h #hash((a . "apple") (b . "banana")))
> (define i (dict-iterate-first h))
> i
0
> (dict-iterate-next h i)
1
> (dict-iterate-next h (dict-iterate-next h i))
#f
```

```
(dict-iterate-key dict pos) → any
  dict : dict?
  pos : any/c
```

Returns the key for the element in `dict` at index `pos`. If `pos` is not a valid index for `dict`, the `exn:fail:contract` exception is raised. The `dict-iterate-key` operation should take constant time.

Examples:
> (define h '(((a . "apple") (b . "banana")))
> (define i (dict-iterate-first h))
> (dict-iterate-key h i)
  'a
> (dict-iterate-key h (dict-iterate-next h i))
  'b

(dict-iterate-value dict pos) → any
  dict : dict?
  pos : any/c

Returns the value for the element in dict at index pos. If pos is not a valid index for dict, the exn:fail:contract exception is raised. The dict-iterate-key operation should take constant time.

Examples:

> (define h '(((a . "apple") (b . "banana")))
> (define i (dict-iterate-first h))
> (dict-iterate-value h i)
  "apple"
> (dict-iterate-value h (dict-iterate-next h i))
  "banana"

Derived Dictionary Methods

These methods of gen:dict have fallback implementations in terms of the other methods; they may be supported even by dictionary types that do not directly implement them.

(dict-has-key? dict key) → boolean?
  dict : dict?
  key : any/c

Returns #t if dict contains a value for the given key, #f otherwise.

Supported for any dict that implements dict-ref.

Examples:

> (dict-has-key? #hash((a . "apple") (b . "beer")) 'a)
  #t
> (dict-has-key? #hash((a . "apple") (b . "beer")) 'c)
  #f
> (dict-has-key? '(((a . "apple") (b . "banana"))) 'b)
  #t
> (dict-has-key? #("apple" "banana") 1)
#t
> (dict-has-key? #("apple" "banana") 3)
#f
> (dict-has-key? #("apple" "banana") -3)
#f

(dict-set*! dict key v ... ...) → void?
dict : (and/c dict? (not/c immutable?))
key : any/c
v : any/c

Maps each key to each v in dict, overwriting any existing mapping for each key. The update can fail with a exn:fail:contract exception if dict is not mutable or if any key is not an allowed key for the dictionary (e.g., not an exact integer in the appropriate range when dict is a vector). The update takes place from the left, so later mappings overwrite earlier mappings.

Supported for any dict that implements dict-set!.

Examples:

> (define h (make-hash))
> (dict-set*! h 'a "apple" 'b "banana")
> h
'#hash((a . "apple") (b . "banana"))
> (define v1 (vector #f #f #f))
> (dict-set*! v1 0 "apple" 1 "banana")
> v1
'#("apple" "banana" #f)
> (define v2 (vector #f #f #f))
> (dict-set*! v2 0 "apple" 0 "banana")
> v2
'#("banana" #f #f)

(dict-set*! dict key v ... ...) → (and/c dict? immutable?)
dict : (and/c dict? immutable?)
key : any/c
v : any/c

Functionally extends dict by mapping each key to each v, overwriting any existing mapping for each key, and returning an extended dictionary. The update can fail with a exn:fail:contract exception if dict does not support functional extension or if any key is not an allowed key for the dictionary. The update takes place from the left, so later mappings overwrite earlier mappings.
Supported for any `dict` that implements `dict-set`.

Examples:

```scheme
> (dict-set* #hash() 'a "apple" 'b "beer")
'#hash((a . "apple") (b . "beer"))
> (dict-set* #hash((a . "apple") (b . "beer")) 'b "banana" 'a "anchor")
'#hash((a . "anchor") (b . "banana"))
> (dict-set* '() 'a "apple" 'b "beer")
'((a . "apple") (b . "beer"))
> (dict-set* '((a . "apple") (b . "beer")) 'b "banana" 'a "anchor")
'((a . "anchor") (b . "banana"))
> (dict-set* '((a . "apple") (b . "beer")) 'b "banana" 'b "ballistic")
'((a . "apple") (b . "ballistic"))
```

Returns the value for `key` in `dict`. If no value is found for `key`, then `to-set` determines the result as in `dict-ref` (i.e., it is either a thunk that computes a value or a plain value), and this result is stored in `dict` for the `key`. (Note that if `to-set` is a thunk, it is not invoked in tail position.)

Supported for any `dict` that implements `dict-ref` and `dict-set!`.

Examples:

```scheme
> (dict-ref! (make-hasheq '((a . "apple") (b . "beer"))) 'a #f)
"apple"
> (dict-ref! (make-hasheq '((a . "apple") (b . "beer"))) 'c 'cabbage)
'cabbage
> (define h (make-hasheq '((a . "apple") (b . "beer"))))
> (dict-ref h 'c)
hash-ref: no value found for key
key: 'c
> (dict-ref! h 'c (λ () 'cabbage))
'cabbage
> (dict-ref h 'c)
'cabbage
```

```
(dict-update! dict
   key
   updater
   [failure-result]) → void?
```
Composes `dict-ref` and `dict-set!` to update an existing mapping in `dict`, where the optional `failure-result` argument is used as in `dict-ref` when no mapping exists for `key` already.

Supported for any `dict` that implements `dict-ref` and `dict-set!`.

Examples:

```scheme
> (define h (make-hash))
> (dict-update! h 'a add1)
hash-update: no value found for key: 'a
> (dict-update! h 'a add1 0)
> h
'#hash((a . 1))
> (define v (vector #f #f #f))
> (dict-update! v 0 not)
> v
'#(#t #f #f)
```

```
(dict-update dict key updater [failure-result])
→ (and/c dict? immutable?)
dict : dict?
key : any/c
updater : (any/c . -> . any/c)
failure-result : failure-result/c
               = (lambda () (raise (make-exn:fail ....)))
```

Composes `dict-ref` and `dict-set` to functionally update an existing mapping in `dict`, where the optional `failure-result` argument is used as in `dict-ref` when no mapping exists for `key` already.

Supported for any `dict` that implements `dict-ref` and `dict-set`.

Examples:

```scheme
> (dict-update #hash() 'a add1)
hash-update: no value found for key: 'a
> (dict-update #hash() 'a add1 0)
'#hash((a . 1))
> (dict-update #hash((a . "apple") (b . "beer")) 'b string-length)
'#hash((a . "apple") (b . 4))
```
(dict-map dict proc) → (listof any/c)
  dict : dict?
  proc : (any/c any/c . -> . any/c)

Applies the procedure proc to each element in dict in an unspecified order, accumulating the results into a list. The procedure proc is called each time with a key and its value.

Supported for any dict that implements dict-iterate-first, dict-iterate-next, dict-iterate-key, and dict-iterate-value.

Example:
> (dict-map #hash((a . "apple") (b . "banana")) vector)
'(#(b "banana") #(a "apple"))

(dict-for-each dict proc) → void?
  dict : dict?
  proc : (any/c any/c . -> . any)

Applies proc to each element in dict (for the side-effects of proc) in an unspecified order. The procedure proc is called each time with a key and its value.

Supported for any dict that implements dict-iterate-first, dict-iterate-next, dict-iterate-key, and dict-iterate-value.

Example:
> (dict-for-each #hash((a . "apple") (b . "banana"))
  (lambda (k v)
    (printf "a = ~s\n" k v))
  (b "banana")
  (a "apple")

(dict-empty? dict) → boolean?
  dict : dict?

Reports whether dict is empty.

Supported for any dict that implements dict-iterate-first.

Examples:
> (dict-empty? #hash((a . "apple") (b . "banana")))
#f
> (dict-empty? (vector))
#t

442
(dict-count dict) → exact-nonnegative-integer?
dict : dict?

Returns the number of keys mapped by dict, usually in constant time.

Supported for any dict that implements dict-iterate-first and dict-iterate-next.

Examples:

> (dict-count #hash((a . "apple") (b . "banana")))
2
> (dict-count #("apple" "banana"))
2

(dict-copy dict) → dict?
dict : dict?

Produces a new, mutable dictionary of the same type as dict and with the same key/value associations.

Supported for any dict that implements dict-clear, dict-set!, dict-iterate-first, dict-iterate-next, dict-iterate-key, and dict-iterate-value.

Examples:

> (define original (vector "apple" "banana"))
> (define copy (dict-copy original))
> original
'#("apple" "banana")
> copy
'#("apple" "banana")
> (dict-set! copy 1 "carrot")
> original
'#("apple" "banana")
> copy
'#("apple" "carrot")

(dict-clear dict) → dict?
dict : dict?

Produces an empty dictionary of the same type as dict. If dict is mutable, the result must be a new dictionary.
Supported for any `dict` that supports `dict-remove`, `dict-iterate-first`, `dict-iterate-next`, and `dict-iterate-key`.

Examples:

```scheme
> (dict-clear #hash((a . "apple") ("banana" . b)))
'#hash()
> (dict-clear '((1 . two) (three . "four")))
'
```

```
(dict-clear! dict) → void?
dict : dict?
```

Removes all of the key/value associations in `dict`.

Supported for any `dict` that supports `dict-remove!`, `dict-iterate-first`, and `dict-iterate-key`.

Examples:

```scheme
> (define table (make-hash))
> (dict-set! table 'a "apple")
> (dict-set! table "banana" 'b)
> table
'#hash((a . "apple") ("banana" . b))
> (dict-clear! table)
> table
'#hash()
```

```
(dict-keys dict) → list?
dict : dict?
```

Returns a list of the keys from `dict` in an unspecified order.

Supported for any `dict` that implements `dict-iterate-first`, `dict-iterate-next`, and `dict-iterate-key`.

Examples:

```scheme
> (define h #hash((a . "apple") ("banana") (b . "four")))
> (dict-keys h)
'(b a)
```

```
(dict-values dict) → list?
dict : dict?
```
Returns a list of the values from `dict` in an unspecified order.

Supported for any `dict` that implements `dict-iterate-first`, `dict-iterate-next`, and `dict-iterate-value`.

Examples:

```
> (define h #hash((a . "apple") (b . "banana")))
> (dict-values h)
'("banana" "apple")
```

```
(dict->list dict) → list?
dict : dict?
```

Returns a list of the associations from `dict` in an unspecified order.

Supported for any `dict` that implements `dict-iterate-first`, `dict-iterate-next`, `dict-iterate-key`, and `dict-iterate-value`.

Examples:

```
> (define h #hash((a . "apple") (b . "banana")))
> (dict->list h)
'((b . "banana") (a . "apple"))
```

```
4.16.3 Dictionary Sequences
```

```
(in-dict dict) → sequence?
dict : dict?
```

Returns a sequence whose each element is two values: a key and corresponding value from `dict`.

Supported for any `dict` that implements `dict-iterate-first`, `dict-iterate-next`, `dict-iterate-key`, and `dict-iterate-value`.

Examples:

```
> (define h #hash((a . "apple") (b . "banana")))
> (for/list ([[(k v) (in-dict h)]]
            (format "~a = ~s" k v))
  "b = \"banana\"" "a = \"apple\"")
```

```
(in-dict-keys dict) → sequence?
dict : dict?
```

445
Returns a sequence whose elements are the keys of `dict`.

Supported for any `dict` that implements `dict-iterate-first`, `dict-iterate-next`, and `dict-iterate-key`.

Examples:

```scheme
> (define h #hash((a . "apple") (b . "banana")))
> (for/list ([k (in-dict-keys h)]) k)
'(b a)
```

**(in-dict-values dict) → sequence?**

dict : dict?

Returns a sequence whose elements are the values of `dict`.

Supported for any `dict` that implements `dict-iterate-first`, `dict-iterate-next`, and `dict-iterate-value`.

Examples:

```scheme
> (define h #hash((a . "apple") (b . "banana")))
> (for/list ([v (in-dict-values h)]) v)
'("banana" "apple")
```

**(in-dict-pairs dict) → sequence?**

dict : dict?

Returns a sequence whose elements are pairs, each containing a key and its value from `dict` (as opposed to using `in-dict`, which gets the key and value as separate values for each element).

Supported for any `dict` that implements `dict-iterate-first`, `dict-iterate-next`, `dict-iterate-key`, and `dict-iterate-value`.

Examples:

```scheme
> (define h #hash((a . "apple") (b . "banana")))
> (for/list ([p (in-dict-pairs h)]) p)
'((b . "banana") (a . "apple"))
```

### 4.16.4 Contracted Dictionaries

```scheme
| prop:dict/contract : struct-type-property? |
```
A structure type property for defining dictionaries with contracts. The value associated with prop:dict/contract must be a list of two immutable vectors:

```
(list dict-vector
     (vector type-key-contract
type-value-contract
type-iter-contract
instance-key-contract
instance-value-contract
instance-iter-contract))
```

The first vector must be a vector of 10 procedures which match the gen:dict generic interface (in addition, it must be an immutable vector). The second vector must contain six elements; each of the first three is a contract for the dictionary type’s keys, values, and positions, respectively. Each of the second three is either #f or a procedure used to extract the contract from a dictionary instance.

```
(dict-key-contract d) → contract?
d : dict?
(dict-value-contract d) → contract?
d : dict?
(dict-iter-contract d) → contract?
d : dict?
```

Returns the contract that d imposes on its keys, values, or iterators, respectively, if d implements the prop:dict/contract interface.

### 4.16.5 Custom Hash Tables

```
(define-custom-hash-types name
optional-predicate
comparison-expr
optional-hash-functions)
```

optional-predicate =
| #:key? predicate-expr

optional-hash-functions =
| hash1-expr
| hash1-expr hash2-expr

Creates a new dictionary type based on the given comparison comparison-expr, hash functions hash1-expr and hash2-expr, and key predicate predicate-expr; the interfaces for these functions are the same as in make-custom-hash-types. The new dictio-
nary type has three variants: immutable, mutable with strongly-held keys, and mutable with weakly-held keys.

Defines seven names:

- **name?** recognizes instances of the new type,
- **immutable-name?** recognizes immutable instances of the new type,
- **mutable-name?** recognizes mutable instances of the new type with strongly-held keys,
- **weak-name?** recognizes mutable instances of the new type with weakly-held keys,
- **make-immutable-name** constructs immutable instances of the new type,
- **make-mutable-name** constructs mutable instances of the new type with strongly-held keys, and
- **make-weak-name** constructs mutable instances of the new type with weakly-held keys.

The constructors all accept a dictionary as an optional argument, providing initial key/value pairs.

Examples:

```
> (define-custom-hash-types string-hash
    #:key? string?
    string=?
    string-length)
> (define imm
    (make-immutable-string-hash
     '(("apple" . a) ("banana" . b))))
> (define mut
    (make-mutable-string-hash
     '(("apple" . a) ("banana" . b))))
> (dict? imm)
 #t
> (dict? mut)
 #t
> (string-hash? imm)
 #t
> (string-hash? mut)
 #t
> (immutable-string-hash? imm)
 #t
> (immutable-string-hash? mut)
```

448
#f
> (dict-ref imm "apple")
'a
> (dict-ref mut "banana")
'b
> (dict-set! mut "banana" 'berry)
> (dict-ref mut "banana")
'berry
> (equal? imm mut)
#f
> (equal? (dict-remove (dict-remove imm "apple") "banana")
(make-immutable-string-hash))
#t

(make-custom-hash-types eql? [hash1 hash2 #:key? key? #:name name #:for who]) → (any/c . -> . boolean?)
(any/c . -> . boolean?)
(any/c . -> . boolean?)
(any/c . -> . boolean?)
(->* [] [dict?] dict?)
(->* [] [dict?] dict?)
(->* [] [dict?] dict?)
eql?: (or/c (any/c any/c . -> . any/c)
    (any/c any/c (any/c any/c . -> . any/c) . -> . any/c))
hash1: (or/c (any/c . -> . exact-integer?)
    (any/c (any/c . -> . exact-integer?) . -> . exact-integer?))
    = (const 1)
hash2: (or/c (any/c . -> . exact-integer?)
    (any/c (any/c . -> . exact-integer?) . -> . exact-integer?))
    = (const 1)
key?: (any/c . -> . boolean?) = (const #true)
name : symbol? = 'custom-hash
who : symbol? = 'make-custom-hash-types

Creates a new dictionary type based on the given comparison function eql?, hash functions hash1 and hash2, and predicate key?. The new dictionary type has variants that are immutable, mutable with strongly-held keys, and mutable with weakly-held keys. The given name is used when printing instances of the new dictionary type, and the symbol who is used for reporting errors.

The comparison function eql? may accept 2 or 3 arguments. If it accepts 2 arguments, it given two keys to compare them. If it accepts 3 arguments and does not accept 2 arguments,
it is also given a recursive comparison function that handles data cycles when comparing sub-parts of the keys.

The hash functions $hash1$ and $hash2$ may accept 1 or 2 arguments. If either hash function accepts 1 argument, it is applied to a key to compute the corresponding hash value. If either hash function accepts 2 arguments and does not accept 1 argument, it is also given a recursive hash function that handles data cycles when computing hash values of sub-parts of the keys.

The predicate $key?$ must accept 1 argument and is used to recognize valid keys for the new dictionary type.

Produces seven values:

- a predicate recognizing all instances of the new dictionary type,
- a predicate recognizing immutable instances,
- a predicate recognizing mutable instances,
- a predicate recognizing weak instances,
- a constructor for immutable instances,
- a constructor for mutable instances, and
- a constructor for weak instances.

See `define-custom-hash-types` for an example.

```
(make-custom-hash eql?  
[hsh1  
[hsh2  
#:key? key?])  
-> dict?  
eql? : (or/c (any/c any/c . -> . any/c)  
(any/c any/c (any/c any/c . -> . any/c) . -> . any/c))  
hash1 : (or/c (any/c . -> . exact-integer?)  
(any/c (any/c . -> . exact-integer?) . -> . exact-integer?))  
= (const 1)  
hash2 : (or/c (any/c . -> . exact-integer?)  
(any/c (any/c . -> . exact-integer?) . -> . exact-integer?))  
= (const 1)  
key? : (any/c . -> . boolean?) = (λ (x) #true)
```
(make-weak-custom-hash  eql?
  [hash1
   hash2
   #:key?  key?]) → dict?

  eql? : (or/c (any/c any/c . -> . any/c)
           (any/c any/c (any/c any/c . -> . any/c) . -> . any/c))
hash1 : (or/c (any/c . -> . exact-integer?)
         (any/c (any/c . -> . exact-integer?) . -> . exact-integer?))
      = (const 1)
hash2 : (or/c (any/c . -> . exact-integer?)
         (any/c (any/c . -> . exact-integer?) . -> . exact-integer?))
      = (const 1)
key? : (any/c . -> . boolean?) = (λ (x) #true)

(make-immutable-custom-hash  eql?
  [hash1
   hash2
   #:key?  key?]) → dict?

  eql? : (or/c (any/c any/c . -> . any/c)
           (any/c any/c (any/c any/c . -> . any/c) . -> . any/c))
hash1 : (or/c (any/c . -> . exact-integer?)
         (any/c (any/c . -> . exact-integer?) . -> . exact-integer?))
      = (const 1)
hash2 : (or/c (any/c . -> . exact-integer?)
         (any/c (any/c . -> . exact-integer?) . -> . exact-integer?))
      = (const 1)
key? : (any/c . -> . boolean?) = (λ (x) #true)

Creates an instance of a new dictionary type, implemented in terms of a hash table where keys are compared with \texttt{eql?}, hashed with \texttt{hash1} and \texttt{hash2}, and where the key predicate is \texttt{key?}. See \texttt{gen:equal+hash} for information on suitable equality and hashing functions.

The \texttt{make-custom-hash} and \texttt{make-weak-custom-hash} functions create a mutable dictionary that does not support functional update, while \texttt{make-immutable-custom-hash} creates an immutable dictionary that supports functional update. The dictionary created by \texttt{make-weak-custom-hash} retains its keys weakly, like the result of \texttt{make-weak-hash}.

Dictionaries created by \texttt{make-custom-hash} and company are \texttt{equal?} when they have the same mutability and key strength, the associated procedures are \texttt{equal?}, and the key–value mappings are the same when keys and values are compared with \texttt{equal?}.

See also \texttt{define-custom-hash-types}.

Examples:

```scheme
> (define h (make-custom-hash (lambda (a b)
       (string=? (format "~a" a)
                  (format "~a" b)))
```

451
(lambda (a)
  (equal-hash-code
   (format "~a" a))))

> (dict-set! h 1 'one)
> (dict-ref h "1")
'one

4.16.6 Passing keyword arguments in dictionaries

(keyword-apply/dict proc
  kw-dict
  pos-arg ...
  pos-args
  #:<kw> kw-arg ...) → any

proc : procedure?
kw-dict : dict?
pos-arg : any/c
pos-args : (listof any/c)
kw-arg : any/c

Applies the proc using the positional arguments from (list* pos-arg ... pos-args), and the keyword arguments from kw-dict in addition to the directly supplied keyword arguments in the #:<kw> kw-arg sequence.

All the keys in kw-dict must be keywords. The keywords in the kw-dict do not have to be sorted. However, the keywords in kw-dict and the directly supplied #:<kw> keywords must not overlap. The given proc must accept all of the keywords in kw-dict plus the #:<kw>s.

Examples:

> (define (sundae #:ice-cream [ice-cream "vanilla"]
  #:toppings [toppings "brownie-bits"]
  #:sprinkles [sprinkles "chocolate"]
  #:syrup [syrup "caramel"])
(format 
  "A sundae with ~a ice cream, ~a, ~a sprinkles, and ~a syrup."
  (string-join ice-cream #:before-last " and ")
  (string-join toppings #:before-last " and ")
  sprinkles
  syrup))
> (keyword-apply/dict sundae "((#:ice-cream "chocolate") "()"
"A sundae with chocolate ice cream, brownie-bits, chocolate sprinkles, and caramel syrup."

452
> (keyword-apply/dict sundae
  (hash #:toppings "cookie-dough"
       #:sprinkles "rainbow"
       #:syrup "chocolate"
       '()))
"A sundae with vanilla ice cream, cookie-dough, rainbow sprinkles, and chocolate syrup."

> (keyword-apply/dict sundae
  #:sprinkles "rainbow"
  (hash #:toppings "cookie-dough"
       #:syrup "chocolate"
       '()))
"A sundae with vanilla ice cream, cookie-dough, rainbow sprinkles, and chocolate syrup."

Added in version 7.9 of package base.

4.17 Sets

A set represents a collection of distinct elements. The following datatypes are all sets:

- hash sets;
- lists using equal? to compare elements; and
- structures whose types implement the gen:set generic interface.

(require racket/set)  package:base

The bindings documented in this section are provided by the racket/set and racket libraries, but not racket/base.

4.17.1 Hash Sets

A hash set is a set whose elements are compared via equal?, eqv?, or eq? and partitioned via equal-hash-code, eqv-hash-code, or eq-hash-code. A hash set is either immutable or mutable; mutable hash sets retain their elements either strongly or weakly.

A hash set can be used as a stream (see §4.15.2 "Streams") and thus as a single-valued sequence (see §4.15.1 "Sequences"). The elements of the set serve as elements of the stream or sequence. If an element is added to or removed from the hash set during iteration, then an iteration step may fail with exn:fail:contract, or the iteration may skip or duplicate elements. See also in-set.
Two hash sets are equal? when they use the same element-comparison procedure (equal?, eqv?, or eq?), both hold elements strongly or weakly, have the same mutability, and have equivalent elements. Immutable hash sets support effectively constant-time access and update, just like mutable hash sets; the constant on immutable operations is usually larger, but the functional nature of immutable hash sets can pay off in certain algorithms.


Operations on sets that contain elements that are mutated are unpredictable in much the same way that hash table operations are unpredictable when keys are mutated.

```
(set-equal? x) -> boolean?
  x : any/c
(set-eqv? x) -> boolean?
  x : any/c
(set-eq? x) -> boolean?
  x : any/c
```

Returns #t if x is a hash set that compares elements with equal?, eqv?, or eq?, respectively; returns #f otherwise.

```
(set? x) -> boolean?
  x : any/c
(set-mutable? x) -> boolean?
  x : any/c
(set-weak? x) -> boolean?
  x : any/c
```

Returns #t if x is a hash set that is respectively immutable, mutable with strongly-held keys, or mutable with weakly-held keys; returns #f otherwise.

```
(set v ...) -> (and/c generic-set? set-equal? set?)
  v : any/c
(seteqv v ...) -> (and/c generic-set? set-eqv? set?)
  v : any/c
(seteq v ...) -> (and/c generic-set? set-eq? set?)
  v : any/c
(mutable-set v ...)
  -> (and/c generic-set? set-equal? set-mutable?)
  v : any/c
```
(mutable-seteqv v ...)  
→ (and/c generic-set? set-eqv? set-mutable?)
  v : any/c
(mutable-seteq v ...)  
→ (and/c generic-set? set-eq? set-mutable?)
  v : any/c
(weak-set v ...)  
→ (and/c generic-set? set-equal? set-weak?)
  v : any/c
(weak-seteqv v ...)  
→ (and/c generic-set? set-eqv? set-weak?)
  v : any/c
(weak-seteq v ...)  
→ (and/c generic-set? set-eq? set-weak?)
  v : any/c

Creates a hash set with the given vs as elements. The elements are added in the order that they appear as arguments, so in the case of sets that use equal? or eqv?, an earlier element may be replaced by a later element that is equal? or eqv? but not eq?.

(list->set lst)  
→ (and/c generic-set? set-equal? set?)
  lst : list?
(list->seteqv lst)  
→ (and/c generic-set? set-eqv? set?)
  lst : list?
(list->seteq lst)  
→ (and/c generic-set? set-eq? set?)
  lst : list?
(list->mutable-set lst)  
→ (and/c generic-set? set-equal? set-mutable?)
  lst : list?
(list->mutable-seteqv lst)  
→ (and/c generic-set? set-eqv? set-mutable?)
  lst : list?
(list->mutable-seteq lst)  
→ (and/c generic-set? set-eq? set-mutable?)
  lst : list?
(list->weak-set lst)  
→ (and/c generic-set? set-equal? set-weak?)
  lst : list?
(list->weak-seteqv lst)  
→ (and/c generic-set? set-eqv? set-weak?)
  lst : list?
(list->weak-seteq lst)  
→ (and/c generic-set? set-eq? set-weak?)
  lst : list?
(list->weak-seteqv lst)  
→ (and/c generic-set? set-eqv? set-weak?)
  lst : list?

Creates a hash set with the elements of the given lst as the elements of the set. Equivalent to (apply set lst), (apply seteqv lst), (apply seteq lst), and so on, respectively.
Analogous to for/list and for*/list, but to construct a hash set instead of a list.

\[
\text{(in-immutable-set } st \text{)} \rightarrow \text{sequence?}
\]
\[
\text{st : set?}
\]
\[
\text{(in-mutable-set } st \text{)} \rightarrow \text{sequence?}
\]
\[
\text{st : set-mutable?}
\]
\[
\text{(in-weak-set } st \text{)} \rightarrow \text{sequence?}
\]
\[
\text{st : set-weak?}
\]

Explicitly converts a specific kind of hash set to a sequence for use with for forms.

As with in-list and some other sequence constructors, in-immutable-set performs better when it appears directly in a for clause.

These sequence constructors are compatible with §4.17.4 “Custom Hash Sets”.

Added in version 6.4.0.7 of package base.

### 4.17.2 Set Predicates and Contracts

\[
\text{(generic-set? } v \text{)} \rightarrow \text{boolean?}
\]
\[
v : \text{any/c}
\]

Returns #t if v is a set; returns #f otherwise.

Examples:
> (generic-set? (list 1 2 3))
#t
> (generic-set? (set 1 2 3))
#t
> (generic-set? (mutable-seteq 1 2 3))
#t
> (generic-set? (vector 1 2 3))
#f

(set-implements? st sym ...) → boolean?
st : generic-set?
sym : symbol?

Returns #t if st implements all of the methods from gen:set named by the syms; returns #f otherwise. Fallback implementations do not affect the result; st may support the given methods via fallback implementations yet produce #f.

Examples:

> (set-implements? (list 1 2 3) 'set-add)
#t
> (set-implements? (list 1 2 3) 'set-add!)
#f
> (set-implements? (set 1 2 3) 'set-add)
#t
> (set-implements? (set 1 2 3) 'set-add!)
#t
> (set-implements? (mutable-seteq 1 2 3) 'set-add)
#t
> (set-implements? (mutable-seteq 1 2 3) 'set-add!)
#t
> (set-implements? (weak-seteqv 1 2 3) 'set-remove 'set-remove!)
#t

(set-implements/c sym ...) → flat-contract?
sym : symbol?

Recognizes sets that support all of the methods from gen:set named by the syms.
Constructs a contract that recognizes sets whose elements match \texttt{elem/c}.

If \texttt{kind} is \texttt{'immutable}, \texttt{'mutable}, or \texttt{'weak}, the resulting contract accepts only hash sets that are respectively immutable, mutable with strongly-held keys, or mutable with weakly-held keys. If \texttt{kind} is \texttt{'mutable-or-weak}, the resulting contract accepts any mutable hash sets, regardless of key-holding strength.

If \texttt{cmp} is \texttt{'equal}, \texttt{'eqv}, or \texttt{'eq}, the resulting contract accepts only hash sets that compare elements using \texttt{equal?}, \texttt{eqv?}, or \texttt{eq?}, respectively.

If \texttt{cmp} and \texttt{kind} are both \texttt{'dont-care}, then the resulting contract will accept any kind of set, not just hash sets.

If \texttt{lazy?} is not \#f, then the elements of the set are not checked immediately by the contract and only the set itself is checked (according to the \texttt{cmp} and \texttt{kind} arguments). If \texttt{lazy?} is \#f, then the elements are checked immediately by the contract. The \texttt{lazy?} argument is ignored when the set contract accepts generic sets (i.e., when \texttt{cmp} and \texttt{kind} are both \texttt{'dont-care}); in that case, the value being checked in that case is a \texttt{list?}, then the contract is not lazy otherwise the contract is lazy.

If \texttt{kind} allows mutable sets (i.e., is \texttt{'dont-care}, \texttt{'mutable}, \texttt{'weak}, or \texttt{'mutable-or-weak}) and \texttt{lazy?} is \#f, then the elements are checked both immediately and when they are accessed from the set.

The \texttt{equal-key/c} contract is used when values are passed to the comparison and hashing functions used internally.

The result contract will be a flat contract when \texttt{elem/c} and \texttt{equal-key/c} are both flat contracts, \texttt{lazy?} is \#f, and \texttt{kind} is \texttt{'immutable}. The result will be a chaperone contract when \texttt{elem/c} is a chaperone contract.

4.17.3 Generic Set Interface

\texttt{gen:set}

A generic interface (see §5.4 "Generic Interfaces") that supplies set method implementations
for a structure type via the #:methods option of struct definitions. This interface can be used to implement any of the methods documented as §4.17.3.1 “Set Methods”.

Examples:

```scheme
> (struct binary-set [integer]
   #:transparent
   #:methods gen:set
   [(define (set-member? st i)
      (bitwise-bit-set? (binary-set-integer st) i))
   (define (set-add st i)
      (binary-set (bitwise-ior (binary-set-integer st)
          (arithmetic-shift 1 i))))
   (define (set-remove st i)
      (binary-set (bitwise-and (binary-set-integer st)
          (bitwise-not (arithmetic-shift 1 i)))))))
> (define bset (binary-set 5))
> bset
(binary-set 5)
> (generic-set? bset)
#t
> (set-member? bset 0)
#t
> (set-member? bset 1)
#f
> (set-member? bset 2)
#t
> (set-add bset 4)
(binary-set 21)
> (set-remove bset 2)
(binary-set 1)
```

Set Methods

The methods of gen:set can be classified into three categories, as determined by their fallback implementations:

1. methods with no fallbacks,
2. methods whose fallbacks depend on other, non-fallback methods,
3. and methods whose fallbacks can depend on either fallback or non-fallback methods.

As an example, implementing the following methods would guarantee that all the methods in gen:set would at least have a fallback method:
• set-member?
• set-add
• set-add!
• set-remove
• set-remove!
• set-first
• set-empty?
• set-copy-clear

There may be other such subsets of methods that would guarantee at least a fallback for every method.

\[(\text{set-member? } st \ v) \rightarrow \text{boolean?}\]
\[st : \text{generic-set}\?\]
\[v : \text{any/c}\]

Returns #t if \(v\) is in \(st\), #f otherwise. Has no fallback.

\[(\text{set-add } st \ v) \rightarrow \text{generic-set}\?\]
\[st : \text{generic-set}\?\]
\[v : \text{any/c}\]

Produces a set that includes \(v\) plus all elements of \(st\). This operation runs in constant time for hash sets. Has no fallback.

\[(\text{set-add! } st \ v) \rightarrow \text{void}\?\]
\[st : \text{generic-set}\?\]
\[v : \text{any/c}\]

Adds the element \(v\) to \(st\). This operation runs in constant time for hash sets. Has no fallback.

For hash sets, see also the caveats concerning concurrent modification for hash tables, which applies to hash sets.

\[(\text{set-remove } st \ v) \rightarrow \text{generic-set}\?\]
\[st : \text{generic-set}\?\]
\[v : \text{any/c}\]

Produces a set that includes all elements of \(st\) except \(v\). This operation runs in constant time for hash sets. Has no fallback.
(set-remove! st v) → void?
  st : generic-set?
  v : any/c

Removes the element v from st. This operation runs in constant time for hash sets. Has no fallback.

For hash sets, see also the caveats concerning concurrent modification for hash tables, which applies to hash sets.

(set-empty? st) → boolean?
  st : generic-set?

Returns #t if st has no members; returns #f otherwise.

Supported for any st that implements set->stream or set-count.

(set-count st) → exact-nonnegative-integer?
  st : generic-set?

Returns the number of elements in st.

Supported for any st that supports set->stream.

(set-first st) → any/c
  st : (and/c generic-set? (not/c set-empty?))

Produces an unspecified element of st. Multiple uses of set-first on st produce the same result.

Supported for any st that implements set->stream.

(set-rest st) → generic-set?
  st : (and/c generic-set? (not/c set-empty?))

Produces a set that includes all elements of st except (set-first st).

Supported for any st that implements set-remove and either set-first or set->stream.

(set->stream st) → stream?
  st : generic-set?

 Produces a stream containing the elements of st.

Supported for any st that implements:
• set->list
• in-set
• set-empty?, set-first, set-rest
• set-empty?, set-first, set-remove
• set-count, set-first, set-rest
• set-count, set-first, set-remove

(set-copy st) → generic-set?
  st : generic-set?

Produces a new, mutable set of the same type and with the same elements as st.
Supported for any st that supports set->stream and implements set-copy-clear and set-add!.

(set-copy-clear st) → (and/c generic-set? set-empty?)
  st : generic-set?

Produces a new, empty set of the same type, mutability, and key strength as st.
A difference between set-copy-clear and set-clear is that the latter conceptually iterates set-remove on the given set, and so it preserves any contract on the given set. The set-copy-clear function produces a new set without any contracts.
The set-copy-clear function must call concrete set constructors and thus has no generic fallback.

(set-clear st) → (and/c generic-set? set-empty?)
  st : generic-set?

Produces a set like st but with all elements removed.
Supported for any st that implements set-remove and supports set->stream.

(set-clear! st) → void?
  st : generic-set?

Removes all elements from st.
Supported for any st that implements set-remove! and either supports set->stream or implements set-first and either set-count or set-empty?.

For hash sets, see also the caveats concerning concurrent modification for hash tables, which applies to hash sets.
(set-union st0 st ...) → generic-set?
st0 : generic-set?
st : generic-set?

Produces a set of the same type as st0 that includes the elements from st0 and all of the sts.

If st0 is a list, each st must also be a list. This operation runs on lists in time proportional to the total size of the sts times the size of the result.

If st0 is a hash set, each st must also be a hash set that uses the same comparison function (equal?, eqv?, or eq?). The mutability and key strength of the hash sets may differ. This operation runs on hash sets in time proportional to the total size of all of the sets except the largest immutable set.

At least one set must be provided to set-union to determine the type of the resulting set (list, hash set, etc.). If there is a case where set-union may be applied to zero arguments, instead pass an empty set of the intended type as the first argument.

Supported for any st that implements set-add and supports set->stream.

Examples:

> (set-union (set))
(set)
> (set-union (seteq))
(seteq)
> (set-union (set 1 2) (set 2 3))
(set 1 2 3)
> (set-union (list 1 2) (list 2 3))
'(3 1 2)
> (set-union (set 1 2) (seteq 2 3))
set-union: set arguments have incompatible equivalence predicates
  first set: (set 1 2)
  incompatible set: (seteq 2 3)
; Sets of different types cannot be unioned

(set-union! st0 st ...) → void?
st0 : generic-set?
st : generic-set?

Adds the elements from all of the sts to st0.

If st0 is a hash set, each st must also be a hash set that uses the same comparison function
(equal?, eqv?, or eq?). The mutability and key strength of the hash sets may differ. This operation runs on hash sets in time proportional to the total size of the \textit{sts}.

Supported for any \textit{st} that implements \texttt{set-add!} and supports \texttt{set->stream}.

For hash sets, see also the caveats concerning concurrent modification for hash tables, which applies to hash sets.

\begin{verbatim}
(set-intersect st0 st ... ) → generic-set?
  st0 : generic-set?
  st  : generic-set?
\end{verbatim}

Produces a set of the same type as \textit{st0} that includes the elements from \textit{st0} that are also contained by all of the \textit{sts}.

If \textit{st0} is a list, each \textit{st} must also be a list. This operation runs on lists in time proportional to the total size of the \textit{sts} times the size of \textit{st0}.

If \textit{st0} is a hash set, each \textit{st} must also be a hash set that uses the same comparison function (equal?, eqv?, or eq?). The mutability and key strength of the hash sets may differ. This operation runs on hash sets in time proportional to the size of the smallest immutable set.

Supported for any \textit{st} that implements either \texttt{set-remove} or both \texttt{set-clear} and \texttt{set-add}, and supports \texttt{set->stream}.

\begin{verbatim}
(set-intersect! st0 st ... ) → void?
  st0 : generic-set?
  st  : generic-set?
\end{verbatim}

Removes every element from \textit{st0} that is not contained by all of the \textit{sts}.

If \textit{st0} is a hash set, each \textit{st} must also be a hash set that uses the same comparison function (equal?, eqv?, or eq?). The mutability and key strength of the hash sets may differ. This operation runs on hash sets in time proportional to the size of \textit{st0}.

Supported for any \textit{st} that implements \texttt{set-remove!} and supports \texttt{set->stream}.

For hash sets, see also the caveats concerning concurrent modification for hash tables, which applies to hash sets.

\begin{verbatim}
(set-subtract st0 st ... ) → generic-set?
  st0 : generic-set?
  st  : generic-set?
\end{verbatim}

Produces a set of the same type as \textit{st0} that includes the elements from \textit{st0} that are not contained by any of the \textit{sts}.

464
If \( st0 \) is a list, each \( st \) must also be a list. This operation runs on lists in time proportional to the total size of the \( st \)s times the size of \( st0 \).

If \( st0 \) is a hash set, each \( st \) must also be a hash set that uses the same comparison function (\( \text{equal?} \), \( \text{eqv?} \), or \( \text{eq?} \)). The mutability and key strength of the hash sets may differ. This operation runs on hash sets in time proportional to the size of \( st0 \).

Supported for any \( st \) that implements either \text{set-remove} or both \text{set-clear} and \text{set-add}, and supports \text{set->stream}.

\[
\begin{align*}
\text{(set-subtract! } st0 \text{ } st \ldots) \to \text{void} \\
\quad st0 : \text{generic-set} \\
\quad st : \text{generic-set}
\end{align*}
\]

Removes every element from \( st0 \) that is contained by any of the \( st \)s.

If \( st0 \) is a hash set, each \( st \) must also be a hash set that uses the same comparison function (\( \text{equal?} \), \( \text{eqv?} \), or \( \text{eq?} \)). The mutability and key strength of the hash sets may differ. This operation runs on hash sets in time proportional to the size of \( st0 \).

Supported for any \( st \) that implements \text{set-remove!} and supports \text{set->stream}.

For hash sets, see also the caveats concerning concurrent modification for hash tables, which applies to hash sets.

\[
\begin{align*}
\text{(set-symmetric-difference } st0 \text{ } st \ldots) \to \text{generic-set} \\
\quad st0 : \text{generic-set} \\
\quad st : \text{generic-set}
\end{align*}
\]

Produces a set of the same type as \( st0 \) that includes all of the elements contained an odd number of times in \( st0 \) and the \( st \)s.

If \( st0 \) is a list, each \( st \) must also be a list. This operation runs on lists in time proportional to the total size of the \( st \)s times the size of \( st0 \).

If \( st0 \) is a hash set, each \( st \) must also be a hash set that uses the same comparison function (\( \text{equal?} \), \( \text{eqv?} \), or \( \text{eq?} \)). The mutability and key strength of the hash sets may differ. This operation runs on hash sets in time proportional to the total size of all of the sets except the largest immutable set.

Supported for any \( st \) that implements \text{set-remove} or both \text{set-clear} and \text{set-add}, and supports \text{set->stream}.

Example:

\[
\begin{align*}
> \text{(set-symmetric-difference (set 1) (set 1 2) (set 1 2 3))} \\
\quad \text{(set 1 3)}
\end{align*}
\]
(set-symmetric-difference! st0 st ...) → void?
st0 : generic-set?
   st : generic-set?

Adds and removes elements of st0 so that it includes all of the elements contained an odd number of times in the sts and the original contents of st0.

If st0 is a hash set, each st must also be a hash set that uses the same comparison function (equal?, eqv?, or eq?). The mutability and key strength of the hash sets may differ. This operation runs on hash sets in time proportional to the total size of the sts.

Supported for any st that implements set-remove! and supports set->stream.

For hash sets, see also the caveats concerning concurrent modification for hash tables, which applies to hash sets.

(set=? st st2) → boolean?
st : generic-set?
   st2 : generic-set?

Returns #t if st and st2 contain the same members; returns #f otherwise.

If st0 is a list, each st must also be a list. This operation runs on lists in time proportional to the size of st times the size of st2.

If st0 is a hash set, each st must also be a hash set that uses the same comparison function (equal?, eqv?, or eq?). The mutability and key strength of the hash sets may differ. This operation runs on hash sets in time proportional to the size of st plus the size of st2.

Supported for any st and st2 that both support subset?; also supported for any if st2 that implements set=? regardless of st.

Examples:

> (set=? (list 1 2) (list 2 1))
#t
> (set=? (set 1) (set 1 2 3))
#f
> (set=? (set 1 2 3) (set 1))
#f
> (set=? (set 1 2 3) (set 1 2 3))
#t
> (set=? (seteq 1 2) (mutable-seteq 2 1))
#t
> (set=? (seteq 1 2) (seteqv 1 2))
set=?: set arguments have incompatible equivalence
predicates

first set: (seteq 1 2)
incompatible set: (seteqv 1 2)
; Sets of different types cannot be compared

(subset? st st2) → boolean?
st : generic-set?
st2 : generic-set?

Returns #t if st2 contains every member of st; returns #f otherwise.

If st is a list, then st2 must also be a list. This operation runs on lists in time proportional to the size of st times the size of st2.

If st is a hash set, then st2 must also be a hash set that uses the same comparison function (equal?, eqv?, or eq?). The mutability and key strength of the hash sets may differ. This operation runs on hash sets in time proportional to the size of st.

Supported for any st that supports set->stream.

Examples:

> (subset? (set 1) (set 1 2 3))
#t
> (subset? (set 1 2 3) (set 1))
#f
> (subset? (set 1 2 3) (set 1 2 3))
#t

(proper-subset? st st2) → boolean?
st : generic-set?
st2 : generic-set?

Returns #t if st2 contains every member of st and at least one additional element; returns #f otherwise.

If st is a list, then st2 must also be a list. This operation runs on lists in time proportional to the size of st times the size of st2.

If st is a hash set, then st2 must also be a hash set that uses the same comparison function (equal?, eqv?, or eq?). The mutability and key strength of the hash sets may differ. This operation runs on hash sets in time proportional to the size of st plus the size of st2.

Supported for any st and st2 that both support subset?.

Examples:
> (proper-subset? (set 1) (set 1 2 3))
#t
> (proper-subset? (set 1 2 3) (set 1))
#f
> (proper-subset? (set 1 2 3) (set 1 2 3))
#f

(set->list st) → list?

st : generic-set?

Produces a list containing the elements of st.

Supported for any st that supports set->stream.

(set-map st proc) → (listof any/c)

st : generic-set?
proc : (any/c . -> . any/c)

Applies the procedure proc to each element in st in an unspecified order, accumulating the results into a list.

Supported for any st that supports set->stream.

(set-for-each st proc) → void?

st : generic-set?
proc : (any/c . -> . any)

Applies proc to each element in st (for the side-effects of proc) in an unspecified order.

Supported for any st that supports set->stream.

(in-set st) → sequence?

st : generic-set?

Explicitly converts a set to a sequence for use with for and other forms.

Supported for any st that supports set->stream.

(impersonate-hash-set st
 inject-proc
 add-proc
 shrink-proc
 extract-proc
 [clear-proc
  equal-key-proc]
 prop
 prop-val ...
 ...)

468
Impersonates \( st \), redirecting various set operations via the given procedures.

The \texttt{inject-proc} procedure is called whenever an element is temporarily put into the set for the purposes of comparing it with other elements that may already be in the set. For example, when evaluating \((\text{set-member? } s \ e)\), \( e \) will be passed to the \texttt{inject-proc} before comparing it with other elements of \( s \).

The \texttt{add-proc} procedure is called when adding an element to a set, e.g., via \texttt{set-add} or \texttt{set-add!}. The result of the \texttt{add-proc} is stored in the set.

The \texttt{shrink-proc} procedure is called when building a new set with one fewer element. For example, when evaluating \((\text{set-remove } s \ e)\) or \((\text{set-remove! } s \ e)\), an element is removed from a set, e.g., via \texttt{set-remove} or \texttt{set-remove!}. The result of the \texttt{shrink-proc} is the element actually removed from the set.

The \texttt{extract-proc} procedure is called when an element is pulled out of a set, e.g., by \texttt{set-first}. The result of the \texttt{extract-proc} is the element actually produced by from the set.

The \texttt{clear-proc} is called by \texttt{set-clear} and \texttt{set-clear!} and if it returns (as opposed to escaping, perhaps via raising an exception), the clearing operation is permitted. Its result is ignored. If \texttt{clear-proc} is \#f, then clearing is done element by element (via calls into the other supplied procedures).

The \texttt{equal-key-proc} is called when an element’s hash code is needed of when an element is supplied to the underlying equality in the set. The result of \texttt{equal-key-proc} is used when computing the hash or comparing for equality.

If any of the \texttt{inject-proc}, \texttt{add-proc}, \texttt{shrink-proc}, or \texttt{extract-proc} arguments are \#f, then they all must be \#f, the \texttt{clear-proc} and \texttt{equal-key-proc} must also be \#f, and there must be at least one property supplied.

Pairs of \( prop \) and \( prop-val \) (the number of arguments to \texttt{impersonate-hash-set} must be odd) add impersonator properties or override impersonator property values of \( st \).
Chaperones \( st \). Like \texttt{impersonate-hash-set} but with the constraints that the results of the \texttt{inject-proc}, \texttt{add-proc}, \texttt{shrink-proc}, \texttt{extract-proc}, and \texttt{equal-key-proc} must be \texttt{chaperone-of?} their second arguments. Also, the input may be an \texttt{immutable?} set.

### 4.17.4 Custom Hash Sets

\[
\text{(define-custom-set-types } \text{name} \\
\hspace{-1cm} \text{optional-predicate} \\
\hspace{-1cm} \text{comparison-expr} \\
\hspace{-1cm} \text{optional-hash-functions})
\]

\[
\text{optional-predicate } = \text{ | #:elem? predicate-expr}
\]

\[
\text{optional-hash-functions } = \text{ | hash1-expr} \\
\hspace{-1cm} \text{| hash1-expr hash2-expr}
\]

Creates a new hash set type based on the given comparison \texttt{comparison-expr}, hash functions \texttt{hash1-expr} and \texttt{hash2-expr}, and element predicate \texttt{predicate-expr}; the interfaces for these functions are the same as in \texttt{make-custom-set-types}. The new set type has
three variants: immutable, mutable with strongly-held elements, and mutable with weakly-held elements.

Defines seven names:

- `name?` recognizes instances of the new type,
- `immutable-name?` recognizes immutable instances of the new type,
- `mutable-name?` recognizes mutable instances of the new type with strongly-held elements,
- `weak-name?` recognizes mutable instances of the new type with weakly-held elements,
- `make-immutable-name` constructs immutable instances of the new type,
- `make-mutable-name` constructs mutable instances of the new type with strongly-held elements, and
- `make-weak-name` constructs mutable instances of the new type with weakly-held elements.

The constructors all accept a stream as an optional argument, providing initial elements.

Examples:

```scheme
> (define-custom-set-types string-set
  #:elem? string?
  string=?
  string-length)
> (define imm
  (make-immutable-string-set '("apple" "banana")))
> (define mut
  (make-mutable-string-set '("apple" "banana")))
> (generic-set? imm) #t
> (generic-set? mut) #t
> (set? imm) #t
> (generic-set? imm) #t
> (string-set? imm) #t
> (string-set? mut) #t
> (immutable-string-set? imm)
```

471
#t
> (immutable-string-set? mut)
#f
> (set-member? imm "apple")
#t
> (set-member? mut "banana")
#t
> (equal? imm mut)
#f
> (set=? imm mut)
#t
> (set-remove! mut "banana")
> (set-member? mut "banana")
#f
> (equal? (set-remove (set-remove imm "apple") "banana")
  (make-immutable-string-set))
#t

(make-custom-set-types  eql?
 [hash1
   hash2
   #:elem? elem?
   #:name name
   #:for who]
  
  -> (any/c . -> . boolean?)
  (any/c . -> . boolean?)
  (any/c . -> . boolean?)
  (any/c . -> . boolean?)
  (->* [] [stream?] generic-set?)
  (->* [] [stream?] generic-set?)
  (->* [] [stream?] generic-set?)
  eql?: (or/c (any/c any/c any/c . -> . any/c)
    (any/c any/c (any/c any/c . -> . any/c) . -> . any/c))
  hash1: (or/c (any/c . -> . exact-integer?)
    (any/c (any/c . -> . exact-integer?) . -> . exact-integer?)
    = (const 1)
  hash2: (or/c (any/c . -> . exact-integer?)
    (any/c (any/c . -> . exact-integer?) . -> . exact-integer?)
    = (const 1)
  elem?: (any/c . -> . boolean?) = (const #true)
  name : symbol? = 'custom-set
  who : symbol? = 'make-custom-set-types

Creates a new set type based on the given comparison function eql?, hash functions hash1 and hash2, and predicate elem?. The new set type has variants that are immutable, mutable
with strongly-held elements, and mutable with weakly-held elements. The given name is used when printing instances of the new set type, and the symbol who is used for reporting errors.

The comparison function eql? may accept 2 or 3 arguments. If it accepts 2 arguments, it given two elements to compare them. If it accepts 3 arguments and does not accept 2 arguments, it is also given a recursive comparison function that handles data cycles when comparing sub-parts of the elements.

The hash functions hash1 and hash2 may accept 1 or 2 arguments. If either hash function accepts 1 argument, it is applied to a element to compute the corresponding hash value. If either hash function accepts 2 arguments and does not accept 1 argument, it is also given a recursive hash function that handles data cycles when computing hash values of sub-parts of the elements.

The predicate elem? must accept 1 argument and is used to recognize valid elements for the new set type.

Produces seven values:

- a predicate recognizing all instances of the new set type,
- a predicate recognizing weak instances,
- a predicate recognizing mutable instances,
- a predicate recognizing immutable instances,
- a constructor for weak instances,
- a constructor for mutable instances, and
- a constructor for immutable instances.

See define-custom-set-types for an example.

### 4.18 Procedures

```scheme
(procedure? v) → boolean?
  v : any/c
```

Returns #t if v is a procedure, #f otherwise.

```scheme
(apply proc v ... lst #:<kw> kw-arg ...) → any
  proc : procedure?
  v : any/c
  lst : list?
  kw-arg : any/c
```
Applies \texttt{proc} using the content of \texttt{(list* v ... lst)} as the (by-position) arguments. The \#:<\texttt{kw}> \texttt{kv-arg} sequence is also supplied as keyword arguments to \texttt{proc}, where \#:<\texttt{kw}> stands for any keyword.

The given \texttt{proc} must accept as many arguments as the number of \texttt{v}s plus length of \texttt{lst}, it must accept the supplied keyword arguments, and it must not require any other keyword arguments; otherwise, the \texttt{exn:fail:contract} exception is raised. The given \texttt{proc} is called in tail position with respect to the \texttt{apply} call.

Examples:

\begin{verbatim}
> (apply + '(1 2 3))  
  6
> (apply + 1 2 '(3))  
  6
> (apply + '())  
  0
> (apply sort (list (list '2) '(1)) #:key car)  
  '((1) (2))
\end{verbatim}

\begin{verbatim}
((compose proc ...) → procedure?
 proc : procedure?
(compose1 proc ...) → procedure?
 proc : procedure?)
\end{verbatim}

Returns a procedure that composes the given functions, applying the last \texttt{proc} first and the first \texttt{proc} last. The \texttt{compose} function allows the given functions to consume and produce any number of values, as long as each function produces as many values as the preceding function consumes, while \texttt{compose1} restricts the internal value passing to a single value. In both cases, the input arity of the last function and the output arity of the first are unrestricted, and they become the corresponding arity of the resulting composition (including keyword arguments for the input side).

When no \texttt{proc} arguments are given, the result is \texttt{values}. When exactly one is given, it is returned.

Examples:

\begin{verbatim}
> ((compose1 - sqrt) 10)  
  -3.1622776601683795
> ((compose1 sqrt -) 10)  
  0.0+3.1622776601683795i
> ((compose list split-path) (bytes->path "/a" 'unix))  
  '(/a)  
  '(#<path:/> #<path:a> #f)
\end{verbatim}
Note that in many cases, \texttt{compose1} is preferred. For example, using \texttt{compose} with two library functions may lead to problems when one function is extended to return two values, and the preceding one has an optional input with different semantics. In addition, \texttt{compose1} may create faster compositions.

\begin{verbatim}
(procedure-rename proc name) \to procedure?
  proc : procedure?
  name : symbol?
\end{verbatim}

Returns a procedure that is like \texttt{proc}, except that its name as returned by \texttt{object-name} (and as printed for debugging) is \texttt{name}.

The given \texttt{name} is used for printing an error message if the resulting procedure is applied to the wrong number of arguments. In addition, if \texttt{proc} is an accessor or mutator produced by \texttt{struct}, \texttt{make-struct-field-accessor}, or \texttt{make-struct-field-mutator}, the resulting procedure also uses \texttt{name} when its (first) argument has the wrong type. More typically, however, \texttt{name} is not used for reporting errors, since the procedure name is typically hard-wired into an internal check.

\begin{verbatim}
(procedure->method proc) \to procedure?
  proc : procedure?
\end{verbatim}

Returns a procedure that is like \texttt{proc} except that, when applied to the wrong number of arguments, the resulting error hides the first argument as if the procedure had been compiled with the 'method-arity-error syntax property.

\begin{verbatim}
(procedure-closure-contents-eq? proc1 proc2) \to boolean?
  proc1 : procedure?
  proc2 : procedure?
\end{verbatim}

Compares the contents of the closures of \texttt{proc1} and \texttt{proc2} for equality by comparing closure elements pointwise using \texttt{eq}?

### 4.18.1 Keywords and Arity

\begin{verbatim}
(keyword-apply proc
  ku-lst
  kw-val-lst
  v ...
  lst
  #:<kw> kw-arg ...) \to any
  proc : procedure?
  kw-lst : (listof keyword?)
\end{verbatim}
Like `apply`, but `kw-lst` and `kw-val-lst` supply by-keyword arguments in addition to the by-position arguments of the `vs` and `lst`, and in addition to the directly supplied keyword arguments in the `#:<kw> kw-arg` sequence, where `#:<kw>` stands for any keyword.

The given `kw-lst` must be sorted using `keyword<?`. No keyword can appear twice in `kw-lst` or in both `kw-list` and as a `#:<kw>`, otherwise, the `exn:fail:contract` exception is raised. The given `kw-val-lst` must have the same length as `kw-lst`, otherwise, the `exn:fail:contract` exception is raised. The given `proc` must accept all of the keywords in `kw-lst` plus the `#:<kw>s`, it must not require any other keywords, and it must accept as many by-position arguments as supplied via the `vs` and `lst`; otherwise, the `exn:fail:contract` exception is raised.

Examples:

```scheme
(define (f x #:y y #:z [z 10])
  (list x y z))

> (keyword-apply f '('#:y) '(2) '(1))
'(1 2 10)
> (keyword-apply f '('#:y #:z) '(2 3) '(1))
'(1 2 3)
> (keyword-apply f #:z 7 '('#:y) '(2) '(1))
'(1 2 7)
```

§4.3.3 “The apply Function” in The Racket Guide introduces `keyword-apply`.

`(procedure-arity proc) ➝ normalized-arity?
proc : procedure?
```

Returns information about the number of by-position arguments accepted by `proc`. See also `procedure-arity?`, `normalized-arity?`, and `procedure-arity-mask`.

`(procedure-arity? v) ➝ boolean?
 v : any/c
```

A valid arity `a` is one of the following:

- An exact non-negative integer, which means that the procedure accepts `a` arguments, only.
- A `arity-at-least` instance, which means that the procedure accepts `(arity-at-least-value a)` or more arguments.
A list containing integers and `arity-at-least` instances, which means that the procedure accepts any number of arguments that can match one of the elements of `a`.

The result of `procedure-arity` is always normalized in the sense of `normalized-arity?`.

Examples:

```scheme
> (procedure-arity cons)
2
> (procedure-arity list)
(arity-at-least 0)
> (arity-at-least? (procedure-arity list))
#t
> (arity-at-least-value (procedure-arity list))
0
> (arity-at-least-value (procedure-arity (lambda (x . y) x)))
1
> (procedure-arity (case-lambda [(x) 0] [(x y) 1]))
'(1 2)
```

```scheme
(procedure-arity-mask proc) → exact-integer?
proc : procedure?
```

Returns the same information as `procedure-arity`, but encoded differently. The arity is encoded as an exact integer `mask` where `(bitwise-bit-set? mask n)` returns true if `proc` accepts `n` arguments.

The mask encoding of an arity is often easier to test and manipulate, and `procedure-arity-mask` is sometimes faster than `procedure-arity` while always being at least as fast.

Added in version 7.0.0.11 of package base.

```scheme
(procedure-arity-includes? proc k [kws-ok?]) → boolean?
proc : procedure?
k : exact-nonnegative-integer?
kws-ok? : any/c = #f
```

Returns #t if the procedure can accept `k` by-position arguments, #f otherwise. If `kws-ok?` is #f, the result is #t only if `proc` has no required keyword arguments.

Examples:

```scheme
> (procedure-arity-includes? cons 2)
#t
```
(procedure-reduce-arity proc arity [name]) → procedure?
   proc : procedure?
   arity : procedure-arity?
   name : (or/c symbol? #f) = #f

Returns a procedure that is the same as proc (including the same name returned by object-name), but that accepts only arguments consistent with arity. In particular, when procedure-arity is applied to the generated procedure, it returns a value that is equal? to the normalized form of arity.

If the arity specification allows arguments that are not in (procedure-arity proc), the exn:fail:contract exception is raised. If proc accepts keyword argument, either the keyword arguments must be all optional (and they are not accepted in by the arity-reduced procedure) or arity must be the empty list (which makes a procedure that cannot be called); otherwise, the exn:fail:contract exception is raised.

If name is not #f, then object-name of the result procedure produces name. Otherwise, object-name of the result procedure produces the same result as for proc.

Examples:

> (define my+ (procedure-reduce-arity + 2))
> (my+ 1 2)
3
> (my+ 1 2 3)
+: arity mismatch:
   the expected number of arguments does not match the given number
   expected: 2
   given: 3

> (define also-my+ (procedure-reduce-arity + 2 'also-my+)))
> (also-my+ 1 2 3)
also-my+: arity mismatch:
   the expected number of arguments does not match the given number
   expected: 2
   given: 3
(procedure-reduce-arity-mask proc mask [name]) → procedure?

proc : procedure?
mask : exact-integer?
name : (or/c symbol? #f) = #f

The same as procedure-reduce-arity, but using the representation of arity described with procedure-arity-mask.

The mask encoding of an arity is often easier to test and manipulate, and procedure-reduce-arity-mask is sometimes faster than procedure-reduce-arity while always being at least as fast.

Added in version 7.0.0.11 of package base.

(procedure-keywords proc) → (listof keyword?)

(or/c (listof keyword?) #f)

proc : procedure?

Returns information about the keyword arguments required and accepted by a procedure. The first result is a list of distinct keywords (sorted by keyword<?) that are required when applying proc. The second result is a list of distinct accepted keywords (sorted by keyword<?), or #f to mean that any keyword is accepted. When the second result is a list, every element in the first list is also in the second list.

Examples:

> (procedure-keywords +)
'(())
'(())
> (procedure-keywords (lambda (#:tag t #:mode m) t))
'(#:mode #:tag)
'(#:mode #:tag)
> (procedure-keywords (lambda (#:tag #:mode [m #f]) t))
'(#:tag)
'(#:mode #:tag)

(procedure-result-arity proc) → (or/c #f procedure-arity?)

proc : procedure?

Returns the arity of the result of the procedure proc or #f if the number of results are not known, perhaps due to shortcomings in the implementation of procedure-result-arity or because proc’s behavior is not sufficiently simple.

Examples:
> (procedure-result-arity car)
1
> (procedure-result-arity values)
#f
> (procedure-result-arity
  (λ (x)
    (apply values
      (let loop ()
        (cond
          [(zero? (random 10)) '()]
          [else (cons 1 (loop))]))))))
#f

Added in version 6.4.0.3 of package base.

(make-keyword-procedure proc [plain-proc]) → procedure?
proc : (((listof keyword?) list?) () #:rest list? . ->* . any)
plain-proc : procedure?
  = (lambda args (apply proc null null args))

Returns a procedure that accepts all keyword arguments (without requiring any keyword arguments).

When the procedure returned by make-keyword-procedure is called with keyword arguments, then proc is called; the first argument is a list of distinct keywords sorted by keyword<, the second argument is a parallel list containing a value for each keyword, and the remaining arguments are the by-position arguments.

When the procedure returned by make-keyword-procedure is called without keyword arguments, then plain-proc is called—possibly more efficiently than dispatching through proc. Normally, plain-proc should have the same behavior as calling proc with empty lists as the first two arguments, but that correspondence is in no way enforced.

The result of procedure-arity and object-name on the new procedure is the same as for plain-proc. See also procedure-reduce-keyword-arity and procedure-rename.

Examples:

(define show
  (make-keyword-procedure
    (lambda (kws kw-args . rest)
      (list kws kw-args rest))))

> (show 1)
'((1))
> (show #:init 0 1 2 3 #:extra 4)
'(#:extra #:init) (4 0) (1 2 3))

(define show2
  (make-keyword-procedure
   (lambda (kws kw-args . rest)
     (list kws kw-args rest))
   (lambda args
     (list->vector args))))

> (show2 1)
'#(1)
> (show2 #:init 0 1 2 3 #:extra 4)
'(#:extra #:init) (4 0) (1 2 3))

(procedure-reduce-keyword-arity proc
  arity
  required-kws
  allowed-kws) → procedure?

proc : procedure?
arity : procedure-arity?
required-kws : (listof keyword?)
allowed-kws : (or/c (listof keyword?) #f)

Like procedure-reduce-arity, but constrains the keyword arguments according to required-kws and allowed-kws, which must be sorted using keyword<? and contain no duplicates. If allowed-kws is #f, then the resulting procedure still accepts any keyword, otherwise the keywords in required-kws must be a subset of those in allowed-kws.
The original proc must require no more keywords than the ones listed in required-kws, and it must allow at least the keywords in allowed-kws (or it must allow all keywords if allowed-kws is #f).

Examples:

(define orig-show
  (make-keyword-procedure
   (lambda (kws kw-args . rest)
     (list kws kw-args rest))
   (lambda args
     (list->vector args))))

(define show (procedure-reduce-keyword-arity
  orig-show 3 '(#:init) '(#:extra #:init)))

> (show #:init 0 1 2 3 #:extra 4)
'((#:extra #:init) (4 0) (1 2 3))
> (show 1)
unknown: arity mismatch;
  the expected number of arguments does not match the given number
expected: 3 plus an argument with keyword #:init plus an
optional argument with keyword #:extra
given: 1
arguments...:
1
> (show #:init 0 1 2 3 #:extra 4 #:more 7)
application: procedure does not expect an argument with
given keyword
procedure: unknown
given keyword: #:more
arguments...:
1
2
3
#:extra 4
#:init 0
#:more 7

(procedure-reduce-keyword-arity-mask proc
  mask
  required-kws
  allowed-kws) → procedure?

proc : procedure?
mask : exact-integer?
required-kws : (listof keyword?)
allowed-kws : (or/c (listof keyword?) #f)

The same as procedure-reduce-keyword-arity, but using the representation of arity
described with procedure-arity-mask.

Added in version 7.0.0.11 of package base.

(struct arity-at-least (value)
  #:extra-constructor-name make-arity-at-least)
value : exact-nonnegative-integer?

A structure type used for the result of procedure-arity. See also procedure-arity?.

prop:procedure : struct-type-property?

A structure type property to identify structure types whose instances can be applied as proce-
dures. In particular, when procedure? is applied to the instance, the result will be #t, and
when an instance is used in the function position of an application expression, a procedure
is extracted from the instance and used to complete the procedure call.
If the `prop:procedure` property value is an exact non-negative integer, it designates a field within the structure that should contain a procedure. The integer must be between 0 (inclusive) and the number of non-automatic fields in the structure type (exclusive, not counting supertype fields). The designated field must also be specified as immutable, so that after an instance of the structure is created, its procedure cannot be changed. (Otherwise, the arity and name of the instance could change, and such mutations are generally not allowed for procedures.) When the instance is used as the procedure in an application expression, the value of the designated field in the instance is used to complete the procedure call. (This procedure can be another structure that acts as a procedure; the immutability of procedure fields disallows cycles in the procedure graph, so that the procedure call will eventually continue with a non-structure procedure.) That procedure receives all of the arguments from the application expression. The procedure’s name (see `object-name`), arity (see `procedure-arity`), and keyword protocol (see `procedure-keywords`) are also used for the name, arity, and keyword protocol of the structure. If the value in the designated field is not a procedure, then the instance behaves like `(case-lambda)` (i.e., a procedure which does not accept any number of arguments). See also `procedure-extract-target`.

Providing an integer `proc-spec` argument to `make-struct-type` is the same as both supplying the value with the `prop:procedure` property and designating the field as immutable (so that a property binding or immutable designation is redundant and disallowed).

Examples:

```scheme
> (struct annotated-proc (base note)
   #:property prop:procedure
   (struct-field-index base))
> (define plus1 (annotated-proc
     (lambda (x) (+ x 1))
     "adds 1 to its argument")
> (procedure? plus1)
#t
> (annotated-proc? plus1)
#t
> (plus1 10)
11
> (annotated-proc-note plus1)
"adds 1 to its argument"
```

When the `prop:procedure` value is a procedure, it should accept at least one non-keyword argument. When an instance of the structure is used in an application expression, the property-value procedure is called with the instance as the first argument. The remaining arguments to the property-value procedure are the arguments from the application expression (including keyword arguments). Thus, if the application expression provides five non-keyword arguments, the property-value procedure is called with six non-keyword arguments. The name of the instance (see `object-name`) and its keyword protocol (see `procedure-keywords`) are unaffected by the property-value procedure, but the instance’s arity is deter-
mined by subtracting one from every possible non-keyword argument count of the property-value procedure. If the property-value procedure cannot accept at least one argument, then the instance behaves like (case-lambda).

Providing a procedure proc-spec argument to make-struct-type is the same as supplying the value with the prop:procedure property (so that a specific property binding is disallowed).

Examples:

```scheme
> (struct fish (weight color)
   #:mutable
   #:property
   prop:procedure
   (lambda (f n)
     (let ([w (fish-weight f)])
       (set-fish-weight! f (+ n w)))))
> (define wanda (fish 12 'red))
> (fish? wanda)
#t
> (procedure? wanda)
#t
> (fish-weight wanda)
12
> (for-each wanda '(1 2 3))
> (fish-weight wanda)
18
```

If the value supplied for the prop:procedure property is not an exact non-negative integer or a procedure, the exn:fail:contract exception is raised.

```scheme
(procedure-struct-type? type) -> boolean?
```

Returns #t if instances of the structure type represented by type are procedures (according to procedure?), #f otherwise.

```scheme
(procedure-extract-target proc) -> (or/c #f procedure?)
```

If proc is an instance of a structure type with property prop:procedure, and if the property value indicates a field of the structure, and if the field value is a procedure, then procedure-extract-target returns the field value. Otherwise, the result is #f.

When a prop:procedure property value is a procedure, the procedure is not returned by procedure-extract-target. Such a procedure is different from one accessed through a
structure field, because it consumes an extra argument, which is always the structure that was applied as a procedure. Keeping the procedure private ensures that it is always called with a suitable first argument.

### prop:arity-string : struct-type-property?

A structure type property that is used for reporting arity-mismatch errors when a structure type with the prop:procedure property is applied to the wrong number of arguments. The value of the prop:arity-string property must be a procedure that takes a single argument, which is the misapplied structure, and returns a string. The result string is used after the word “expects,” and it is followed in the error message by the number of actual arguments.

Arity-mismatch reporting automatically uses procedure-extract-target when the prop:arity-string property is not associated with a procedure structure type.

Examples:

```scheme
> (struct evens (proc)
   #:property prop:procedure (struct-field-index proc)
   #:property prop:arity-string
   (lambda (p)
     "an even number of arguments"))
> (define pairs
  (evens
   (case-lambda
    [(() null)
     [(a b . more)
      (cons (cons a b)
        (apply pairs more))]]))
> (pairs 1 2 3 4)
'((1 . 2) (3 . 4))
> (pairs 5)
arity mismatch;
  the expected number of arguments does not match the given number
  expected: an even number of arguments
given: 1
arguments...:
  5
```

### prop:checked-procedure : struct-type-property?

A structure type property that is used with checked-procedure-check-and-extract, which is a hook to allow the compiler to improve the performance of keyword arguments. The property can only be attached to a structure type without a supertype and with at least two fields.
(checked-procedure-check-and-extract type v proc v1 v2) \(\rightarrow\) any/c

\begin{align*}
\text{type} & : \text{struct-type}\? \\text{v} & : \text{any/c} \\
\text{proc} & : (\text{any/c any/c any/c . -> . any/c}) \\text{v1} & : \text{any/c} \\
\text{v2} & : \text{any/c}
\end{align*}

Extracts a value from \(v\) if it is an instance of \(\text{type}\), which must have the property prop:checked-procedure. If \(v\) is such an instance, then the first field of \(v\) is extracted and applied to \(v1\) and \(v2\); if the result is a true value, the result is the value of the second field of \(v\).

If \(v\) is not an instance of \(\text{type}\), or if the first field of \(v\) applied to \(v1\) and \(v2\) produces \#f, then \(\text{proc}\) is applied to \(v, v1, \) and \(v2,\) and its result is returned by checked-procedure-check-and-extract.

\begin{align*}
(\text{procedure-specialize proc}) & \rightarrow \text{procedure}\? \\
\text{proc} & : \text{procedure}\?
\end{align*}

Returns \(\text{proc}\) or its equivalent, but provides a hint to the run-time system that it should spend extra time and memory to specialize the implementation of \(\text{proc}\).

The hint is currently used when \(\text{proc}\) is the value of a \text{lambda} or \text{case-lambda} form that references variables bound outside of the \text{lambda} or \text{case-lambda}, and when \(\text{proc}\) has not been previously applied.

Added in version 6.3.0.10 of package \text{base}.

4.18.2 Reflecting on Primitives

A primitive procedure is a built-in procedure that is implemented in low-level language. Not all procedures of racket/base are primitives, but many are. The distinction is mainly useful to other low-level code.

\begin{align*}
(\text{primitive? v}) & \rightarrow \text{boolean}\? \\
\text{v} & : \text{any/c}
\end{align*}

Returns \#t if \(v\) is a primitive procedure, \#f otherwise.

\begin{align*}
(\text{primitive-closure? v}) & \rightarrow \text{boolean} \\
\text{v} & : \text{any/c}
\end{align*}

486
Returns \( \texttt{#t} \) if \( v \) is internally implemented as a primitive closure rather than a simple primitive procedure, \( \texttt{#f} \) otherwise.

\[
(\text{primitive-result-arity \textit{prim}}) \rightarrow \text{procedure-arity}?
\]
\[
\textit{prim} : \text{primitive}?
\]

Returns the arity of the result of the primitive procedure \( \textit{prim} \) (as opposed to the procedure’s input arity as returned by \texttt{procedure-arity}). For most primitives, this procedure returns 1, since most primitives return a single value when applied.

### 4.18.3 Additional Higher-Order Functions

\[
\text{(require racket/function)} \quad \text{package: \texttt{base}}
\]

The bindings documented in this section are provided by the \texttt{racket/function} and \texttt{racket} libraries, but not \texttt{racket/base}.

\[
(\text{identity \textit{v}}) \rightarrow \text{any/c}
\]
\[
\textit{v} : \text{any/c}
\]

Returns \( \textit{v} \).

\[
(\text{const \textit{v}}) \rightarrow \text{procedure}?
\]
\[
\textit{v} : \text{any}
\]

Returns a procedure that accepts any arguments (including keyword arguments) and returns \( \textit{v} \).

Examples:

\[
> ((\text{const 'foo}) 1 2 3)
\]

'foo

\[
> ((\text{const 'foo}))
\]

'foo

\[
(\text{thunk \textit{body} ...+})
\]
\[
(\text{thunk* \textit{body} ...+})
\]

The \texttt{thunk} form creates a nullary function that evaluates the given body. The \texttt{thunk*} form is similar, except that the resulting function accepts any arguments (including keyword arguments).

Examples:
(define th1 (thunk (define x 1) (printf "\n\n a\n x\n")))

> (th1)
1
> (th1 'x)
th1: arity mismatch:
the expected number of arguments does not match the given number
expected: 0
given: 1
> (th1 #:y 'z)
application: procedure does not accept keyword arguments
procedure: th1
arguments...:
#:y 'z

(define th2 (thunk* (define x 1) (printf "\n\n a\n x\n")))

> (th2)
1
> (th2 'x)
1
> (th2 #:y 'z)
1

(define th2 (thunk* (define x 1) (printf "\n\n a\n x\n")))

> (th2)
1
> (th2 'x)
1
> (th2 #:y 'z)
1

(negate proc) → procedure?
proc : procedure?

Returns a procedure that is just like proc, except that it returns the not of proc’s result.

Examples:

> (filter (negate symbol?) '(1 a 2 b 3 c))
'(1 2 3)
> (map (negate =) '(1 2 3) '(1 1 1))
'(##f #t #t)

((conjoin f ... x ...) → any
f : procedure?
x : any/c

Combines calls to each function with and. Equivalent to (and (f x ...) ...)

Examples:
(define f (conjoin exact? integer?))

> (f 1)
#t
> (f 1.0)
#f
> (f 1/2)
#f
> (f 0.5)
#f
> ((conjoin (λ (x) (values 1 2))) 0)
1
2

((disjoin f ...) x ...) -> any
  f : procedure?
  x : any/c

Combines calls to each function with or. Equivalent to (or (f x ...) ...)

Examples:

(define f (disjoin exact? integer?))

> (f 1)
#t
> (f 1.0)
#t
> (f 1/2)
#t
> (f 0.5)
#f
> ((disjoin (λ (x) (values 1 2))) 0)
1
2

(curry proc) -> procedure?
  proc : procedure?
(curry proc v ...+) -> any/c
  proc : procedure?
  v : any/c

Returns a procedure that is a curried version of proc. When the resulting procedure is first applied, unless it is given the maximum number of arguments that it can accept, the result is a procedure to accept additional arguments.
Examples:

```lisp
> ((curry list) 1 2)
#<procedure:curried:list>
> ((curry cons) 1)
#<procedure:curried:cons>
> ((curry cons) 1 2)
'(1 . 2)
```

After the first application of the result of `curry`, each further application accumulates arguments until an acceptable number of arguments have been accumulated, at which point the original `proc` is called.

Examples:

```lisp
> (((curry list) 1 2) 3)
'(1 2 3)
> (((curry list) 1) 3)
'(1 3)
> (((curry foldl) +) 0) '(1 2 3))
6
> (define foo (curry (lambda (x y z) (list x y z))))
> (foo 1 2 3)
'(1 2 3)
> (((((foo) 1) 2)) 3)
'(1 2 3)
```

A function call `(curry proc v ...)` is equivalent to `((curry proc) v ...)`. In other words, `curry` itself is curried.

Examples:

```lisp
> (map ((curry +) 10) '(1 2 3))
'(11 12 13)
> (map (curry + 10) '(1 2 3))
'(11 12 13)
> (map (compose (curry * 2) (curry + 10)) '(1 2 3))
'(22 24 26)
```

The `curry` function also supports functions with keyword arguments: keyword arguments will be accumulated in the same way as positional arguments until all required keyword arguments have been supplied.

Examples:

```lisp
(define (f #:a a #:b b #:c c)
  ...)
```
(list a b c))

> (((((curry f) #:a 1) #:b 2) #:c 3)
  '(1 2 3)
> (((((curry f) #:b 1) #:c 2) #:a 3)
  '(3 1 2)
> (((curry f #:a 1 #:c 2) #:b 3)
  '(1 3 2)

Changed in version 7.0.0.7 of package base: Added support for keyword arguments.

(curryr proc) → procedure?
  proc : procedure?
(curryr proc v ...+) → any/c
  proc : procedure?
  v : any/c

Like curry, except that the arguments are collected in the opposite direction: the first step collects the rightmost group of arguments, and following steps add arguments to the left of these.

Example:

> (map (curryr list 'foo) '(1 2 3))
  '(((1 foo) (2 foo) (3 foo))

(normalized-arity? arity) → boolean?
  arity : any/c

A normalized arity has one of the following forms:

- the empty list;
- an exact non-negative integer;
- an arity-at-least instance;
- a list of two or more strictly increasing, exact non-negative integers; or
- a list of one or more strictly increasing, exact non-negative integers followed by a single arity-at-least instance whose value is greater than the preceding integer by at least 2.

Every normalized arity is a valid procedure arity and satisfies procedure-arity?. Any two normalized arity values that are arity=? must also be equal?.

Examples:
> (normalized-arity? (arity-at-least 1))
#t
> (normalized-arity? (list (arity-at-least 1)))
#f
> (normalized-arity? (list 0 (arity-at-least 2)))
#t
> (normalized-arity? (list (arity-at-least 2) 0))
#f
> (normalized-arity? (list 0 2 (arity-at-least 3)))
#f

(normalize-arity
arity)
→ (and/c normalized-arity? (lambda (x) (arity=? x
arity)))
arity : procedure-arity?

Produces a normalized form of arity. See also normalized-arity? and arity=?.

Examples:
> (normalize-arity 1)
1
> (normalize-arity (list 1))
1
> (normalize-arity (arity-at-least 2))
(arity-at-least 2)
> (normalize-arity (list (arity-at-least 2)))
(arity-at-least 2)
> (normalize-arity (list 1 (arity-at-least 2)))
(arity-at-least 1)
> (normalize-arity (list (arity-at-least 2) 1))
(arity-at-least 1)
> (normalize-arity (list (arity-at-least 2) 3))
(arity-at-least 2)
> (normalize-arity (list 3 (arity-at-least 2)))
(arity-at-least 2)
> (normalize-arity (list (arity-at-least 6) 0 2 (arity-at-least 4)))
(list 0 2 (arity-at-least 4))

(arity=? a b) → boolean?
a : procedure-arity?
b : procedure-arity?

Returns #true if procedures with arity a and b accept the same numbers of arguments, and #false otherwise. Equivalent to both (and (arity-includes? a b) (arity-includes? b a)) and (equal? (normalize-arity a) (normalize-arity b)).
Examples:

```scheme
> (arity=? 1 1)
#t
> (arity=? (list 1) 1)
#t
> (arity=? 1 (list 1))
#t
> (arity=? 1 (arity-at-least 1))
#f
> (arity=? (arity-at-least 1) 1)
#f
> (arity=? (arity-at-least 1) (list 1 (arity-at-least 2)))
#t
> (arity=? (list 1 (arity-at-least 2)) (arity-at-least 1))
#t
> (arity=? (arity-at-least 1) (list 1 (arity-at-least 3)))
#f
> (arity=? (list 1 (arity-at-least 3)) (arity-at-least 1))
#f
> (arity=? (list 0 1 2 (arity-at-least 3)) (list (arity-at-least 0)))
#t
> (arity=? (list (arity-at-least 0)) (list 0 1 2 (arity-at-least 3)))
#t
> (arity=? (list 0 2 (arity-at-least 3)) (list (arity-at-least 0)))
#f
> (arity=? (list (arity-at-least 0)) (list 0 2 (arity-at-least 3)))
#f
```

(arity-includes? a b) → boolean?
  
  a : procedure-arity?
  b : procedure-arity?

Returns #true if procedures with arity a accept any number of arguments that procedures with arity b accept.

Examples:

```scheme
> (arity-includes? 1 1)
#t
> (arity-includes? (list 1) 1)
#t
```
> (arity-includes? 1 (list 1)) #t
> (arity-includes? 1 (arity-at-least 1)) #f
> (arity-includes? (arity-at-least 1) 1) #t
> (arity-includes? (arity-at-least 1) (list 1 (arity-at-least 2))) #t
> (arity-includes? (list 1 (arity-at-least 2)) (arity-at-least 1)) #t
> (arity-includes? (arity-at-least 1) (list 1 (arity-at-least 3))) #t
> (arity-includes? (arity-at-least 1) (list 0 2 (arity-at-least 3))) #f
> (arity-includes? (list 0 1 2 (arity-at-least 3)) (list (arity-at-least 0))) #t
> (arity-includes? (list (arity-at-least 0)) (list 0 1 2 (arity-at-least 3))) #t
> (arity-includes? (list 0 1 2 (arity-at-least 3)) (list (arity-at-least 0))) #f
> (arity-includes? (list (arity-at-least 0)) (list 0 2 (arity-at-least 3))) #t

4.19 Void

The constant #<void> is returned by most forms and procedures that have a side-effect and no useful result.

The #<void> value is always eq? to itself.

(define (void? v) boolean? 
  v : any/c)

Returns #t if v is the constant #<void>, #f otherwise.

(define (void v ...) void? 
  v : any/c)

Returns the constant #<void>. Each v argument is ignored.
4.20 Undefined

(\texttt{require racket/undefined}) \hspace{1em} \texttt{package: base}

The bindings documented in this section are provided by the \texttt{racket/undefined} library, not \texttt{racket/base} or \texttt{racket}.

The constant \texttt{undefined} can be used as a placeholder value for a value to be installed later, especially for cases where premature access of the value is either difficult or impossible to detect or prevent.

The \texttt{undefined} value is always \texttt{eq?} to itself.

Added in version 6.0.0.6 of package \texttt{base}.

\hspace{1em} \texttt{undefined : any/c}

The “undefined” constant.
5 Structures

A structure type is a record datatype composing a number of fields. A structure, an instance of a structure type, is a first-class value that contains a value for each field of the structure type. A structure instance is created with a type-specific constructor procedure, and its field values are accessed and changed with type-specific accessor and mutator procedures. In addition, each structure type has a predicate procedure that answers \#t for instances of the structure type and \#f for any other value.

A structure type’s fields are essentially unnamed, though names are supported for error-reporting purposes. The constructor procedure takes one value for each field of the structure type, except that some of the fields of a structure type can be automatic fields; the automatic fields are initialized to a constant that is associated with the structure type, and the corresponding arguments are omitted from the constructor procedure. All automatic fields in a structure type follow the non-automatic fields.

A structure type can be created as a structure subtype of an existing base structure type. An instance of a structure subtype can always be used as an instance of the base structure type, but the subtype gets its own predicate procedure, and it may have its own fields in addition to the fields of the base type.

A structure subtype “inherits” the fields of its base type. If the base type has \( m \) fields, and if \( n \) fields are specified for the new structure subtype, then the resulting structure type has \( m+n \) fields. The value for automatic fields can be different in a subtype than in its base type.

If \( m' \) of the original \( m \) fields are non-automatic (where \( m' < m \)), and \( n' \) of the new fields are non-automatic (where \( n' < n \)), then \( m'+n' \) field values must be provided to the subtype’s constructor procedure. Values for the first \( m \) fields of a subtype instance are accessed with selector procedures for the original base type (or its supertypes), and the last \( n \) are accessed with subtype-specific selectors. Subtype-specific accessors and mutators for the first \( m \) fields do not exist.

The struct form and make-struct-type procedure typically create a new structure type, but they can also access prefab (i.e., previously fabricated) structure types that are globally shared, and whose instances can be parsed and written by the default reader (see §1.3 “The Reader”) and printer (see §1.4 “The Printer”). Prefab structure types can inherit only from other prefab structure types, and they cannot have guards (see §5.2 “Creating Structure Types”) or properties (see §5.3 “Structure Type Properties”). Exactly one prefab structure type exists for each combination of name, supertype, field count, automatic field count, automatic field value (when there is at least one automatic field), and field mutability.

Two structure values are eqv? if and only if they are eq?. Two structure values are equal? if they are eq?. By default, two structure values are also equal? if they are instances of the same structure type, no fields are opaque, and the results of applying struct->vector to the structs are equal?. (Consequently, equal? testing for structures may depend on the current inspector.) A structure type can override the default equal? definition through the

§5 “Programmer-Defined Datatypes” in The Racket Guide introduces structure types via struct.
5.1 Defining Structure Types: `struct`

```
(struct id maybe-super (field ...)  
    struct-option ...)  

maybe-super =  
    | super-id

field = field-id  
    | [field-id field-option ...]

struct-option = #:mutable  
    | #:super super-expr  
    | #:inspector inspector-expr  
    | #:auto-value auto-expr  
    | #:guard guard-expr  
    | #:property prop-expr val-expr  
    | #:transparent  
    | #:prefab  
    | #:sealed  
    | #:authentic  
    | #:name name-id  
    | #:extra-name name-id  
    | #:constructor-name constructor-id  
    | #:extra-constructor-name constructor-id  
    | #:reflection-name symbol-expr  
    | #:methods gen:name-id method-defs  
    | #:omit-define-syntaxes  
    | #:omit-define-values

field-option = #:mutable  
    | #:auto

method-defs = (definition ...)
```

Creates a new structure type (or uses a pre-existing structure type if #:prefab is specified), and binds transformers and variables related to the structure type.

A `struct` form with `n fields` defines up to `4+2n` names:

- `struct:id`, a `structure type descriptor` value that represents the structure type.
• **constructor-id** (which defaults to **id**), a *constructor* procedure that takes *m* arguments and returns a new instance of the structure type, where *m* is the number of *fields* that do not include an #:auto option.

• **name-id** (which defaults to **id**), a transformer binding that encapsulates information about the structure type declaration. This binding is used to define subtypes, and it also works with the shared and match forms. For detailed information about the binding of **name-id**, see §5.7 “Structure Type Transformer Binding”.

The **constructor-id** and **name-id** can be the same, in which case **name-id** performs both roles. In that case, the expansion of **name-id** as an expression produces an otherwise inaccessible identifier that is bound to the constructor procedure; the expanded identifier has a 'constructor-for property whose value is an identifier that is free-identifier=? to **name-id** as well as a syntax property accessible via syntax-procedure-alias-property with an identifier that is free-identifier=? to **name-id**.

• **id?**, a predicate procedure that returns #t for instances of the structure type (constructed by **constructor-id** or the constructor for a subtype) and #f for any other value.

• **id-field-id**, for each field; an accessor procedure that takes an instance of the structure type and extracts the value for the corresponding field.

• **set-id-field-id!**, for each field that includes a #:mutable option, or when the #:mutable option is specified as a struct-option; a mutator procedure that takes an instance of the structure type and a new field value. The structure is destructively updated with the new value, and #<void> is returned.

If **super-id** is provided, it must have a transformer binding of the same sort bound to **name-id** (see §5.7 “Structure Type Transformer Binding”), and it specifies a supertype for the structure type. Alternately, the #:super option can be used to specify an expression that must produce a structure type descriptor. See §5 “Structures” for more information on structure subtypes and supertypes. If both **super-id** and #:super are provided, a syntax error is reported.

Examples:

> (struct document (author title content))
> (struct book document (publisher))
> (struct paper (journal) #:super struct:document)

If the #:mutable option is specified for an individual field, then the field can be mutated in instances of the structure type, and a mutator procedure is bound. Supplying #:mutable as a struct-option is the same as supplying it for all fields. If #:mutable is specified as both a field-option and struct-option, a syntax error is reported.

Examples:
> (struct cell ([content #:mutable]) #:transparent)
> (define a-cell (cell 0))
> (set-cell-content! a-cell 1)

The #:inspector, #:auto-value, and #:guard options specify an inspector, value for automatic fields, and guard procedure, respectively. See make-struct-type for more information on these attributes of a structure type. The #:property option, which can be supplied multiple times, attaches a property value to the structure type; see §5.3 “Structure Type Properties” for more information on properties. The #:transparent option is a shorthand for #:inspector #f.

Examples:

> (struct point (x y) #:inspector #f)
> (point 3 5)
(point 3 5)
> (struct celsius (temp)
   #:guard (λ (temp name)
     (unless (and (real? temp) (>= temp -273.15))
       (error "not a valid temperature"))
     temp))
> (celsius -275)
not a valid temperature

The #:prefab option obtains a prefab (pre-defined, globally shared) structure type, as opposed to creating a new structure type. Such a structure type is inherently transparent and non-sealed, and it cannot have a guard or properties, so using #:prefab with #:transparent, #:inspector, #:guard, #:property, #:sealed, #:authentic, or #:methods is a syntax error. If a supertype is specified, it must also be a prefab structure type.

Examples:

> (struct prefab-point (x y) #:prefab)
> (prefab-point 1 2)
'(prefab-point 1 2)
> (prefab-point? #s(prefab-point 1 2))
#t

The #:sealed option is a shorthand for #:property prop:sealed #t, which prevents the structure type from being used as the supertype of another structure type. See prop:sealed for more information.

The #:authentic option is a shorthand for #:property prop:authentic #t, which prevents instances of the structure type from being impersonated (see impersonate-struct), chaperoned (see chaperone-struct), or acquiring a non-flat contract (see
struct/c). See prop:authentic for more information. If a supertype is specified, it must also have the prop:authentic property.

If name-id is supplied via #:extra-name and it is not id, then both name-id and id are bound to information about the structure type. Only one of #:extra-name and #:name can be provided within a struct form, and #:extra-name cannot be combined with #:omit-define-syntaxes.

Examples:

> (struct ghost (color name) #:prefab #:extra-name GHOST)
> (match (ghost 'red 'blinky)
  [(GHOST c n) c])
'red

If constructor-id is supplied, then the transformer binding of name-id records constructor-id as the constructor binding; as a result, for example, struct-out includes constructor-id as an export. If constructor-id is supplied via #:extra-constructor-name and it is not id, applying object-name on the constructor produces the symbolic form of id rather than constructor-id. If constructor-id is supplied via #:constructor-name and it is not the same as name-id, then name-id does not serve as a constructor, and object-name on the constructor produces the symbolic form of constructor-id. Only one of #:extra-constructor-name and #:constructor-name can be provided within a struct form.

Examples:

> (struct color (r g b) #:constructor-name -color)
> (struct rectangle (w h color) #:extra-constructor-name rect)
> (rectangle 13 50 (-color 192 157 235))
#<rectangle>
> (rect 50 37 (-color 35 183 252))
#<rectangle>

If #:reflection-name symbol-expr is provided, then symbol-expr must produce a symbol that is used to identify the structure type in reflective operations such as struct-type-info. It corresponds to the first argument of make-struct-type. Structure printing uses the reflective name, as do the various procedures that are bound by struct.

Examples:

> (struct circle (radius) #:reflection-name '<circle>)
> (circle 15)
#<circle>
> (circle-radius "bad")
<circle>-radius: contract violation
If #:methods gen:name-id method-defs is provided (potentially multiple times), then gen:name-id must be a transformer binding for the static information about a generic interface produced by define-generics. The method-defs define the methods of the gen:name-id interface. A define/generic form or auxiliary definitions and expressions may also appear in method-defs.

Examples:

```scheme
> (struct constant-stream (val)
  #:methods gen:stream
  [(define (stream-empty? stream) #f)
   (define (stream-first stream)
     (constant-stream-val stream))
   (define (stream-rest stream) stream)]
> (stream-ref (constant-stream 'forever) 0)
'forever
> (stream-ref (constant-stream 'forever) 50)
'forever
```

If the #:omit-define-syntaxes option is supplied, then name-id (and id, if #:extra-name is specified) is not bound as a transformer. If the #:omit-define-values option is supplied, then none of the usual variables are bound, but id is bound. If both are supplied, then the struct form is equivalent to (begin).

Examples:

```scheme
> (struct square (side) #:omit-define-syntaxes)
> (match (square 5)
  ; fails to match because syntax is omitted
  [(struct square x) x])
```

Expressions supplied to #:auto-value are evaluated once and shared between every instance of the structure type. In particular, updates to a mutable #:auto-value affect all current and future instances.
automatic value is used for the field, as specified by the #:auto-value option, or as defaults to #:auto when #:auto-value is not supplied. The field is mutable (e.g., through reflective operations), but a mutator procedure is bound only if #:mutable is specified.

If a field includes the #:auto option, then all fields after it must also include #:auto, otherwise a syntax error is reported. If any field-option or struct-option keyword is repeated, other than #:property, a syntax error is reported.

Examples:

```lisp
(struct posn (x y [z #:auto #:mutable])
    #:auto-value 0
    #:transparent)
```

> (posn 1 2)
(posn 1 2 0)
> (posn? (posn 1 2))
#t
> (posn-y (posn 1 2))
2
> (posn-z (posn 1 2))
0

```lisp
(struct color-posn posn (hue) #:mutable)
(define cp (color-posn 1 2 "blue"))
```

> (color-posn-hue cp)
"blue"
> cp
(color-posn 1 2 0 ...)
> (set-posn-z! cp 3)

For serialization, see define-serializable-struct.

Changed in version 6.9.0.4 of package base: Added #:authentic.
Changed in version 8.0.0.7: Added #:sealed.

```lisp
(struct-field-index field-id)
```

This form can only appear as an expression within a struct form; normally, it is used with #:property, especially for a property like prop:procedure. The result of a struct-field-index expression is an exact, non-negative integer that corresponds to the position within the structure declaration of the field named by field-id.

Examples:

```lisp
(struct mood-procedure (base rating)
```
#:property prop:procedure (struct-field-index base))

(define happy+ (mood-procedure add1 10))

> (happy+ 2)
3
> (mood-procedure-rating happy+)
10

(define-struct id-maybe-super (field ...)
struct-option ...)

id-maybe-super = id
    | (id super-id)

Like struct, except that the syntax for supplying a super-id is different, and a constructor-id that has a make-prefix on id is implicitly supplied via #:extra-constructor-name if neither #:extra-constructor-name nor #:constructor-name is provided.

This form is provided for backwards compatibility; struct is preferred.

Examples:

(define-struct posn (x y [z #:auto])
    #:auto-value 0
    #:transparent)

> (make-posn 1 2)
(posn 1 2 0)
> (posn? (make-posn 1 2))
#t
> (posn-y (make-posn 1 2))
2

(struct/derived (id . rest-form)
    id (field ...) struct-option ...)
(struct/derived (id . rest-form)
    id super-id (field ...) struct-option ...)

The same as struct, but with an extra (id . rest-form) sub-form that is treated as the overall form for syntax-error reporting and otherwise ignored. The only constraint on the sub-form for error reporting is that it starts with id. The struct/derived form is intended for use by macros that expand to struct.

Examples:
(define-syntax (fruit-struct stx)
  (syntax-case stx ()
    [(ds name . rest)
      (with-syntax ([orig stx])
        '#'(struct/derived orig name (seeds color) . rest))]))

> (fruit-struct apple)
> (apple-seeds (apple 12 "red"))
12
> (fruit-struct apple #:mutable)
> (set-apple-seeds! (apple 12 "red") 8)
; this next line will cause an error due to a bad keyword
> (fruit-struct apple #:bad-option)
eval:54:0: fruit-struct: unrecognized struct-specification keyword
  at: #:bad-option
  in: (fruit-struct apple #:bad-option)

Added in version 7.5.0.16 of package base.

(define-struct/derived (id . rest-form)
  id-maybe-super (field ...) struct-option ...)

Like struct/derived, except that the syntax for supplying a super-id is different, and a constructor-id that has a make-prefix on id is implicitly supplied via #:extra-constructor-name if neither #:extra-constructor-name nor #:constructor-name is provided. The define-struct/derived form is intended for use by macros that expand to define-struct.

Examples:

(define-syntax (define-xy-struct stx)
  (syntax-case stx ()
    [(ds name . rest)
      (with-syntax ([orig stx])
        '#'(define-struct/derived orig name (x y) . rest))]))

> (define-xy-struct posn)
> (posn-x (make-posn 1 2))
1
> (define-xy-struct posn #:mutable)
> (set-posn-x! (make-posn 1 2) 0)
; this next line will cause an error due to a bad keyword
> (define-xy-struct posn #:bad-option)
eval:60:0: define-xy-struct: unrecognized struct-specification keyword

504
5.2 Creating Structure Types

\[
\text{(make-struct-type} \text{ name super-type init-field-cnt auto-field-cnt [auto-v props inspector proc-spec immutables guard constructor-name])}
\]

\]

Creates a new structure type, unless inspector is 'prefab, in which case make-struct-type accesses a prefab structure type. The name argument is used as the type name. If super-type is not #f, the resulting type is a subtype of the corresponding structure type.

The resulting structure type has init-field-cnt+auto-field-cnt fields (in addition to
any fields from super-type), but only init-field-cnt constructor arguments (in addition to any constructor arguments from super-type). The remaining fields are initialized with auto-v. The total field count (including super-type fields) must be no more than 32768.

The props argument is a list of pairs, where the car of each pair is a structure type property descriptor, and the cdr is an arbitrary value. A property can be specified multiple times in props (including properties that are automatically added by properties that are directly included in props) only if the associated values are eq?, otherwise the exn:fail:contract exception is raised. See §5.3 "Structure Type Properties" for more information about properties. When inspector is ’prefab, then props must be null.

The inspector argument normally controls access to reflective information about the structure type and its instances; see §14.9 "Structure Inspectors" for more information. If inspector is ’prefab, then the resulting prefab structure type and its instances are always transparent. If inspector is #f, then the structure type’s instances are transparent.

If proc-spec is an integer or procedure, instances of the structure type act as procedures. See prop:procedure for further information. Providing a non-#f value for proc-spec is the same as pairing the value with prop:procedure at the end of props, plus including proc-spec in immutables when proc-spec is an integer.

The immutables argument provides a list of field positions. Each element in the list must be unique, otherwise exn:fail:contract exception is raised. Each element must also fall in the range 0 (inclusive) to init-field-cnt (exclusive), otherwise exn:fail:contract exception is raised.

The guard argument is either a procedure of n+1 arguments or #f, where n is the number of arguments for the new structure type’s constructor (i.e., init-field-cnt plus constructor arguments implied by super-type, if any). If guard is a procedure, then the procedure is called whenever an instance of the type is constructed, or whenever an instance of a subtype is created. The arguments to guard are the values provided for the structure’s first n fields, followed by the name of the instantiated structure type (which is name, unless a subtype is instantiated). The guard result must be n values, which become the actual values for the structure’s fields. The guard can raise an exception to prevent creation of a structure with the given field values. If a structure subtype has its own guard, the subtype guard is applied first, and the first n values produced by the subtype’s guard procedure become the first n arguments to guard. When inspector is ’prefab, then guard must be #f.

If constructor-name is not #f, it is used as the name of the generated constructor procedure as returned by object-name or in the printed form of the constructor value.

The result of make-struct-type is five values:

- a structure type descriptor,
- a constructor procedure,
- a predicate procedure,
- an accessor procedure, which consumes a structure and a field index between 0 (inclusive) and \( \text{init-field-cnt} + \text{auto-field-cnt} \) (exclusive), and
- a mutator procedure, which consumes a structure, a field index, and a field value.

Examples:

```scheme
(define-values (struct:a make-a a? a-ref a-set!)
    (make-struct-type 'a #f 2 1 'uninitialized))
(define an-a (make-a 'x 'y))

> (a-ref an-a 1)
y
> (a-ref an-a 2)
'uninitialized
> (define a-first (make-struct-field-accessor a-ref 0))
> (a-first an-a)
'x

(define-values (struct:b make-b b? b-ref b-set!)
    (make-struct-type 'b struct:a 1 2 'b-uninitialized))
(define a-b (make-b 'x 'y 'z))

> (a-ref a-b 1)
y
> (a-ref a-b 2)
'uninitialized
> (b-ref a-b 0)
'z
> (b-ref a-b 1)
'b-uninitialized
> (b-ref a-b 2)
'b-uninitialized

(define-values (struct:c make-c c? c-ref c-set!)
    (make-struct-type
        'c struct:b 0 0 #f null (make-inspector) #f null
        ; guard checks for a number, and makes it inexact
        (lambda (a1 a2 b1 name)
            (unless (number? a2)
                (error (string->symbol (format "make-~a" name))
                        "second field must be a number"))
            (values a1 (exact->inexact a2 b1)))))
```

507
> (make-c 'x 'y 'z)
make-c: second field must be a number
> (define a-c (make-c 'x 2 'z))
> (a-ref a-c 1)
2.0

(define p1 #s(p a b c))
(define-values (struct:p make-p p-ref p-set!)
  (make-struct-type 'p #f 3 0 #f null 'prefab #f '(0 1 2)))

> (p? p1)
#t
> (p-ref p1 0)
'a
> (make-p 'x 'y 'z)
'#s(p x y z)

(make-struct-field-accessor accessor-proc
  field-pos
  [field-name]) → procedure?
accessor-proc : struct-accessor-procedure?
field-pos : exact-nonnegative-integer?
field-name : (or/c symbol? #f)
  = (symbol->string (format "field-" a field-pos))

Returns a field accessor that is equivalent to (lambda (s) (accessor-proc s field-pos)). The accessor-proc must be an accessor returned by make-struct-type. The name of the resulting procedure for debugging purposes is derived from field-name and the name of accessor-proc’s structure type if field-name is a symbol.

For examples, see make-struct-type.

(make-struct-field-mutator mutator-proc
  field-pos
  [field-name]) → procedure?
mutator-proc : struct-mutator-procedure?
field-pos : exact-nonnegative-integer?
field-name : (or/c symbol? #f)
  = (symbol->string (format "field-~a" field-pos))

Returns a field mutator that is equivalent to (lambda (s v) (mutator-proc s field-pos v)). The mutator-proc must be a mutator returned by make-struct-type. The name of the resulting procedure for debugging purposes is derived from field-name and the name of mutator-proc’s structure type if field-name is a symbol.

For examples, see make-struct-type.
**prop:sealed**: `struct-type-property?`

A structure type property that declares a structure type as *sealed*. The value associated with the property is ignored; the presence of the property itself makes the structure type sealed.

A sealed structure type cannot be used as the supertype of another structure type. Declaring a structure type as sealed is typically just a performance hint, since checking for an instance of a sealed structure type can be slightly faster than checking for an instance of a structure type that might have subtypes.

Added in version 8.0.0.7 of package base.

### 5.3 Structure Type Properties

A *structure type property* allows per-type information to be associated with a structure type (as opposed to per-instance information associated with a structure value). A property value is associated with a structure type through the `make-struct-type` procedure (see §5.2 “Creating Structure Types”) or through the `#:property` option of `struct`. Subtypes inherit the property values of their parent types, and subtypes can override an inherited property value with a new value.

```scheme
(make-struct-type-property name [guard supers can-impersonate?])
```

→ `struct-type-property?`  
  procedure?  
  procedure?  
  name : symbol?  
  guard : (or/c procedure? #f 'can-impersonate) = #f  
  supers : (listof (cons/c struct-type-property? = null (any/c . -> . any/c)))  
  can-impersonate? : any/c = #f

Creates a new structure type property and returns three values:

- a *structure type property descriptor*, for use with `make-struct-type` and `struct`;
- a *property predicate* procedure, which takes an arbitrary value and returns `#t` if the value is a descriptor or instance of a structure type that has a value for the property, `#f` otherwise;
- a *property accessor* procedure, which returns the value associated with the structure type given its descriptor or one of its instances; if the structure type does
If the optional `guard` is supplied as a procedure, it is called by `make-struct-type` before attaching the property to a new structure type. The `guard` must accept two arguments: a value for the property supplied to `make-struct-type`, and a list containing information about the new structure type. The list contains the values that `struct-type-info` would return for the new structure type if it skipped the immediate current-inspector control check (but not the check for exposing an ancestor structure type, if any; see §14.9 "Structure Inspectors").

The result of calling `guard` is associated with the property in the target structure type, instead of the value supplied to `make-struct-type`. To reject a property association (e.g., because the value supplied to `make-struct-type` is inappropriate for the property), the `guard` can raise an exception. Such an exception prevents `make-struct-type` from returning a structure type descriptor.

If `guard` is `can-impersonate`, then the property’s accessor can be redirected through `impersonate-struct`. This option is identical to supplying `#t` as the `can-impersonate?` argument and is provided for backwards compatibility.

The optional `supers` argument is a list of properties that are automatically associated with some structure type when the newly created property is associated to the structure type. Each property in `supers` is paired with a procedure that receives the value supplied for the new property (after it is processed by `guard`) and returns a value for the associated property (which is then sent to that property’s guard, of any).

The optional `can-impersonate?` argument determines if the structure type property can be redirected through `impersonate-struct`. If the argument is `#f`, then redirection is not allowed. Otherwise, the property accessor may be redirected by a struct impersonator.

Examples:

```scheme
> (define-values (prop:p p? p-ref) (make-struct-type-property 'p))
> (define-values (struct:a make-a a? a-ref a-set!) (make-struct-type 'a #f 2 1 'uninitialized (list (cons prop:p 8))))
> (p? struct:a)
#t
> (p? 13)
#f
> (define an-a (make-a 'x 'y))
> (p? an-a)
```

510
> (define-values (struct:b make-b b-ref b-set!) (make-struct-type 'b #f 0 0 #f))
> (p? struct:b) #f
> (define-values (prop:q q? q-ref) (make-struct-type-property 'q (lambda (v si) (add1 v)) (list (cons prop:p sqrt))))
> (define-values (struct:c make-c c? c-ref c-set!) (make-struct-type 'c #f 0 0 'uninit (list (cons prop:q 8))))
> (q-ref struct:c) 9
> (p-ref struct:c) 3

(struct-type-property? v) → boolean?
  v : any/c

Returns #t if v is a structure type property descriptor value, #f otherwise.

(struct-type-property-accessor-procedure? v) → boolean?
  v : any/c

Returns #t if v is an accessor procedure produced by make-struct-type-property, #f otherwise.

(struct-type-property-predicate-procedure? v [prop]) → boolean?
  v : any/c
  prop : (or/c struct-type-property? #f) = #f

Returns #t if v is a predicate procedure produced by make-struct-type-property and either prop is #f or it was produced by the same call to make-struct-type-property, #f otherwise.

Added in version 7.5.0.11 of package base.

5.4 Generic Interfaces

(require racket/generic)
A generic interface allows per-type methods to be associated with generic functions. Generic functions are defined using a define-generics form. Method implementations for a structure type are defined using the #:methods keyword (see §5.1 “Defining Structure Types: struct”).

\[
\begin{align*}
\text{(define-generics id} & \text{generics-opt ...} \\
& \text{method-id . kw-formals*] ...} \\
& \text{generics-opt ...) } \\
\text{generics-opt} = & \ #:\text{defaults } ([\text{default-pred? default-impl ...}] ...) \\
& | \ #:\text{fast-defaults } ([\text{fast-pred? fast-impl ...}] ...) \\
& | \ #:\text{fallbacks } [\text{fallback-impl ...}] \\
& | \ #:\text{defined-predicate } \text{defined-pred-id} \\
& | \ #:\text{defined-table } \text{defined-table-id} \\
& | \ #:\text{derive-property } \text{prop-expr prop-value-expr} \\
\text{kw-formals*} = & \ (\text{arg* ...}) \\
& | \ (\text{arg* ...}+ . \text{rest-id}) \\
& | \ \text{rest-id} \\
\text{arg*} = & \ \text{arg-id} \\
& | \ [\text{arg-id}] \\
& | \ \text{keyword arg-id} \\
& | \ \text{keyword [arg-id]}
\end{align*}
\]

Defines the following names, plus any specified by keyword options.

- \text{gen:id} as a transformer binding for the static information about a new generic interface;
- \text{id?} as a predicate identifying instances of structure types that implement this generic group; and
- each \text{method-id} as a generic method that calls the corresponding method on values where \text{id?} is true. Each \text{method-id}'s \text{kw-formals*} must include a required by-position argument that is \text{free-identifier=?} to \text{id}. That argument is used in the generic definition to locate the specialization.
- \text{id/c} as a contract combinator that recognizes instances of structure types which implement the \text{gen:id} generic interface. The combinator takes pairs of \text{method-ids} and contracts. The contracts will be applied to each of the corresponding method implementations. The \text{id/c} combinator is intended to be used to contract the range of a constructor procedure for a struct type that implements the generic interface.

The #:defaults option may be provided at most once. When it is provided, each generic function uses default-pred?s to dispatch to the given default method implementations,
The #:fast-defaults option may be provided at most once. It works the same as #:defaults, except the fast-pred? is checked before dispatching to the generic method table. This option is intended to provide a fast path for dispatching to built-in datatypes, such as lists and vectors, that do not overlap with structures implementing gen:id.

The #:fallbacks option may be provided at most once. When it is provided, the fallback-impls define fallback method implementations that are used for any instance of the generic interface that does not supply a specific implementation. The syntax of the fallback-impls is the same as the methods provided for the #:methods keyword for struct.

The #:defined-predicate option may be provided at most once. When it is provided, defined-pred-id is defined as a procedure that reports whether a specific instance of the generic interface implements a given set of methods. Specifically, \( (\text{defined-pred-id } v \ '\text{name ...} ) \) produces \#t if v has implementations for each method name, not counting #:fallbacks implementations, and produces \#f otherwise. This procedure is intended for use by higher-level APIs to adapt their behavior depending on method availability.

The #:defined-table option may be provided at most once. When it is provided, defined-table-id is defined as a procedure that takes an instance of the generic interface and returns an immutable hash table that maps symbols corresponding to method names to booleans representing whether or not that method is implemented by the instance. This option is deprecated; use #:defined-predicate instead.

If a value v satisfies id?, then v is a generic instance of gen:id.

If a generic instance v has a corresponding implementation for some method-id provided via #:methods in struct or via #:defaults or #:fast-defaults in define-generics, then method-id is an implemented generic method of v.

If method-id is not an implemented generic method of a generic instance v, and method-id has a fallback implementation that does not raise an exn:fail:support exception when given v, then method-id is a supported generic method of v.

\[
\text{(raise-support-error name v)} \rightarrow \text{none/c} \\
\text{name : symbol?} \\
\text{v : any/c}
\]
Raises an `exn:fail:support` exception for a generic method called `name` that does not support the generic instance `v`.

Example:

```lisp
> (raise-support-error 'some-method-name '("arbitrary" "instance" "value"))
some-method-name: not implemented for ('"arbitrary" "instance" "value")
```

```lisp
(struct exn:fail:support exn:fail () #:transparent)
```

Raised for generic methods that do not support the given generic instance.

```lisp
(define/generic local-id method-id)
```

When used inside the method definitions associated with the `#:methods`, `#:fallbacks`, `#:defaults` or `#:fast-defaults` keywords, binds `local-id` to the generic for `method-id`. This form is useful for method specializations to use generic methods (as opposed to the local specialization) on other values.

The `define/generic` form is only allowed inside:

- a `#:methods` specification in `struct` (or `define-struct`)
- the specification of `#:fallbacks`, `#:defaults` or `#:fast-defaults` in `define-generics`

Using `define/generic` elsewhere is a syntax error.

Examples:

```lisp
> (define-generics printable
   (gen-print printable [port])
   (gen-port-print port printable)
   (gen-print* printable [port] #:width width #:height [height])
   #:defaults ([string?
     (define/generic super-print gen-print)
     (define (gen-print s [port (current-output-port)])
       (printf port "String: ~a" s))
     (define (gen-port-print port s)
       ; we can call gen-print alternatively
       (super-print s port))
```

514
(define (gen-print* s [port (current-output-port)])
    #:width w #:height [h 0])
    (fprintf port "String (~ax~a): ~a" w h s)))

> (struct num (v)
    #:methods gen:printable
    [(define (gen-print n [port (current-output-port)])
        (fprintf port "Num: ~a" (num-v n)))
      (define (gen-port-print port n)
        (gen-print n port))
      (define (gen-print* n [port (current-output-port)]
        #:width w #:height [h 0])
        (fprintf port "Num (~ax~a): ~a" w h (num-v n)))]

> (struct string+num (v n)
    #:methods gen:printable
    [(define/generic super-print gen-print)
     (define/generic super-print* gen-print*)
      (define (gen-print b [port (current-output-port)])
        (super-print (string+num-v b) port)
        (fprintf port " ")
        (super-print (string+num-n b) port))
      (define (gen-port-print port b)
        (gen-print b port))
      (define (gen-print* b [port (current-output-port)]
        #:width w #:height [h 0])
        (super-print* (string+num-v b) #:width w #:height h)
        (fprintf port " ")
        (super-print* (string+num-n b) #:width w #:height h)))]

> (define x (num 10))
> (gen-print x)
Num: 10
> (gen-port-print (current-output-port) x)
Num: 10
> (gen-print* x #:width 100 #:height 90)
Num (100x90): 10
> (define str "Strings are printable too!"
> (gen-print str)
String: Strings are printable too!
> (define y (string+num str x))
> (gen-print y)
String: Strings are printable too! Num: 10
> (gen-port-print (current-output-port) y)
String: Strings are printable too! Num: 10
> (gen-print* y #:width 100 #:height 90)
String (100x90): Strings are printable too! Num (100x90): 10
> (define/contract make-num-contracted
(-> number? (printable/c [gen-print (->* (printable?) (output-port?) void?)]) [gen-port-print (-> output-port? printable? void?)]) [gen-print* (->* (printable? #:width exact-nonnegative-integer?) (output-port? #:height exact-nonnegative-integer?) void?)])

num)

> (define z (make-num-contracted 10))
> (gen-print* z #:width "not a number" #:height 5)

make-num-contracted: contract violation

expected: natural?
given: "not a number"
in: the #:width argument of
    method gen-print*
    the range of
    (-> number? (printable/c (gen-print (->* (printable?) (output-port?) void?)) (gen-port-print (-> output-port? printable? void?)) (gen-print* (->* (printable? #:width natural?) (output-port? #:height natural?) void?))))

contract from:
    (definition make-num-contracted)
blaming: top-level
    (assuming the contract is correct)
at: eval:16:0

(generic-instance/c gen-id [method-id method-ctc] ...)

method-ctc : contract?

Creates a contract that recognizes structures that implement the generic interface gen-id, and constrains their implementations of the specified method-ids with the corresponding method-ctcs.

(impersonate-generics gen-id val-exp
    [method-id method-proc-exp] ...
    maybe-properties)
maybe-properties =
  | #:properties props-expr

method-proc-expr : (any/c . -> . any/c)
props-expr : (list/c impersonator-property? any/c ... ...)

Creates an impersonator of `val-expr`, which must be a structure that implements the generic interface `gen-id`. The impersonator applies the results of the `method-proc-exprs` to the structure’s implementation of the corresponding `method-ids`, and replaces the method implementation with the result.

A `props-expr` can provide properties to attach to the impersonator. The result of `props-expr` must be a list with an even number of elements, where the first element of the list is an impersonator property, the second element is its value, and so on.

Changed in version 6.1.1.8 of package `base`: Added `#:properties`.

```lisp
(chaperone-generics gen-id val-expr
  [method-id method-proc-expr] ...
  maybe-properties)
```

Like `impersonate-generics`, but creates a chaperone of `val-expr`, which must be a structure that implements the generic interface `gen-id`. The chaperone applies the specified `method-procs` to the structure’s implementation of the corresponding `method-ids`, and replaces the method implementation with the result, which must be a chaperone of the original.

```lisp
(redirect-generics mode gen-id val-expr
  [method-id method-proc-expr] ...
  maybe-properties)
```

Like `impersonate-generics`, but creates an impersonator of `val-expr` if `mode` evaluates to `#f`, or creates a chaperone of `val-expr` otherwise.

```lisp
(make-struct-type-property/generic
  name-expr
  maybe-guard-expr
  maybe-supers-expr
  maybe-can-impersonate?-expr
  property-option
  ...
)```
maybe-guard-expr =
   | guard-expr

maybe-supers-expr =
   | supers-expr

maybe-can-impersonate?-expr =
   | can-impersonate?-expr

property-option = #:property prop-expr val-expr
   | #:methods gen:name-id method-defs

method-defs = (definition ...)

name-expr : symbol?

guard-expr : (or/c procedure? #f 'can-impersonate)
supers-expr : (listof (cons/c struct-type-property? (-> any/c any/c)))
can-impersonate?-expr : any/c
prop-expr : struct-type-property?
val-expr : any/c

Creates a new structure type property and returns three values, just like make-struct-type-property would:

• a structure type property descriptor
• a property predicate procedure
• a property accessor procedure

Any struct that implements this property will also implement the properties and generic interfaces given in the #:property and #:methods declarations. The property val-exprs and method-defs are evaluated eagerly when the property is created, not when it is attached to a structure type.

(make-generic-struct-type-property
   gen:name-id
   method-def
   ...
)

Creates a new structure type property and returns the structure type property descriptor.

Any struct that implements this property will also implement the generic interface given by gen:name-id with the given method-defs. The method-defs are evaluated eagerly when the property is created, not when it is attached to a structure type.
null
> (define not-really-chum
  (struct-copy fish bruce
              [weight 90]))
> not-really-chum
(fish 'grey 90)

5.6 Structure Utilities

(struct->vector v [opaque-v]) → vector?
  v : any/c
  opaque-v : any/c = '...

Creates a vector representing v. The first slot of the result vector contains a symbol whose
pattern name has the form struct:id. Each remaining slot contains either the value of a
field in v, if it is accessible via the current inspector, or opaque-v for a field that is not
accessible. A single opaque-v value is used in the vector for contiguous inaccessible fields.
(Consequently, the size of the vector does not match the size of the struct if more than one
field is inaccessible.)

(struct? v) → any
  v : any/c

Returns #t if struct-info exposes any structure types of v with the current inspector, #f
otherwise.

Typically, when (struct? v) is true, then (struct->vector v) exposes at least one
field value. It is possible, however, for the only visible types of v to contribute zero fields.

(struct-type? v) → boolean?
  v : any/c

Returns #t if v is a structure type descriptor value, #f otherwise.

(struct-constructor-procedure? v) → boolean?
  v : any/c

Returns #t if v is a constructor procedure generated by struct or make-struct-type, #f
otherwise.

(struct-predicate-procedure? v) → boolean?
  v : any/c

Returns #t if v is a predicate procedure generated by struct or make-struct-type, #f
otherwise.
Returns \#t if \(v\) is an accessor procedure generated by \texttt{struct}, \texttt{make-struct-type}, or \texttt{make-struct-field-accessor}, \#f otherwise.

\[
\text{(struct-accessor-procedure? } v) \rightarrow \text{ boolean?}
\]
\[
v : \text{ any/c}
\]

\[
\text{Returns \#t if } v \text{ is a mutator procedure generated by } \texttt{struct}, \texttt{make-struct-type}, \texttt{or make-struct-field-mutator}, \#f otherwise.}
\]
\[
\text{(struct-mutator-procedure? } v) \rightarrow \text{ boolean?}
\]
\[
v : \text{ any/c}
\]

\[
\text{Returns \#f if } v \text{ is not an instance of a prefab structure type. Otherwise, the result is the shorted key that could be used with } \texttt{make-prefab-struct} \text{ to create an instance of the structure type.}
\]
\[
\text{Examples:}
\]
\[
> \text{(prefab-struct-key } \text{#s(cat "Garfield"))}
\]
\[
'\text{cat}
\]
\[
> \text{(struct cat (name) #:prefab)}
\]
\[
> \text{(struct cute-cat cat (shipping-dest) #:prefab)}
\]
\[
> \text{(cute-cat "Nermel" "Abu Dhabi")}
\]
\[
'>\text{(cute-cat cat 1)}
\]

\[
\text{(make-prefab-struct } \text{key } v \ldots) \rightarrow \text{ struct?}
\]
\[
\text{key} : \text{ prefab-key?}
\]
\[
v : \text{ any/c}
\]

Creates an instance of a prefab structure type, using the \(v\)s as field values. The \textit{key} and the number of \(v\)s determine the prefab structure type.

A \textit{key} identifies a structure type based on a list with the following items:

- A symbol for the structure type's name.
- An exact, nonnegative integer representing the number of non-automatic fields in the structure type, not counting fields from the supertype (if any).
• A list of two items, where the first is an exact, nonnegative integer for the number of automatic fields in the structure type that are not from the supertype (if any), and the second element is an arbitrary value that is the value for the automatic fields.

• A vector of exact, nonnegative integers that indicate mutable non-automatic fields in the structure type, counting from 0 and not including fields from the supertype (if any).

• Nothing else, if the structure type has no supertype. Otherwise, the rest of the list is the key for the supertype.

An empty vector and an auto-field list that starts with 0 can be omitted. Furthermore, the first integer (which indicates the number of non-automatic fields) can be omitted, since it can be inferred from the number of supplied vs. Finally, a single symbol can be used instead of a list that contains only a symbol (in the case that the structure type has no supertype, no automatic fields, and no mutable fields).

The total field count must be no more than 32768. If the number of fields indicated by key is inconsistent with the number of supplied vs, the exn:fail:contract exception is raised.

Examples:

```lisp
> (make-prefab-struct 'clown "Binky" "pie")
'#s(clown "Binky" "pie")
> (make-prefab-struct '(clown 2) "Binky" "pie")
'#s(clown Binky "pie")
> (make-prefab-struct '(clown 2 (0 #f) #()) "Binky" "pie")
'#s(clown Binky "pie")
> (make-prefab-struct '(clown 1 (1 #f) #()) "Binky" "pie")
'#s((clown 1 #f) "Binky" "pie")
> (make-prefab-struct '(clown 1 (1 #f) #(0)) "Binky" "pie")
'#s((clown 1 #f) #(0)) "Binky" "pie")
```

```
(prefab-key->struct-type key field-count) -> struct-type?
key : prefab-key?
field-count : (integer-in 0 32768)
```

Returns a structure type descriptor for the prefab structure type specified by the combination of key and field-count.

If the number of fields indicated by key is inconsistent with field-count, the exn:fail:contract exception is raised.

```
(prefab-key? v) -> boolean?
v : any/c
```

522
Return `#t` if `v` can be a prefab structure type key, `#f` otherwise.

See `make-prefab-struct` for a description of valid key shapes.

### 5.6.1 Additional Structure Utilities

```racket
(define write-proc
  (make-constructor-style-printer
    (lambda (obj) 'point)
    (lambda (obj) (list (point-x obj) (point-y obj))))))
```

Examples:

```racket
> (struct point (x y)
  #:methods gen:custom-write
  [(define write-proc
    (make-constructor-style-printer
      (lambda (obj) 'point)
      (lambda (obj) (list (point-x obj) (point-y obj))))))]
> (print (point 1 2))
(point 1 2)
> (write (point 1 2))
#:point: 1 2>
```

The function also cooperates with `pretty-print`:

```racket
> (parameterize ((pretty-print-columns 10))
  (pretty-print (point 3000000 4000000)))
(point
  3000000
  4000000)
> (parameterize ((pretty-print-columns 10))
  (pretty-write (point 3000000 4000000)))
```
Note that the printer uses a separate property, prop:custom-print-quotable, to determine whether a struct instance is quotable. If so, the printer may print it in write mode in certain contexts, such as within a list. For example:

```
> (print (list (point 1 2) (point 3 4)))
'(#:point: 1 2) #:point: 3 4)
```

Use #:property prop:custom-print-quotable 'never to prevent a struct instance from being considered quotable. For example:

```
> (struct point2 (x y)
  #:property prop:custom-print-quotable 'never
  #:methods gen:custom-write
  [(define write-proc
    (make-constructor-style-printer
      (lambda (obj) 'point)
      (lambda (obj) (list (point2-x obj) (point2-y obj))))))]
> (print (list (point2 1 2) (point2 3 4)))
(list (point 1 2) (point 3 4))
```

Keyword arguments can be simulated with unquoted-printing-string:

```
(define write-proc
  (make-constructor-style-printer
    (lambda (obj) 'kwpoint)
    (lambda (obj) (list (unquoted-printing-string "#:x")
                        (kwpoint-impl-x obj)
                        (unquoted-printing-string "#:y")
                        (kwpoint-impl-y obj)))))))
```

Private implementation

```
> (struct kwpoint-impl (x y)
  #:methods gen:custom-write
  [(define write-proc
    (make-constructor-style-printer
      (lambda (obj) 'kwpoint)
      (lambda (obj)
        (list (unquoted-printing-string "#:x")
              (kwpoint-impl-x obj)
              (unquoted-printing-string "#:y")
              (kwpoint-impl-y obj))))))]
```

Public `\``constructor``'

```
(define (kwpoint #:x x #:y y)
  (kwpoint-impl x y))
```

Example use

```
> (print (kwpoint #:x 1 #:y 2))
(kwpoint #:x 1 #:y 2)
> (write (kwpoint #:x 3 #:y 4))
#<kwpoint: #:x 3 #:y 4>
```

524
Added in version 6.3 of package base.

```
(struct->list v [#:on-opaque on-opaque]) → (or/c list? #f)
```

\[ v : \text{any/c} \]
\[ \text{on-opaque} : (\text{or/c \ 'error \ 'return-false \ 'skip}) = \text{\ 'error} \]

Returns a list containing the struct instance \( v \)'s fields. Unlike \( \text{struct->vector} \), the struct name itself is not included.

If any fields of \( v \) are inaccessible via the current inspector the behavior of \( \text{struct->list} \) is determined by \( \text{on-opaque} \). If \( \text{on-opaque} \) is \'error\ (the default), an error is raised. If it is \'return-false, \( \text{struct->list} \) returns \#f. If it is \'skip, the inaccessible fields are omitted from the list.

Examples:

\[
> (\text{struct open (u v) #:transparent})
\]
\[
> (\text{struct->list (open \ 'a \ 'b)})
\]
\[
'(a b)
\]
\[
> (\text{struct->list s(pre 1 2 3)})
\]
\[
'(1 2 3)
\]
\[
> (\text{struct secret open (x y)})
\]
\[
> (\text{struct->list (secret 0 1 17 22)})
\]
\[
\text{struct->list: expected argument of type <non-opaque struct>;
given: (secret 0 1 ...)}
\]
\[
> (\text{struct->list (secret 0 1 17 22) #:on-opaque \ 'return-false})
\]
\[
#f
\]
\[
> (\text{struct->list (secret 0 1 17 22) #:on-opaque \ 'skip})
\]
\[
'(0 1)
\]
\[
> (\text{struct->list \ 'not-a-struct #:on-opaque \ 'return-false})
\]
\[
#f
\]
\[
> (\text{struct->list \ 'not-a-struct #:on-opaque \ 'skip})
\]
\[
'()\]

Added in version 6.3 of package base.

## 5.7 Structure Type Transformer Binding

The \text{struct} form binds the name of a structure type as a transformer binding that records the other identifiers bound to the structure type, the constructor procedure, the predicate procedure, and the field accessor and mutator procedures. This information can be used during the expansion of other expressions via \text{syntax-local-value}.

For example, the \text{struct} variant for subtypes uses the base type name \( t \) to find the variable \text{struct:}\( t \) containing the base type’s descriptor; it also folds the field accessor and mutator
information for the base type into the information for the subtype. As another example, the match form uses a type name to find the predicates and field accessors for the structure type. The struct form in an imported signature for unit causes the unit transformer to generate information about imported structure types, so that match and subtyping struct forms work within the unit.

The expansion-time information for a structure type can be represented directly as a list of six elements (of the same sort that the encapsulated procedure must return):

- an identifier that is bound to the structure type’s descriptor, or #f if none is known;
- an identifier that is bound to the structure type’s constructor, or #f if none is known;
- an identifier that is bound to the structure type’s predicate, or #f if none is known;
- a list of identifiers bound to the field accessors of the structure type, optionally with #f as the list’s last element. A #f as the last element indicates that the structure type may have additional fields, otherwise the list is a reliable indicator of the number of fields in the structure type. Furthermore, the accessors are listed in reverse order for the corresponding constructor arguments. (The reverse order enables sharing in the lists for a subtype and its base type.)
- a list of identifiers bound to the field mutators of the structure type, or #f for each field that has no known mutator, and optionally with an extra #f as the list’s last element (if the accessor list has such a #f). The list’s order and the meaning of a final #f are the same as for the accessor identifiers, and the length of the mutator list is the same as the accessor list’s length.
- an identifier that determines a super-type for the structure type, #f if the super-type (if any) is unknown, or #t if there is no super-type. If a super-type is specified, the identifier is also bound to structure-type expansion-time information.

Instead of this direct representation, the representation can be a structure created by make-struct-info (or an instance of a subtype of struct:struct-info), which encapsulates a procedure that takes no arguments and returns a list of six elements. Alternately, the representation can be a structure whose type has the prop:struct-info structure type property. Finally, the representation can be an instance of a structure type derived from struct:struct-info or with the prop:struct-info property that also implements prop:procedure, and where the instance is further is wrapped by make-set!-transformer. In addition, the representation may implement the prop:struct-auto-info and prop:struct-field-info properties.

Use struct-info? to recognize all allowed forms of the information, and use extract-struct-info to obtain a list from any representation.

The implementor of a syntactic form can expect users of the form to know what kind of information is available about a structure type. For example, the match implementation
works with structure information containing an incomplete set of accessor bindings, because
the user is assumed to know what information is available in the context of the match ex-
pression. In particular, the match expression can appear in a unit form with an imported
structure type, in which case the user is expected to know the set of fields that are listed in
the signature for the structure type.

```
(require racket/struct-info)  package: base
```

The bindings documented in this section are provided by the `racket/struct-info` library,
not `racket/base` or `racket`.

```
(struct-info? v) ➔ boolean?
  v : any/c
```

Returns `#t` if `v` is either a six-element list with the correct shape for representing structure-
type information, a procedure encapsulated by `make-struct-info`, a structure with the
`prop:struct-info` property, or a structure type derived from `struct:struct-info` or
with `prop:struct-info` and wrapped with `make-set!-transformer`.

```
(checked-struct-info? v) ➔ boolean?
  v : any/c
```

Returns `#t` if `v` is a procedure encapsulated by `make-struct-info` and produced by
`struct`, but only when no parent type is specified or the parent type is also specified through
a transformer binding to such a value.

```
(make-struct-info thunk) ➔ struct-info?
  thunk : (→ (and/c struct-info? list?))
```

Encapsulates a thunk that returns structure-type information in list form. Note that accessors
are listed in reverse order, as mentioned in §5.7 "Structure Type Transformer Binding". Note
that the field names are not well-defined for struct-type informations that are created with this
method, so it is likely not going to work well with forms like `struct-copy` and `struct*`.

Examples:

```
> (define (new-pair? x) (displayln "new pair?" ) (pair? x))
> (define (new-car x) (displayln "new car" ) (car x))
> (define (new-cdr x) (displayln "new cdr" ) (cdr x))
> (define-syntax new-list
      (make-struct-info
       (λ () (list #f
         #!cons
         #!new-pair?
        (list #!new-cdr #!new-car)
        (list #f #f)
        #t)))
```

527
> (match (list 1 2 3)
    [(new-list hd tl) (append tl (list hd))])
new pair?
new car
new cdr
'(2 3 1)

Examples:
> (struct A (x y))
> (define (new-A-x a) (displayln "A-x") (A-x a))
> (define (new-A-y a) (displayln "A-y") (A-y a))
> (define (new-A? a) (displayln "A?") (A? a))
> (define-syntax A-info
    (make-struct-info
      (λ () (list #'A
        #'A
        #'new-A?
        (list #'new-A-y #'new-A-x)
        (list #f #f)
        #t))))
> (define-match-expander B
    (syntax-rules () [(_ x ...) (A-info x ...)]))
> (match (A 10 20)
    [(B x y) (list y x)])
A?
A-x
A-y
'(20 10)

(extract-struct-info v) → (and/c struct-info? list?)
v : struct-info?

Extracts the list form of the structure type information represented by v.

struct:struct-info : struct-type?
The structure type descriptor for the structure type returned by make-struct-info. This structure type descriptor is mostly useful for creating structure subtypes. The structure type includes a guard that checks an instance’s first field in the same way as make-struct-info.

prop:struct-info : struct-type-property?
The structure type property for creating new structure types like struct:struct-info. The property value must be a procedure of one argument that takes an instance structure and returns structure-type information in list form.
prop:struct-auto-info : struct-type-property?
(struct-auto-info? v) → boolean?
  v : any/c
  (struct-auto-info-lists sai)
  → (list/c (listof identifier?) (listof identifier?))
  sai : struct-auto-info?

The prop:struct-auto-info property is implemented to provide static information about which of the accessor and mutator identifiers for a structure type correspond to #:auto fields (so that they have no corresponding argument in the constructor). The property value must be a procedure that accepts an instance structure to which the property is given, and the result must be two lists of identifiers suitable as a result from struct-auto-info-lists.

The struct-auto-info? predicate recognizes values that implement the prop:struct-auto-info property.

The struct-auto-info-lists function extracts two lists of identifiers from a value that implements the prop:struct-auto-info property. The first list should be a subset of the accessor identifiers for the structure type described by sai, and the second list should be a subset of the mutator identifiers. The two subsets correspond to #:auto fields.

prop:struct-field-info : struct-type-property?
(struct-field-info? v) → boolean?
  v : any/c
  (struct-field-info-list sfi) → (listof symbol?)
  sfi : struct-field-info?

The prop:struct-field-info property is implemented to provide static information about field names in a structure type. The property value must be a procedure that accepts an instance structure to which the property is given, and the result must be a list of symbols suitable as a result from struct-field-info-list.

The struct-field-info? predicate recognizes values that implement the prop:struct-field-info property.

The struct-field-info-list function extracts a list of symbols from a value that implements the prop:struct-field-info property. The list should contain every immediate field name (that is, not including fields from its super struct type) in the reverse order.

Examples:

> (struct foo (x))
> (struct bar foo (y z))
> (define-syntax (get-bar-field-names stx)
  #'(,(struct-field-info-list (syntax-local-value #'bar))))

529
> (get-bar-field-names)
'(z y)

Added in version 7.7.0.9 of package base.
6 Classes and Objects

(require racket/class) package: base

The bindings documented in this section are provided by the racket/class and racket libraries, but not racket/base.

A class specifies

- a collection of fields;
- a collection of methods;
- initial value expressions for the fields; and
- initialization variables that are bound to initialization arguments.

In the context of the class system, an object is a collection of bindings for fields that are instantiated according to a class description.

The class system allows a program to define a new class (a derived class) in terms of an existing class (the superclass) using inheritance, overriding, and augmenting:

- inheritance: An object of a derived class supports methods and instantiates fields declared by the derived class’s superclass, as well as methods and fields declared in the derived class expression.
- overriding: Some methods declared in a superclass can be replaced in the derived class. References to the overridden method in the superclass use the implementation in the derived class.
- augmenting: Some methods declared in a superclass can be merely extended in the derived class. The superclass method specifically delegates to the augmenting method in the derived class.

An interface is a collection of method names to be implemented by a class, combined with a derivation requirement. A class implements an interface when it

- declares (or inherits) a public method for each variable in the interface;
- is derived from the class required by the interface, if any; and
- specifically declares its intention to implement the interface.
A class can implement any number of interfaces. A derived class automatically implements any interface that its superclass implements. Each class also implements an implicitly-defined interface that is associated with the class. The implicitly-defined interface contains all of the class’s public method names, and it requires that all other implementations of the interface are derived from the class.

A new interface can extend one or more interfaces with additional method names; each class that implements the extended interface also implements the original interfaces. The derivation requirements of the original interface must be consistent, and the extended interface inherits the most specific derivation requirement from the original interfaces.

Classes, objects, and interfaces are all values. However, a class or interface is not an object (i.e., there are no “meta-classes” or “meta-interfaces”).

### 6.1 Creating Interfaces

```racket
(interface (super-interface-expr ...) name-clause ...)
name-clause = id | (id contract-expr)
```

Produces an interface. The `ids` must be mutually distinct.

Each `super-interface-expr` is evaluated (in order) when the `interface` expression is evaluated. The result of each `super-interface-expr` must be an interface value, otherwise the `exn:fail:object` exception is raised. The interfaces returned by the `super-interface-exps` are the new interface’s superinterfaces, which are all extended by the new interface. Any class that implements the new interface also implements all of the superinterfaces.

The result of an `interface` expression is an interface that includes all of the specified `ids`, plus all identifiers from the superinterfaces. Duplicate identifier names among the superinterfaces are ignored, but if a superinterface contains one of the `ids` in the `interface` expression, the `exn:fail:object` exception is raised. A given `id` may be paired with a corresponding `contract-expr`.

If no `super-interface-exps` are provided, then the derivation requirement of the resulting interface is trivial: any class that implements the interface must be derived from `object%`. Otherwise, the implementation requirement of the resulting interface is the most specific requirement from its superinterfaces. If the superinterfaces specify inconsistent derivation requirements, the `exn:fail:object` exception is raised.

Examples:

```racket
(define file-interface<%>
  (interface () open close read-byte write-byte))
```
(define directory-interface<%>
  (interface (file-interface<%>)
    [file-list (-m (listof (is-a?/c file-interface<%>)))
     parent-directory])

(interface* (super-interface-expr ...)
  ([property-expr val-expr] ...)
  name-clause ...)

name-clause = id
  | (id contract-expr)

Like interface, but also associates to the interface the structure-type properties produced by the property-exprs with the corresponding val-exprs.

Whenever the resulting interface (or a sub-interface derived from it) is explicitly implemented by a class through the class* form, each property is attached with its value to a structure type that instantiated by instances of the class. Specifically, the property is attached to a structure type with zero immediate fields, which is extended to produce the internal structure type for instances of the class (so that no information about fields is accessible to the structure type property’s guard, if any).

Example:

(define i<%> (interface* () ([prop:custom-write
  (lambda (obj port mode) (void))]
    method1 method2 method3))

6.2 Creating Classes

object% : class?

A built-in class that has no methods fields, implements only its own interface (class->interface object%), and is transparent (i.e., its inspector is #f, so all immediate instances are equal?). All other classes are derived from object%.

(class* superclass-expr (interface-expr ...)
  class-clause
  ...)

§13 “Classes and Objects” in The Racket Guide introduces classes and objects.
class-clause = (inspect inspector-expr)
        | (init init-decl ...)
        | (init-field init-decl ...)
        | (field field-decl ...)
        | (inherit-field maybe-renamed ...)
        | (init-rest id)
        | (init-rest)
        | (public maybe-renamed ...)
        | (pubment maybe-renamed ...)
        | (public-final maybe-renamed ...)
        | (override maybe-renamed ...)
        | (overment maybe-renamed ...)
        | (override-final maybe-renamed ...)
        | (augment maybe-renamed ...)
        | (augride maybe-renamed ...)
        | (augment-final maybe-renamed ...)
        | (private id ...)
        | (abstract id ...)
        | (inherit maybe-renamed ...)
        | (inherit/super maybe-renamed ...)
        | (inherit/inner maybe-renamed ...)
        | (rename-super renamed ...)
        | (rename-inner renamed ...)
method-definition
  definition
  expr
        | (begin class-clause ...)

init-decl = id
        | (renamed)
        | (maybe-renamed default-value-expr)

field-decl = (maybe-renamed default-value-expr)

maybe-renamed = id
        | renamed

renamed = (internal-id external-id)

method-definition = (define-values (id) method-procedure)

method-procedure = (lambda kw-formals expr ...+)
        | (case-lambda (formals expr ...+) ...)
        | (#%plain-lambda formals expr ...+)
        | (let-values (((id) method-procedure] ...) method-procedure)
        | (letrec-values (((id) method-procedure] ...) method-procedure)
        | (let-values (((id) method-procedure] ...+)
                   id)
        | (letrec-values (((id) method-procedure] ...+)
                   id)
        | (chaperone-procedure method-procedure wrapper-arg other-arg-expr ...)

$\text{class-clause} = (\text{inspect inspector-expr})$
| (\text{init init-decl ...})
| (\text{init-field init-decl ...})
| (\text{field field-decl ...})
| (\text{inherit-field maybe-renamed ...})
| (\text{init-rest id})
| (\text{init-rest})
| (\text{public maybe-renamed ...})
| (\text{pubment maybe-renamed ...})
| (\text{public-final maybe-renamed ...})
| (\text{override maybe-renamed ...})
| (\text{overment maybe-renamed ...})
| (\text{override-final maybe-renamed ...})
| (\text{augment maybe-renamed ...})
| (\text{augride maybe-renamed ...})
| (\text{augment-final maybe-renamed ...})
| (\text{private id ...})
| (\text{abstract id ...})
| (\text{inherit maybe-renamed ...})
| (\text{inherit/super maybe-renamed ...})
| (\text{rename-super renamed ...})
| (\text{rename-inner renamed ...})

$\text{method-definition}$
\text{definition}
\text{expr}
| (begin \text{class-clause ...})

$\text{init-decl} = \text{id}$
| (\text{renamed})
| (\text{maybe-renamed default-value-expr})

$\text{field-decl} = (\text{maybe-renamed default-value-expr})$

$\text{maybe-renamed} = \text{id}$
| \text{renamed}

$\text{renamed} = (\text{internal-id external-id})$

$\text{method-definition} = (\text{define-values (id) method-procedure})$

$\text{method-procedure} = (\text{lambda kw-formals expr ...+})$
| (\text{case-lambda (formals expr ...+) ...})
| (\text{#%plain-lambda formals expr ...+})
| (\text{let-values (((id) method-procedure] ...) method-procedure})
| (\text{letrec-values (((id) method-procedure] ...) method-procedure})
| (\text{let-values (((id) method-procedure] ...+) id})
| (\text{letrec-values (((id) method-procedure] ...+) id})
| (\text{chaperone-procedure method-procedure wrapper-arg other-arg-expr ...})$
Produces a class value.

The superclass-expr expression is evaluated when the class* expression is evaluated. The result must be a class value (possibly object%), otherwise the exn:fail:object exception is raised. The result of the superclass-expr expression is the new class’s superclass.

The interface-expr expressions are also evaluated when the class* expression is evaluated, after superclass-expr is evaluated. The result of each interface-expr must be an interface value, otherwise the exn:fail:object exception is raised. The interfaces returned by the interface-exprs are all implemented by the class. For each identifier in each interface, the class (or one of its ancestors) must declare a public method with the same name, otherwise the exn:fail:object exception is raised. The class’s superclass must satisfy the implementation requirement of each interface, otherwise the exn:fail:object exception is raised.

An inspect class-clause selects an inspector (see §14.9 “Structure Inspectors”) for the class extension. The inspector-expr must evaluate to an inspector or #f when the class* form is evaluated. Just as for structure types, an inspector controls access to the class’s fields, including private fields, and also affects comparisons using equal?. If no inspect clause is provided, access to the class is controlled by the parent of the current inspector (see §14.9 “Structure Inspectors”). A syntax error is reported if more than one inspect clause is specified.

The other class-clauses define initialization arguments, public and private fields, and public and private methods. For each id or maybe-renamed in a public, override, augment, pubment, overment, augride, public-final, override-final, augment-final, or private clause, there must be one method-definition. All other definition class-clauses create private fields. All remaining exprs are initialization expressions to be evaluated when the class is instantiated (see §6.3 “Creating Objects”).

The result of a class* expression is a new class, derived from the specified superclass and implementing the specified interfaces. Instances of the class are created with the instantiate form or make-object procedure, as described in §6.3 “Creating Objects”.

Each class-clause is (partially) macro-expanded to reveal its shapes. If a class-clause is a begin expression, its sub-expressions are lifted out of the begin and treated as class-clauses, in the same way that begin is flattened for top-level and embedded definitions.

Within a class* form for instances of the new class, this is bound to the object itself; this% is bound to the class of the object; super-instantiate, super-make-object, and super-new are bound to forms to initialize fields in the superclass (see §6.3 “Creating Objects”); super is available for calling superclass methods (see §6.2.3.1 “Method Definitions”), and inner is available for calling subclass augmentations of methods (see §6.2.3.1 “Method Definitions”).

(class superclass-expr class-clause ...)
Like `class*`, but omits the `interface-expr`s, for the case that none are needed.

Example:

```scheme
(define book-class%
  (class object%
    (field (pages 5))
    (define/public (letters)
      (* pages 500))
    (super-new)))
```

Within a `class*` form, this refers to the current object (i.e., the object being initialized or whose method was called). Use outside the body of a `class*` form is a syntax error.

Examples:

```scheme
(define (describe obj)
  (printf "Hello ~a\n" obj))
(define table%
  (class object%
    (define/public (describe-self)
      (describe this))
    (super-new)))
> (send (new table%) describe-self)
Hello #(struct:object:table% ...)
```

Within a `class*` form, this% refers to the class of the current object (i.e., the object being initialized or whose method was called). Use outside the body of a `class*` form is a syntax error.

Examples:

```scheme
(define account%
  (class object%
    (super-new)
    (init-field balance)
    (define/public (add n)
      (new this% [balance (+ n balance)]))
  (define savings%
    (class account%
      ...)
(super-new)
(inherit-field balance)
(define interest 0.04)
(define/public (add-interest)
  (send this add (* interest balance))))

> (let* ([acct (new savings% [balance 500]])
        [acct (send acct add 500)]
        [acct (send acct add-interest)])
  (printf "Current balance: ~a\n" (get-field balance acct)))
Current balance: 1040.0

See class*: use outside the body of a class* form is a syntax error.

(init init-decl ...)

See class* and §6.2.1 “Initialization Variables” use outside the body of a class* form is a syntax error.

Example:

> (class object%
  (super-new)
  (init turnip
    [(internal-potato potato)]
    [carrot 'good]
    [(internal-rutabaga rutabaga) 'okay]))
#<class:eval:10:0>

(init-field init-decl ...)

See class*, §6.2.1 “Initialization Variables” and §6.2.2 “Fields” use outside the body of a class* form is a syntax error.

Example:

> (class object%
  (super-new)
  (init-field turkey
    [(internal-ostrich ostrich)]
    [chicken 7]
    [(internal-emu emu) 13]))
#<class:eval:11:0>
(field field-\texttt{decl} ...)

See class* and §6.2.2 "Fields" use outside the body of a class* form is a syntax error.

Example:

\begin{verbatim}
> (class object%
  (super-new)
  (field [minestrone 'ready]
    [(internal-coq-au-vin coq-au-vin) 'stewing]))
#<class:eval:12:0>
\end{verbatim}

(inherit-field maybe-renamed ...)

See class* and §6.2.2 "Fields" use outside the body of a class* form is a syntax error.

Examples:

\begin{verbatim}
(define cookbook%
  (class object%
    (super-new)
    (field [recipes '(caldo-verde oyakodon eggs-benedict)]
      [pages 389])))

> (class cookbook%
  (super-new)
  (inherit-field recipes
    [internal-pages pages]))
#<class:eval:14:0>
\end{verbatim}

(init-rest id)

See class* and §6.2.1 "Initialization Variables" use outside the body of a class* form is a syntax error.

Examples:

\begin{verbatim}
(define fruit-basket%
  (class object%
    (super-new)
    (init-rest fruits)
\end{verbatim}
(displayln fruits)))

> (make-object fruit-basket% 'kiwi 'lychee 'melon)
  (kiwi lychee melon)
  (object:fruit-basket% ...)

(public maybe-renamed ...

See class* and §6.2.3.1 “Method Definitions”; use outside the body of a class* form is a syntax error.

Examples:

(define jumper%
  (class object%
    (super-new)
    (define (skip) 'skip)
    (define (hop) 'hop)
    (public skip [hop jump])))

> (send (new jumper%) skip)
  'skip
> (send (new jumper%) jump)
  'hop

(pubment maybe-renamed ...

See class* and §6.2.3.1 “Method Definitions”; use outside the body of a class* form is a syntax error.

Examples:

(define runner%
  (class object%
    (super-new)
    (define (run) 'run)
    (define (trot) 'trot)
    (pubment run [trot jog])))

> (send (new runner%) run)
  'run
> (send (new runner%) jog)
  'trot

539
(public-final maybe-renamed ...)

See class* and §6.2.3.1 “Method Definitions” use outside the body of a class* form is a syntax error.

Examples:

(define point%
  (class object%
    (super-new)
    (init-field [x 0] [y 0])
    (define (get-x) x)
    (define (do-get-y) y)
    (public-final get-x [do-get-y get-y])))

> (send (new point% [x 1] [y 3]) get-y)
3
> (class point%
    (super-new)
    (define (get-x) 3.14)
    (override get-x))
class*: cannot override or augment final method
  method name: get-x
  class name: eval:25:0

(override maybe-renamed ...)

See class* and §6.2.3.1 “Method Definitions” use outside the body of a class* form is a syntax error.

Examples:

(define sheep%
  (class object%
    (super-new)
    (define/public (bleat)
      (displayln "baaaaaaaaah"))))

(define confused-sheep%
  (class sheep%
    (super-new)
    (define (bleat)
      (super bleat)
      (displayln "???"))
    (override bleat)))
> (send (new sheep%) bleat)
  baaaaaaaaah
> (send (new confused-sheep%) bleat)
  baaaaaaaaah
  ???

(overment maybe-renamed ...)

See class* and §6.2.3.1 “Method Definitions” use outside the body of a class* form is a syntax error.

Examples:

```
(define turkey%
  (class object%
    (super-new)
    (define/public (gobble)
      (displayln "gobble gobble")))))

(define extra-turkey%
  (class turkey%
    (super-new)
    (define (gobble)
      (super gobble)
      (displayln "gobble gobble gobble")
      (inner (void) gobble)
      (overment gobble)))

(define cyborg-turkey%
  (class extra-turkey%
    (super-new)
    (define/augment (gobble)
      (displayln "110011111111110001011011001100101"))))
```

> (send (new extra-turkey%) gobble)
gobble gobble
gobble gobble gobble
gobble gobble gobble
gobble gobble
gobble gobble gobble
gobble gobble gobble
gobble gobble
gobble gobble
gobble
gobble
11001111101111100010110011011001100101

(override-final maybe-renamed ...)
See class* and §6.2.3.1 “Method Definitions”; use outside the body of a class* form is a syntax error.

Examples:

```scheme
(define meeper%
  (class object%
    (super-new)
    (define/public (meep)
      (displayln "meep")))))

(define final-meeper%
  (class meeper%
    (super-new)
    (define (meep)
      (super meep)
      (displayln "This meeping ends with me"))
    (override-final meep))

> (send (new meeper%) meep)
meep
> (send (new final-meeper%) meep)
meep
This meeping ends with me

;; (augment maybe-renamed ...)```

See class* and §6.2.3.1 “Method Definitions”; use outside the body of a class* form is a syntax error.

Examples:

```scheme
(define buzzer%
  (class object%
    (super-new)
    (define/pubment (buzz)
      (displayln "bzzzt")
      (inner (void) buzz)))

(define loud-buzzer%
  (class buzzer%
    (super-new)
    (define (buzz)
      (displayln "BZZZZZZZZZT")
    (augment buzz)))
```

542
> (send (new buzzer%) buzz)
bzzzt
> (send (new loud-buzzer%) buzz)
bzzzt
 BZZZZZZZZZT

\( \text{(augride } \text{maybe-renamed } \ldots) \)\n
See class* and §6.2.3.1 “Method Definitions”, use outside the body of a class* form is a syntax error.

\( \text{(augment-final } \text{maybe-renamed } \ldots) \)\n
See class* and §6.2.3.1 “Method Definitions”, use outside the body of a class* form is a syntax error.

\( \text{(private } \text{id } \ldots) \)\n
See class* and §6.2.3.1 “Method Definitions”, use outside the body of a class* form is a syntax error.

Examples:

\( \text{(define light\%} \)
\( \text{class object\%} \)
\( \text{(super-new)} \)
\( \text{(define on? #t)} \)
\( \text{(define (toggle) (set! on? (not on?)) )} \)
\( \text{(private toggle)} \)
\( \text{(define (flick) (toggle))} \)
\( \text{(public flick))} \)

> (send (new light\%) toggle)
send: no such method
method name: toggle
class name: light\%
> (send (new light\%) flick)
See \texttt{class*} and \textbf{§6.2.3.1 “Method Definitions”}; use outside the body of a \texttt{class*} form is a syntax error.

Examples:

\begin{verbatim}
(define train%
  (class object%
    (super-new)
    (abstract get-speed)
    (init-field [position 0])
    (define/public (move)
      (new this% [position (+ position (get-speed))])))

(define acela%
  (class train%
    (super-new)
    (define/override (get-speed) 241)))

(define talgo-350%
  (class train%
    (super-new)
    (define/override (get-speed) 330)))

> (new train%)
instantiate: cannot instantiate class with abstract methods
  class: #<class:train%>
  abstract methods:
    get-speed
> (send (new acela%) move)
(object:acela% ...)

(inherit maybe-renamed ...)
\end{verbatim}

See \texttt{class*} and \textbf{§6.2.3.2 “Inherited and Superclass Methods”}; use outside the body of a \texttt{class*} form is a syntax error.

Examples:

\begin{verbatim}
(define alarm%
  (class object%
    (super-new)
    (define/public (alarm)
      (displayln "beeeeeeeep")))
\end{verbatim}
(define car-alarm%
  (class alarm%
    (super-new)
    (init-field proximity)
    (inherit alarm)
    (when (< proximity 10)
      (alarm)))
>
  (new car-alarm% [proximity 5])
  beeeeeeeep
  (object:car-alarm% ...)

(inherit/super maybe-renamed ...)

See class* and §6.2.3.2 "Inherited and Superclass Methods" use outside the body of a class* form is a syntax error.

(inherit/inner maybe-renamed ...)

See class* and §6.2.3.2 "Inherited and Superclass Methods" use outside the body of a class* form is a syntax error.

(rename-super renamed ...)

See class* and §6.2.3.2 "Inherited and Superclass Methods" use outside the body of a class* form is a syntax error.

(rename-inner renamed ...)

See class* and §6.2.3.2 "Inherited and Superclass Methods" use outside the body of a class* form is a syntax error.

(public* (id expr) ...)

Shorthand for (begin (public id) ... (define id expr) ...).

(pubment* (id expr) ...)

Shorthand for (begin (pubment id) ... (define id expr) ...).
Shorthand for (begin (public-final id) ... (define id expr) ...).

Shorthand for (begin (override id) ... (define id expr) ...).

Shorthand for (begin (overment id) ... (define id expr) ...).

Shorthand for (begin (override-final id) ... (define id expr) ...).

Shorthand for (begin (augment id) ... (define id expr) ...).

Shorthand for (begin (augride id) ... (define id expr) ...).

Shorthand for (begin (augment-final id) ... (define id expr) ...).

Shorthand for (begin (private id) ... (define id expr) ...).

Shorthand for (define/public id expr)
(define/public (id . formals) body ...+)}

Shorthand for (begin (public id) (define id expr)) or (begin (public id) (define (id . formals) body ...+))
(define/pubment id expr)
(define/pubment (id . formals) body ...+)

Shorthand for (begin (pubment id) (define id expr)) or (begin (pubment id) (define (id . formals) body ...+))

(define/public-final id expr)
(define/public-final (id . formals) body ...+)

Shorthand for (begin (public-final id) (define id expr)) or (begin (public-final id) (define (id . formals) body ...+))

(define/override id expr)
(define/override (id . formals) body ...+)

Shorthand for (begin (override id) (define id expr)) or (begin (override id) (define (id . formals) body ...+))

(define/overment id expr)
(define/overment (id . formals) body ...+)

Shorthand for (begin (overment id) (define id expr)) or (begin (overment id) (define (id . formals) body ...+))

(define/override-final id expr)
(define/override-final (id . formals) body ...+)

Shorthand for (begin (override-final id) (define id expr)) or (begin (override-final id) (define (id . formals) body ...+))

(define/augment id expr)
(define/augment (id . formals) body ...+)

Shorthand for (begin (augment id) (define id expr)) or (begin (augment id) (define (id . formals) body ...+))

(define/augride id expr)
(define/augride (id . formals) body ...+)
Shorthand for `(begin (augride id) (define id expr))` or `(begin (augride id) (define (id . formals) body ...))`

```scheme
(define/augment-final id expr)
(define/augment-final (id . formals) body ...+)
```

Shorthand for `(begin (augment-final id) (define id expr))` or `(begin (augment-final id) (define (id . formals) body ...))`

```scheme
(define/private id expr)
(define/private (id . formals) body ...+)
```

Shorthand for `(begin (private id) (define id expr))` or `(begin (private id) (define (id . formals) body ...))`

```scheme
(class/derived original-datum
  (name-id super-expr (interface-expr ...) deserialize-id-expr)
  class-clause
  ...)
```

Like class*, but includes a sub-expression to be used as the source for all syntax errors within the class definition. For example, define-serializable-class expands to class/derived so that errors in the body of the class are reported in terms of define-serializable-class instead of class.

The original-datum is the original expression to use for reporting errors.

The name-id is used to name the resulting class; if it is #f, the class name is inferred.

The super-expr, interface-exprs, and class-clauses are as for class*.

If the deserialize-id-expr is not literally #f, then a serializable class is generated, and the result is two values instead of one: the class and a deserialize-info structure produced by make-deserialize-info. The deserialize-id-expr should produce a value suitable as the second argument to make-serialize-info, and it should refer to an export whose value is the deserialize-info structure.

Future optional forms may be added to the sequence that currently ends with deserialize-id-expr.

### 6.2.1 Initialization Variables

A class’s initialization variables, declared with init, init-field, and init-rest, are instantiated for each object of a class. Initialization variables can be used in the initial
value expressions of fields, default value expressions for initialization arguments, and in initialization expressions. Only initialization variables declared with init-field can be accessed from methods; accessing any other initialization variable from a method is a syntax error.

The values bound to initialization variables are

- the arguments provided with instantiate or passed to make-object, if the object is created as a direct instance of the class; or,
- the arguments passed to the superclass initialization form or procedure, if the object is created as an instance of a derived class.

If an initialization argument is not provided for an initialization variable that has an associated default-value-expr, then the default-value-expr expression is evaluated to obtain a value for the variable. A default-value-expr is only evaluated when an argument is not provided for its variable. The environment of default-value-expr includes all of the initialization variables, all of the fields, and all of the methods of the class. If multiple default-value-exps are evaluated, they are evaluated from left to right. Object creation and field initialization are described in detail in §6.3 “Creating Objects”.

If an initialization variable has no default-value-expr, then the object creation or superclass initialization call must supply an argument for the variable, otherwise the exn:fail:object exception is raised.

Initialization arguments can be provided by name or by position. The external name of an initialization variable can be used with instantiate or with the superclass initialization form. Those forms also accept by-position arguments. The make-object procedure and the superclass initialization procedure accept only by-position arguments.

Arguments provided by position are converted into by-name arguments using the order of init and init-field clauses and the order of variables within each clause. When an instantiate form provides both by-position and by-name arguments, the converted arguments are placed before by-name arguments. (The order can be significant; see also §6.3 “Creating Objects”).

Unless a class contains an init-rest clause, when the number of by-position arguments exceeds the number of declared initialization variables, the order of variables in the superclass (and so on, up the superclass chain) determines the by-name conversion.

If a class expression contains an init-rest clause, there must be only one, and it must be last. If it declares a variable, then the variable receives extra by-position initialization arguments as a list (similar to a dotted “rest argument” in a procedure). An init-rest variable can receive by-position initialization arguments that are left over from a by-name conversion for a derived class. When a derived class’s superclass initialization provides even more by-position arguments, they are prefixed onto the by-position arguments accumulated so far.
If too few or too many by-position initialization arguments are provided to an object creation or superclass initialization, then the `exn:fail:object` exception is raised. Similarly, if extra by-position arguments are provided to a class with an `init-rest` clause, the `exn:fail:object` exception is raised.

Unused (by-name) arguments are to be propagated to the superclass, as described in §6.3 “Creating Objects”. Multiple initialization arguments can use the same name if the class derivation contains multiple declarations (in different classes) of initialization variables with the same name. See §6.3 “Creating Objects” for further details.

See also §6.2.3.3 “Internal and External Names” for information about internal and external names.

### 6.2.2 Fields

Each `field`, `init-field`, and non-method `define-values` clause in a class declares one or more new fields for the class. Fields declared with `field` or `init-field` are public. Public fields can be accessed and mutated by subclasses using `inherit-field`. Public fields are also accessible outside the class via `class-field-accessor` and mutable via `class-field-mutator` (see §6.4 “Field and Method Access”). Fields declared with `define-values` are accessible only within the class.

A field declared with `init-field` is both a public field and an initialization variable. See §6.2.1 “Initialization Variables” for information about initialization variables.

An `inherit-field` declaration makes a public field defined by a superclass directly accessible in the class expression. If the indicated field is not defined in the superclass, the `exn:fail:object` exception is raised when the class expression is evaluated. Every field in a superclass is present in a derived class, even if it is not declared with `inherit-field` in the derived class. The `inherit-field` clause does not control inheritance, but merely controls lexical scope within a class expression.

When an object is first created, all of its fields have the `#<undefined>` value (see §4.19 “Void”). The fields of a class are initialized at the same time that the class’s initialization expressions are evaluated; see §6.3 “Creating Objects” for more information.

See also §6.2.3.3 “Internal and External Names” for information about internal and external names.

### 6.2.3 Methods

#### Method Definitions

Each public, `override`, `augment`, `pubment`, `overment`, `augride`, `public-final`,
override-final, augment-final, and private clause in a class declares one or more method names. Each method name must have a corresponding method-definition. The order of public, etc., clauses and their corresponding definitions (among themselves, and with respect to other clauses in the class) does not matter.

As shown in the grammar for class*, a method definition is syntactically restricted to certain procedure forms, as defined by the grammar for method-procedure; in the last two forms of method-procedure, the body id must be one of the ids bound by let-values or letrec-values. A method-procedure expression is not evaluated directly. Instead, for each method, a class-specific method procedure is created; it takes an initial object argument, in addition to the arguments the procedure would accept if the method-procedure expression were evaluated directly. The body of the procedure is transformed to access methods and fields through the object argument.

A method declared with public, pubment, or public-final introduces a new method into a class. The method must not be present already in the superclass, otherwise the exn:fail:object exception is raised when the class expression is evaluated. A method declared with public can be overridden in a subclass that uses override, overment, or override-final. A method declared with pubment can be augmented in a subclass that uses augment, augride, or augment-final. A method declared with public-final cannot be overridden or augmented in a subclass.

A method declared with override, overment, or override-final overrides a definition already present in the superclass. If the method is not already present, the exn:fail:object exception is raised when the class expression is evaluated. A method declared with override can be overridden again in a subclass that uses override, overment, or override-final. A method declared with overment can be augmented in a subclass that uses augment, augride, or augment-final. A method declared with override-final cannot be overridden further or augmented in a subclass.

A method declared with augment, augride, or augment-final augments a definition already present in the superclass. If the method is not already present, the exn:fail:object exception is raised when the class expression is evaluated. A method declared with augment can be augmented further in a subclass that uses augment, augride, or augment-final. A method declared with augride can be overridden in a subclass that uses override, overment, or override-final. (Such an override merely replaces the augmentation, not the method that is augmented.) A method declared with augment-final cannot be overridden or augmented further in a subclass.

A method declared with private is not accessible outside the class expression, cannot be overridden, and never overrides a method in the superclass.

When a method is declared with override, overment, or override-final, then the superclass implementation of the method can be called using super form.

When a method is declared with pubment, augment, or overment, then a subclass augmenting method can be called using the inner form. The only difference between public-
final and public without a corresponding inner is that public-final prevents the declaration of augmenting methods that would be ignored.

A method declared with abstract must be declared without an implementation. Subclasses may implement abstract methods via the override, overment, or override-final forms. Any class that contains or inherits any abstract methods is considered abstract and cannot be instantiated.

```
(super id arg ...)
(super id arg ... . arg-list-expr)
```

Always accesses the superclass method, independent of whether the method is overridden again in subclasses. Using the super form outside of class* is a syntax error. Each arg is as for #%app: either arg-expr or keyword arg-expr.

The second form is analogous to using apply with a procedure; the arg-list-expr must not be a parenthesized expression.

```
(inner default-expr id arg ...)
(inner default-expr id arg ... . arg-list-expr)
```

If the object’s class does not supply an augmenting method, then default-expr is evaluated, and the arg expressions are not evaluated. Otherwise, the augmenting method is called with the arg results as arguments, and default-expr is not evaluated. If no inner call is evaluated for a particular method, then augmenting methods supplied by subclasses are never used. Using the inner form outside of class* is a syntax error.

The second form is analogous to using apply with a procedure; the arg-list-expr must not be a parenthesized expression.

**Inherited and Superclass Methods**

Each inherit, inherit/super, inherit/inner, rename-super, and rename-inner clause declares one or more methods that are defined in the class, but must be present in the superclass. The rename-super and rename-inner declarations are rarely used, since inherit/super and inherit/inner provide the same access. Also, superclass and augmenting methods are typically accessed through super and inner in a class that also declares the methods, instead of through inherit/super, inherit/inner, rename-super, or rename-inner.

Method names declared with inherit, inherit/super, or inherit/inner access overriding declarations, if any, at run time. Method names declared with inherit/super can also be used with the super form to access the superclass implementation, and method names declared with inherit/inner can also be used with the inner form to access an augmenting method, if any.

Method names declared with rename-super always access the superclass’s implementation
at run-time. Methods declared with `rename-inner` access a subclass’s augmenting method, if any, and must be called with the form

```
(id (lambda () default-expr) arg ...)
```

so that a `default-expr` is available to evaluate when no augmenting method is available. In such a form, `lambda` is a literal identifier to separate the `default-expr` from the `arg`. When an augmenting method is available, it receives the results of the `arg` expressions as arguments.

Methods that are present in the superclass but not declared with `inherit`, `inherit/super`, or `inherit/inner` or `rename-super` are not directly accessible in the class (though they can be called with `send`). Every public method in a superclass is present in a derived class, even if it is not declared with `inherit` in the derived class; the `inherit` clause does not control inheritance, but merely controls lexical scope within a class expression.

If a method declared with `inherit`, `inherit/super`, `inherit/inner`, `rename-super`, or `rename-inner` is not present in the superclass, the `exn:fail:object` exception is raised when the class expression is evaluated.

**Internal and External Names**

Each method declared with `public`, `override`, `augment`, `pubment`, `overment`, `augment-final`, `public-final`, `override-final`, `augment-final`, `inherit`, `inherit/super`, `inherit/inner`, `rename-super`, and `rename-inner` can have separate internal and external names when `(internal-id external-id)` is used for declaring the method. The internal name is used to access the method directly within the class expression (including within `super` or `inner` forms), while the external name is used with `send` and `generic` (see §6.4 "Field and Method Access"). If a single `id` is provided for a method declaration, the identifier is used for both the internal and external names.

Method inheritance, overriding, and augmentation are based on external names only. Separate internal and external names are required for `rename-super` and `rename-inner` (for historical reasons, mainly).

Each `init`, `init-field`, `field`, or `inherit-field` variable similarly has an internal and an external name. The internal name is used within the class to access the variable, while the external name is used outside the class when providing initialization arguments (e.g., to `instantiate`), inheriting a field, or accessing a field externally (e.g., with `class-field-accessor`). As for methods, when inheriting a field with `inherit-field`, the external name is matched to an external field name in the superclass, while the internal name is bound in the `class` expression.

A single identifier can be used as an internal identifier and an external identifier, and it is possible to use the same identifier as internal and external identifiers for different bindings. Furthermore, within a single class, a single name can be used as an external method name, an external field name, and an external initialization argument name. Overall, each internal

553
identifier must be distinct from all other internal identifiers, each external method name must be distinct from all other method names, each external field name must be distinct from all other field names, and each initialization argument name must be distinct from all other initialization argument names.

By default, external names have no lexical scope, which means, for example, that an external method name matches the same syntactic symbol in all uses of send. The define-local-member-name and define-member-name forms introduce scoped external names.

When a class expression is compiled, identifiers used in place of external names must be symbolically distinct (when the corresponding external names are required to be distinct), otherwise a syntax error is reported. When no external name is bound by define-member-name, then the actual external names are guaranteed to be distinct when class expression is evaluated. When any external name is bound by define-member-name, the exn:fail:object exception is raised by class if the actual external names are not distinct.

(define-local-member-name id . .)

Unless it appears as the top-level definition, binds each id so that, within the scope of the definition, each use of each id as an external name is resolved to a hidden name generated by the define-local-member-name declaration. Thus, methods, fields, and initialization arguments declared with such external-name ids are accessible only in the scope of the define-local-member-name declaration. As a top-level definition, define-local-member-name binds id to its symbolic form.

The binding introduced by define-local-member-name is a syntax binding that can be exported and imported with modules. Each evaluation of a define-local-member-name declaration generates a distinct hidden name (except as a top-level definition). The interface->method-names procedure does not expose hidden names.

Examples:

(define-values (r o)
  (let ()
    (define-local-member-name m)
    (define c% (class object%)
      (define/public (m) 10)
      (super-new)))
    (define o (new c%))
    (values (send o m)
       o)))

> r
10
> (send o m)
send: no such method

554
(define-member-name  id  key-expr)

Maps a single external name to an external name that is determined by an expression. The value of \textit{key-expr} must be the result of either a \texttt{member-name-key} expression or a \texttt{generate-member-key} call.

\texttt{(member-name-key  \texttt{identifier})}

Produces a representation of the external name for \texttt{id} in the environment of the \texttt{member-name-key} expression.

\texttt{(generate-member-key) \rightarrow member-name-key?}

Produces a hidden name, just like the binding for \texttt{define-local-member-name}.

\texttt{(member-name-key?  \texttt{v}) \rightarrow boolean?}

\texttt{v : any/c}

Returns \#t for values produced by \texttt{member-name-key} and \texttt{generate-member-key}, \#f otherwise.

\texttt{(member-name-key=?  \texttt{a-key}  \texttt{b-key}) \rightarrow boolean?}

\texttt{a-key : member-name-key?}

\texttt{b-key : member-name-key?}

Produces \#t if member-name keys \texttt{a-key} and \texttt{b-key} represent the same external name, \#f otherwise.

\texttt{(member-name-key-hash-code  \texttt{a-key}) \rightarrow integer?}

\texttt{a-key : member-name-key?}

Produces an integer hash code consistent with \texttt{member-name-key=?} comparisons, analogous to \texttt{equal-hash-code}.

Examples:

\begin{verbatim}
(define (make-c% key)
   (define-member-name m key)
   (class object%
      (define/public (m) 10)
      (super-new)))
\end{verbatim}
6.3 Creating Objects

The make-object procedure creates a new object with by-position initialization arguments, the new form creates a new object with by-name initialization arguments, and the instantiate form creates a new object with both by-position and by-name initialization arguments.

All fields in the newly created object are initially bound to the special #<undefined> value (see §4.19 “Void”). Initialization variables with default value expressions (and no provided value) are also initialized to #<undefined>. After argument values are assigned to initialization variables, expressions in field clauses, init-field clauses with no provided argument, init clauses with no provided argument, private field definitions, and other expressions are evaluated. Those expressions are evaluated as they appear in the class expression, from left to right.

Sometime during the evaluation of the expressions, superclass-declared initializations must be evaluated once by using the super-make-object procedure, super-new form, or super-instantiate form.

By-name initialization arguments to a class that have no matching initialization variable are
implicitly added as by-name arguments to a super-make-object, super-new, or super-
instantiate invocation, after the explicit arguments. If multiple initialization arguments
are provided for the same name, the first (if any) is used, and the unused arguments are
propagated to the superclass. (Note that converted by-position arguments are always placed
before explicit by-name arguments.) The initialization procedure for the object\% class
accepts zero initialization arguments; if it receives any by-name initialization arguments,
then \texttt{exn:fail:object} exception is raised.

If the end of initialization is reached for any class in the hierarchy without invoking the
superclass’s initialization, the \texttt{exn:fail:object} exception is raised. Also, if superclass
initialization is invoked more than once, the \texttt{exn:fail:object} exception is raised.

Fields inherited from a superclass are not initialized until the superclass’s initialization pro-
cedure is invoked. In contrast, all methods are available for an object as soon as the object is
created; the overriding of methods is not affected by initialization (unlike objects in C++)

\begin{verbatim}
(make-object class init-v ...) \rightarrow object?
  class : class?
  init-v : any/c
\end{verbatim}

Creates an instance of \texttt{class}. The \texttt{init-v}s are passed as initialization arguments, bound
to the initialization variables of \texttt{class} for the newly created object as described in \texttt{§6.2.1}
"Initialization Variables" If \texttt{class} is not a class, the \texttt{exn:fail:contract} exception is
raised.

\begin{verbatim}
(new class-expr (id by-name-expr) ...)
\end{verbatim}

Creates an instance of the value of \texttt{class-expr} (which must be a class), and the value of
each \texttt{by-name-expr} is provided as a by-name argument for the corresponding \texttt{id}.

\begin{verbatim}
(instantiate class-expr (by-pos-expr ...) (id by-name-expr) ...)
\end{verbatim}

Creates an instance of the value of \texttt{class-expr} (which must be a class), and the values
of the \texttt{by-pos-expr}s are provided as by-position initialization arguments. In addition, the
value of each \texttt{by-name-expr} is provided as a by-name argument for the corresponding \texttt{id}.

\begin{verbatim}
super-make-object
\end{verbatim}

Produces a procedure that takes by-position arguments an invokes superclass initialization.
See \texttt{§6.3 “Creating Objects”} for more information.

\begin{verbatim}
(super-instantiate (by-pos-expr ...) (id by-expr ...) ...)
\end{verbatim}

Invokes superclass initialization with the specified by-position and by-name arguments. See
\texttt{§6.3 “Creating Objects”} for more information.
Invokes superclass initialization with the specified by-name arguments. See §6.3 “Creating Objects” for more information.

6.4 Field and Method Access

In expressions within a class definition, the initialization variables, fields, and methods of the class are all part of the environment. Within a method body, only the fields and other methods of the class can be referenced; a reference to any other class-introduced identifier is a syntax error. Elsewhere within the class, all class-introduced identifiers are available, and fields and initialization variables can be mutated with set!.

6.4.1 Methods

Method names used within a class can only be used in the procedure position of an application expression; any other use is a syntax error.

To allow methods to be applied to lists of arguments, a method application can have the following form:

\[
\text{(method-id arg ... . arg-list-expr)}
\]

This form calls the method in a way analogous to (apply method-id arg ... arg-list-expr). The arg-list-expr must not be a parenthesized expression.

Methods are called from outside a class with the send, send/apply, and send/keyword-apply forms.

\[
\text{(send obj-expr method-id arg ...)}
\]
\[
\text{(send obj-expr method-id arg ... . arg-list-expr)}
\]

Evaluates obj-expr to obtain an object, and calls the method with (external) name method-id on the object, providing the arg results as arguments. Each arg is as for #%app: either arg-expr or keyword arg-expr. In the second form, arg-list-expr cannot be a parenthesized expression.

If obj-expr does not produce an object, the exn:fail:contract exception is raised. If the object has no public method named method-id, the exn:fail:object exception is raised.
Like the dotted form of `send`, but `arg-list-expr` can be any expression.

Like `send/apply`, but with expressions for keyword and argument lists like `keyword-apply`.

Calls the method on `obj` whose name matches `method-name`, passing along all given `v`s and `kw-args`.

Calls multiple methods (in order) of the same object. Each `msg` corresponds to a use of `send`.

For example,

```
(send* edit (begin-edit-sequence)
 (insert "Hello")
 (insert #\newline)
 (end-edit-sequence))
```

is the same as

```
(let ([o edit])
 (send o begin-edit-sequence)
 (send o insert "Hello")
 (send o insert #\newline)
 (send o end-edit-sequence))
```
\( (\text{send+} \; \text{obj-expr} \; \text{msg} \; \ldots) \)

\( \text{msg} = (\text{method-id} \; \text{arg} \; \ldots) \)
| \( (\text{method-id} \; \text{arg} \; \ldots \; \text{arg-list-expr}) \)

Calls methods (in order) starting with the object produced by \( \text{obj-expr} \). Each method call will be invoked on the result of the last method call, which is expected to be an object. Each \( \text{msg} \) corresponds to a use of \( \text{send} \).

This is the functional analogue of \( \text{send*} \).

Examples:

\[
(\text{define point\%}
\quad (\text{class object\%}
\quad (\text{super-new})
\quad (\text{init-field} \; [x \; 0] \; [y \; 0])
\quad (\text{define/public} \; (\text{move-x} \; \text{dx})
\quad (\text{new this\%} \; [x \; (+ \; x \; \text{dx})]))
\quad (\text{define/public} \; (\text{move-y} \; \text{dy})
\quad (\text{new this\%} \; [y \; (+ \; y \; \text{dy})])))))
\]

\[
> \; (\text{send+} \; (\text{new point\%})
\quad (\text{move-x} \; 5)
\quad (\text{move-y} \; 7)
\quad (\text{move-x} \; 12))
\quad (\text{object:point\%} \; \ldots)
\]

\[
(\text{with-method} \; ((\text{id} \; \text{obj-expr} \; \text{method-id}) \; \ldots)
\quad \text{body} \; \ldots+)
\]

Extracts methods from an object and binds a local name that can be applied directly (in the same way as declared methods within a class) for each method. Each \( \text{obj-expr} \) must produce an object, which must have a public method named by the corresponding \( \text{method-id} \). The corresponding \( \text{id} \) is bound so that it can be applied directly (see §6.4.1 "Methods").

Example:

\[
(\text{let} \; ([s \; (\text{new stack\%})])
\quad (\text{with-method} \; ([\text{push} \; (s \; \text{push!})]
\quad \quad (\text{push} \; 10)
\quad \quad (\text{push} \; 9)
\quad \quad (\text{pop})))
\]

560
is the same as

```scheme
(let ([s (new stack%)])
  (send s push! 10)
  (send s push! 9)
  (send s pop!))
```

### 6.4.2 Fields

```scheme
(get-field id obj-expr)
```

Extracts the field with (external) name `id` from the value of `obj-expr`.

If `obj-expr` does not produce an object, the `exn:fail:contract` exception is raised. If the object has no `id` field, the `exn:fail:object` exception is raised.

```scheme
(dynamic-get-field field-name obj)
```

field-name : symbol?
obj : object?

Extracts the field from `obj` with the (external) name that matches `field-name`. If the object has no field matching `field-name`, the `exn:fail:object` exception is raised.

```scheme
(set-field! id obj-expr expr)
```

Sets the field with (external) name `id` from the value of `obj-expr` to the value of `expr`.

If `obj-expr` does not produce an object, the `exn:fail:contract` exception is raised. If the object has no `id` field, the `exn:fail:object` exception is raised.

```scheme
(dynamic-set-field! field-name obj v)
```

field-name : symbol?
obj : object?
v : any/c

Sets the field from `obj` with the (external) name that matches `field-name` to `v`. If the object has no field matching `field-name`, the `exn:fail:object` exception is raised.

```scheme
(field-bound? id obj-expr)
```

Produces `#t` if the object result of `obj-expr` has a field with (external) name `id`, `#f` otherwise.

If `obj-expr` does not produce an object, the `exn:fail:contract` exception is raised.
(class-field-accessor class-expr field-id)

Returns an accessor procedure that takes an instance of the class produced by class-expr and returns the value of the object’s field with (external) name field-id.

If class-expr does not produce a class, the exn:fail:contract exception is raised. If the class has no field-id field, the exn:fail:object exception is raised.

(class-field-mutator class-expr field-id)

Returns a mutator procedure that takes an instance of the class produced by class-expr and a value, and sets the value of the object’s field with (external) name field-id to the given value. The result is #<void>.

If class-expr does not produce a class, the exn:fail:contract exception is raised. If the class has no field-id field, the exn:fail:object exception is raised.

6.4.3 Generics

A generic can be used instead of a method name to avoid the cost of relocating a method by name within a class.

(generic class-or-interface-expr id)

Produces a generic that works on instances of the class or interface produced by class-or-interface-expr (or an instance of a class/interface derived from class-or-interface) to call the method with (external) name id.

If class-or-interface-expr does not produce a class or interface, the exn:fail:contract exception is raised. If the resulting class or interface does not contain a method named id, the exn:fail:object exception is raised.

(sendDate generic obj-expr generic-expr arg ...)

Calls a method of the object produced by obj-expr as indicated by the generic produced by generic-expr. Each arg is as for #%app: either arg-expr or keyword arg-expr. The second form is analogous to calling a procedure with apply, where arg-list-expr is not a parenthesized expression.

If obj-expr does not produce an object, or if generic-expr does not produce a generic, the exn:fail:contract exception is raised. If the result of obj-expr is not an instance of the class or interface encapsulated by the result of generic-expr, the exn:fail:object exception is raised.
Like the generic form, but as a procedure that accepts a symbolic method name.

### 6.5 Mixins

```
(mixin (interface-expr ...) (interface-expr ...) class-clause ...)
```

Produces a mixin, which is a procedure that encapsulates a class extension, leaving the superclass unspecified. Each time that a mixin is applied to a specific superclass, it produces a new derived class using the encapsulated extension.

The given class must implement interfaces produced by the first set of interface-vars. The result of the procedure is a subclass of the given class that implements the interfaces produced by the second set of interface-vars. The class-clauses are as for class*, to define the class extension encapsulated by the mixin.

Evaluation of a mixin form checks that the class-clauses are consistent with both sets of interface-vars.

### 6.6 Traits

```
(require racket/trait)   package:base
```

The bindings documented in this section are provided by the racket/trait library, not racket/base or racket.

A trait is a collection of methods that can be converted to a mixin and then applied to a class. Before a trait is converted to a mixin, the methods of a trait can be individually renamed, and multiple traits can be merged to form a new trait.

```
(trait trait-clause ...)
```
trait-clause = (public maybe-renamed ...)  
| (pubment maybe-renamed ...)  
| (public-final maybe-renamed ...)  
| (override maybe-renamed ...)  
| (overment maybe-renamed ...)  
| (override-final maybe-renamed ...)  
| (augment maybe-renamed ...)  
| (augride maybe-renamed ...)  
| (augment-final maybe-renamed ...)  
| (inherit maybe-renamed ...)  
| (inherit/super maybe-renamed ...)  
| (inherit/inner maybe-renamed ...)  
| method-definition  
| (field field-declaration ...)  
| (inherit-field maybe-renamed ...)  

Creates a trait. The body of a trait form is similar to the body of a class* form, but restricted to non-private method definitions. In particular, the grammar of maybe-renamed, method-definition, and field-declaration are the same as for class*, and every method-definition must have a corresponding declaration (one of public, override, etc.). As in class, uses of method names in direct calls, super calls, and inner calls depend on bringing method names into scope via inherit, inherit/super, inherit/inner, and other method declarations in the same trait; an exception, compared to class, is that overment binds a method name only in the corresponding method, and not in other methods of the same trait. Finally, macros such as public* and define/public work in trait as in class.

External identifiers in trait, trait-exclude, trait-exclude-field, trait-alias, trait-rename, and trait-rename-field forms are subject to binding via define-member-name and define-local-member-name. Although private methods or fields are not allowed in a trait form, they can be simulated by using a public or field declaration and a name whose scope is limited to the trait form.

(trait? v) → boolean?  
| v : any/c  

Returns #t if v is a trait, #f otherwise.

(trait->mixin tr) → (class? . -> . class?)  
| tr : trait?  

Converts a trait to a mixin, which can be applied to a class to produce a new class. An expression of the form

(trait->mixin
  (trait
    trait-clause ...))
is equivalent to

```scheme
(lambda (\%)
  (class \%
    trait-clause ...
    (super-new)))
```

Normally, however, a trait’s methods are changed and combined with other traits before converting to a mixin.

```scheme
(trait-sum tr ...+) \to trait?
  tr : trait?
```

Produces a trait that combines all of the methods of the given \textit{trs}. For example,

```scheme
(define t1
  (trait
    (define/public (m1) 1)))
(define t2
  (trait
    (define/public (m2) 2)))
(define t3 (trait-sum t1 t2))
```

creates a trait \textit{t3} that is equivalent to

```scheme
(trait
  (define/public (m1) 1)
  (define/public (m2) 2))
```

but \textit{t1} and \textit{t2} can still be used individually or combined with other traits.

When traits are combined with \textit{trait-sum}, the combination drops \textit{inherit}, \textit{inherit/super}, \textit{inherit/inner}, and \textit{inherit-field} declarations when a definition is supplied for the same method or field name by another trait. The \textit{trait-sum} operation fails (the \texttt{exn:fail:contract} exception is raised) if any of the traits to combine define a method or field with the same name, or if an \textit{inherit/super} or \textit{inherit/inner} declaration to be dropped is inconsistent with the supplied definition. In other words, declaring a method with \textit{inherit}, \textit{inherit/super}, or \textit{inherit/inner}, does not count as defining the method; at the same time, for example, a trait that contains an \textit{inherit/super} declaration for a method \textit{m} cannot be combined with a trait that defines \textit{m} as \textit{augment}, since no class could satisfy the requirements of both \textit{augment} and \textit{inherit/super} when the trait is later converted to a mixin and applied to a class.

```scheme
(trait-exclude trait-expr id)
```
Produces a new trait that is like the trait result of \texttt{trait-expr}, but with the definition of a method named by \texttt{id} removed; as the method definition is removed, either an \texttt{inherit}, \texttt{inherit/super}, or \texttt{inherit/inner} declaration is added:

- A method declared with \texttt{public}, \texttt{pubment}, or \texttt{public-final} is replaced with an \texttt{inherit} declaration.
- A method declared with \texttt{override} or \texttt{override-final} is replaced with an \texttt{inherit/super} declaration.
- A method declared with \texttt{augment}, \texttt{augride}, or \texttt{augment-final} is replaced with an \texttt{inherit/inner} declaration.
- A method declared with \texttt{overment} is not replaced with any \texttt{inherit} declaration.

If the trait produced by \texttt{trait-expr} has no method definition for \texttt{id}, the \texttt{exn:fail:contract} exception is raised.

\begin{verbatim}
\texttt{(trait-exclude-field trait-expr id)}
\end{verbatim}

Produces a new trait that is like the trait result of \texttt{trait-expr}, but with the definition of a field named by \texttt{id} removed; as the field definition is removed, an \texttt{inherit-field} declaration is added.

\begin{verbatim}
\texttt{(trait-alias trait-expr id new-id)}
\end{verbatim}

Produces a new trait that is like the trait result of \texttt{trait-expr}, but the definition and declaration of the method named by \texttt{id} is duplicated with the name \texttt{new-id}. The consistency requirements for the resulting trait are the same as for \texttt{trait-sum}, otherwise the \texttt{exn:fail:contract} exception is raised. This operation does not rename any other use of \texttt{id}, such as in method calls (even method calls to \texttt{identifier} in the cloned definition for \texttt{new-id}).

\begin{verbatim}
\texttt{(trait-rename trait-expr id new-id)}
\end{verbatim}

Produces a new trait that is like the trait result of \texttt{trait-expr}, but all definitions and references to methods named \texttt{id} are replaced by definitions and references to methods named by \texttt{new-id}. The consistency requirements for the resulting trait are the same as for \texttt{trait-sum}, otherwise the \texttt{exn:fail:contract} exception is raised.

\begin{verbatim}
\texttt{(trait-rename-field trait-expr id new-id)}
\end{verbatim}

Produces a new trait that is like the trait result of \texttt{trait-expr}, but all definitions and references to fields named \texttt{id} are replaced by definitions and references to fields named by \texttt{new-id}. The consistency requirements for the resulting trait are the same as for \texttt{trait-sum}, otherwise the \texttt{exn:fail:contract} exception is raised.
6.7 Object and Class Contracts

```
(class/c maybe-opaque member-spec ...)  

maybe-opaque =  
    #:opaque  
    #:opaque #:ignore-local-member-names  

member-spec = method-spec  
    | (field field-spec ...)  
    | (init field-spec ...)  
    | (init-field field-spec ...)  
    | (inherit method-spec ...)  
    | (inherit-field field-spec ...)  
    | (super method-spec ...)  
    | (inner method-spec ...)  
    | (override method-spec ...)  
    | (augment method-spec ...)  
    | (augride method-spec ...)  
    | (absent absent-spec ...)  

method-spec = method-id  
    | (method-id method-contract-expr)  

field-spec = field-id  
    | (field-id contract-expr)  

absent-spec = method-id  
    | (field field-id ...)  
```

Produces a contract for a class.

There are two major categories of contracts listed in a class/c form: external and internal contracts. External contracts govern behavior when an object is instantiated from a class or when methods or fields are accessed via an object of that class. Internal contracts govern behavior when method or fields are accessed within the class hierarchy. This separation allows for stronger contracts for class clients and weaker contracts for subclasses.

Method contracts must contain an additional initial argument which corresponds to the implicit this parameter of the method. This allows for contracts which discuss the state of the object when the method is called (or, for dependent contracts, in other parts of the contract). Alternative contract forms, such as ->m, are provided as a shorthand for writing method contracts.

Methods and fields listed in an absent clause must not be present in the class.
A class contract can be specified to be `opaque` with the `#:opaque` keyword. An opaque class contract will only accept a class that defines exactly the external methods and fields specified by the contract. A contract error is raised if the contracted class contains any methods or fields that are not specified. Methods or fields with local member names (i.e., defined with `define-local-member-name`) are ignored for this check if `#:ignore-local-member-names` is provided.

The external contracts are as follows:

- An external method contract without a tag describes the behavior of the implementation of `method-id` on method sends to an object of the contracted class. This contract will continue to be checked in subclasses until the contracted class’s implementation is no longer the entry point for dynamic dispatch.

  If only the field name is present, this is equivalent to insisting only that the method is present in the class.

  Examples:

  ```scheme
  (define woody%
    (class object%
      (define/public (draw who)
        (format "reach for the sky, ~a" who))
      (super-new)))
  
  (define/contract woody+c%
    (class/c [draw (->m symbol? string?)])
    woody%)
  
  > (send (new woody%) draw #f)
  "reach for the sky, #f"
  > (send (new woody+c%) draw 'zurg)
  "reach for the sky, zurg"
  > (send (new woody+c%) draw #f)
  `draw: contract violation`
  expected: symbol?
  given: #f
  in: the 1st argument of
  the draw method in
  (class/c (draw (->m symbol? string?)))
  contract from: (definition woody+c%)
  contract on: woody+c%
  blaming: top-level
  (assuming the contract is correct)
  at: eval:68:0
  ```

- An external field contract, tagged with `field`, describes the behavior of the value contained in that field when accessed from outside the class. Since fields may be
mutated, these contracts are checked on any external access (via get-field) and external mutations (via set-field!) of the field.

If only the field name is present, this is equivalent to using the contract any/c (but it is checked more efficiently).

Examples:

```
(define woody/hat%
  (class woody%
    (field [hat-location 'uninitialized])
    (define/public (lose-hat) (set! hat-location 'lost))
    (define/public (find-hat) (set! hat-location 'on-head))
    (super-new))

(define/contract woody/hat+c%
  (class/c [draw (->m symbol? string?)
            [lose-hat (->m void?)
            [find-hat (->m void?)
            (field [hat-location (or/c 'on-head 'lost)])]
  woody/hat%)

> (get-field hat-location (new woody/hat%))
'uninitialized
> (let ([woody (new woody/hat+c%)])
  (send woody lose-hat)
  (get-field hat-location woody))
'lost
> (get-field hat-location (new woody/hat+c%))
woody/hat+c%: broke its own contract
  promised: (or/c (quote on-head) (quote lost))
  produced: 'uninitialized
  in: the hat-location field in
    (class/c
      (draw (->m symbol? string?))
      (lose-hat (->m void?))
      (find-hat (->m void?))
      (field (hat-location
        (or/c 'on-head 'lost)))))
  contract from: (definition woody/hat+c%)
  blaming: (definition woody/hat+c%)
  (assuming the contract is correct)
  at: eval:73:0
> (let ([woody (new woody/hat+c%)])
  (set-field! hat-location woody 'under-the-dresser))
woody/hat+c%: contract violation
  expected: (or/c (quote on-head) (quote lost))
  given: 'under-the-dresser
  in: the hat-location field in
```
An initialization argument contract, tagged with `init`, describes the expected behavior of the value paired with that name during class instantiation. The same name can be provided more than once, in which case the first such contract in the `class/c` form is applied to the first value tagged with that name in the list of initialization arguments, and so on.

If only the initialization argument name is present, this is equivalent to using the contract `any/c` (but it is checked more efficiently).

Examples:

```scheme
(define woody/init-hat% (class woody% (init init-hat-location) (field [hat-location init-hat-location]))
(define/public (lose-hat) (set! hat-location 'lost))
(define/public (find-hat) (set! hat-location 'on-head))
(super-new))

(define/contract woody/init-hat+c% (class/c [draw (->m symbol? string?)] [lose-hat (->m void?)] [find-hat (->m void?)]
  (init [init-hat-location (or/c 'on-head 'lost)])
  (field [hat-location (or/c 'on-head 'lost)]))
woody/init-hat%)

> (get-field hat-location (new woody/init-hat+c% [init-hat-location 'lost]))
'lost
> (get-field hat-location (new woody/init-hat+c% [init-hat-location 'slinkys-mouth]))
woody/init-hat+c%: contract violation
  expected: (or/c (quote on-head) (quote lost))
given: 'slinkys-mouth
```

570
in: the init-hat-location init argument in
  (class/c
    (draw (->m symbol? string?))
    (lose-hat (->m void?))
    (find-hat (->m void?))
    (init (init-hat-location
      (or/c 'on-head 'lost)))
    (field (hat-location
      (or/c 'on-head 'lost))))
contract from:
  (definition woody/init-hat+c%)
blaming: top-level
  (assuming the contract is correct)
at: eval:79:0

• The contracts listed in an init-field section are treated as if each contract appeared
  in an init section and a field section.

The internal contracts restrict the behavior of method calls made between classes and their
subclasses; such calls are not controlled by the class contracts described above.

As with the external contracts, when a method or field name is specified but no contract ap-
pears, the contract is satisfied merely with the presence of the corresponding field or method.

• A method contract tagged with inherit describes the behavior of the method when
  invoked directly (i.e., via inherit) in any subclass of the contracted class. This
  contract, like external method contracts, applies until the contracted class’s method
  implementation is no longer the entry point for dynamic dispatch.

Examples:
> (new (class woody+c%
      (inherit draw)
      (super-new)
      (printf "woody sez: "~a\n" (draw "evil dr pork-
      chop"))))
woody sez: ‘reach for the sky, evil dr porkchop’
(object:eval:82:0 ...)

(define/contract woody+c-inherit%
  (class/c (inherit [draw (->m symbol? string?)]))
  woody+c%)

> (new (class woody+c-inherit%
      (inherit draw)
      (printf "woody sez: ~a\n" (draw "evil dr pork-
      chop"))))

571
A method contract tagged with super describes the behavior of method-id when called by the super form in a subclass. This contract only affects super calls in subclasses which call the contract class’s implementation of method-id.

This example shows how to extend the draw method so that if it is passed two arguments, it combines two calls to the original draw method, but with a contract the controls how the super methods must be invoked.

Examples:

```scheme
(define/contract woody%+
  (class/c (super [draw (->m symbol? string?)]))
  (class object%
    (define/public (draw who)
      (format "reach for the sky, ~a" who))
    (super-new)))

(define woody2+c%
  (class woody%+
    (define/override draw
      (case-lambda
        [(a) (super draw a)]
        [(a b) (string-append (super draw a)
          " and "
          (super draw b))]
      (super-new)))
    (super-new)))

> (send (new woody2+c%) draw 'evil-dr-porkchop 'zurg)
"reach for the sky, evil-dr-porkchop and reach for the sky, zurg"
> (send (new woody2+c%) draw "evil dr porkchop" "zurg")
```

```scheme
draw: contract violation
expected: symbol?
given: "evil dr porkchop"
```
in: the 1st argument of
draw method in
class/c
(supercall (draw (->m symbol? string?)))
contract from: (definition woody%+s)
contract on: woody%+s
blaming: top-level
(assuming the contract is correct)
at: eval:85:0

The last call signals an error blaming woody2%+c because there is no contract checking the initial draw call and the super-call violates its contract.

- A method contract tagged with inner describes the behavior the class expects of an augmenting method in a subclass. This contract affects any implementations of method-id in subclasses which can be called via inner from the contracted class. This means a subclass which implements method-id via augment or overment stop future subclasses from being affected by the contract, since further extension cannot be reached via the contracted class.

- A method contract tagged with override describes the behavior expected by the contracted class for method-id when called directly (i.e. by the application (method-id ...)). This form can only be used if overriding the method in subclasses will change the entry point to the dynamic dispatch chain (i.e., the method has never been augmentable).

This time, instead of overriding draw to support two arguments, we can make a new method, draw2 that takes the two arguments and calls draw. We also add a contract to make sure that overriding draw doesn’t break draw2.

Examples:

(define/contract woody2+override/c%
class/c (override [draw (->m symbol? string?)])
class woody+c%
(inherit draw)
(define/public (draw2 a b)
  (string-append (draw a)
                " and "
                (draw b)))
(super-new)))

(define woody2+broken-draw
class woody2+override/c%
(define/override (draw x)
  'not-a-string)
(super-new)))
> (send (new woody2+broken-draw) draw2
evil-dr-porkchop
'zurg)
draw: contract violation
  expected: string?
given: 'not-a-string
in: the range of
  the draw method in
  (class/c
    (override (draw (->m symbol? string?))))
contract from:
  (definition woody2+override/c%)
contract on: woody2+override/c%
blaming: top-level
  (assuming the contract is correct)
at: eval:89:0

• A method contract tagged with either augment or augride describes the behavior provided by the contracted class for method-id when called directly from subclasses. These forms can only be used if the method has previously been augmentable, which means that no augmenting or overriding implementation will change the entry point to the dynamic dispatch chain. augment is used when subclasses can augment the method, and augride is used when subclasses can override the current augmentation.

• A field contract tagged with inherit-field describes the behavior of the value contained in that field when accessed directly (i.e., via inherit-field) in any subclass of the contracted class. Since fields may be mutated, these contracts are checked on any access and/or mutation of the field that occurs in such subclasses.

• Changed in version 6.1.1.8 of package base: Opaque class/c now optionally ignores local member names if an additional keyword is supplied.

(absent absent-spec ...)

See class/c; use outside of a class/c form is a syntax error.

(->m dom ... range)

Similar to ->, except that the domain of the resulting contract contains one more element than the stated domain, where the first (implicit) argument is contracted with any/c. This contract is useful for writing simpler method contracts when no properties of this need to be checked.

(->*m (mandatory-dom ...) (optional-dom ...) rest range)
Similar to \( \rightarrow\star\), except that the mandatory domain of the resulting contract contains one more element than the stated domain, where the first (implicit) argument is contracted with \( \text{any/c} \). This contract is useful for writing simpler method contracts when no properties of this need to be checked.

\[
\text{(case-}\rightarrow\text{m } (\rightarrow \text{ dom } \ldots \text{ rest range}) \ldots)\]

Similar to \( \text{case-}\rightarrow\), except that the mandatory domain of each case of the resulting contract contains one more element than the stated domain, where the first (implicit) argument is contracted with \( \text{any/c} \). This contract is useful for writing simpler method contracts when no properties of this need to be checked.

\[
(\rightarrow\text{dm } (\text{mandatory-dependent-dom } \ldots) \\
\quad (\text{optional-dependent-dom } \ldots) \\
\quad \text{dependent-rest} \\
\quad \text{pre-cond} \\
\quad \text{dep-range})\]

Similar to \( \rightarrow\text{d} \), except that the mandatory domain of the resulting contract contains one more element than the stated domain, where the first (implicit) argument is contracted with \( \text{any/c} \). In addition, this is appropriately bound in the body of the contract. This contract is useful for writing simpler method contracts when no properties of this need to be checked.

\[
(\text{object/c } \text{member-spec } \ldots)\]

\[\text{member-spec } = \text{method-spec} \]
\[\quad | \text{field field-spec } \ldots\]

\[\text{method-spec } = \text{method-id} \]
\[\quad | \text{(method-id method-contract)}\]

\[\text{field-spec } = \text{field-id} \]
\[\quad | \text{(field-id contract-expr)}\]

Produces a contract for an object.

Unlike the older form \( \text{object-contract} \), but like \( \text{class/c} \), arbitrary contract expressions are allowed. Also, method contracts for \( \text{object/c} \) follow those for \( \text{class/c} \). An object wrapped with \( \text{object/c} \) behaves as if its class had been wrapped with the equivalent \( \text{class/c} \) contract.

\[
(\text{instanceof/c class-contract}) \rightarrow \text{contract?} \\
\text{class-contract : contract?}\]

Produces a contract for an object, where the object is an instance of a class that conforms to \( \text{class-contract} \).
(dynamic-object/c method-names
  method-contracts
  field-names
  field-contracts) → contract?
method-names : (listof symbol?)
method-contracts : (listof contract?)
field-names : (listof symbol?)
field-contracts : (listof contract?)

Produces a contract for an object, similar to object/c but where the names and contracts for both methods and fields can be computed dynamically. The list of names and contracts for both methods and field respectively must have the same lengths.

(object-contract member-spec ...)

576
member-spec = (method-id method-contract)
  | (field field-id contract-expr)

method-contract = (-> dom ... range)
  | (->* (mandatory-dom ...) (optional-dom ...)
       rest range)
  | (->d (mandatory-dependent-dom ...) (optional-dependent-dom ...)
       dependent-rest pre-cond dep-range)

  dom = dom-expr
  | keyword dom-expr

  range = range-expr
  | (values range-expr ...)
  | any

  mandatory-dom = dom-expr
  | keyword dom-expr

  optional-dom = dom-expr
  | keyword dom-expr

  rest = 
  | #:rest rest-expr

  mandatory-dependent-dom = [id dom-expr]
  | keyword [id dom-expr]

  optional-dependent-dom = [id dom-expr]
  | keyword [id dom-expr]

  dependent-rest = 
  | #:rest id rest-expr

  pre-cond = 
  | #:pre-cond boolean-expr

  dep-range = any
  | [id range-expr] post-cond
  | (values [id range-expr] ...) post-cond

  post-cond = 
  | #:post-cond boolean-expr
Produces a contract for an object.

Each of the contracts for a method has the same semantics as the corresponding function contract, but the syntax of the method contract must be written directly in the body of the object-contract—much like the way that methods in class definitions use the same syntax as regular function definitions, but cannot be arbitrary procedures. Unlike the method contracts for \texttt{class/c}, the implicit \texttt{this} argument is not part of the contract. To allow for the use of \texttt{this} in dependent contracts, \texttt{-cd} contracts implicitly bind \texttt{this} to the object itself.

\texttt{mixin-contract : contract?}

A function contract that recognizes mixins. It guarantees that the input to the function is a class and the result of the function is a subclass of the input.

\begin{verbatim}
(make-mixin-contract type ...) -> contract?
type : (or/c class? interface?)
\end{verbatim}

Produces a function contract that guarantees the input to the function is a class that implements/subclasses each \texttt{type}, and that the result of the function is a subclass of the input.

\begin{verbatim}
(is-a?/c type) -> flat-contract?
type : (or/c class? interface?)
\end{verbatim}

Accepts a class or interface and returns a flat contract that recognizes objects that instantiate the class/interface.

See \texttt{is-a}.

\begin{verbatim}
(implementation?/c interface) -> flat-contract?
interface : interface?
\end{verbatim}

Returns a flat contract that recognizes classes that implement \texttt{interface}.

See \texttt{implementation}.

\begin{verbatim}
(subclass?/c class) -> flat-contract?
class : class?
\end{verbatim}

Returns a flat contract that recognizes classes that are subclasses of \texttt{class}.

See \texttt{subclass}.

\section{Object Equality and Hashing}

By default, objects that are instances of different classes or that are instances of a non-transparent class are \texttt{equal?} only if they are \texttt{eq?}. Like transparent structures, two objects
that are instances of the same transparent class (i.e., every superclass of the class has \#f as its inspector) are equal? when their field values are equal?.

To customize the way that a class instance is compared to other instances by equal?, implement the equal<%> interface.

equal<%> : interface?

The equal<%> interface includes three methods, which are analogous to the functions provided for a structure type with prop:equal+hash:

• equal-to? — Takes two arguments. The first argument is an object that is an instance of the same class (or a subclass that does not re-declare its implementation of equal<%>) and that is being compared to the target object. The second argument is an equal?-like procedure of two arguments that should be used for recursive equality testing. The result should be a true value if the object and the first argument of the method are equal, #f otherwise.

• equal-hash-code-of — Takes one argument, which is a procedure of one argument that should be used for recursive hash-code computation. The result should be an exact integer representing the target object’s hash code.

• equal-secondary-hash-code-of — Takes one argument, which is a procedure of one argument that should be used for recursive hash-code computation. The result should be an exact integer representing the target object’s secondary hash code.

The equal<%> interface is unusual in that declaring the implementation of the interface is different from inheriting the interface. Two objects can be equal only if they are instances of classes whose most specific ancestor to explicitly implement equal<%> is the same ancestor.

See prop:equal+hash for more information on equality comparisons and hash codes. The equal<%> interface is implemented with interface* and prop:equal+hash.

Example:

#lang racket

;; Case insensitive words:
(define ci-word% (class* object% (equal<%>)

;;; Initialization
(init-field word)
(super-new)
;; We define equality to ignore case:
(define/public (equal-to? other recur)
  (string-ci=? word (get-field word other)))

;; The hash codes need to be insensitive to casing as well.
;; We'll just downcase the word and get its hash code.
(define/public (equal-hash-code-of hash-code)
  (hash-code (string-downcase word))))

(define/public (equal-secondary-hash-code-of hash-code)
  (hash-code (string-downcase word))))

;; We can create a hash with a single word:
(define h (make-hash))
(hash-set! h (new ci-word% [word "inconceivable!"]) 'value)

;; Lookup into the hash should be case-insensitive, so that
;; both of these should return 'value.
(hash-ref h (new ci-word% [word "inconceivable!"]))
(hash-ref h (new ci-word% [word "INCONCEIVABLE!"]))

;; Comparison fails if we use a non-ci-word%:
(hash-ref h "inconceivable!" 'i-dont-think-it-means-what-you-think-it-means)

6.9 Object Serialization

(define-serializable-class* class-id superclass-expr ...
  (interface-expr ...)
  class-clause ...)

Binds class-id to a class, where superclass-expr, the interface-exprs, and the
class-clauses are as in class*.

This form can only be used at the top level, either within a module or outside. The class-id
identifier is bound to the new class, and deserialize-info:class-id is also defined;
if the definition is within a module, then the latter is provided from a deserialize-info
submodule via module+.

Serialization for the class works in one of two ways:

- If the class implements the built-in interface externalizable<%, then an object is
  serialized by calling its externalize method; the result can be anything that is se-
rializable (but, obviously, should not be the object itself). Deserialization creates an instance of the class with no initialization arguments, and then calls the object’s internalize method with the result of externalize (or, more precisely, a deserialized version of the serialized result of a previous call).

To support this form of serialization, the class must be instantiable with no initialization arguments. Furthermore, cycles involving only instances of the class (and other such classes) cannot be serialized.

- If the class does not implement externalizable<%>, then every superclass of the class must be either serializable or transparent (i.e., have #f as its inspector). Serialization and deserialization are fully automatic, and may involve cycles of instances.

To support cycles of instances, deserialization may create an instance of the call with all fields as the undefined value, and then mutate the object to set the field values. Serialization support does not otherwise make an object’s fields mutable.

In the second case, a serializable subclass can implement externalizable<%>, in which case the externalize method is responsible for all serialization (i.e., automatic serialization is lost for instances of the subclass). In the first case, all serializable subclasses implement externalizable<%>, since a subclass implements all of the interfaces of its parent class.

In either case, if an object is an immediate instance of a subclass (that is not itself serializable), the object is serialized as if it was an immediate instance of the serializable class. In particular, overriding declarations of the externalize method are ignored for instances of non-serializable subclasses.

(define-serializable-class class-id superclass-expr
class-clause ...)

Like define-serializable-class*, but without interface expressions (analogous to class).

externalizable<%> : interface?

The externalizable<%> interface includes only the externalize and internalize methods. See define-serializable-class* for more information.

6.10 Object Printing

To customize the way that a class instance is printed by print, write and display, implement the printable<%> interface.
The **printable** interface includes only the `custom-print`, `custom-write`, and `custom-display` methods. The `custom-print` method accepts two arguments: the destination port and the current quasiquote depth as an exact nonnegative integer. The `custom-write` and `custom-display` methods each accepts a single argument, which is the destination port to write or display the object.

Calls to the `custom-print`, `custom-write`, or `custom-display` methods are like calls to a procedure attached to a structure type through the `prop:custom-write` property. In particular, recursive printing can trigger an escape from the call.

See `prop:custom-write` for more information. The **printable** interface is implemented with `interface*` and `prop:custom-write`.

**writable** interface?

Like **printable**, but includes only the `custom-write` and `custom-display` methods. A print request is directed to `custom-write`.

### 6.11 Object, Class, and Interface Utilities

**(object? v) → boolean?**

\[ v : \text{any/c} \]

Returns `#t` if \( v \) is an object, `#f` otherwise.

Examples:

\[
> (object? (new object))
#t
> (object? object)
#f
> (object? "clam chowder")
#f
\]

**(class? v) → boolean?**

\[ v : \text{any/c} \]

Returns `#t` if \( v \) is a class, `#f` otherwise.

Examples:
> (class? object%)
#t
> (class? (class object% (super-new)))
#t
> (class? (new object%))
#f
> (class? "corn chowder")
#f

(interface? v) → boolean?
 v : any/c

Returns #t if v is an interface, #f otherwise.

Examples:

> (interface? (interface () empty cons first rest))
#t
> (interface? object%)
#f
> (interface? "gazpacho")
#f

(generic? v) → boolean?
 v : any/c

Returns #t if v is a generic, #f otherwise.

Examples:

> (define c%
   (class object%
      (super-new)
      (define/public (m x)
        (+ 3.14 x)))
> (generic? (generic c% m))
#t
> (generic? c%)
#f
> (generic? "borscht")
#f

(object=? a b) → boolean?
a : object?
b : object?
Determines whether \(a\) and \(b\) were returned from the same call to `new` or not. If the two objects have fields, this procedure determines whether mutating a field of one would change that field in the other.

This procedure is similar in spirit to `eq?` but also works properly with contracts (and has a stronger guarantee).

Examples:

```scheme
> (define obj-1 (new object%))
> (define obj-2 (new object%))
> (define/contract obj-3 (object/c) obj-1)
> (object=? obj-1 obj-1)
#t
> (object=? obj-1 obj-2)
#f
> (object=? obj-1 obj-3)
#t
> (eq? obj-1 obj-1)
#t
> (eq? obj-1 obj-2)
#f
> (eq? obj-1 obj-3)
#f
```

\[(\text{object-or-false=} \ a \ b) \rightarrow \text{boolean?}\]

- \(a: (\text{or/c object? #f})\)
- \(b: (\text{or/c object? #f})\)

Like `object=?`, but accepts `#f` for either argument and returns `#t` if both arguments are `#f`.

Examples:

```scheme
> (object-or-false=? #f (new object%))
#f
> (object-or-false=? (new object%) #f)
#f
> (object-or-false=? #f #f)
#t
```

Added in version 6.1.1.8 of package base.

\[(\text{object=}\text{-hash-code} \ o) \rightarrow \text{fixnum?}\]

- \(o: \text{object?}\)

Returns the hash code for \(o\) that corresponds to the equality relation `object=?`. 584
Added in version 7.1.0.6 of package base.

\[
\text{(object->vector } \text{ object } [\text{opaque-v}]) \rightarrow \text{ vector?}
\]
\[
\text{object} : \text{ object?}
\]
\[
\text{opaque-v} : \text{ any/c } = \#f
\]

Returns a vector representing \text{ object} that shows its inspectable fields, analogous to \text{ struct->vector}.

Examples:

\[
> \text{(object->vector } \text{ (new object%))}
\]
\[
\text{'(object:object% ...)}
\]
\[
> \text{(object->vector } \text{ (new class object%}
\quad \text{(super-new)}
\quad \text{(field } [\text{x 5}][\text{y 10}])))
\]
\[
\text{'(object:eval:107:0 ...)}
\]

\[
\text{(class->interface } \text{ class}) \rightarrow \text{ interface?}
\]
\[
\text{class} : \text{ class?}
\]

Returns the interface implicitly defined by \text{ class}.

Example:

\[
> \text{(class->interface object%)}
\]
\[
\text{#<interface:object%>}
\]

\[
\text{(object-interface } \text{ object}) \rightarrow \text{ interface?}
\]
\[
\text{object} : \text{ object?}
\]

Returns the interface implicitly defined by the class of \text{ object}.

Example:

\[
> \text{(object-interface } \text{ (new object%))}
\]
\[
\text{#<interface:object%>}
\]

\[
\text{(is-a? } \text{ v } \text{ type}) \rightarrow \text{ boolean?}
\]
\[
\text{v} : \text{ any/c}
\]
\[
\text{type} : \text{ (or/c interface? class?)}
\]

Returns \#t if \text{ v} is an instance of a class \text{ type} or a class that implements an interface \text{ type}, \#f otherwise.

Examples:
> (define point<%> (interface () get-x get-y))
> (define 2d-point%
  (class* object% (point<%>)
    (super-new)
    (field [x 0] [y 0])
    (define/public (get-x) x)
    (define/public (get-y) y)))
> (is-a? (new 2d-point%) 2d-point%) #t
> (is-a? (new 2d-point%) point<%) #t
> (is-a? (new object%) 2d-point%) #f
> (is-a? (new object%) point<%) #f

(subclass? v cls) \rightarrow boolean?
  v : any/c
  cls : class?

Returns #t if v is a class derived from (or equal to) cls, #f otherwise.

Examples:
> (subclass? (class object% (super-new)) object%) #t
> (subclass? object% (class object% (super-new))) #f
> (subclass? object% object%) #t

(implementation? v intf) \rightarrow boolean?
  v : any/c
  intf : interface?

Returns #t if v is a class that implements intf, #f otherwise.

Examples:
> (define i<%> (interface () go))
> (define c%
  (class* object% (i<%>)
    (super-new)
    (define/public (go) 'go)))
> (implementation? c% i<%) #t
> (implementation? object% i<%) #f

(interface-extension? v intf) → boolean?
  v : any/c
  intf : interface?

Returns #t if v is an interface that extends intf, #f otherwise.

Examples:
> (define point<%> (interface () get-x get-y))
> (define colored-point<%> (interface (point<%) color))
> (interface-extension? colored-point<%> point<%) #t
> (interface-extension? point<%> colored-point<%) #f
> (interface-extension? (interface () get-x get-y get-z) point<%) #f

(method-in-interface? sym intf) → boolean?
  sym : symbol?
  intf : interface?

Returns #t if intf (or any of its ancestor interfaces) includes a member with the name sym, #f otherwise.

Examples:
> (define i<%> (interface () get-x get-y))
> (method-in-interface? 'get-x i<%>) #t
> (method-in-interface? 'get-z i<%>) #f

(interface->method-names intf) → (listof symbol?)
  intf : interface?

Returns a list of symbols for the method names in intf, including methods inherited from superinterfaces, but not including methods whose names are local (i.e., declared with define-local-member-name).

Examples:
> (define i<%> (interface () get-x get-y))
> (interface->method-names i<%>)
'(get-x get-y)

(object-method-arity-includes? object sym cnt) → boolean?

object : object?
sym : symbol?
cnt : exact-nonnegative-integer?

Returns #t if object has a method named sym that accepts cnt arguments, #f otherwise.

Examples:

> (define c% (class object% (super-new) (define/public (m x [y 0]) (+ x y))))
> (object-method-arity-includes? (new c%) 'm 1)
#t
> (object-method-arity-includes? (new c%) 'm 2)
#t
> (object-method-arity-includes? (new c%) 'm 3)
#f
> (object-method-arity-includes? (new c%) 'n 1)
#f

(field-names object) → (listof symbol?)

object : object?

Returns a list of all of the names of the fields bound in object, including fields inherited from superinterfaces, but not including fields whose names are local (i.e., declared with define-local-member-name).

Examples:

> (field-names (new object%))
'()
> (field-names (new (class object% (super-new) (field [x 0] [y 0]))) )
'(x y)
Returns two values, analogous to the return values of `struct-info`:

- **class**: a class or `#f`; the result is `#f` if the current inspector does not control any class for which the `object` is an instance.
- **skipped?**: `#f` if the first result corresponds to the most specific class of `object`, `#t` otherwise.

Returns seven values, analogous to the return values of `struct-type-info`:

- **name**: the class’s name as a symbol;
- **field-cnt**: the number of fields (public and private) defined by the class;
- **field-name-list**: a list of symbols corresponding to the class’s public fields; this list can be larger than `field-cnt` because it includes inherited fields;
- **field-accessor**: an accessor procedure for obtaining field values in instances of the class; the accessor takes an instance and a field index between 0 (inclusive) and `field-cnt` (exclusive);
- **field-mutator**: a mutator procedure for modifying field values in instances of the class; the mutator takes an instance, a field index between 0 (inclusive) and `field-cnt` (exclusive), and a new field value;
- **super-class**: a class for the most specific ancestor of the given class that is controlled by the current inspector, or `#f` if no ancestor is controlled by the current inspector;
- **skipped?**: `#f` if the sixth result is the most specific ancestor class, `#t` otherwise.
Raised for class-related failures, such as attempting to call a method that is not supplied by an object.

\begin{verbatim}
(class-seal class key unsealed-init unsealed-fields unsealed-methods inst-proc member-proc) \rightarrow class?
\end{verbatim}

\begin{itemize}
\item \texttt{class : class?}
\item \texttt{key : symbol?}
\item \texttt{unsealed-init : (listof symbol?)}
\item \texttt{unsealed-fields : (listof symbol?)}
\item \texttt{unsealed-methods : (listof symbol?)}
\item \texttt{inst-proc : (\rightarrow class? any)}
\item \texttt{member-proc : (\rightarrow class? (listof symbol?) any)}
\end{itemize}

Adds a seal to a given class keyed with the symbol \texttt{key}. The given \texttt{unsealed-init}, \texttt{unsealed-fields}, and \texttt{unsealed-methods} list corresponding class members that are unaffected by sealing.

When a class has any seals, the \texttt{inst-proc} procedure is called on instantiation (normally, this is used to raise an error on instantiation) and the \texttt{member-proc} function is called (again, this is normally used to raise an error) when a subclass attempts to add class members that are not listed in the unsealed lists.

The \texttt{inst-proc} is called with the class value on which an instantiation was attempted. The \texttt{member-proc} is called with the class value and the list of initialization argument, field, or method names.

\begin{verbatim}
(class-unseal class key wrong-key-proc) \rightarrow class?
\end{verbatim}

\begin{itemize}
\item \texttt{class : class?}
\item \texttt{key : symbol?}
\item \texttt{wrong-key-proc : (\rightarrow class? any)}
\end{itemize}

Removes a seal on a class that has been previously sealed with the \texttt{class-seal} function and the given \texttt{key}.

If the unseal removed all of the seals in the class, the class value can be instantiated or subclassed freely. If the given class value does not contain or any seals or does not contain any seals with the given key, the \texttt{wrong-key-proc} function is called with the class value.

### 6.12 Surrogates

\begin{verbatim}
(require racket/surrogate)
\end{verbatim}

package: \texttt{base}
The bindings documented in this section are provided by the \texttt{racket/surrogate} library, not \texttt{racket/base} or \texttt{racket}.

The \texttt{racket/surrogate} library provides an abstraction for building an instance of the \textit{proxy design pattern}. The pattern consists of two objects, a \textit{host} and a \textit{surrogate} object. The host object delegates method calls to its surrogate object. Each host has a dynamically assigned surrogate, so an object can completely change its behavior merely by changing the surrogate.

\begin{verbatim}
(surrogate use-wrapper-proc method-spec ...)

use-wrapper-proc = #:use-wrapper-proc

method-spec = (augment default-expr method-id arg-spec ...)

| (override method-id arg-spec ...)

arg-spec = (id ...) | id
\end{verbatim}

The \texttt{surrogate} form produces four values: a host mixin (a procedure that accepts and returns a class), a host interface, a surrogate class, and a surrogate interface.

If \texttt{#:use-wrapper-proc} does not appear, the host mixin adds a single private field to its argument. It also adds getter and setter methods \texttt{get-surrogate} and \texttt{set-surrogate} to get and set the value of the field. The \texttt{set-surrogate} method accepts instances of the class returned by the \texttt{surrogate} form or \texttt{#f}, and it updates the field with its argument; then, \texttt{set-surrogate} calls the \texttt{on-disable-surrogate} on the previous value of the field and \texttt{on-enable-surrogate} for the new value of the field. The \texttt{get-surrogate} method returns the current value of the field.

If \texttt{#:use-wrapper-proc} does appear, the the host mixin adds a second private field and its getter and setter methods \texttt{get-surrogate-wrapper-proc} and \texttt{set-surrogate-wrapper-proc}. The additional field holds a wrapper procedure whose contract is \texttt{(-> (-> any) (-> any) any)}, so the procedure is invoked with two thunks. The first thunk is a fallback that invokes the original object’s method, skipping the surrogate. The second thunk invokes the surrogate. The default wrapper procedure is

\begin{verbatim}
(λ (fallback-thunk surrogate-thunk)
  (surrogate-thunk))
\end{verbatim}

That is, it simply defers to the method being invoked on the surrogate. Note that wrapper procedure can adjust the dynamic extent of calls to the surrogate by, for example, changing the values of parameters. The wrapper procedure is also invoked when calling the \texttt{on-disable-surrogate} and \texttt{on-enable-surrogate} methods of the surrogate.

The host mixin has a single overriding method for each \texttt{method-id} in the \texttt{surrogate} form (even the ones specified with \texttt{augment}). Each of these methods is defined with a case-
Each arm has the variables as arguments in the \texttt{arg-spec}. The body of each method tests the private surrogate field. If the field value is \texttt{#f}, the method just returns the result of invoking the super or inner method. If the field value is not \texttt{#f}, the corresponding method of the object in the field is invoked. This method receives the same arguments as the original method, plus two extras. The extra arguments come at the beginning of the argument list. The first is the original object. The second is a procedure that calls the super or inner method (i.e., the method of the class that is passed to the mixin or an extension, or the method in an overriding class), with the arguments that the procedure receives.

For example, the host-mixin for this surrogate:

\begin{verbatim}
(surrogate (override m (x y z)))
\end{verbatim}

will override the \texttt{m} method and call the surrogate like this:

\begin{verbatim}
(define/override (m x y z)
  (if surrogate
     (send surrogate m
       this
       (\lambda (x y z) (super m x y z))
       x y z)
     (super m x y z)))
\end{verbatim}

where \texttt{surrogate} is bound to the value most recently passed to the host mixin’s \texttt{set-surrogate} method.

The host interface has the names \texttt{set-surrogate}, \texttt{get-surrogate}, and each of the \texttt{method-ids} in the original form.

The surrogate class has a single public method for each \texttt{method-id} in the surrogate form. These methods are invoked by classes constructed by the mixin. Each has a corresponding method signature, as described in the above paragraph. Each method just passes its argument along to the super procedure it receives.

In the example above, this is the \texttt{m} method in the surrogate class:

\begin{verbatim}
(define/public (m original-object original-super x y z)
  (original-super x y z))
\end{verbatim}

If you derive a class from the surrogate class, do not both call the \texttt{super} argument and the super method of the surrogate class itself. Only call one or the other, since the default methods call the \texttt{super} argument.

Finally, the interface contains all of the names specified in surrogate’s argument, plus \texttt{on-enable-surrogate} and \texttt{on-disable-surrogate}. The class returned by \texttt{surrogate} implements this interface.
7 Units

Units organize a program into separately compilable and reusable components. The imports and exports of a unit are grouped into a signature, which can include “static” information (such as macros) in addition to placeholders for run-time values. Units with suitably matching signatures can be linked together to form a larger unit, and a unit with no imports can be invoked to execute its body.

```
(require racket/unit) package: base
```

The bindings documented in this section are provided by the racket/unit and racket libraries, but not racket/base. The racket/unit module name can be used as a language name with #lang; see §7.10 “Single-Unit Modules”.

7.1 Creating Units

```
(unit
  (import tagged-sig-spec ...)
  (export tagged-sig-spec ...)
  init-depends-decl
  unit-body-expr-or-defn
  ...)
```

- `tagged-sig-spec = sig-spec`  
  - (tag id sig-spec)

- `sig-spec = sig-id`
  - (prefix id sig-spec)
  - (rename sig-spec (id id) ...)
  - (only sig-spec id ...)
  - (except sig-spec id ...)

- `init-depends-decl =`
  - (init-depend tagged-sig-id ...)

- `tagged-sig-id = sig-id`
  - (tag id sig-id)

Produces a unit that encapsulates its `unit-body-expr-or-defns`. Expressions in the unit body can refer to identifiers bound by the `sig-specs` of the import clause, and the body must include one definition for each identifier of a `sig-spec` in the export clause. An identifier that is exported cannot be set in either the defining unit or in importing units, although the implicit assignment to initialize the variable may be visible as a mutation.
Each import or export $\text{sig-spec}$ ultimately refers to a $\text{sig-id}$, which is an identifier that is bound to a signature by define-signature.

In a specific import or export position, the set of identifiers bound or required by a particular $\text{sig-id}$ can be adjusted in a few ways:

- **(prefix $\text{id}$ $\text{sig-spec}$)** as an import binds the same as $\text{sig-spec}$, except that each binding is prefixed with $\text{id}$. As an export, this form causes definitions using the $\text{id}$ prefix to satisfy the exports required by $\text{sig-spec}$.

- **(rename $\text{sig-spec}$ ($\text{id}$ $\text{id}$) ...)** as an import binds the same as $\text{sig-spec}$, except that the first $\text{id}$ is used for the binding instead of the second $\text{id}$ (where $\text{sig-spec}$ by itself must imply a binding that is $\text{bound-identifier}=$? to second $\text{id}$). As an export, this form causes a definition for the first $\text{id}$ to satisfy the export named by the second $\text{id}$ in $\text{sig-spec}$.

- **(only $\text{sig-spec}$ $\text{id}$ ...)** as an import binds the same as $\text{sig-spec}$, but restricted to just the listed $\text{ids}$ (where $\text{sig-spec}$ by itself must imply a binding that is $\text{bound-identifier}=$? to each $\text{id}$). This form is not allowed for an export.

- **(except $\text{sig-spec}$ $\text{id}$ ...)** as an import binds the same as $\text{sig-spec}$, but excluding all listed $\text{ids}$ (where $\text{sig-spec}$ by itself must imply a binding that is $\text{bound-identifier}=$? to each $\text{id}$). This form is not allowed for an export.

As suggested by the grammar, these adjustments to a signature can be nested arbitrarily.

A unit’s declared imports are matched with actual supplied imports by signature. That is, the order in which imports are supplied to a unit when linking is irrelevant; all that matters is the signature implemented by each supplied import. One actual import must be provided for each declared import. Similarly, when a unit implements multiple signatures, the order of the export signatures does not matter.

To support multiple imports or exports for the same signature, an import or export can be tagged using the form **(tag $\text{id}$ $\text{sig-spec}$)**. When an import declaration of a unit is tagged, then one actual import must be given the same tag (with the same signature) when the unit is linked. Similarly, when an export declaration is tagged for a unit, then references to that particular export must explicitly use the tag.

A unit is prohibited syntactically from importing two signatures that are not distinct, unless they have different tags; two signatures are distinct only if they share no ancestor through extends. The same syntactic constraint applies to exported signatures. In addition, a unit is prohibited syntactically from importing the same identifier twice (after renaming and other transformations on a $\text{sig-spec}$), exporting the same identifier twice (again, after renaming), or exporting an identifier that is imported.

When units are linked, the bodies of the linked units are executed in an order that is specified at the linking site. An optional (init-depend tagged-sig-id ...) declaration constrains the allowed orders of linking by specifying that the current unit must be initialized
after the unit that supplies the corresponding import. Each tagged-sig-id in an init-depend declaration must have a corresponding import in the import clause.

```
(define-signature id extension-decl
  (sig-elem ...))
```

```
extension-decl =
  | extends sig-id

sig-elem = id
  | (define-syntaxes (id ...) expr)
  | (define-values (id ...) expr)
  | (define-values-for-export (id ...) expr)
  | (contracted [id contract] ...)
  | (open sig-spec)
  | (struct id (field ...) struct-option ...)
  | (sig-form-id . datum)

field = id
  | [id #:mutable]
```

```
struct-option = #:mutable
  | #:constructor-name constructor-id
  | #:extra-constructor-name constructor-id
  | #:omit-constructor
  | #:omit-define-syntaxes
  | #:omit-define-values
```

Binds an identifier to a signature that specifies a group of bindings for import or export:

- Each `id` in a signature declaration means that a unit implementing the signature must supply a variable definition for the `id`. That is, `id` is available for use in units importing the signature, and `id` must be defined by units exporting the signature.

- Each `define-syntaxes` form in a signature declaration introduces a macro that is available for use in any unit that imports the signature. Free variables in the definition’s `expr` refer to other identifiers in the signature first, or the context of the `define-signature` form if the signature does not include the identifier.

- Each `define-values` form in a signature declaration introduces code that effectively prefixes every unit that imports the signature. Free variables in the definition’s `expr` are treated the same as for `define-syntaxes`.

- Each `define-values-for-export` form in a signature declaration introduces code that effectively suffixes every unit that exports the signature. Free variables in the definition’s `expr` are treated the same as for `define-syntaxes`. 

595
• Each contracted form in a signature declaration means that a unit exporting the signature must supply a variable definition for each id in that form. If the signature is imported, then uses of id inside the unit are protected by the appropriate contracts using the unit as the negative blame. If the signature is exported, then the exported values are protected by the appropriate contracts which use the unit as the positive blame, but internal uses of the exported identifiers are not protected. Variables in the contract expressions are treated the same as for define-syntaxes.

• Each (open sig-spec) adds to the signature everything specified by sig-spec.

• Each (struct id (field ...) struct-option ...) adds all of the identifiers that would be bound by (struct id (field ...) field-option ...), where the extra option #:omit-constructor omits the constructor identifier.

• Each (sig-form-id . datum) extends the signature in a way that is defined by sig-form-id, which must be bound by define-signature-form. One such binding is for struct/ctc.

When a define-signature form includes an extends clause, then the define signature automatically includes everything in the extended signature. Furthermore, any implementation of the new signature can be used as an implementation of the extended signature.

(only sig-spec id ...)

Allowed only in a sig-spec; see unit.

(except sig-spec id ...)

Allowed only in a sig-spec; see unit.
(rename sig-spec (id id) ...)

Allowed only in a sig-spec; see unit.

(prefix id sig-spec)

Allowed only in a sig-spec; see unit.

(import tagged-sig-spec ...)

Allowed only in certain forms; see, for example, unit.

(export tagged-sig-spec ...)

Allowed only in certain forms; see, for example, unit.

(link linkage-decl ...)

Allowed only in certain forms; see, for example, compound-unit.

(tag id sig-spec)

(tag id sig-id)

Allowed only in certain forms; see, for example, unit.

(init-depend tagged-sig-id ...)

Allowed only in a init-depend-decl; see unit.

extends

Allowed only within define-signature.

7.2 Invoking Units

(invoke-unit unit-expr)

(invoke-unit unit-expr (import tagged-sig-spec ...))
Invokes the unit produced by `unit-expr`. For each of the unit’s imports, the invoke-unit expression must contain a `tagged-sig-spec` in the import clause; see `unit` for the grammar of `tagged-sig-spec`. If the unit has no imports, the import clause can be omitted.

When no `tagged-sig-specs` are provided, `unit-expr` must produce a unit that expects no imports. To invoke the unit, all bindings are first initialized to the `#<undefined>` value. Next, the unit’s body definitions and expressions are evaluated in order; in the case of a definition, evaluation sets the value of the corresponding variable(s). Finally, the result of the last expression in the unit is the result of the `invoke-unit` expression.

Each supplied `tagged-sig-spec` takes bindings from the surrounding context and turns them into imports for the invoked unit. The unit need not declare an import for every provided `tagged-sig-spec`, but one `tagged-sig-spec` must be provided for each declared import of the unit. For each variable identifier in each provided `tagged-sig-spec`, the value of the identifier’s binding in the surrounding context is used for the corresponding import in the invoked unit.

```lisp
(define-values/invoke-unit unit-expr
  (import tagged-sig-spec ...)
  (export tagged-sig-spec ...))
```

Like `invoke-unit`, but the values of the unit’s exports are copied to new bindings.

The unit produced by `unit-expr` is linked and invoked as for `invoke-unit`. In addition, the export clause is treated as a kind of import into the local definition context. That is, for every binding that would be available in a unit that used the export clause’s `tagged-sig-spec` as an import, a definition is generated for the context of the `define-values/invoke-unit` form.

### 7.3 Linking Units and Creating Compound Units

```lisp
(compound-unit
  (import link-binding ...)
  (export tagged-link-id ...)
  (link linkage-decl ...))
```

The `link-binding` is:

```lisp
link-binding = (link-id : tagged-sig-id)
```

The `tagged-link-id` is:

```lisp
tagged-link-id = (tag id link-id)
  | link-id
```

The `linkage-decl` is:

```lisp
linkage-decl = ((link-binding ...) unit-expr tagged-link-id ...)
```

Links several units into one new compound unit without immediately invoking any of the...
linked units. The unit-exprs in the link clause determine the units to be linked in creating the compound unit. The unit-exprs are evaluated when the compound-unit form is evaluated.

The import clause determines the imports of the compound unit. Outside the compound unit, these imports behave as for a plain unit; inside the compound unit, they are propagated to some of the linked units. The export clause determines the exports of the compound unit. Again, outside the compound unit, these exports are treated the same as for a plain unit; inside the compound unit, they are drawn from the exports of the linked units. Finally, the left-hand and right-hand parts of each declaration in the link clause specify how the compound unit’s imports and exports are propagated to the linked units.

Individual elements of an imported or exported signature are not available within the compound unit. Instead, imports and exports are connected at the level of whole signatures. Each specific import or export (i.e., an instance of some signature, possibly tagged) is given a link-id name. Specifically, a link-id is bound by the import clause or the left-hand part of a declaration in the link clause. A bound link-id is referenced in the right-hand part of a declaration in the link clause or by the export clause.

The left-hand side of a link declaration gives names to each expected export of the unit produced by the corresponding unit-expr. The actual unit may export additional signatures, and it may export an extension of a specific signature instead of just the specified one. If the unit does not export one of the specified signatures (with the specified tag, if any), the exn:fail:contract exception is raised when the compound-unit form is evaluated.

The right-hand side of a link declaration specifies the imports to be supplied to the unit produced by the corresponding unit-expr. The actual unit may import fewer signatures, and it may import a signature that is extended by the specified one. If the unit imports a signature (with a particular tag) that is not included in the supplied imports, the exn:fail:contract exception is raised when the compound-unit form is evaluated. Each link-id supplied as an import must be bound either in the import clause or in some declaration within the link clause.

The order of declarations in the link clause determines the order of invocation of the linked units. When the compound unit is invoked, the unit produced by the first unit-expr is invoked first, then the second, and so on. If the order specified in the link clause is inconsistent with init-depend declarations of the actual units, then the exn:fail:contract exception is raised when the compound-unit form is evaluated.

### 7.4 Inferred Linking
(define-unit unit-id
  (import tagged-sig-spec ...)  
  (export tagged-sig-spec ...)  
  init-depends-decl  
  unit-body-expr-or-defn
  ...)

Binds unit-id to both a unit and static information about the unit.

Evaluating a reference to a unit-id bound by define-unit produces a unit, just like evaluating an id bound by (define id (unit ...)). In addition, however, unit-id can be used in compound-unit/infer. See unit for information on tagged-sig-spec, init-depends-decl, and unit-body-expr-or-defn.

(compound-unit/infer
  (import tagged-infer-link-import ...)  
  (export tagged-infer-link-export ...)  
  (link infer-linkage-decl ...) )

  tagged-infer-link-import = tagged-sig-id
                  | (link-id : tagged-sig-id)

  tagged-infer-link-export = (tag id infer-link-export)
                  | infer-link-export

                 infer-link-export = link-id
                    | sig-id

                 infer-linkage-decl = ((link-binding ...) unit-id
                                         tagged-link-id ...)  
                    | unit-id

Like compound-unit. Syntactically, the difference between compound-unit and compound-unit/infer is that the unit-expr for a linked unit is replaced with a unit-id, where a unit-id is bound by define-unit (or one of the other unit-binding forms that we introduce later in this section). Furthermore, an import can name just a sig-id without locally binding a link-id, and an export can be based on a sig-id instead of a link-id, and a declaration in the link clause can be simply a unit-id with no specified exports or imports.

The compound-unit/infer form expands to compound-unit by adding sig-ids as needed to the import clause, by replacing sig-ids in the export clause by link-ids, and by completing the declarations of the link clause. This completion is based on static information associated with each unit-id. Links and exports can be inferred when all signatures exported by the linked units are distinct from each other and from all imported
signatures, and when all imported signatures are distinct. Two signatures are *distinct* only if they share no ancestor through extends.

The long form of a link declaration can be used to resolve ambiguity by giving names to some of a unit’s exports and supplying specific bindings for some of a unit’s imports. The long form need not name all of a unit’s exports or supply all of a unit’s imports if the remaining parts can be inferred.

When a unit declares initialization dependencies, compound-unit/infer checks that the link declaration is consistent with those dependencies, and it reports a syntax error if not.

Like compound-unit, the compound-unit/infer form produces a (compound) unit without statically binding information about the result unit’s imports and exports. That is, compound-unit/infer consumes static information, but it does not generate it. Two additional forms, define-compound-unit and define-compound-unit/infer, generate static information (where the former does not consume static information).

Changed in version 6.1.1.8 of package base: Added static checking of the link clause with respect to declared initialization dependencies.

```
(define-compound-unit id
  (import link-binding ...)
  (export tagged-link-id ...)
  (link linkage-decl ...))
```

Like compound-unit, but binds static information about the compound unit like define-unit, including the propagation of initialization-dependency information (on remaining imports) from the linked units.

```
(define-compound-unit/infer id
  (import link-binding ...)
  (export tagged-infer-link-export ...)
  (link infer-linkage-decl ...))
```

Like compound-unit/infer, but binds static information about the compound unit like define-compound-unit.

```
(define-unit-binding unit-id
  unit-expr
  (import tagged-sig-spec ...+)
  (export tagged-sig-spec ...+)
  init-depends-decl)
```

Like define-unit, but the unit implementation is determined from an existing unit produced by unit-expr. The imports and exports of the unit produced by unit-expr must be consistent with the declared imports and exports, otherwise the exn:fail:contract exception is raised when the define-unit-binding form is evaluated.
(invoke-unit/infer unit-spec)

unit-spec = unit-id
  | (link link-unit-id ...)

Like invoke-unit, but uses static information associated with unit-id to infer which imports must be assembled from the current context. If given a link form containing multiple link-unit-ids, then the units are first linked via define-compound-unit/infer.

(define-values/invoke-unit/infer maybe-exports unit-spec)

maybe-exports =
  | (export tagged-sig-spec ...)

unit-spec = unit-id
  | (link link-unit-id ...)

Like define-values/invoke-unit, but uses static information associated with unit-id to infer which imports must be assembled from the current context and which exports should be bound by the definition. If given a link form containing multiple link-unit-ids, then the units are first linked via define-compound-unit/infer.

7.5 Generating A Unit from Context

(unit-from-context tagged-sig-spec)

Creates a unit that implements an interface using bindings in the enclosing environment. The generated unit is essentially the same as

(unit
  (import)
  (export tagged-sig-spec)
  (define id expr) ...)

for each id that must be defined to satisfy the exports, and each corresponding expr produces the value of id in the environment of the unit-from-context expression. (The unit cannot be written as above, however, since each id definition within the unit shadows the binding outside the unit form.)

See unit for the grammar of tagged-sig-spec.

(define-unit-from-context id tagged-sig-spec)

Like unit-from-context, in that a unit is constructed from the enclosing environment, and like define-unit, in that id is bound to static information to be used later with inference.
7.6 Structural Matching

```
(define-unit/new-import-export
  unit-id
  (import tagged-sig-spec ...)
  (export tagged-sig-spec ...)
  init-depends-decl
  ,((tagged-sig-spec ...) unit-expr tagged-sig-spec))
```

Similar to `unit`, except the body of the unit is determined by an existing unit produced by `unit-expr`. The result is a unit whose implementation is `unit-expr`, but whose imports, exports, and initialization dependencies are as in the `unit/new-import-export` form (instead of as in the unit produced by `unit-expr`).

The final clause of the `unit/new-import-export` form determines the connection between the old and new imports and exports. The connection is similar to the way that `compound-unit` propagates imports and exports; the difference is that the connection between import and the right-hand side of the link clause is based on the names of elements in signatures, rather than the names of the signatures. That is, a `tagged-sig-spec` on the right-hand side of the link clause need not appear as a `tagged-sig-spec` in the import clause, but each of the bindings implied by the linking `tagged-sig-spec` must be implied by some `tagged-sig-spec` in the import clause. Similarly, each of the bindings implied by an export `tagged-sig-spec` must be implied by some left-hand-side `tagged-sig-spec` in the linking clause.

```
(define-unit/new-import-export unit-id
  (import tagged-sig-spec ...)
  (export tagged-sig-spec ...)
  init-depends-decl
  ,((tagged-sig-spec ...) unit-expr tagged-sig-spec))
```

Like `unit/new-import-export`, but binds static information to `unit-id` like `define-unit`.

```
(define-unit/s name-id
  (import tagged-sig-spec ...)
  (export tagged-sig-spec ...)
  init-depends-decl
  unit-id)
```

Like `unit/new-import-export`, but the linking clause is inferred, so `unit-id` must have the appropriate static information.
Like unit/s, but binds static information to name-id like define-unit.

### 7.7 Extending the Syntax of Signatures

```scheme
(define-signature-form sig-form-id expr)
(define-signature-form (sig-form-id id) body ...+)
```

Binds **sig-form-id** for use within a define-signature form.

In the first form, the result of **expr** must be a transformer procedure. In the second form, **sig-form-id** is bound to a transformer procedure whose argument is **id** and whose body is the **bodys**. The result of the transformer must be a list of syntax objects, which are substituted for a use of **sig-form-id** in a define-signature expansion. (The result is a list so that the transformer can produce multiple declarations; define-signature has no splicing begin form.)

```scheme
(struct/ctc id ([field contract-expr] ...) struct-option ...) struct-option = #:mutable | #:omit-constructor | #:omit-define-syntaxes | #:omit-define-values
```

For use with define-signature. The struct/ctc form works similarly to struct, but the constructor, predicate, field accessors, and field mutators are contracted appropriately.

### 7.8 Unit Utilities

```scheme
(unit? v) -> boolean? v : any/c
```

Returns #t if **v** is a unit, #f otherwise.

```scheme
(provide-signature-elements sig-spec ...)
```

Expands to a provide of all identifiers implied by the **sig-specs**. See unit for the grammar of **sig-spec**.

---

604
7.9 Unit Contracts

A unit contract wraps a unit and checks both its imported and exported identifiers to ensure that they match the appropriate contracts. This allows the programmer to add contract checks to a single unit value without adding contracts to the imported and exported signatures.

The unit value must import a subset of the import signatures and export a superset of the export signatures listed in the unit contract. Additionally, the unit value must declare initialization dependencies that are a subset of those specified in the unit contract. Any identifier which is not listed for a given signature is left alone. Variables used in a given contract expression first refer to other variables in the same signature, and then to the context of the unit/c expression. If a body contract is specified then the result of invoking the unit value is wrapped with the given contract, if no body contract is supplied then no wrapping occurs when the unit value is invoked.
The `define-unit/contract` form defines a unit compatible with link inference whose imports and exports are contracted with a unit contract. The unit name is used for the positive blame of the contract.

### 7.10 Single-Unit Modules

When `racket/unit` is used as a language name with `#lang`, the module body is treated as a unit body. The body must match the following `module-body` grammar:

\[
\text{module-body} = \text{require-decl} \ldots \\
\quad (\text{import} \ tagged-sig-expr \ldots) \\
\quad (\text{export} \ tagged-sig-expr \ldots) \\
\quad \text{init-depends-decl} \\
\quad \text{unit-body-expr-or-defn} \\
\ldots
\]

\[
\text{require-decl} = (\text{require} \ \text{require-spec} \ldots) \\
\quad | (\text{begin} \ \text{require-decl} \ldots) \\
\quad | \text{derived-require-form}
\]

After any number of `require-decls`, the content of the module is the same as a unit body.

The resulting unit is exported as `base@`, where `base` is derived from the enclosing module’s name (i.e., its symbolic name, or its path without the directory and file suffix). If the module name ends in `-unit`, then `base` corresponds to the module name before `-unit`. Otherwise, the module name serves as `base`.

### 7.11 Single-Signature Modules

`#lang racket/signature` package: `base`

The `racket/signature` language treats a module body as a unit signature.

The body must match the following `module-body` grammar:

\[
\text{module-body} = (\text{require} \ \text{require-spec} \ldots) \ldots \text{sig-spec} \ldots
\]

See §7.1 “Creating Units” for the grammar of `sig-spec`. Unlike the body of a `racket/unit` module, a `require` in a `racket/signature` module must be a literal use of `require`.

The resulting signature is exported as `base^`, where `base` is derived from the enclosing module’s name (i.e., its symbolic name, or its path without the directory and file suffix). If the module name ends in `-sig`, then `base` corresponds to the module name before `-sig`. 

606
Otherwise, the module name serves as `base`.

A `struct` form as a `sig-spec` is consistent with the definitions introduced by `define-struct`, as opposed to definitions introduced `struct`. (That behavior was originally a bug, but it is preserved for compatibility.)

### 7.12 Transformer Helpers

```scheme
(require racket/unit-exptime) package:base
```

The `racket/unit-exptime` library provides procedures that are intended for use by macro transformers. In particular, the library is typically imported using `for-syntax` into a module that defines macro with `define-syntax`.

```scheme
(unit-static-signatures unit-identifier err-syntax) → (list/c (cons/c (or/c symbol? #f) identifier?))
      (list/c (cons/c (or/c symbol? #f) identifier?))

unit-identifier : identifier?
err-syntax : syntax?
```

If `unit-identifier` is bound to static unit information via `define-unit` (or other such forms), the result is two values. The first value is for the unit’s imports, and the second is for the unit’s exports. Each result value is a list, where each list element pairs a symbol or `#f` with an identifier. The symbol or `#f` indicates the import’s or export’s tag (where `#f` indicates no tag), and the identifier indicates the binding of the corresponding signature.

If `unit-identifier` is not bound to static unit information, then the `exn:fail:syntax` exception is raised. In that case, the given `err-syntax` argument is used as the source of the error, where `unit-identifier` is used as the detail source location.

```scheme
(signature-members sig-identifier err-syntax) → (or/c identifier? #f)
      (listof identifier?)
      (listof identifier?)
      (listof identifier?)

sig-identifier : identifier?
err-syntax : syntax?
```

If `sig-identifier` is bound to static unit information via `define-signature` (or other such forms), the result is four values:
• an identifier or #f indicating the signature (of any) that is extended by the sig-identifier binding;

• a list of identifiers representing the variables supplied/required by the signature;

• a list of identifiers for variable definitions in the signature (i.e., variable bindings that are provided on import, but not defined by units that implement the signature); and

• a list of identifiers with syntax definitions in the signature.

Each of the result identifiers is given a lexical context that is based on sig-identifier, so the names are suitable for reference or binding in the context of sign-identifier.

If sig-identifier is not bound to a signature, then the exn:fail:syntax exception is raised. In that case, the given err-syntax argument is used as the source of the error, where sig-identifier is used as the detail source location.

```lisp
(unit-static-init-dependencies unit-identifier err-syntax)
→ (list/c (cons/c (or/c symbol? #f)
               identifier?))
unit-identifier : identifier?
err-syntax : syntax?
```

If unit-identifier is bound to static unit information via define-unit (or other such forms), the result is a list of pairs. Each pair combines a tag (or #f for no tag) and a signature name, indicating an initialization dependency of the unit on the specified import (i.e., the same tag and signature are included in the first result from unit-static-signatures).

If unit-identifier is not bound to static unit information, then the exn:fail:syntax exception is raised. In that case, the given err-syntax argument is used as the source of the error, where unit-identifier is used as the detail source location.

Added in version 6.1.1.8 of package base.
8 Contracts

The contract system guards one part of a program from another. Programmers specify the behavior of a module’s exports via (provide (contract-out ....)), and the contract system enforces those constraints.

(require racket/contract) package: base

The bindings documented in this section are provided by the racket/contract and racket libraries, but not racket/base.

Contracts come in two forms: those constructed by the various operations listed in this section of the manual, and various ordinary Racket values that double as contracts, including

- symbols, booleans, keywords, and null, which are treated as contracts that recognize themselves, using eq?,
- strings, byte strings, characters, +nan.0, and +nan.0, which are treated as contracts that recognize themselves using equal?,
- numbers (except +nan.0 and +nan.0), which are treated as contracts that recognize themselves using =,
- regular expressions, which are treated as contracts that recognize byte strings and strings that match the regular expression, and
- predicates: any procedure of arity 1 is treated as a predicate. During contract checking, it is applied to the values that appear and should return #f to indicate that the contract failed, and anything else to indicate it passed.

Contract combinators are functions such as -> and listof that take contracts and produce other contracts.

Contracts in Racket are subdivided into three different categories:

- Flat contracts can be fully checked immediately for a given value. These kinds of contracts are essentially predicate functions. Using flat-contract-predicate, you can extract the predicate from an arbitrary flat contract; some flat contracts can be applied like functions, in which case they accept a single argument and return #t or #f to indicate if the given value would be accepted by the contract. All of the flat contracts returned by functions in this library can be used directly as predicates, but ordinary Racket values that double as flat contracts (e.g., numbers or symbols) cannot. The function flat-contract? recognizes a flat contract.
- Chaperone contracts may wrap a value in such a way that it signals contract violations later, as the value is used, but are guaranteed to not otherwise change behavior. For
example, a function contract wraps a function value and later checks inputs and outputs; any properties that the function value had before being wrapped by the contract are preserved by the contract wrapper.

All flat contracts may be used where chaperone contracts are expected (but not vice-versa). The function `chaperone-contract?` recognizes a chaperone contract.

- **Impersonator contracts** may wrap values and do not provide any guarantees. Impersonator contracts may hide properties of values, or even make them completely opaque (e.g. `new-\$/c`).

All contracts may be used where impersonator contracts are expected. The function `impersonator-contract?` recognizes an impersonator contract.

For more about this hierarchy, see the section [§14.5 “Impersonators and Chaperones”] as well as a research paper [Strickland12] on chaperones, impersonators, and how they can be used to implement contracts.

Changed in version 6.1.1.8 of package base: Changed `+nan.0` and `+nan.0` to be `equal?`-based contracts.

### 8.1 Data-structure Contracts

```
(flat-contract-with-explanation get-explanation
  [#:name name])

-> flat-contract?
get-explanation : (-> any/c (or/c boolean? (-> blame? any)))
name : any/c = (object-name get-explanation)
```

Provides a way to use flat contracts that, when a contract fails, provide more information about the failure.

If `get-explanation` returns a boolean, then that boolean value is treated as the predicate in a flat contract. If it returns a procedure, then it is treated similarly to returning `#f`, except the result procedure is called to actually signal the contract violation.

The `name` argument is used as the name of the contract; it defaults to the name of the `get-explanation` function.

```
(flat-contract-with-explanation
  (\ (val)
    (cond
      [(even? val) #t]
      [else
        (\ (blame)
          (define more-information ...do-some-complex-computation-here...))])

```

610
(raise-blame-error blame val
  '(expected: "an even num-
  ber" given: "~e"
    "and, here is more help:
    ~s")
  val more-information)))))

(flat-named-contract name
  flat-contract
  [generator])  \rightarrow flat-contract?

name : any/c
flat-contract : flat-contract?
generator : (or/c #f (\rightarrow contract (\rightarrow int? any))) = #f

Produces a flat contract like flat-contract, but with the name name.

For example,

(define/contract i
  (flat-named-contract
    'odd-integer
    (lambda (x) (and (integer? x) (odd? x))))
  2)

The generator argument adds a generator for the flat-named-contract. See contract-
random-generate for more information.

any/c : flat-contract?

A flat contract that accepts any value.

When using this contract as the result portion of a function contract, consider using any
instead; using any leads to better memory performance, but it also allows multiple results.

none/c : flat-contract?

A flat contract that accepts no values.

(\or/c contract \ldots) \rightarrow contract?
contract : contract?

Takes any number of contracts and returns a contract that accepts any value that any one of
the contracts accepts individually.
The `or/c` result tests any value by applying the contracts in order, from left to right, with the exception that it always moves the non-flat contracts (if any) to the end, checking them last. Thus, a contract such as `(or/c (not/c real?) positive?)` is guaranteed to only invoke the `positive?` predicate on real numbers.

If all of the arguments are procedures or flat contracts, the result is a flat contract. If only one of the arguments is a higher-order contract, the result is a contract that just checks the flat contracts and, if they don’t pass, applies the higher-order contract.

If there are multiple higher-order contracts, `or/c` uses `contract-first-order-passes?` to distinguish between them. More precisely, when an `or/c` is checked, it first checks all of the flat contracts. If none of them pass, it calls `contract-first-order-passes?` with each of the higher-order contracts. If only one returns true, `or/c` uses that contract. If none of them return true, it signals a contract violation. If more than one returns true, it also signals a contract violation. For example, this contract

```
(or/c (-> number? number?)
     (-> string? string? string?))
```

does not accept a function like this one: `(lambda args ...)` since it cannot tell which of the two arrow contracts should be used with the function.

If all of its arguments are `list-contract?s`, then `or/c` returns a `list-contract?`.

```
(first-or/c contract ...) -> contract?
contract : contract?
```

Takes any number of contracts and returns a contract that accepts any value that any one of the contracts accepts individually.

The `first-or/c` result tests any value by applying the contracts in order from left to right. Thus, a contract such as `(first-or/c (not/c real?) positive?)` is guaranteed to only invoke the `positive?` predicate on real numbers.

If all of the arguments are procedures or flat contracts, the result is a flat contract and similarly if all of the arguments are chaperone contracts the result is too. Otherwise, the result is an impersonator contract.

If there are multiple higher-order contracts, `first-or/c` uses `contract-first-order-passes?` to distinguish between them. More precisely, when an `first-or/c` is checked, it checks the first order passes of the first contract against the value. If it succeeds, then it uses only that contract. If it fails, then it moves to the second contract, continuing until it finds one of the contracts where the first order check succeeds. If none of them do, a contract violation is signaled.

For example, this contract
(first-or/c (→ number? number?)
  (→ string? string? string?))

accepts the function (λ args 0), applying the (→ number? number?) contract to the function because it comes first, even though (→ string? string? string?) also applies.

If all of its arguments are list-contract?, then first-or/c returns a list-contract?.

(and/c contract ...) → contract?
  contract : contract?

Takes any number of contracts and returns a contract that accepts any value that satisfies all of the contracts simultaneously.

If all of the arguments are procedures or flat contracts, the result is a flat contract.

The contract produced by and/c tests any value by applying the contracts in order, from left to right.

This means that and/c can be used to guard predicates that are not total in contracts. For example, this contract is well-behaved, correctly blaming the definition of whoops-not-a-number for not being a number:

Example:

> (define/contract whoops-not-a-number
|    (and/c real? even?)
|      "four")
whoops-not-a-number: broke its own contract
  promised: real?
  produced: "four"
  in: an and/c case of
    (and/c real? even?)
  contract from:
    (definition whoops-not-a-number)
  blaming: (definition whoops-not-a-number)
    (assuming the contract is correct)
  at: eval:2:0

but if the arguments to and/c are reversed, then the contract itself raises an error:

Example:

> (define/contract whoops-not-a-number
(and/c even? real?)
"four")

even?: contract violation
expected: integer?
given: "four"

If more than one of the contracts are not flat contracts, then the order in which the higher-order parts of the contract are tested can be counter-intuitive. As an example, consider this function that uses and/c in a higher-order manner with contracts that always succeed, but that print when they are called, in order for us to see the order in which they are called.

Examples:

> (define ((show-me n) x)
  (printf "show-me ~a\n" n)
  #t)
> (define/contract identity-with-complex-printing-contract
  (and/c (-> (show-me 4) (show-me 5))
     (-> (show-me 3) (show-me 6))
     (-> (show-me 2) (show-me 7))
     (-> (show-me 1) (show-me 8)))
  (λ (x) x))
> (identity-with-complex-printing-contract 101)
show-me 1
show-me 2
show-me 3
show-me 4
show-me 5
show-me 6
show-me 7
show-me 8
101

The checking order is just like the usual ordering when a contract is double-wrapped. The contract that is first put on has its domain checked second but its range checked first and we see a similar pattern here in this example, because and/c simply applies the contracts in order.

(not/c flat-contract) → flat-contract?
flat-contract : flat-contract?

Accepts a flat contract or a predicate and returns a flat contract that checks the inverse of the argument.

(=/c z) → flat-contract?
z : real?
Returns a flat contract that requires the input to be a number and = to z.

```scheme
(</c n) → flat-contract?
  n : real?
```

Returns a flat contract that requires the input to be a number and < than n.

```scheme
(>/c n) → flat-contract?
  n : real?
```

Like </c, but for >.

```scheme
(<=/c n) → flat-contract?
  n : real?
```

Like </c, but for <=.

```scheme
(>=/c n) → flat-contract?
  n : real?
```

Like </c, but for >=.

```scheme
(between/c n m) → flat-contract?
  n : real?
  m : real?
```

Returns a flat contract that requires the input to be a real number between n and m or equal to one of them.

```scheme
(real-in n m) → flat-contract?
  n : real?
  m : real?
```

An alias for between/c.

```scheme
(integer-in j k) → flat-contract?
  j : (or/c exact-integer? #f)
  k : (or/c exact-integer? #f)
```

Returns a flat contract that requires the input to be an exact integer between j and k, inclusive. If either j or k is #f, then the range is unbounded on that end.

Examples:
> (define/contract two-digit-number
  (integer-in 10 99)
  23)
> (define/contract not-a-two-digit-number
  (integer-in 10 99)
  124)

not-a-two-digit-number: broke its own contract
  promised: (integer-in 10 99)
  produced: 124
  in: (integer-in 10 99)
  contract from:
    (definition not-a-two-digit-number)
  blaming: (definition not-a-two-digit-number)
    (assuming the contract is correct)
  at: eval:3:0

> (define/contract negative-number
  (integer-in #f -1)
  -4)
> (define/contract not-a-negative-number
  (integer-in #f -1)
  4)

not-a-negative-number: broke its own contract
  promised: (integer-in #f -1)
  produced: 4
  in: (integer-in #f -1)
  contract from:
    (definition not-a-negative-number)
  blaming: (definition not-a-negative-number)
    (assuming the contract is correct)
  at: eval:5:0

Changed in version 6.8.0.2 of package base: Allow j and k to be #f

(char-in a b) -> flat-contract?
a : char?
b : char?

Returns a flat contract that requires the input to be a character whose code point number is between the code point numbers of a and b, inclusive.

natural-number/c : flat-contract?

A flat contract that requires the input to be an exact non-negative integer.

(string-len/c len) -> flat-contract?
len : real?
Returns a flat contract that recognizes strings that have fewer than \texttt{len} characters.

\begin{verbatim}
false/c : flat-contract?
\end{verbatim}

An alias for \texttt{#f} for backwards compatibility.

\begin{verbatim}
printable/c : flat-contract?
\end{verbatim}

A flat contract that recognizes values that can be written out and read back in with \texttt{write} and \texttt{read}.

\begin{verbatim}
(one-of/c v ...+) → flat-contract?
v : any/c
\end{verbatim}

Accepts any number of atomic values and returns a flat contract that recognizes those values, using \texttt{eqv?} as the comparison predicate. For the purposes of \texttt{one-of/c}, atomic values are defined to be: characters, symbols, booleans, \texttt{null}, keywords, numbers, \texttt{#<void>}, and \texttt{#<undefined>}.

This is a backwards compatibility contract constructor. If neither \texttt{#<void>} nor \texttt{#<undefined>} are arguments, it simply passes its arguments to \texttt{or/c}.

\begin{verbatim}
(symbols sym ...+) → flat-contract?
sym : symbol?
\end{verbatim}

Accepts any number of symbols and returns a flat contract that recognizes those symbols.

This is a backwards compatibility constructor; it merely passes its arguments to \texttt{or/c}.

\begin{verbatim}
(vectorof c [#:immutable immutable #:flat? flat? #:eager eager]) → contract?
c : contract?
immutable : (or/c #t #f 'dont-care) = 'dont-care
flat? : boolean? = #f
eager : (or/c #t #f exact-nonnegative-integer?) = #t
\end{verbatim}

Returns a contract that recognizes vectors. The elements of the vector must match \texttt{c}.

If the \texttt{flat?} argument is \texttt{#t}, then the resulting contract is a flat contract, and the \texttt{c} argument must also be a flat contract. Such flat contracts will be unsound if applied to mutable vectors, as they will not check future operations on the vector.

If the \texttt{immutable} argument is \texttt{#t} and the \texttt{c} argument is a flat contract and the \texttt{eager} argument is \texttt{#t}, the result will be a flat contract. If the \texttt{c} argument is a chaperone contract, then the result will be a chaperone contract.
If the `eager` argument is `#t`, then immutable vectors are checked eagerly when `c` is a flat contract. If the `eager` argument is a number `n`, then immutable vectors are checked eagerly when `c` is a flat contract and the length of the vector is less than or equal to `n`.

When a higher-order `vectorof` contract is applied to a vector, the result is not `eq?` to the input. The result will be a copy for immutable vectors and a chaperone or impersonator of the input for mutable vectors, unless the `c` argument is a flat contract and the vector is immutable, in which case the result is the original vector.

Changed in version 6.3.0.5 of package `base`: Changed flat vector contracts to not copy immutable vectors.
Changed in version 6.7.0.3: Added the `#:eager` option.

```
(vector-immutableof c) -> contract?
  c : contract?
```

Returns the same contract as `(vectorof c #:immutable #t)`. This form exists for backwards compatibility.

```
(vector/c c ...
  #:immutable immutable
  #:flat? flat?) -> contract?
  c : contract?
  immutable : (or/c #t #f 'dont-care) = 'dont-care
  flat? : boolean? = #f
```

Returns a contract that recognizes vectors whose lengths match the number of contracts given. Each element of the vector must match its corresponding contract.

If the `flat?` argument is `#t`, then the resulting contract is a flat contract, and the `c` arguments must also be flat contracts. Such flat contracts will be unsound if applied to mutable vectors, as they will not check future operations on the vector.

If the `immutable` argument is `#t` and the `c` arguments are flat contracts, the result will be a flat contract. If the `c` arguments are chaperone contracts, then the result will be a chaperone contract.

When a higher-order `vector/c` contract is applied to a vector, the result is not `eq?` to the input. The result will be a copy for immutable vectors and a chaperone or impersonator of the input for mutable vectors.

```
(vector-immutable/c c ...) -> contract?
  c : contract?
```

Returns the same contract as `(vector/c c ... #:immutable #t)`. This form exists for reasons of backwards compatibility.
(box/c in-c
[c
  #:immutable immutable
  #:flat? flat?]) -> contract?
in-c : contract?
c : contract? = in-c
immutable : (or/c #t #f 'dont-care) = 'dont-care
flat? : boolean? = #f

Returns a contract that recognizes boxes. The content of the box must match c, and mutations on mutable boxes must match in-c.

If the flat? argument is #t, then the resulting contract is a flat contract, and the out argument must also be a flat contract. Such flat contracts will be unsound if applied to mutable boxes, as they will not check future operations on the box.

If the immutable argument is #t and the c argument is a flat contract, the result will be a flat contract. If the c argument is a chaperone contract, then the result will be a chaperone contract.

When a higher-order box/c contract is applied to a box, the result is not eq? to the input. The result will be a copy for immutable boxes and either a chaperone or impersonator of the input for mutable boxes.

(box-immutable/c c) -> contract?
c : contract?

Returns the same contract as (box/c c #:immutable #t). This form exists for reasons of backwards compatibility.

(listof c) -> list-contract?
c : contract?

Returns a contract that recognizes a list whose every element matches the contract c. Beware that when this contract is applied to a value, the result is not necessarily eq? to the input.

Examples:
> (define/contract some-numbers
  (listof number?)
  (list 1 2 3))
> (define/contract just-one-number
  (listof number?)
  11)
just-one-number: broke its own contract
promised: list?
produced: 11
in: (listof number?)
contract from: (definition just-one-number)
blaming: (definition just-one-number)
(assuming the contract is correct)
at: eval:3:0

(non-empty-listof c) → list-contract?
c : contract?

Returns a contract that recognizes non-empty lists whose elements match the contract c. Beware that when this contract is applied to a value, the result is not necessarily eq? to the input.

Examples:

> (define/contract some-numbers
   (non-empty-listof number?)
   (list 1 2 3))
> (define/contract not-enough-numbers
   (non-empty-listof number?)
   (list))
not-enough-numbers: broke its own contract
promised: "(and/c list? pair?)"
produced: ()
in: (non-empty-listof number?)
contract from:
   (definition not-enough-numbers)
blaming: (definition not-enough-numbers)
(assuming the contract is correct)
at: eval:3:0

(list*of ele-c [last-c]) → contract?
  ele-c : contract?
  last-c : contract? = ele-c

Returns a contract that recognizes improper lists whose elements match the contract ele-c and whose last position matches last-c. If an improper list is created with cons, then its car position is expected to match ele-c and its cdr position is expected to be (list*of ele-c list-c). Otherwise, it is expected to match last-c. Beware that when this contract is applied to a value, the result is not necessarily eq? to the input.

Examples:
> (define/contract improper-numbers
  (list*of number?)
  (cons 1 (cons 2 3)))
> (define/contract not-improper-numbers
  (list*of number?)
  (list 1 2 3))
not-improper-numbers: broke its own contract
  promised: number?
  produced: '()
  in: an element of
    (list*of number?)
  contract from:
    (definition not-improper-numbers)
  blaming: (definition not-improper-numbers)
  (assuming the contract is correct)
at: eval:3:0

Added in version 6.1.1.1 of package base.
Changed in version 6.4.0.4: Added the last-c argument.

(cons/c car-c cdr-c) → contract?
car-c : contract?
cdr-c : contract?

Produces a contract that recognizes pairs whose first and second elements match car-c and cdr-c, respectively. Beware that when this contract is applied to a value, the result is not necessarily eq? to the input.

If the cdr-c contract is a list-contract?, then cons/c returns a list-contract?.

Examples:

> (define/contract a-pair-of-numbers
  (cons/c number? number?)
  (cons 1 2))
> (define/contract not-a-pair-of-numbers
  (cons/c number? number?)
  (cons #f #t))
not-a-pair-of-numbers: broke its own contract
  promised: number?
  produced: #f
  in: the car of
    (cons/c number? number?)
  contract from:
    (definition not-a-pair-of-numbers)
  blaming: (definition not-a-pair-of-numbers)
(assuming the contract is correct)
at: eval:3:0

Changed in version 6.0.1.13 of package base: Added the list-contract? propagating behavior.

```
(cons/dc [car-id contract-expr] [cdr-id (car-id) contract-expr] cons/dc-option)
(cons/dc [car-id (cdr-id) contract-expr] [cdr-id contract-expr] cons/dc-option)
```

```
cons/dc-option =
  #:flat
  #:chaperone
  #:impersonator
```

Produces a contract that recognizes pairs whose first and second elements match the expressions after car-id and cdr-id, respectively.

In the first case, the contract on the cdr-id portion of the contract may depend on the value in the car-id portion of the pair and in the second case, the reverse is true.

Examples:

```
> (define/contract an-ordered-pair-of-reals
  (cons/dc [hd real?] [tl (hd) (>=/c hd)])
  (cons 1 2))

> (define/contract not-an-ordered-pair-of-reals
  (cons/dc [hd real?] [tl (hd) (>=/c hd)])
  (cons 2 1))
```

not-an-ordered-pair-of-reals: broke its own contract
promised: (>=/c 2)
produced: 1
in: the cdr of
  (cons/dc (hd real?) (tl (hd) (>=/c hd)))
contract from:
  (definition not-an-ordered-pair-of-reals)
blaming: (definition not-an-ordered-pair-of-reals)
  (assuming the contract is correct)
at: eval:3:0

Added in version 6.1.1.6 of package base.

```
(list/c c ...) → list-contract?
c : contract?
```

Produces a contract for a list. The number of elements in the list must match the number of arguments supplied to list/c, and each element of the list must match the corresponding
contract. Beware that when this contract is applied to a value, the result is not necessarily
\(eq?\) to the input.

\[
(*\text{list/c prefix suffix \ldots}) \rightarrow \text{list-contract?}
\]
\[
\begin{align*}
\text{prefix} & : \text{contract?} \\
\text{suffix} & : \text{contract?}
\end{align*}
\]

Produces a contract for a list. The number of elements in the list must be at least as long
as the number of \text{suffix} contracts and the tail of the list must match those contracts, one
for each element. The beginning portion of the list can be arbitrarily long, and each element
must match \text{prefix}.

Beware that when this contract is applied to a value, the result is not necessarily \(eq?\) to the
input.

Examples:

\[
> (\text{define/contract a-list-of-numbers-ending-with-two-integers}
\]
\[
(*\text{list/c number? integer? integer?})
\]
\[
\text{(list 1/2 4/5 0+1i -11 322)}
\]
\[
> (\text{define/contract not-enough-integers-at-the-end}
\]
\[
(*\text{list/c number? integer? integer? integer?})
\]
\[
\text{(list 1/2 4/5 1/2 321 322)}
\]
\[
\text{not-enough-integers-at-the-end: broke its own contract}
\]
\[
\text{promised: integer?}
\]
\[
\text{produced: 1/2}
\]
\[
\text{in: the 3rd to the last element of}
\]
\[
(*\text{list/c number? integer? integer? integer?})
\]
\[
\text{contract from:}
\]
\[
(\text{definition not-enough-integers-at-the-end})
\]
\[
\text{blaming: (definition not-enough-integers-at-the-end)}
\]
\[
(\text{assuming the contract is correct})
\]
\[
\text{at: eval:3:0}
\]

\[
(*\text{syntax/c c \ldots}) \rightarrow \text{flat-contract?}
\]
\[
\begin{align*}
\text{c} & : \text{flat-contract?}
\end{align*}
\]

Produces a flat contract that recognizes syntax objects whose \text{syntax-e content matches c}.

\[
(*\text{struct/c struct-id contract-expr \ldots})
\]

Produces a contract that recognizes instances of the structure type named by \text{struct-id},
and whose field values match the contracts produced by the \text{contract-exprs}.

Contracts for immutable fields must be either flat or chaperone contracts. Contracts for
mutable fields may be impersonator contracts. If all fields are immutable and the \text{contract-exprs}
evaluate to flat contracts, a flat contract is produced. If all the \text{contract-exprs} are
chaperone contracts, a chaperone contract is produced. Otherwise, an impersonator contract is produced.

```
(struct/dc struct-id field-spec ... maybe-inv)
```

```
field-spec = [field-name maybe-lazy contract-expr]
    | [field-name (dep-field-name ...)]
    maybe-lazy
    maybe-contract-type
    maybe-dep-state
    contract-expr

field-name = field-id
    | (#:selector selector-id)
    | (field-id #:parent struct-id)

maybe-lazy =
    | #:lazy

maybe-contract-type =
    | #:flat
    | #:chaperone
    | #:impersonator

maybe-dep-state =
    | #:depends-on-state

maybe-inv =
    | #:inv (dep-field-name ...) invariant-expr
```

Produces a contract that recognizes instances of the structure type named by `struct-id`, and whose field values match the contracts produced by the `field-specs`.

If the `field-spec` lists the names of other fields, then the contract depends on values in those fields, and the `contract-expr` expression is evaluated each time a selector is applied, building a new contract for the fields based on the values of the `dep-field-name` fields (the `dep-field-name` syntax is the same as the `field-name` syntax). If the field is a dependent field and no `contract-type` annotation appears, then it is assumed that the contract is a chaperone, but not always a flat contract (and thus the entire `struct/dc` contract is not a flat contract). If this is not the case, and the contract is always flat then the field must be annotated with the `#:flat`, or the field must be annotated with `#:impersonator` (in which case, it must be a mutable field).

A `field-name` is either an identifier naming a field in the first case, an identifier naming a selector in the second case indicated by the `#:selector` keyword, or a field id for a struct that is a parent of `struct-id`, indicated by the `#:parent` keyword.
If the #:lazy keyword appears, then the contract on the field is checked lazily (only when a selector is applied); #:lazy contracts cannot be put on mutable fields.

If a dependent contract depends on some mutable state, then use the #:depends-on-state keyword argument (if a field’s dependent contract depends on a mutable field, this keyword is automatically inferred). The presence of this keyword means that the contract expression is evaluated each time the corresponding field is accessed (or mutated, if it is a mutable field). Otherwise, the contract expression for a dependent field contract is evaluated when the contract is applied to a value.

If the #:inv clause appears, then the invariant expression is evaluated (and must return a non-#f value) when the contract is applied to a struct.

Contracts for immutable fields must be either flat or chaperone contracts. Contracts for mutable fields may be impersonator contracts. If all fields are immutable and the contract-exprs evaluate to flat contracts, a flat contract is produced. If all the contract-exprs are chaperone contracts, a chaperone contract is produced. Otherwise, an impersonator contract is produced.

As an example, the function bst/c below returns a contract for binary search trees whose values are all between lo and hi. The lazy annotations ensure that this contract does not change the running time of operations that do not inspect the entire tree.

Examples:

```scheme
> (struct bt (val left right))
> (define (bst/c lo hi)
  (or/c #f
    (struct/dc bt
      [val (between/c lo hi)]
      [left (val) #:lazy (bst/c lo val)]
      [right (val) #:lazy (bst/c val hi)])))
> (define/contract not-really-a-bst
  (bst/c -inf.0 +inf.0)
  (bt 5
    (bt 4
      (bt 2 #f #f)
      (bt 6 #f #f))
    #f))
> (bt-right not-really-a-bst)
#f
> (bt-val (bt-left (bt-left not-really-a-bst)))
2
> (bt-right (bt-left not-really-a-bst))
not-really-a-bst: broke its own contract
  promised: (between/c 4 5)
  produced: 6
```
in: the val field of
  a part of the or/c of
  the right field of
  a part of the or/c of
  the left field of
  a part of the or/c of
  (or/c
  #f
  (struct/dc
   bt
   (val (between/c -inf.0 +inf.0))
   (left (val) #:lazy ...)
   (right (val) #:lazy ...)))
contract from: (definition not-really-a-bst)
blaming: (definition not-really-a-bst)
  (assuming the contract is correct)
at: eval:4:0

Changed in version 6.0.1.6 of package base: Added #:inv.

```
(parameter/c in [out]) -> contract?
in : contract?
out : contract? = in
```

Produces a contract on parameters whose values must match out. When the value in the
contracted parameter is set, it must match in.

Examples:

> (define/contract current-snack
  (parameter/c string?)
  (make-parameter "potato-chip")
> (define baked/c
  (flat-named-contract 'baked/c (\(s\) (regexp-match #rx"baked" s))))
> (define/contract current-dinner
  (parameter/c string? baked/c)
  (make-parameter "turkey" (\(s\) (string-append "roasted" s))))
> (current-snack 'not-a-snack)
current-snack: contract violation
  expected: string?
given: 'not-a-snack
in: the parameter of
  (parameter/c string?)
contract from: (definition current-snack)
blaming: top-level
(assuming the contract is correct)
at: eval:2:0

> (parameterize ([current-dinner "tofurkey"])
  (current-dinner))

current-dinner: broke its own contract
promised: baked/c
produced: "roasted tofurkey"
in: the parameter of
  (parameter/c string? baked/c)
contract from: (definition current-dinner)
blaming: (definition current-dinner)
(assuming the contract is correct)
at: eval:4:0

(procedure-arity-includes/c n) → flat-contract?
  n : exact-nonnegative-integer?

Produces a contract for procedures that accept n argument (i.e., the procedure? contract is implied).

[hash/c key
  val
  [#:immutable immutable
   #:flat? flat?]) → contract?
  key : chaperone-contract?
  val : contract?
  immutable : (or/c #t #f 'dont-care) = 'dont-care
  flat? : boolean? = #f

Produces a contract that recognizes hash tables with keys and values as specified by the key and val arguments.

Examples:

> (define/contract good-hash
    (hash/c integer? boolean?)
    (hash 1 #t
       2 #f
       3 #t))

> (define/contract bad-hash
    (hash/c integer? boolean?)
    (hash 1 "elephant"
       2 "monkey"
       3 "manatee"))
bad-hash: broke its own contract
promised: boolean?
produced: "elephant"
in: the values of
(hash/c integer? boolean?)
contract from: (definition bad-hash)
blaming: (definition bad-hash)
(assuming the contract is correct)
at: eval:3:0

There are a number of technicalities that control how hash/c contracts behave.

• If the flat? argument is #t, then the resulting contract is a flat contract, and the key and val arguments must also be flat contracts.

Examples:

> (flat-contract? (hash/c integer? boolean?))
#f
> (flat-contract? (hash/c integer? boolean? #:flat? #t))
#t
> (hash/c integer? (-> integer? integer?) #:flat? #t)
hash/c: contract violation
expected: flat-contract?
given: (-> integer? integer?)

Such flat contracts will be unsound if applied to mutable hash tables, as they will not check future mutations to the hash table.

Examples:

> (define original-h (make-hasheq))
> (define/contract ctc-h
   (hash/c integer? boolean? #:flat? #t)
   original-h)
> (hash-set! original-h 1 "not a boolean")
> (hash-ref ctc-h 1)
"not a boolean"

• If the immutable argument is #t and the key and val arguments are flat-contract?s, the result will be a flat-contract?.

Example:

> (flat-contract? (hash/c integer? boolean? #:immutable #t))
#t
If either the domain or the range is a `chaperone-contract?`, then the result will be a `chaperone-contract?`.

Examples:

> (flat-contract? (hash/c (-> integer? integer?) boolean? #:immutable #t))
  #f
> (chaperone-contract? (hash/c (-> integer? integer?) boolean? #:immutable #t))
  #t

• If the `key` argument is a `chaperone-contract?` but not a `flat-contract?`, then the resulting contract can be applied only to `equal?`-based hash tables.

Example:

> (define/contract h
      (hash/c (-> integer? integer?) any/c)
      (make-hasheq))

  h: broke its own contract; promised `equal?`-based hash table due to higher-order domain contract
  produced: `#hashq()`
  in: (hash/c (-> integer? integer?) any/c)
  contract from: (definition h)
  blaming: (definition h)
  (assuming the contract is correct)
  at: eval:2:0

Also, when such a `hash/c` contract is applied to a hash table, the result is not `eq?` to the input. The result of applying the contract will be a copy for immutable hash tables, and either a chaperone or impersonator of the original hash table for mutable hash tables.

```
(hash/dc [key-id key-contract-expr] [value-id (key-id) value-contract-expr]
         hash/dc-option)

hash/dc-option =
    | #:immutable immutable?-expr hash/dc-option
    | #:kind kind-expr hash/dc-option
```

Creates a contract for `hash?` tables with keys matching `key-contract-expr` and where the contract on the values can depend on the key itself, since `key-id` will be bound to the corresponding key before evaluating the `values-contract-expr`. If `immutable?-expr` is `#t`, then only `immutable?` hashes are accepted. If it is `#f` then `immutable?` hashes are always rejected. It defaults to `'dont-care`, in which case both mutable and immutable hashes are accepted.
If `kind-expr` evaluates to `flat`, then `key-contract-expr` and `value-contract-expr` are expected to evaluate to `flat-contract`s. If it is `chaperone`, then they are expected to be `chaperone-contract`s, and it may also be `impersonator`, in which case they may be any `contract`s. The default is `chaperone`.

Examples:

> (define/contract h
  (hash/dc [k real?] [v (k) (>=/c k)])
  (hash 1 3
        2 4))
> (define/contract h
  (hash/dc [k real?] [v (k) (>=/c k)])
  (hash 3 1
        4 2))

h: broke its own contract
  promised: (>=/c 3)
  produced: 1
  in: the values of
    (hash/dc (k real?) (v (k) (>=/c k)))
  contract from: (definition h)
  blaming: (definition h)
(assuming the contract is correct)
at: eval:3:0

(channel/c val) → contract?
val : contract?

Produces a contract that recognizes channels that communicate values as specified by the `val` argument.

If the `val` argument is a chaperone contract, then the resulting contract is a chaperone contract. Otherwise, the resulting contract is an impersonator contract. When a channel contract is applied to a channel, the resulting channel is not `eq?` to the input.

Examples:

> (define/contract chan
  (channel/c string?)
  (make-channel))
> (thread (λ () (channel-get chan)))
#<thread>
> (channel-put chan 'not-a-string)
chan: contract violation
  expected: string?
given: 'not-a-string
in: (channel/c string?)
contract from: (definition chan)
blaming: top-level
(assuming the contract is correct)
at: eval:2:0

(prompt-tag/c contract ... maybe-call/cc)

maybe-call/cc =
| #:call/cc contract
| #:call/cc (values contract ...)

contract : contract?

Takes any number of contracts and returns a contract that recognizes continuation prompt tags and will check any aborts or prompt handlers that use the contracted prompt tag.

Each contract will check the corresponding value passed to an abort-current-continuation and handled by the handler of a call to call-with-continuation-prompt.

If all of the contracts are chaperone contracts, the resulting contract will also be a chaperone contract. Otherwise, the contract is an impersonator contract.

If maybe-call/cc is provided, then the provided contracts are used to check the return values from a continuation captured with call-with-current-continuation.

Examples:

```
> (define/contract tag
  (prompt-tag/c (- number? string?))
  (make-continuation-prompt-tag))
> (call-with-continuation-prompt
  (lambda ()
    (number->string
     (call-with-composable-continuation
      (lambda (k)
      (abort-current-continuation tag k)))))
  tag
  (lambda (k) (k "not a number")))
tag: contract violation
expected: number?
given: "not a number"
in: the 1st argument of
  (prompt-tag/c
  (- number? string?))
```

631
(continuation-mark-key/c contract) \rightarrow contract?
contract : contract?

Takes a single contract and returns a contract that recognizes continuation marks and will check any mappings of marks to values or any accesses of the mark value.

If the argument contract is a chaperone contract, the resulting contract will also be a chaperone contract. Otherwise, the contract is an impersonator contract.

Examples:

```
> (define/contract mark-key
  (continuation-mark-key/c (-> symbol? (listof symbol?)))
  (make-continuation-mark-key))
> (with-continuation-mark mark-key
  (lambda (s) (append s '(truffle fudge ganache)))
  (let ([mark-value (continuation-mark-set-first
                     (current-continuation-marks) mark-key)])
    (mark-value "chocolate-bar")))
mark-key: contract violation
  expected: symbol?
given: "chocolate-bar"
in: the 1st argument of
  (continuation-mark-key/c
   (-> symbol? (listof symbol?)))
contract from: (definition mark-key)
  blaming: top-level
  (assuming the contract is correct)
at: eval:2:0
```

(evt/c contract ...) \rightarrow chaperone-contract?
contract : chaperone-contract?

Returns a contract that recognizes synchronizable events whose synchronization results are checked by the given contracts.

The resulting contract is always a chaperone contract and its arguments must all be chaperone contracts.

Examples:
> (define/contract my-evt
  [(evt/c evt?)
    always-evt])
> (define/contract failing-evt
  [(evt/c number? number?)
    (alarm-evt (+ (current-inexact-milliseconds) 50))])
> (sync my-evt)
#<always-evt>
> (sync failing-evt)
failing-evt: broke its own contract
  promised: event that produces 2 values
  produced: event that produces 1 values
  in: (evt/c number? number?)
  contract from: (definition failing-evt)
  blaming: (definition failing-evt)
  (assuming the contract is correct)
at: eval:3:0

(assume-contract id flat-contract-expr ...)

Constructs a recursive flat contract. A flat-contract-expr can refer to id to refer recursively to the generated contract.

For example, the contract

(flat-rec-contract sexp
  (cons/c sexp sexp)
  number?
  symbol?)

is a flat contract that checks for (a limited form of) S-expressions. It says that a sexp is either two sexps combined with cons, or a number, or a symbol.

Note that if the contract is applied to a circular value, contract checking will not terminate.

(flat-murec-contract ([id flat-contract-expr ...] ...) body ...+)

A generalization of flat-rec-contract for defining several mutually recursive flat contracts simultaneously. Each id is visible in the entire flat-murec-contract form, and the result of the final body is the result of the entire form.

any

Represents a contract that is always satisfied. In particular, it can accept multiple values. It can only be used in a result position of contracts like ->. Using any elsewhere is a syntax error.
(promise/c c) → contract?
c : contract?

Constructs a contract on a promise. The contract does not force the promise, but when the promise is forced, the contract checks that the result value meets the contract c.

(flat-contract predicate) → flat-contract?
predicate : (-> any/c any/c)

Constructs a flat contract from predicate. A value satisfies the contract if the predicate returns a true value.

This function is a holdover from before predicates could be used directly as flat contracts. It exists today for backwards compatibility.

(flat-contract-predicate v) → (-> any/c any/c)
v : flat-contract?

Extracts the predicate from a flat contract.

Note that most flat contracts can be used directly as predicates, but not all. This function can be used to build predicates for ordinary Racket values that double as contracts, such as numbers and symbols. When building a contract combinator that needs to explicitly convert ordinary racket values to flat contracts, consider using coerce-flat-contract instead of flat-contract-predicate so that the combinator can raise errors that use the combinator’s name in the error message.

(property/c accessor ctc [#:name name]) → flat-contract?
accessor : (-> any/c any/c)
ctc : flat-contract?
name : any/c = (object-name accessor)

Constructs a flat contract that checks that the first-order property accessed by accessor satisfies ctc. The resulting contract is equivalent to

(lambda (v) (ctc (accessor v)))

except that more information is included in error messages produced by violations of the contract. The name argument is used to describe the property being checked in error messages.

Examples:

> (define/contract (sum-triple lst)
-> (and/c (listof number?)
  (property/c length (=/c 3)))
  number?)
(+ (first lst) (second lst) (third lst)))

> (sum-triple '(1 2 3))
6
> (sum-triple '(1 2))
sum-triple: contract violation
expected: (=/c 3)
given: 2
in: the length of
an and/c case of
the 1st argument of
->
(and/c
  (listof number?)
  (property/c length (=/c 3)))
  number?)
contract from: (function sum-triple)
blaming: top-level
(assuming the contract is correct)
at: eval:2:0

Added in version 7.3.0.11 of package base.

(suggest/c c field message) → contract?
  c : contract?
  field : string?
  message : string?

Returns a contract that behaves like c, except that it adds an extra line to the error message
on a contract violation.

The field and message strings are added following the guidelines in §10.2.1 “Error Mes-
sage Conventions”

Examples:

> (define allow-calls? #f)
> (define/contract (f)
  (suggest/c (->* () #:pre allow-calls? any)
    "suggestion" "maybe you should set! allow-calls? to
#t")
  5)
> (f)
f: contract violation

635
8.2 Function Contracts

A function contract wraps a procedure to delay checks for its arguments and results. There are three primary function contract combinators that have increasing amounts of expressiveness and increasing additional overheads. The first --> is the cheapest. It generates wrapper functions that can call the original function directly. Contracts built with ->* require packaging up arguments as lists in the wrapper function and then using either keyword-apply or apply. Finally, ->i is the most expensive (along with ->d), because it requires delaying the evaluation of the contract expressions for the domain and range until the function itself is called or returns.

The case--> contract is a specialized contract, designed to match case-lambda and unconstrained-domain--> allows range checking without requiring that the domain have any particular shape (see below for an example use).

```
(-> dom ... range)
(-> dom ... ellipsis dom-expr ... range)
```

```
  dom = dom-expr
   | keyword dom-expr

  range = range-expr
   | (values range-expr ...) 
   | any

  ellipsis = ...
```

Produces a contract for a function that accepts the argument specified by the dom-expr contracts and returns either a fixed number of results or completely unspecified results (the latter when any is specified).

Each dom-expr is a contract on an argument to a function, and each range-expr is a contract on a result of the function.

If the domain contain ... then the function accepts as many arguments as the rest of the contracts in the domain portion specify, as well as arbitrarily many more that match the contract just before the ... Otherwise, the contract accepts exactly the argument specified.
For example,

\[
(integer? \ boolean? \ .\ ->\ .\ integer?)
\]

produces a contract on functions of two arguments. The first argument must be an integer, and the second argument must be a boolean. The function must produce an integer.

Examples:

\[
(define\contract\ (maybe-invert\ i\ b)
  \rightarrow\ integer?\ boolean?\ integer?)
\]

\[
(if\ b\ (-\ i)\ i))\]

\[
(maybe-invert\ 1\ \#t)\]

\[-1\]

\[
(maybe-invert\ #f\ 1)\]

maybe-invert: contract violation

expected: integer?
given: #f

in: the 1st argument of

\[
(\rightarrow\ integer?\ boolean?\ integer?)\]

contract from: (function maybe-invert)

blaming: top-level

(assuming the contract is correct)

at: eval:2:0

A domain specification may include a keyword. If so, the function must accept corresponding (mandatory) keyword arguments, and the values for the keyword arguments must match the corresponding contracts. For example:

\[
(integer? \ #:invert?\ boolean? \ .\ ->\ .\ integer?)
\]

is a contract on a function that accepts a by-position argument that is an integer and an #:invert? argument that is a boolean.

Examples:

\[
(define\contract\ (maybe-invert\ i\ #:invert?\ b)
  \rightarrow\ integer?\ #:invert?\ boolean?\ integer?)
\]

\[
(if\ b\ (-\ i)\ i))\]

\[
(maybe-invert\ 1\ #:invert?\ \#t)\]

\[-1\]

\[
(maybe-invert\ 1\ #f)\]

maybe-invert: arity mismatch:

the expected number of arguments does not match the given number
expected: 1 plus an argument with keyword #:invert?
given: 2
arguments...:
|
#f

As an example that uses . . . , this contract:

(integer? string? ... integer? . -> . any)

on a function insists that the first and last arguments to the function must be integers (and there must be at least two arguments) and any other arguments must be strings.

Examples:

> (define/contract (string-length/between? lower-bound s1 . more-args)
  (-> integer? string? ... integer? boolean?))

  (define all-but-first-arg-backwards (reverse (cons s1 more-args)))
  (define upper-bound (first all-but-first-arg-backwards))
  (define strings (rest all-but-first-arg-backwards))
  (define strings-length
    (for/sum ([str (in-list strings)])
      (string-length str)))
  (<= lower-bound strings-length upper-bound))
> (string-length/between? 4 "farmer" "john" 40)
#t
> (string-length/between? 4 "farmer" 'john 40)
string-length/between?: contract violation
  expected: string?
given: 'john
in: the repeated argument of
  (-> integer? string? ... integer? boolean?)
contract from:
  (function string-length/between?)
blaming: top-level
(assuming the contract is correct)
at: eval:2:0
> (string-length/between? 4 "farmer" "john" "fourty")
string-length/between?: contract violation
  expected: integer?
given: "fourty"
in: the last argument of
If any is used as the last sub-form for ->, no contract checking is performed on the result of the function, and thus any number of values is legal (even different numbers on different invocations of the function).

Examples:

> (define/contract (multiple-xs n x)  
  (-> natural? any/c any)  
  (apply  
    values  
    (for/list ([_ (in-range n)])  
      n)))
> (multiple-xs 4 "four")
  4
  4
  4
  4

If (values range-exp ... ) is used as the last sub-form of ->, the function must produce a result for each contract, and each value must match its respective contract.

Examples:

> (define/contract (multiple-xs n x)  
  (-> natural? any/c (values any/c any/c any/c))  
  (apply  
    values  
    (for/list ([_ (in-range n)])  
      n)))
> (multiple-xs 3 "three")
  3
  3
  3
> (multiple-xs 4 "four")
  multiple-xs: broke its own contract;
  expected 3 values, returned 4 values
  in: the range of
  (->
    natural?)

639
any/c
(values any/c any/c any/c))
contract from: (function multiple-xs)
blaming: (function multiple-xs)
(assuming the contract is correct)
at: eval:2:0

Changed in version 6.4.0.5 of package base: Added support for ellipses

(->* (mandatory-dom ...) optional-doms rest pre range post)

mandatory-dom = dom-expr
| keyword dom-expr

optional-doms =
| (optional-dom ...)

optional-dom = dom-expr
| keyword dom-expr

rest =
| #:rest rest-expr

pre =
| #:pre pre-cond-expr
| #:pre/desc pre-cond-expr

range = range-expr
| (values range-expr ...)
| any

post =
| #:post post-cond-expr
| #:post/desc post-cond-expr

The ->* contract combinator produces contracts for functions that accept optional arguments (either keyword or positional) and/or arbitrarily many arguments. The first clause of a ->* contract describes the mandatory arguments, and is similar to the argument description of a -> contract. The second clause describes the optional arguments. The range of description can either be any or a sequence of contracts, indicating that the function must return multiple values.

If present, the rest-expr contract governs the arguments in the rest parameter. Note that the rest-expr contract governs only the arguments in the rest parameter, not those in mandatory arguments. For example, this contract:
does not match the function

\( (\lambda (x . \text{rest}) x) \)

because the contract insists that the function accept zero arguments (because there are no mandatory arguments listed in the contract). The \(\rightarrow\star\) contract does not know that the contract on the rest argument is going to end up disallowing empty argument lists.

The \texttt{pre-cond-expr} and \texttt{post-cond-expr} expressions are checked as the function is called and returns, respectively, and allow checking of the environment without an explicit connection to an argument (or a result). If the \#:\texttt{pre} or \#:\texttt{post} keywords are used, then a \#f result is treated as a failure and any other result is treated as success. If the \#:\texttt{pre/desc} or \#:\texttt{post/desc} keyword is used, the result of the expression must be either a boolean, a string, or a list of strings, where \#t means success and any of the other results mean failure. If the result is a string or a list of strings, the strings are expected to have at exactly one space after each newline and multiple are used as lines in the error message; the contract itself adds single space of indentation to each of the strings in that case. The formatting requirements are not checked but they match the recommendations in §10.2.1 "Error Message Conventions".

As an example, the contract

\[
(\rightarrow\star () \text{(boolean? #:x integer?) #:rest (listof symbol?) symbol?})
\]

matches functions that optionally accept a boolean, an integer keyword argument \#:\texttt{x} and arbitrarily more symbols, and that return a symbol.
The \( \rightarrow_i \) contract combinator differs from the \( 
\rightarrow^* \) combinator in that each argument and result is named and these names can be used in the subcontracts and in the pre-/post-condition clauses. In other words, \( \rightarrow_i \) expresses dependencies among arguments and results.

The optional first keyword argument to \( \rightarrow_i \) indicates if the result contract will be a chaperone. If it is \#:chaperone, all of the contract for the arguments and results must be chaperone.
contracts and the result of ->i will be a chaperone contract. If it is not present, then the result contract will not be a chaperone contract.

The first sub-form of a ->i contract covers the mandatory and the second sub-form covers the optional arguments. Following that is an optional rest-args contract, and an optional pre-condition. The pre-condition is introduced with the #:pre keyword followed by the list of names on which it depends. If the #:pre/name keyword is used, the string supplied is used as part of the error message; similarly with #:post/name. If #:pre/desc or #:post/desc is used, the the result of the expression is treated the same way as ->*.

The dependent-range non-terminal specifies the possible result contracts. If it is any, then any value is allowed. Otherwise, the result contract pairs a name and a contract or a multiple values return with names and contracts. In the last two cases, the range contract may be optionally followed by a post-condition; the post-condition expression is not allowed if the range contract is any. Like the pre-condition, the post-condition must specify the variables on which it depends.

Consider this sample contract:

(-i (x number?)
  y (x) (>=/c x))
[result (x y) (and/c number? (>=/c (+ x y)))]

It specifies a function of two arguments, both numbers. The contract on the second argument (y) demands that it is greater than the first argument. The result contract promises a number that is greater than the sum of the two arguments. While the dependency specification for y signals that the argument contract depends on the value of the first argument, the dependency sequence for result indicates that the contract depends on both argument values. Since the contract for x does not depend on anything else, it does not come with any dependency sequence, not even ()

This example is like the previous one, except the x and y arguments are now optional keyword arguments, instead of mandatory, by-position arguments:

(-i ()
  #:x [x number?]
  #:y [y (x) (>=/c x)]
[result (x y)
  (and/c number?
    (if (and (number? x) (number? y))
      (>=/c (+ x y)))
    any/c))]

The conditional in the range that tests x and y is necessary to cover the situation where x or y are not supplied by the calling context (meaning they might be bound to the-un supplied-arg).
The contract expressions are not always evaluated in order. First, if there is no dependency for a given contract expression, the contract expression is evaluated at the time that the \(-\rightarrow i\) expression is evaluated rather than the time when the function is called or returns. These dependency-free contract expressions are evaluated in the order in which they are listed. Second, the dependent contract sub-expressions are evaluated when the contracted function is called or returns in some order that satisfies the dependencies. That is, if a contract for an argument depends on the value of some other contract, the former is evaluated first (so that the argument, with its contract checked, is available for the other). When there is no dependency between two arguments (or the result and an argument), then the contract that appears earlier in the source text is evaluated first.

If all of the identifier positions of a range contract with a dependency are _s (underscores), then the range contract expressions are evaluated when the function is called instead of when it returns. Otherwise, dependent range expressions are evaluated when the function returns.

If there are optional arguments that are not supplied, then the corresponding variables will be bound to a special value called the-unsupplied-arg value. For example, in this contract:

\[-\rightarrow i \([x \ (y) \ (if \ (unsupplied-arg\? \ y) \ real? \ (>=/c \ y))]) ([y \ real?]) any\]

the contract on \(x\) depends on \(y\), but \(y\) might not be supplied at the call site. In that case, the value of \(y\) in the contract on \(x\) is the-unsupplied-arg and the \(-\rightarrow i\) contract must check for it and tailor the contract on \(x\) to account for \(y\) not being supplied.

When the contract expressions for unsupplied arguments are dependent, and the argument is not supplied at the call site, the contract expressions are not evaluated at all. For example, in this contract, \(y\)'s contract expression is evaluated only when \(y\) is supplied:

\[-\rightarrow i () ([x \ real?] [y \ (x) \ (>=/c \ x)]) any\]

In contrast, \(x\)'s expression is always evaluated (indeed, it is evaluated when the \(-\rightarrow i\) expression is evaluated because it does not have any dependencies).

\[-\rightarrow d (mandatory-dependent-dom ...) dependent-rest pre-condition dependent-range post-condition\]
This contract is here for backwards compatibility; any new code should use \(-\to i\) instead.

This contract is similar to \(-\to i\), but is “lax”, meaning that it does not enforce contracts internally. For example, using this contract

\[\text{(-}\to d \ (\text{mandatory-dependent-dom } \ldots) \\
(\text{optional-dependent-dom } \ldots) \\
\text{dependent-rest} \\
\text{pre-condition} \\
\text{dependent-range} \\
\text{post-condition})\]

\[
\text{mandatory-dependent-dom} = \ \text{[id dom-expr]} \\
| \text{keyword [id dom-expr]}
\]

\[
\text{optional-dependent-dom} = \ \text{[id dom-expr]} \\
| \text{keyword [id dom-expr]}
\]

\[
\text{dependent-rest} = \\
| \text{#:rest id rest-expr}
\]

\[
\text{pre-condition} = \\
| \text{#:pre boolean-expr} \\
| \text{#:pre-cond boolean-expr}
\]

\[
\text{dependent-range} = \text{any} \\
| \text{[_ range-expr]} \\
| (\text{values [id range-expr] ...}) \\
| [id range-expr] \\
| (\text{values [id range-expr] ...})
\]

\[
\text{post-condition} = \\
| \text{#:post-cond boolean-expr}
\]

will allow \(f\) to be called with \#f, trigger whatever bad behavior the author of \(f\) was trying to prohibit by insisting that \(f\)’s contract accept only integers.

The \text{#:pre-cond} and \text{#:post-cond} keywords are aliases for \text{#:pre} and \text{#:post} and are provided for backwards compatibility.

\[\text{(case-> (-> dom-expr ... rest range) ...)}\]
This contract form is designed to match `case-lambda`. Each argument to `case->` is a contract that governs a clause in the `case-lambda`. If the `#:rest` keyword is present, the corresponding clause must accept an arbitrary number of arguments. The `range` specification is just like that for `->` and `->*`.

For example, this contract matches a function with two cases, one that accepts an integer, returning `void`, and one that accepts no arguments and returns an integer.

```
(case-> (-> integer? void?)
       (-> integer?))
```

Such a contract could be used to guard a function that controls access to a single shared integer.

```
@dynamic->*
[#:mandatory-domain-contracts mandatory-domain-contracts
 #:optional-domain-contracts optional-domain-contracts
 #:mandatory-keywords mandatory-keywords
 #:mandatory-keyword-contracts mandatory-keyword-contracts
 #:optional-keywords optional-keywords
 #:optional-keyword-contracts optional-keyword-contracts
 #:rest-contract rest-contract]
 #:range-contracts range-contracts)
-> contract?
  mandatory-domain-contracts : (listof contract?) = '()
  optional-domain-contracts : (listof contract?) = '()
  mandatory-keywords : (listof keyword?) = '()
  mandatory-keyword-contracts : (listof contract?) = '()
  optional-keywords : (listof keyword?) = '()
  optional-keyword-contracts : (listof contract?) = '()
  rest-contract : (or/c #f contract?) = #f
  range-contracts : (or/c #f (listof contract?))
```

Like `->*`, except the number of arguments and results can be computed at runtime, instead of being fixed at compile-time. Passing `#f` as the `#:range-contracts` argument produces a contract like one where any is used with `->` or `->*`. 
For many uses, `dynamic->*`’s result is slower than `->*` (or `->`), but for some it has comparable speed. The name of the contract returned by `dynamic->*` uses the `->` or `->*` syntax.

```
(union-constrained-domain-> range-expr ...)
```

Constructs a contract that accepts a function, but makes no constraint on the function’s domain. The `range-expr`s determine the number of results and the contract for each result.

Generally, this contract must be combined with another contract to ensure that the domain is actually known to be able to safely call the function itself.

For example, the contract

```scheme
(provide
 (contract-out
 [f (->d ([size natural-number/c]
                          [proc (and/c (union-constrained-domain-> number?)
                                      (lambda (p)
                                        (procedure-arity-includes? p size)))])
                ()
                [_ number?])))
```

says that the function `f` accepts a natural number and a function. The domain of the function that `f` accepts must include a case for `size` arguments, meaning that `f` can safely supply `size` arguments to its input.

For example, the following is a definition of `f` that cannot be blamed using the above contract:

```scheme
(define (f i g)
  (apply g (build-list i add1)))
```

```
predicate/c : contract?
```

Use this contract to indicate that some function is a predicate. It is semantically equivalent to `(-> any/c boolean?)`.

This contract also includes an optimization so that functions returning `#t` from `struct-predicate-procedure?` are just returned directly, without being wrapped. This contract is used by `provide/contract`’s `struct` sub-form so that struct predicates end up not being wrapped.

```
the-unsupplied-arg : unsupplied-arg?
```

Used by `->i` (and `->d`) to bind optional arguments that are not supplied by a call site.
A predicate to determine whether v is the-unsupplied-arg.

### 8.3 Parametric Contracts

```scheme
(require racket/contract/parametric)  ; package: base
```

The most convenient way to use parametric contract is to use contract-out’s #:exists keyword. The racket/contract/parametric provides a few more, general-purpose parametric contracts.

```scheme
(parametric->/c (x ...) c)
```

Creates a contract for parametric polymorphic functions. Each function is protected by c, where each x is bound in c and refers to a polymorphic type that is instantiated each time the function is applied.

At each application of a function, the parametric->/c contract constructs a new opaque wrapper for each x: values flowing into the polymorphic function (i.e. values protected by some x in negative position with respect to parametric->/c) are wrapped in the corresponding opaque wrapper. Values flowing out of the polymorphic function (i.e. values protected by some x in positive position with respect to parametric->/c) are checked for the appropriate wrapper. If they have it, they are unwrapped; if they do not, a contract violation is signaled.

Examples:

```scheme
> (define swap-ctc (parametric->/c [A B] (-> A B (values B A))))
> (define/contract (good-swap a b) swap-ctc (values b a))
> (good-swap 1 2)
2
1
> (define/contract (bad-swap a b) swap-ctc (values a b))
> (bad-swap 1 2)
bad-swap: broke its own contract
   promised: B
   produced: #<A>
in: the range of
```

648
(parametric-=>/c (A B) (-> A B (values B A)))
contract from: (function bad-swap)
blaming: (function bad-swap)
(assuming the contract is correct)
at: eval:5:0
> (define/contract (copy-first a b)
  swap-ctc
  (values a a))
> (let ((v 'same-symbol)) (copy-first v v))
copy-first: broke its own contract
  promised: B
  produced: #<A>
in: the range of
  (parametric-=>/c (A B) (-> A B (values B A)))
contract from: (function copy-first)
blaming: (function copy-first)
(assuming the contract is correct)
at: eval:7:0
> (define/contract (inspect-first a b)
  swap-ctc
  (if (integer? a)
      (+ a b)
      (raise-user-error "an opaque wrapped value is not an integer")))
> (inspect-first 1 2)
an opaque wrapped value is not an integer

(new-∀/c [name]) → contract?
name : (or/c symbol? #f) = #f

Constructs a new universal contract.

Universal contracts accept all values when in negative positions (e.g., function inputs) and
wrap them in an opaque struct, hiding the precise value. In positive positions (e.g. function
returns), a universal contract accepts only values that were previously accepted in negative
positions (by checking for the wrappers).

The name is used to identify the contract in error messages and defaults to a name based on
the lexical context of new-∀/c.

For example, this contract:

(let ([a (new-∀/c 'a)]
    (-> a a))

describes the identity function (or a non-terminating function). That is, the first use of the a
appears in a negative position and thus inputs to that function are wrapped with an opaque struct. Then, when the function returns, it is checked to determine whether the result is wrapped, since the second a appears in a positive position.

The \texttt{new-\textbackslash v/c} contract constructor is dual to \texttt{new-\textbackslash l/c}.

\begin{verbatim}
\(\text{contract/\textbackslash l/c}\ [\text{name}] \rightarrow \text{contract}\?\)
\(\text{name} : (\text{or/c symbol\? #f}) = #f\)
\end{verbatim}

Constructs a new existential contract.

Existential contracts accept all values when in positive positions (e.g., function returns) and wrap them in an opaque struct, hiding the precise value. In negative positions (e.g., function inputs), they accept only values that were previously accepted in positive positions (by checking for the wrappers).

The name is used to identify the contract in error messages and defaults to a name based on the lexical context of \texttt{new-\textbackslash l/c}.

For example, this contract:

\begin{verbatim}
\(\text{let } ([a (\text{new-\textbackslash l/c }'a)])\)
\(\rightarrow (\rightarrow a a\)
\(\text{any/c})\)
\end{verbatim}

describes a function that accepts the identity function (or a non-terminating function) and returns an arbitrary value. That is, the first use of the a appears in a positive position and thus inputs to that function are wrapped with an opaque struct. Then, when the function returns, it is checked to see if the result is wrapped, since the second a appears in a negative position.

The \texttt{new-\textbackslash l/c} contract constructor is dual to \texttt{new-\textbackslash v/c}.

\section{8.4 Lazy Data-structure Contracts}

\begin{verbatim}
\(\text{contract-struct id (field-id ...)}\)
\end{verbatim}

\textbf{NOTE:} This library is deprecated; use \texttt{struct}, instead. Lazy struct contracts no longer require a separate struct declaration; instead \texttt{struct/dc} and \texttt{struct/c} work directly with \texttt{struct} and \texttt{define-struct}.

Like \texttt{struct}, but with two differences: they do not define field mutators, and they define two contract constructors: \texttt{id/c} and \texttt{id/dc}. The first is a procedure that accepts as many
arguments as there are fields and returns a contract for struct values whose fields match the arguments. The second is a syntactic form that also produces contracts on the structs, but the contracts on later fields may depend on the values of earlier fields.

The generated contract combinators are lazy: they only verify the contract holds for the portion of some data structure that is actually inspected. More precisely, a lazy data structure contract is not checked until a selector extracts a field of a struct.

```
(id/dc field-spec ...)
```

```
field-spec = [field-id contract-expr] 
  | [field-id (field-id ...) contract-expr]
```

In each field-spec case, the first field-id specifies which field the contract applies to; the fields must be specified in the same order as the original contract-struct. The first case is for when the contract on the field does not depend on the value of any other field. The second case is for when the contract on the field does depend on some other fields, and the parenthesized field-ids indicate which fields it depends on; these dependencies can only be to earlier fields.

```
(define-contract-struct id (field-id ...))
```

NOTE: This library is deprecated; use struct, instead. Lazy struct contracts no longer require a separate struct declaration; instead struct/dc and struct/c work directly with struct and define-struct.

Like contract-struct, but where the constructor’s name is make-id, much like define-struct.

### 8.5 Structure Type Property Contracts

```
(struct-type-property/c value-contract) -> contract?

value-contract : contract?
```

Produces a contract for a structure type property. When the contract is applied to a struct type property, it produces a wrapped struct type property that applies value-contract to the value associated with the property when it used to create a new struct type (via struct, make-struct-type, etc).

The struct type property’s accessor function is not affected; if it is exported, it must be protected separately.
As an example, consider the following module. It creates a structure type property, prop, whose value should be a function mapping a structure instance to a numeric predicate. The module also exports app-prop, which extracts the predicate from a structure instance and applies it to a given value.

> (module propmod racket
  (require racket/contract)
  (define-values (prop prop? prop-ref)
    (make-struct-type-property 'prop))
  (define (app-prop x v)
    (((prop-ref x) x) v))
  (provide/contract
   [prop? (-> any/c boolean?)]
   [prop (struct-type-property/c
           (-> prop? (-> integer? boolean?)))]
   [app-prop (-> prop? integer? boolean?)])
  (provide prop-ref))

The structmod module creates a structure type named s with a single field; the value of prop is a function that extracts the field value from an instance. Thus the field ought to be an integer predicate, but notice that structmod places no contract on s enforcing that constraint.

> (module structmod racket
  (require propmod)
  (struct s (f) #:property prop (lambda (s) (s-f s)))
  (provide (struct-out s)))
> (require propmod structmod)

First we create an s instance with an integer predicate, so the constraint on prop is in fact satisfied. The first call to app-prop is correct; the second simply violates the contract of app-prop.

> (define s1 (s even?))
> (app-prop s1 5)
#f
> (app-prop s1 'apple)
app-prop: contract violation
  expected: integer?
given: 'apple
  in: the 2nd argument of
       (-> prop? integer? boolean?)
contract from: propmod
  blaming: top-level
       (assuming the contract is correct)
at: eval:2:0
We are able to create `s` instances with values other than integer predicates, but applying `app-prop` on them blames `structmod`, because the function associated with `prop`—that is, `(lambda (s) (s-f s))`—does not always produce a value satisfying `(- -> integer? boolean?)`.

```lisp
> (define s2 (s "not a fun"))
> (app-prop s2 5)
prop: contract violation
  expected: a procedure
  given: "not a fun"
  in: the range of
    the struct property value of
    (struct-type-property/c
     (- -> prop? (- -> integer? boolean?)))
contract from: propmod
blaming: structmod
  (assuming the contract is correct)
at: eval:2:0
> (define s3 (s list))
> (app-prop s3 5)
prop: contract violation
  expected: boolean?
  given: '(5)
  in: the range of
    the range of
    the struct property value of
    (struct-type-property/c
     (- -> prop? (- -> integer? boolean?)))
contract from: propmod
blaming: structmod
  (assuming the contract is correct)
at: eval:2:0
```

The fix would be to propagate the obligation inherited from `prop` to `s`:

```
(provide (contract-out
           [struct s ([f (- -> integer? boolean?)])]))
```

Finally, if we directly apply the property accessor, `prop-ref`, and then misuse the resulting function, the `propmod` module is blamed:

```lisp
> ([prop-ref s3] 'apple)
prop: broke its own contract
  promised: prop?
  produced: 'apple
```
The `propmod` module has an obligation to ensure a function associated with `prop` is applied only to values satisfying `prop?`. By directly providing `prop-ref`, it enables that constraint to be violated (and thus it is blamed), even though the bad application actually occurs elsewhere.

Generally there is no need to provide a structure type property accessor at all; it is typically only used by other functions within the module. But if it must be provided, it should be protected thus:

```scheme
(provide (contract-out [prop-ref (-> prop? (-> prop? (-> integer? boolean?)))])
```

### 8.6 Attaching Contracts to Values

```scheme
(contract-out unprotected-submodule contract-out-item ...)
unprotected-submodule =
  | #:unprotected-submodule submodule-name
contract-out-item = (struct id/ignored ((id contract-expr) ...) struct-option)
  | (rename orig-id id contract-expr)
  | (id contract-expr)
  | #:∀ poly-variables
  | #:exists poly-variables
  | #:forall poly-variables
poly-variables = id
  | (id ...)
id/ignored = id
  | (id ignored-id)
struct-option =
  | #:omit-constructor
```
A **provide-spec** for use in **provide** (currently only for the same phase level as the **provide** form; for example, **contract-out** cannot be nested within **for-syntax**). Each **id** is provided from the module. In addition, clients of the module must live up to the contract specified by **contract-expr** for each export.

The **contract-out** form treats modules as units of blame. The module that defines the provided variable is expected to meet the positive (co-variant) positions of the contract. Each module that imports the provided variable must obey the negative (contra-variant) positions of the contract. Each **contract-expr** in a **contract-out** form is effectively moved to the end of the enclosing module, so a **contract-expr** can refer to variables that are defined later in the same module.

Only uses of the contracted variable outside the module are checked. Inside the module, no contract checking occurs.

The **rename** form of **contract-out** exports the first variable (the internal name) with the name specified by the second variable (the external name).

The **struct** form of **contract-out** provides a structure-type definition **id**, and each field has a contract that dictates the contents of the fields. Unlike a **struct** definition, however, all of the fields (and their contracts) must be listed. The contract on the fields that the sub-struct shares with its parent are only used in the contract for the sub-struct’s constructor, and the selector or mutators for the super-struct are not provided. The exported structure-type name always doubles as a constructor, even if the original structure-type name does not act as a constructor. If the **#:omit-constructor** option is present, the constructor is not provided.

The second form of **id/ignored**, which has both **id** and **ignored-id**, is deprecated and allowed in the grammar only for backward compatibility, where **ignored-id** is ignored. The first form should be used instead.

Note that if the struct is created with **serializable-struct** or **define-serializable-struct**, **contract-out** does not protect struct instances that are created via **deserialize**. Consider using **struct-guard/c** instead.

The **#:D**, **#:exists**, **#:∀**, and **#:forall** clauses define new abstract contracts. The variables are bound in the remainder of the **contract-out** form to new contracts that hide the values they accept and ensure that the exported functions are treated parametrically. See **new-D/c** and **new-∀/c** for details on how the clauses hide the values.

If **#:unprotected-submodule** appears, the identifier that follows it is used as the name of a submodule that **contract-out** generates. The submodule exports all of the names in the **contract-out**, but without contracts.

The implementation of **contract-out** uses **syntax-property** to attach properties to the code it generates that records the syntax of the contracts in the fully expanded program. Specifically, the symbol **`provide/contract-original-contract`** is bound to vectors of two elements, the exported identifier and a syntax object for the expression that produces the contract controlling the export.
A `provide-spec` for use in `provide` (currently, just like `contract-out`, only for the same phase level as the `provide` form).

It re-exports `id`, but with positive blame associated to the module containing `recontract-out` instead of the location of the original site of `id`.

This can be useful when a public module wants to export an identifier from a private module but where any contract violations should be reported in terms of the public module instead of the private one.

Examples:

```
> (module private-implementation racket/base
  (require racket/contract)
  (define (recip x) (/ 1 x))
  (define (non-zero? x) (not (= x 0)))
  (provide/contract [recip -> (and/c real? non-zero?)
                   (between/c -1 1)])
> (module public racket/base
  (require racket/contract 'private-implementation)
  (provide (recontract-out recip)))
> (require 'public)
> (recip +nan.0)
recip: broke its own contract
  promised: (between/c -1 1)
  produced: +nan.0
  in: the range of
      (->
       (and/c real? non-zero?)
       (between/c -1 1))
contract from: public
blaming: public
  (assuming the contract is correct)
at: eval:3:0
```

Replacing the use of `recontract-out` with just `recip` would result in a contract violation blaming the private module.

```
(provide/contract unprotected-submodule contract-out-item ...)
```

656
A legacy shorthand for \((\text{provide (contract-out unprotected-submodule contract-out-item \ldots)})\), except that a \textit{contract-expr} within provide/contract is evaluated at the position of the provide/contract form instead of at the end of the enclosing module.

\[(\text{struct-guard/c contract-expr \ldots})\]

Returns a procedure suitable to be passed as the \#:guard argument to \textit{struct}, \textit{serializable-struct} (and related forms). The guard procedure ensures that each contract protects the corresponding field values, as long as the struct is not mutated. Mutations are not protected.

Examples:

```racket
> (struct snake (weight hungry?)
  #:guard (struct-guard/c real? boolean?))
> (snake 1.5 "yep")

snake, field 2: contract violation
  expected: boolean?
given: "yep"
in: boolean?
contract from: top-level
blaming: top-level
  (assuming the contract is correct)
at: eval:2:0
```

### 8.6.1 Nested Contract Boundaries

\[(\text{require racket/contract/region}) \quad \text{package: base}\]

\[(\text{with-contract blame-id (wc-export \ldots) free-var-list \ldots body \ldots})\]
\[(\text{with-contract blame-id results-spec free-var-list \ldots body \ldots})\]

\[\text{wc-export} = (\text{id contract-expr})\]

\[\text{result-spec} = #:result contract-expr\]
\[\quad | \quad #:results (contract-expr \ldots)\]

\[\text{free-var-list} =\]
\[\quad | \quad #:freevar id contract-expr\]
\[\quad | \quad #:freevars ([id contract-expr] \ldots)\]

Generates a local contract boundary.

The first \textit{with-contract} form cannot appear in expression position. All names defined within the first \textit{with-contract} form are visible externally, but those names listed in the
The `define/contract` form treats the individual definition as a contract region. The definition itself is responsible for positive (co-variant) positions of the contract, and references to `id` outside of the definition must meet the negative positions of the contract. Since the contract boundary is between the definition and the surrounding context, references to `id` inside the `define/contract` form are not checked.
Examples:

; an unusual predicate that prints when called
> (define (printing-int? x)
  (displayln "I was called")
  (exact-integer? x))
> (define/contract (fact n)
  (-> printing-int? printing-int?)
  (if (zero? n)
    1
    (* n (fact (sub1 n)))))
> (fact 5) ; only prints twice, not for each recursive call
I was called
I was called
120

If a free-var-list is given, then any uses of the free variables inside the body will be protected with contracts that blame the context of the define/contract form for the positive positions and the define/contract form for the negative ones.

Examples:

> (define (integer->binary-string n)
  (number->string n 2))
> (define/contract (numbers->strings lst)
  (-> (listof number?) (listof string?))
  #:freevar integer->binary-string (-> exact-integer? string?)
  ; mistake, lst might contain inexact numbers
  (map integer->binary-string lst))
> (numbers->strings '(4.0 3.3 5.8))
integer->binary-string: contract violation
expected: exact-integer?
given: 4.0
in: the 1st argument of
  (-> exact-integer? string?)
contract from: top-level
blaming: (function numbers->strings)
  (assuming the contract is correct)
at: eval:3:0

(struct/contract struct-id ([(field contract-expr] ...) struct-option ...))
(struct/contract struct-id super-struct-id
  ([field contract-expr] ...) struct-option ...)

659
Works like `struct`, except that the arguments to the constructor, accessors, and mutators are protected by contracts. For the definitions of `field` and `struct-option`, see `struct`.

The `struct/contract` form only allows a subset of the `struct-option` keywords: `#:mutable`, `#:transparent`, `#:auto-value`, `#:omit-define-syntaxes`, `#:property` and `#:omit-define-values`.

Examples:

```scheme
> (struct/contract fruit ([seeds number?]))
> (fruit 60)
#<fruit>
> (fruit #f)
fruit: contract violation
  expected: number?
  given: #f
  in: the 1st argument of
    (-> number? symbol? any)
  contract from: (struct fruit)
  blaming: top-level
  (assuming the contract is correct)

> (struct/contract apple fruit ([type string?]))
> (apple 14 "golden delicious")
#<apple>
> (apple 5 30)
apple: contract violation
  expected: string?
  given: 30
  in: the 2nd argument of
    (-> any/c string? symbol? any)
  contract from: (struct apple)
  blaming: top-level
  (assuming the contract is correct)

> (apple #f "granny smith")
fruit: contract violation
  expected: number?
  given: #f
  in: the 1st argument of
    (-> number? symbol? any)
  contract from: (struct fruit)
  blaming: top-level
  (assuming the contract is correct)
```

```scheme
(define-struct/contract struct-id ([field contract-expr] ...) struct-option ...)
```

660
(define-struct/contract (struct-id super-struct-id)
  ([field contract-expr] ...)
  struct-option ...)

Works like struct/contract, except that the syntax for supplying a super-struct-id is different, and a constructor-id that has a make-prefix on struct-id is implicitly supplied. For the definitions of field and struct-option, see define-struct.

The define-struct/contract form only allows a subset of the struct-option keywords: #:mutable, #:transparent, #:auto-value, #:omit-define-syntaxes, #:property and #:omit-define-values.

Examples:

> (define-struct/contract fish ([color number?]))
> (make-fish 5)
#<fish>
> (make-fish #f)
make-fish: contract violation
   expected: number?
   given: #f
   in: the 1st argument of
      (-> number? symbol? any)
   contract from: (struct fish)
   blaming: top-level
      (assuming the contract is correct)

> (define-struct/contract (salmon fish) ([ocean symbol?] ))
> (make-salmon 5 'atlantic)
#<salmon>
> (make-salmon 5 #f)
make-salmon: contract violation
   expected: symbol?
   given: #f
   in: the 2nd argument of
      (-> any/c symbol? symbol? any)
   contract from: (struct salmon)
   blaming: top-level
      (assuming the contract is correct)

> (make-salmon #f 'pacific)
make-fish: contract violation
   expected: number?
   given: #f
   in: the 1st argument of
      (-> number? symbol? any)
   contract from: (struct fish)
   blaming: top-level

661
(assuming the contract is correct)

```
(invariant-assertion invariant-expr expr)
```

Establishes an invariant of \textit{expr}, determined by \textit{invariant-expr}.

Unlike the specification of a contract, an \textit{invariant-assertion} does not establish a boundary between two parties. Instead, it simply attaches a logical assertion to the value. Because the form uses contract machinery to check the assertion, the surrounding module is treated as the party to be blamed for any violations of the assertion.

This means, for example, that the assertion is checked on recursive calls, when an invariant is used on the right-hand side of a definition:

Examples:

```scheme
> (define furlongss->feets
  (invariant-assertion
   (-> (listof real?) (listof real?))
   (λ (l)
    (cond
     [(empty? l) empty]
     [else
      (if (= 327 (car l))
       (furlongss->feets (list "wha?")
        (cons (furlongs->feet (first l))
          (furlongss->feets (rest l))))))))

> (furlongss->feets (list 1 2 3))
'(660 1320 1980)
> (furlongss->feets (list 1 327 3))
furlongss->feets: assertion violation
  expected: real?
given: "wha?"
in: an element of
  the 1st argument of
  (-> (listof real?) (listof real?))
contract from: invariant-assertion
at: eval:5:0
```

Added in version 6.0.1.11 of package \textit{base}.

```
current-contract-region
```

Bound by \texttt{define-syntax-parameter}, this contains information about the current contract region, used by the above forms to determine the candidates for blame assignment.
8.6.2 Low-level Contract Boundaries

\[
\text{(define-module-boundary-contract } id \text{ orig-id contract-expr pos-blame-party source-loc name-for-blame context-limit)}
\]

\[
\text{pos-blame-party} =
\]
\[
\text{source-loc} =
\]
\[
\text{name-for-blame} =
\]
\[
\text{context-limit} =
\]

Defines \text{id} to be \text{orig-id}, but with the contract \text{contract-expr}.

The identifier \text{id} is defined as a macro transformer that consults the context of its use to determine the name for negative blame assignment (using the entire module where a reference appears as the negative party).

The positive party defaults to the module containing the use of \text{define-module-boundary-contract}, but can be specified explicitly via the \#:pos-source keyword.

The source location used in the blame error messages for the location of the place where the contract was put on the value defaults to the source location of the use of \text{define-module-boundary-contract}, but can be specified via the \#:srcloc argument, in which case it can be any of the things that the third argument to \text{datum->syntax} can be.

The name used in the error messages will be \text{orig-id}, unless \#:name-for-blame is supplied, in which case the identifier following it is used as the name in the error messages.

If \#:context-limit is supplied, it behaves the same as it does when supplied to \text{contract}.

Examples:

> (module server racket/base
   (require racket/contract/base)
   (define (f x) #f)
   (define-module-boundary-contract g f (-> integer? integer?))
(provide g))
> (module client racket/base
  (require 'server)
  (define (clients-fault) (g #f))
  (define (servers-fault) (g 1))
  (provide servers-fault clients-fault))
> (require 'client)
> (clients-fault)
g: contract violation
  expected: integer?
given: #f
  in: the 1st argument of
    (-> integer? integer?)
  contract from: 'server
  blaming: client
    (assuming the contract is correct)
at: eval:2:0
> (servers-fault)
g: broke its own contract
  promised: integer?
  produced: #f
  in: the range of
    (-> integer? integer?)
  contract from: 'server
  blaming: (quote server)
    (assuming the contract is correct)
at: eval:2:0

Changed in version 6.7.0.4 of package base: Added the #:name-for-blame argument.
Changed in version 6.90.0.29: Added the #:context-limit argument.

| (contract contract-expr to-protect-expr
  positive-blame-expr negative-blame-expr) |
| (contract contract-expr to-protect-expr
  positive-blame-expr negative-blame-expr
  #:context-limit limit-expr) |
| (contract contract-expr to-protect-expr
  positive-blame-expr negative-blame-expr
  value-name-expr source-location-expr) |

The primitive mechanism for attaching a contract to a value. The purpose of contract is as a target for the expansion of some higher-level contract specifying form.

The contract expression adds the contract specified by contract-expr to the value produced by to-protect-expr. The result of a contract expression is the result of the to-
`protect-expr` expression, but with the contract specified by `contract-expr` enforced on `to-protect-expr`.

The values of `positive-blame-expr` and `negative-blame-expr` indicate how to assign blame for positive and negative positions of the contract specified by `contract-expr`. They may be any value, and are formatted as by `display` for purposes of contract violation error messages.

If specified, `value-name-expr` indicates a name for the protected value to be used in error messages. If not supplied, or if `value-name-expr` produces `#f`, no name is printed. Otherwise, it is also formatted as by `display`. More precisely, the `value-name-expr` ends up in the `blame-name` field of the blame record, which is used as the first portion of the error message.

Examples:

```scheme
> (contract integer? #f 'pos 'neg 'timothy #f)
  timothy: broke its own contract
    promised: integer?
    produced: #f
    in: integer?
    contract from: pos
    blaming: pos
    (assuming the contract is correct)
> (contract integer? #f 'pos 'neg #f #f)
  broke its own contract
    promised: integer?
    produced: #f
    in: integer?
    contract from: pos
    blaming: pos
    (assuming the contract is correct)
```

If specified, `source-location-expr` indicates the source location reported by contract violations. The expression must produce a `srcloc` structure, syntax object, `#f`, or a list or vector in the format accepted by the third argument to `datum->syntax`.

If `#:context-limit` is supplied, the following expression must evaluate to either `#f` or a natural number. If the expression evaluates to an natural number, the number of layers of context information is limited to at most that many. For example, if the number is 0, no context information is recorded and the error messages do not contain the section that starts with `in:`.
8.7 Building New Contract Combinators

```
(require racket/contract/combinator)  
package: base

(make-contract
  #:name name
  #:first-order first-order
  #:late-neg-projection late-neg-proj
  #:collapsible-late-neg-projection collapsible-late-neg-proj
  #:val-first-projection val-first-proj
  #:projection proj
  #:stronger stronger
  #:equivalent equivalent
  #:list-contract? is-list-contract?)
 -> contract?
name : any/c = 'anonymous-contract
first-order : (-> any/c any/c) = (\x) #t
late-neg-proj : (or/c #f (-> blame? (-> any/c any/c any/c)))
                = #f
collapsible-late-neg-proj : (or/c #f (-> blame? (values (-> any/c any/c any/c) collapsible))
                             = #f
val-first-proj : (or/c #f (-> blame? (-> any/c (-> any/c any/c any/c))))
                 = #f
proj : (-> blame? (-> any/c any/c))
      = (\b (\x)
          (if (first-order x)
              x
              (raise-blame-error
               b x
               '(expected: "~a" given: "~e")
               name x))))
stronger : (or/c #f (-> contract? contract? boolean?)) = #f
equivalent : (or/c #f (-> contract? contract? boolean?)) = #f
is-list-contract? : boolean? = #f
```
(make-chaperone-contract
 [#:name name
 #:first-order first-order
 #:late-neg-projection late-neg-proj
 #:collapsible-late-neg-projection collapsible-late-neg-proj
 #:val-first-projection val-first-proj
 #:projection proj
 #:stronger stronger
 #:equivalent equivalent
 #:list-contract? is-list-contract?])

→ chaperone-contract?
name : any/c = 'anonymous-chaperone-contract
first-order : (-> any/c any/c) = (λ (x) #t)
late-neg-proj : (or/c #f (-> blame? (-> any/c any/c any/c)))
   = #f
collapsible-late-neg-proj : (or/c #f (-> blame? (values (-> any/c any/c any/c) collapsible-contract?)))
   = #f
val-first-proj : (or/c #f (-> blame? (-> any/c (-> any/c any/c any/c))))
   = #f
proj : (-> blame? (-> any/c any/c))
   = (λ (b)
        (λ (x)
         (if (first-order x)
             x
             (raise-blame-error
              b x
              '(expected: "~a" given: "~e")
              name x))))
stronger : (or/c #f (-> contract? contract? boolean?)) = #f
equivalent : (or/c #f (-> contract? contract? boolean?)) = #f
is-list-contract? : boolean? = #f
(make-flat-contract
  #:name name
  #:first-order first-order
  #:late-neg-projection late-neg-proj
  #:collapsible-late-neg-projection collapsible-late-neg-proj
  #:val-first-projection val-first-proj
  #:projection proj
  #:stronger stronger
  #:equivalent equivalent
  #:list-contract? is-list-contract?)
→ flat-contract?
name : any/c = 'anonymous-flat-contract
first-order : (-> any/c any/c) = (λ (x) #t)
late-neg-proj : (or/c #f (-> blame? (-> any/c any/c any/c)))
  = #f
collapsible-late-neg-proj : (or/c #f (-> blame? (values (-> any/c any/c any/c) collapsible-contract?)))
  = #f
val-first-proj : (or/c #f (-> blame? (-> any/c (-> any/c any/c))))
  = #f
proj : (-> blame? (-> any/c any/c))
  = (λ (b)
      (λ (x)
        (if (first-order x)
            x
            (raise-blame-error
              b x
              '(expected: "~a" given: "~e")
              name x))))
stronger : (or/c #f (-> contract? contract? boolean?)) = #f
equivalent : (or/c #f (-> contract? contract? boolean?)) = #f
is-list-contract? : boolean? = #f

These functions build simple higher-order contracts, chaperone contracts, and flat contracts, respectively. They all take the same set of three optional arguments: a name, a first-order predicate, and a blame-tracking projection. For make-flat-contract, see also flat-contract-with-explanation.

The name argument is any value to be rendered using display to describe the contract when a violation occurs. The default name for simple higher-order contracts is anonymous-contract, for chaperone contracts is anonymous-chaperone-contract, and for flat contracts is anonymous-flat-contract.

The first-order predicate first-order is used to determine which values the contract applies to. This test is used by contract-first-order-passes?, and indirectly by or/c and first-or/c to determine which higher-order contract to wrap a value with when there are multiple higher-order contracts to choose from. The default value accepts any value, but
it must match the behavior of the projection argument (see below for how). The predicate should be influenced by the value of (contract-first-order-okay-to-give-up?) (see it’s documentation for more explanation).

The late-neg-proj argument defines the behavior of applying the contract via a late neg projection. If it is supplied, this argument accepts a blame object that is missing one party (see also blame-missing-party?). Then it must return a function that accepts both the value that is getting the contract and the name of the missing blame party, in that order. The result must either be the value (perhaps suitably wrapped with a chaperone or impersonator to enforce the contract), or signal a contract violation using raise-blame-error. The default is #f.

The collapsible-late-neg-proj argument takes the place of the late-neg-proj argument for contracts that support collapsing. If it is supplied, this argument accepts a blame object that is missing one party. It must return two values. The first value must be a function that accepts both the value that is getting the contract and the name of the missing blame party, in that order. The second value should be a collapsible representation of the contract.

The projection proj and val-first-proj are older mechanisms for defining the behavior of applying the contract. The proj argument is a curried function of two arguments: the first application accepts a blame object, and the second accepts a value to protect with the contract. The projection must either produce the value, suitably wrapped to enforce any higher-order aspects of the contract, or signal a contract violation using raise-blame-error. The default projection produces an error when the first-order test fails, and produces the value unchanged otherwise. The val-first-proj is like late-neg-proj, except with an extra layer of currying.

At least one of the late-neg-proj, proj, val-first-proj, or first-order must be non-#f.

The projection arguments (late-neg-proj, proj, and val-first-proj) must be in sync with the first-order argument. In particular, if the first-order argument returns #f for some value, then the projections must raise a blame error for that value and if the first-order argument returns #t for some value, then the projection must not signal any blame for this value, unless there are higher-order interactions later. In other words, for flat contracts, the first-order and projection arguments must check the same predicate. For convenience, the the default projection uses the first-order argument, signalling an error when it returns #f and never signalling one otherwise.

Projections for chaperone contracts must produce a value that passes chaperone-of? when compared with the original, uncontracted value. Projections for flat contracts must fail precisely when first-order does, and must produce the input value unchanged otherwise. Applying a flat contract may result in either an application of the predicate, or the projection, or both; therefore, the two must be consistent. The existence of a separate projection only serves to provide more specific error messages. Most flat contracts do not need to supply an explicit projection.
The stronger argument is used to implement contract-stronger?. The first argument is always the contract itself and the second argument is whatever was passed as the second argument to contract-stronger?. If no stronger argument is supplied, then a default that compares its arguments with equal? is used for flat contracts and chaperone contracts. For impersonator contracts constructed with make-contract that do not supply the stronger argument, contract-stronger? returns #f.

Similarly, the equivalent argument is used to implement contract-equivalent?. If it isn’t supplied or #false is supplied, then equal? is used for chaperone and flat contracts, and (λ (x y) #f) is used otherwise.

The is-list-contract? argument is used by the list-contract? predicate to determine if this is a contract that accepts only list? values.

Examples:

```
> (define int/c
   (make-flat-contract #:name 'int/c #:first-order integer?))
> (contract int/c 1 'positive 'negative)
1
> (contract int/c "not one" 'positive 'negative)
   eval:4:0: broke its own contract
      promised: int/c
      produced: "not one"
      in: int/c
      contract from: positive
      blaming: positive
      (assuming the contract is correct)
> (int/c 1)
#t
> (int/c "not one")
#f
> (define int->int/c
   (make-contract
    #:name 'int->int/c
    #:first-order
    (λ (x) (and (procedure? x) (procedure-arity-includes? x 1)))
    #:projection
    (λ (b)
     (let ([domain ((contract-projection int/c) (blame-swap b))]
             [range ((contract-projection int/c) b)]
             [f (if (and (procedure? f) (procedure-arity-includes? f 1))
                   (range (f (domain x))))]
             [b f])
      (raise-blame-error b f))
```
'(expected "a function of one argument" given: "~e")

f)))))))

> (contract int->int/c "not fun" 'positive 'negative)

err:8:0: broke its own contract;
  promised a function of one argument
  produced: "not fun"
  in: int->int/c
  contract from: positive
  blaming: positive
  (assuming the contract is correct)

> (define halve
    (contract int->int/c (λ (x) (/ x 2)) 'positive 'negative))

> (halve 2)
1

> (halve 1/2)
halve: contract violation
  expected: int/c
  given: 1/2
  in: int->int/c
  contract from: positive
  blaming: negative
  (assuming the contract is correct)

> (halve 1)
halve: broke its own contract
  promised: int/c
  produced: 1/2
  in: int->int/c
  contract from: positive
  blaming: positive
  (assuming the contract is correct)

---

(build-compound-type-name c/s ...) → any
  c/s : any/c

Produces an S-expression to be used as a name for a contract. The arguments should be either contracts or symbols. It wraps parentheses around its arguments and extracts the names from any contracts it is supplied with.
Converts a regular Racket value into an instance of a contract struct, converting it according to the description of contracts.

If \( v \) is not one of the coercible values, \texttt{coerce-contract} signals an error, using the first argument in the error message.

\[
\text{(coerce-contracts id vs)} \rightarrow \text{(listof contract?)}
\]
\[
id : \text{symbol}?
\]
\[
vs : \text{(listof any/c)}
\]

Coerces all of the arguments in \( vs \) into contracts (via \texttt{coerce-contract/f}) and signals an error if any of them are not contracts. The error messages assume that the function named by \( id \) got \( vs \) as its entire argument list.

\[
\text{(coerce-chaperone-contract id v)} \rightarrow \text{chaperone-contract?}
\]
\[
id : \text{symbol}?
\]
\[
v : \text{any/c}
\]

Like \texttt{coerce-contract}, but requires the result to be a chaperone contract, not an arbitrary contract.

\[
\text{(coerce-chaperone-contracts id vs)} \rightarrow \text{(listof chaperone-contract?)}
\]
\[
id : \text{symbol}?
\]
\[
vs : \text{(listof any/c)}
\]

Like \texttt{coerce-contracts}, but requires the results to be chaperone contracts, not arbitrary contracts.

\[
\text{(coerce-flat-contract id v)} \rightarrow \text{flat-contract?}
\]
\[
id : \text{symbol}?
\]
\[
v : \text{any/c}
\]

Like \texttt{coerce-contract}, but requires the result to be a flat contract, not an arbitrary contract.

\[
\text{(coerce-flat-contracts id v)} \rightarrow \text{(listof flat-contract?)}
\]
\[
id : \text{symbol}?
\]
\[
v : \text{(listof any/c)}
\]

Like \texttt{coerce-contracts}, but requires the results to be flat contracts, not arbitrary contracts.

\[
\text{(coerce-contract/f v)} \rightarrow \text{(or/c contract? #f)}
\]
\[
v : \text{any/c}
\]

Like \texttt{coerce-contract}, but returns \#f if the value cannot be coerced to a contract.
The functions `make-chaperone-contract` and `build-chaperone-contract-property` wrap their arguments to ensure that the result of the projections are chaperones of the input. This layer of wrapping can, in some cases, introduce unwanted overhead into contract checking. If this parameter’s value is `#t` during the dynamic extent of the call to either of those functions, the wrapping (and thus the checks) are skipped.

```
(with-contract-continuation-mark blame body ...)  
(with-contract-continuation-mark blame+neg-party body ...)  
```

Inserts a continuation mark that informs the contract profiler (see the contract profiling documentation) that contract checking is happening. For the costs from checking your new combinator to be included, you should wrap any deferred, higher-order checks with this form. First-order checks are recognized automatically and do not require this form.

If your combinator’s projections operate on complete blame objects (i.e., no missing blame parties), the blame object should be the first argument to this form. Otherwise (e.g., in the case of `late-neg` projections), a pair of the blame object and the missing party should be used instead.

Added in version 6.4.0.4 of package `base`.

```
(contract-pos/neg-doubling e1 e2)  
```

Some contract combinators need to build projections for subcontracts with both regular and `blame-swap`ed versions of the blame that they are given in order to check both access and mutations (e.g., `vector/c` and `vectorof`). In the case that such combinators are nested deeply inside each other, there is a potential for an exponential explosion of nested projections being built.

To avoid that explosion, wrap each of the calls to the blame-accepting portion of the combinator in `contract-pos/neg-doubling`. It returns three values. The first is a boolean, indicating how to interpret the other two results. If the boolean is `#t`, then the other two results are the values of `e1` and `e2` and we are not too deep in the nesting. If the boolean is `#f`, then we have passed a threshold and it is not safe to evaluate `e1` and `e2` yet, as we are in danger of running into the exponential slowdown. In that case, the last two results are thunks that, when invoked, compute the values of `e1` and `e2`.

As an example, `vectorof` uses `contract-pos/neg-doubling` wrapping its two calls to the blame-accepting part of the projection for its subcontract. When it receives a `#f` as that first boolean, it does not invoke the thunks right away, but waits until the interposition
procedure that it attaches to the chaperoned vector is called. Then it invokes them (and
cares the result). This delays the construction of the projections until they are actually
needed, avoiding the exponential blowup.

Added in version 6.90.0.27 of package base.

8.7.1 Blame Objects

This section describes blame objects and operations on them.

\[(\text{blame? } v) \rightarrow \text{boolean?}\]

\[v : \text{any/c}\]

This predicate recognizes blame objects.

\[(\text{raise-blame-error } b \ #:\text{missing-party } missing-party\ v \ v\_fmt ... ) \rightarrow \text{none/c}\]

\[b : \text{blame?}\]

\[missing-party : \#f\]

\[v : \text{any/c}\]

\[fmt : (\text{or/c string?} (\text{listof (or/c string? ('given 'given: 'expected 'expected:))))}\]

\[v\_fmt : \text{any/c}\]

Signals a contract violation. The first argument, \(b\), records the current blame information,
including positive and negative parties, the name of the contract, the name of the value,
and the source location of the contract application. The #:missing-party argument supplies one of the blame parties. It should be non-\#f when the \(b\) object was created without
supplying a negative party. See blame-add-missing-party and the description of the
late-neg-proj argument of make-contract.

The second positional argument, \(v\), is the value that failed to satisfy the contract.

The remaining arguments are a format string, \(fmt\), and its arguments, \(v\_fmt\) ..., specifying an error message specific to the precise violation.

If \(fmt\) is a list, then the elements are concatenated together (with spaces added, unless
there are already spaces at the ends of the strings), after first replacing symbols with ei-
ther their string counterparts, or replacing 'given with "produced" and 'expected with
"promised", depending on whether or not the \texttt{b} argument has been swapped or not (see \texttt{blame-swap}).

If \texttt{fmt} contains the symbols \texttt{'given} or \texttt{'expected}, they are replaced like \texttt{'given} and \texttt{'expected} are, but the replacements are prefixed with the string \texttt{"\n"} to conform to the error message guidelines in §10.2.1 “Error Message Conventions”.

\begin{verbatim}
(define blame-add-context
  (lambda [context
    [#:important important
      #:swap? swap?]]
    blame)
  blame
  context
  [#:important important
    #:swap? swap?]
  \rightarrow blame?
)

blame : blame?
context : (or/c string? #f)
important : (or/c string? #f) = #f
swap? : boolean? = #f
\end{verbatim}

Adds some context information to blame error messages that explicates which portion of the contract failed (and that gets rendered by \texttt{raise-blame-error}).

The \texttt{context} argument describes one layer of the portion of the contract, typically of the form \texttt{"the 1st argument of"} (in the case of a function contract) or \texttt{"a conjunct of"} (in the case of an \texttt{and/c} contract).

For example, consider this contract violation:

\begin{verbatim}
> (define/contract f
  (list/c (-> integer? integer?))
  (list (λ (x) x)))
> ((car f) #f)
f: contract violation
  expected: integer?
  given: #f
in: the 1st argument of
  the 1st element of
  (list/c (-> integer? integer?))
contract from: (definition f)
blaming: top-level
  (assuming the contract is correct)
at: eval:2:0
\end{verbatim}

It shows that the portion of the contract being violated is the first occurrence of \texttt{integer?}, because the -> and the \texttt{list/c} combinators each internally called \texttt{blame-add-context} to add the two lines following “in” in the error message.

The \texttt{important} argument is used to build the beginning part of the contract violation. The last \texttt{important} argument that gets added to a blame object is used. The \texttt{class/c} con-
tract adds an important argument, as does the -> contract (when -> knows the name of the function getting the contract).

The swap? argument has the effect of calling blame-swap while adding the layer of context, but without creating an extra blame object.

Passing #f as the context string argument is no longer relevant. For backwards compatibility, blame-add-context returns b when context is #f.

Changed in version 6.90.0.29 of package base: The context argument being #f is no longer relevant.

\[
\text{(blame-context blame)} \rightarrow \text{(listof string?)}
\]
\[
\text{blame : blame?}
\]

Returns the context information that would be supplied in an error message, if blame is passed to raise-blame-error.

\[
\text{(blame-positive b)} \rightarrow \text{any/c}
\]
\[
\text{b : blame?}
\]

\[
\text{(blame-negative b)} \rightarrow \text{any/c}
\]
\[
\text{b : blame?}
\]

These functions produce printable descriptions of the current positive and negative parties of a blame object.

\[
\text{(blame-contract b)} \rightarrow \text{any/c}
\]
\[
\text{b : blame?}
\]

This function produces a description of the contract associated with a blame object (the result of contract-name).

\[
\text{(blame-value b)} \rightarrow \text{any/c}
\]
\[
\text{b : blame?}
\]

This function produces the name of the value to which the contract was applied, or #f if no name was provided.

\[
\text{(blame-source b)} \rightarrow \text{srcloc?}
\]
\[
\text{b : blame?}
\]

This function produces the source location associated with a contract. If no source location was provided, all fields of the structure will contain #f.

\[
\text{(blame-swap b)} \rightarrow \text{blame?}
\]
\[
\text{b : blame?}
\]
This function swaps the positive and negative parties of a blame object. (See also blame-
add-context.)

```scheme
(blaime-original? b) → boolean?
  b : blame?
(blaime-swapped? b) → boolean?
  b : blame?
```

These functions report whether the current blame of a given blame object is the same as
in the original contract invocation (possibly of a compound contract containing the current
one), or swapped, respectively. Each is the negation of the other; both are provided for
convenience and clarity.

```scheme
(blaime-replace-negative b neg) → blame?
  b : blame?
  neg : any/c
```

Produces a `blame?` object just like `b` except that it uses `neg` instead of the negative position
`b` has.

```scheme
(blaime-replaced-negative? b) → boolean?
  b : blame?
```

Returns `#t` if `b` is the result of calling `blame-replace-negative` (or the result of some
other function whose input was the result of `blame-replace-negative`).

```scheme
(blaime-update b pos neg) → blame?
  b : blame?
  pos : any/c
  neg : any/c
```

Produces a `blame?` object just like `b` except that it adds `pos` and `neg` to the positive and
negative parties of `b` respectively.

```scheme
(blaime-missing-party? b) → boolean?
  b : blame?
```

Returns `#t` when `b` does not have both parties.

```scheme
(blaime-add-missing-party b missing-party)
  → (and/c blame? (not/c blame-missing-party?))
  b : (and/c blame? blame-missing-party?)
  missing-party : any/c
```

Produces a new blame object like `b`, except that the missing party is replaced with `missing-
party`.  

677
This exception is raised to signal a contract error. The object field contains a blame object associated with a contract violation.

A parameter that is used when constructing a contract violation error. Its value is procedure that accepts three arguments:

- the blame object for the violation,
- the value that the contract applies to, and
- a message indicating the kind of violation.

The procedure then returns a string that is put into the contract error message. Note that the value is often already included in the message that indicates the violation.

Examples:

```scheme
> (define (show-blame-error blame value message)
  (string-append
   "Contract Violation!\n"
   (format "Guilty Party: ~a\n" (blame-positive blame))
   (format "Innocent Party: ~a\n" (blame-negative blame))
   (format "Contracted Value Name: ~a\n" (blame-value blame))
   (format "Contract Location: ~s\n" (blame-source blame))
   (format "Contract Name: ~a\n" (blame-contract blame))
   (format "Offending Value: ~s\n" value)
   (format "Offense: ~a\n" message)))
> (current-blame-format show-blame-error)
>
> (define/contract (f x)
  (-> integer? integer?)
  (/ x 2))
> (f 2)
1
> (f 1)
Contract Violation!
Guilty Party: (function f)
Innocent Party: top-level
```
8.7.2 Contracts as structs

The property prop:contract allows arbitrary structures to act as contracts. The property prop:chaperone-contract allows arbitrary structures to act as chaperone contracts; prop:chaperone-contract inherits prop:contract, so chaperone contract structures may also act as general contracts. The property prop:flat-contract allows arbitrary structures to act as flat contracts; prop:flat-contract inherits both prop:chaperone-contract and prop:procedure, so flat contract structures may also act as chaperone contracts, as general contracts, and as predicate procedures.

These properties declare structures to be contracts or flat contracts, respectively. The value for prop:contract must be a contract property constructed by build-contract-property; likewise, the value for prop:chaperone-contract must be a chaperone contract property constructed by build-chaperone-contract-property and the value for prop:flat-contract must be a flat contract property constructed by build-flat-contract-property.

These properties attach a contract value to the protected structure, chaperone, or impersonator value. The function has-contract? returns #t for values that have one of these
properties, and value-contract extracts the value from the property (which is expected to be the contract on the value).

```scheme
(prop: blame : struct-type-property?)
(impersonator-prop: blame : impersonator-property?)
```

These properties attach a blame information to the protected structure, chaperone, or impersonator value. The function has-blame? returns #t for values that have one of these properties, and value-blame extracts the value from the property.

The value is expected to be the blame record for the contract on the value or a cons-pair of a blame record with a missing party and the missing party. The value-blame function reassembles the arguments of the pair into a complete blame record using blame-add-missing-party. If the value has one of the properties, but the value is not a blame object or a pair whose car position is a blame object, then has-blame? returns #f but value-blame returns #f.
(build-flat-contract-property
[#:name get-name
#:first-order get-first-order
#:late-neg-projection late-neg-proj
#:collapsible-late-neg-projection collapsible-late-neg-proj
#:val-first-projection val-first-proj
#:projection get-projection
#:stronger stronger
#:equivalent equivalent
#:generate generate
#:list-contract? is-list-contract?])

→ flat-contract-property?
get-name : (-> contract? any/c)
  = (λ (c) 'anonymous-flat-contract)
get-first-order : (-> contract? (-> any/c boolean?))
  = (λ (c) (λ (x) #t))
late-neg-proj : (or/c #f (-> contract? (-> blame? (-> any/c any/c any/c))))
  = #f
collapsible-late-neg-proj : (or/c #f (-> contract? blame? (values (-> any/c any/c any/c) collapsible-contract?)))
  = #f
val-first-proj : (or/c #f (-> contract? blame? (-> any/c (-> any/c any/c any/c))))
  = #f
get-projection : (-> contract? (-> blame? (-> any/c any/c)))
  = (λ (c)
      (λ (b)
        (λ (x)
          (if ((get-first-order c) x)
              x
              (raise-blame-error
               b x '(expected: "~a" given: "~e")
               (get-name c) x))))))
stronger : (or/c (-> contract? contract? boolean?) #f) = #f
equivalent : (or/c #f (-> contract? contract? boolean?)) = #f
generate : (->i ([c contract?])
  [generator
    (c)
    (→ (and/c positive? real?)
      (or/c (→ (or/c contract-random-generate-fail? c)
      #f)))]
  = (λ (c) (λ (fuel) #f))
is-list-contract? : (→ contract? boolean?) = (λ (c) #f)
(build-chaperone-contract-property
[#:name get-name
#:first-order get-first-order
#:late-neg-projection late-neg-proj
#:collapsible-late-neg-projection collapsible-late-neg-proj
#:val-first-projection val-first-proj
#:projection get-projection
#:stronger stronger
#:equivalent equivalent
#:generate generate
#:exercise exercise
#:list-contract? is-list-contract?])

→ chaperone-contract-property?
get-name : (-> contract? any/c)
  = (λ (c) 'anonymous-chaperone-contract)
get-first-order : (-> contract? (-> any/c boolean?))
  = (λ (c) (λ (x) #t))
late-neg-proj : (or/c #f (-> contract? (-> blame? (-> any/c any/c any/c))))
  = #f
collapsible-late-neg-proj : (or/c #f (-> contract? (-> blame? (values (-> any/c any/c any/c) collapsible-contract?))))
  = #f
val-first-proj : (or/c #f (-> contract? blame? (-> any/c (-> any/c any/c))))
  = #f
get-projection : (-> contract? (-> blame? (-> any/c any/c)))
  = (λ (c)
      (λ (b)
         (λ (x)
            (if ((get-first-order c) x)
                x
                (raise-blame-error
                 b x '(expected: "~a" given: "~e")
                 (get-name c) x))))))
stronger : (or/c (-> contract? contract? boolean?) #f) = #f
equivalent : (or/c #f (-> contract? contract? boolean?)) = #f
generate : (->i ([c contract?])
  [generator
    (c)
    (-> (and/c positive? real?)
        (or/c (-> (or/c contract-random-generate-fail? c))
        #f)]))
  = (λ (c) (λ (fuel) #f))
exercise : (->i ([c contract?])
  [result
    (c)
    (-> (and/c positive? real?)
        (values
         (-> c void?)
         (listof contract?)))]
  = (λ (c) (λ (fuel) (values void '(()))))
is-list-contract? : (-> contract? boolean?) = (λ (c) #f)
(build-contract-property
[#:name get-name
#:first-order get-first-order
#:late-neg-projection late-neg-proj
#:collapsible-late-neg-projection collapsible-late-neg-proj
#:val-first-projection val-first-proj
#:projection get-projection
#:stronger stronger
#:equivalent equivalent
#:generate generate
#:exercise exercise
#:list-contract? is-list-contract?)
→ contract-property?
  get-name : (→ contract? any/c) = (λ (c) 'anonymous-contract)
  get-first-order : (→ contract? (→ any/c boolean?))
                  = (λ (c) (λ (x) #t))
  late-neg-proj : (or/c #f (→ contract? (→ blame? (→ any/c any/c any/c))))
                 = #f
  collapsible-late-neg-proj : (or/c #f (→ contract? blame? (→ any/c (→ any/c any/c))))
                           = #f
  val-first-proj : (or/c #f (→ contract? blame? (→ any/c (→ any/c any/c))))
                 = #f
  get-projection : (→ contract? (→ blame? (→ any/c any/c)))
                = (λ (c)
                   (λ (b)
                      (λ (x)
                         (if ((get-first-order c) x)
                          x
                          (raise-blame-error
                           b x '(expected: "~a" given: "~e")
                           (get-name c) x))))))
  stronger : (or/c (→ contract? contract? boolean?) #f) = #f
  equivalent : (or/c #f (→ contract? contract? boolean?)) = #f
  generate : (→i (c contract?))
              [generator
               (c)
               (→ (and/c positive? real?)
                  (or/c (→ (or/c contract-random-generate-fail? c))
                       #f))]]
        = (λ (c) (λ (fuel) #f))
  exercise : (→i (c contract?))
           [result
            (c)
            (→ (and/c positive? real?)
               (values
                (→ c void?)
                (listof contract?)))]
        = (λ (c) (λ (fuel) (values void '())))
  is-list-contract? : (→ contract? boolean?) = (λ (c) #f)
These functions build the arguments for `prop:contract`, `prop:chaperone-contract`, and `prop:flat-contract`, respectively.

A contract property specifies the behavior of a structure when used as a contract. It is specified in terms of seven properties:

- **get-name** which produces a description to write as part of a contract violation;
- **get-first-order**, which produces a first-order predicate to be used by `contract-first-order-passes?`;
- **late-neg-proj**, which produces a blame-tracking projection defining the behavior of the contract (The `get-projection` and `val-first-proj` arguments also specify the projection, but using a different signature. They are here for backwards compatibility.);
- **collapsible-late-neg-proj**, similar to `late-neg-proj` which produces a blame-tracking projection defining the behavior of the contract, this function additionally specifies the collapsible behavior of the contract;
- **stronger**, a predicate that determines whether this contract (passed in the first argument) is stronger than some other contract (passed in the second argument) and whose default always returns `#f`;
- **equivalent**, a predicate that determines whether this contract (passed in the first argument) is equivalent to some other contract (passed in the second argument); the default for flat and chaperone contracts is `equal?` and for impersonator contracts returns `#f`;
- **generate**, which returns a thunk that generates random values matching the contract (using `contract-random-generate-fail`) to indicate failure) or `#f` to indicate that random generation for this contract isn’t supported;
- **exercise**, which returns a function that exercises values matching the contract (e.g., if it is a function contract, it may call the function) and a list of contracts whose values will be generated by this process;
- and **is-list-contract?**, which is used by `flat-contract?` to determine if this contract accepts only `list?`.

At least one of the `late-neg-proj`, `collapsible-late-neg-proj`, `get-projection`, `val-first-proj`, or `get-first-order` must be non-`#f`.

These accessors are passed as (optional) keyword arguments to `build-contract-property`, and are applied to instances of the appropriate structure type by the contract system. Their results are used analogously to the arguments of `make-contract`.

A chaperone contract property specifies the behavior of a structure when used as a chaperone contract. It is specified using `build-chaperone-contract-property`, and accepts
exactly the same set of arguments as **build-contract-property**. The only difference is that the projection accessor must return a value that passes **chaperone-of?** when compared with the original, uncontracted value.

A **flat contract property** specifies the behavior of a structure when used as a flat contract. It is specified using **build-flat-contract-property**, and accepts similar arguments as **build-contract-property**. The differences are:

- the projection accessor is expected not to wrap its argument in a higher-order fashion, analogous to the constraint on projections in **make-flat-contract**;
- the #:exercise keyword argument is omitted because it is not relevant for flat contracts.

Changed in version 6.0.1.13 of package base: Added the #:list-contract? argument.
Changed in version 6.1.1.4: Allow generate to return **contract-random-generate-fail**.
Changed in version 6.90.0.30: Added the #:equivalent argument.
Changed in version 7.1.0.10: Added the #:collapsible-late-neg-projection argument.

```scheme
(contract-property? v) → boolean?  
  v : any/c
(chaperone-contract-property? v) → boolean?  
  v : any/c
(flat-contract-property? v) → boolean?  
  v : any/c
```

These predicates detect whether a value is a contract property, chaperone contract property, or a flat contract property, respectively.

### 8.7.3 Obligation Information in Check Syntax

Check Syntax in DrRacket shows obligation information for contracts according to **syntax-property**s that the contract combinators leave in the expanded form of the program. These properties indicate where contracts appear in the source and where the positive and negative positions of the contracts appear.

To make Check Syntax show obligation information for your new contract combinators, use the following properties (some helper macros and functions are below):

- `'racket/contract:contract : (vector/c symbol? (listof syntax?) (listof syntax?))`
  This property should be attached to the result of a transformer that implements a contract combinator. It signals to Check Syntax that this is where a contract begins.
The first element in the vector should be a unique (in the sense of `eq?`) value that Check Syntax can use a tag to match up this contract with its subpieces (specified by the two following syntax properties).

The second and third elements of the vector are syntax objects from pieces of the contract, and Check Syntax will color them. The first list should contain subparts that are the responsibility of parties (typically modules) that provide implementations of the contract. The second list should contain subparts that are the responsibility of clients.

For example, in `(-\#* () #:pre #t any/c #:post #t)`, the `-\#*` and the `#:post` should be in the first list and `#:pre` in the second list.

- `'racket/contract:negative-position : symbol?`
  This property should be attached to sub-expressions of a contract combinator that are expected to be other contracts. The value of the property should be the key (the first element from the vector for the `'racket/contract:contract` property) indicating which contract this is.

  This property should be used when the expression’s value is a contract that clients are responsible for.

- `'racket/contract:positive-position : symbol?`
  This form is just like `'racket/contract:negative-position`, except that it should be used when the expression’s value is a contract that the original party should be responsible for.

- `'racket/contract:contract-on-boundary : symbol?`
  The presence of this property tells Check Syntax that it should start coloring from this point. It expects the expression to be a contract (and, thus, to have the `'racket/contract:contract` property); this property indicates that this contract is on a (module) boundary.

  (The value of the property is not used.)

- `'racket/contract:internal-contract : symbol?`
  Like `'racket/contract:contract-on-boundary`, the presence of this property triggers coloring, but this is meant for use when the party (module) containing the contract (regardless of whether or not this module exports anything matching the contract) can be blamed for violating the contract. This comes into play for `->i` contracts, since the contract itself has access to values under contract via the dependency.

```scheme
(define final-prop
  (define/final-prop header body ...))
header = main-id
  | (main-id id ...)
  | (main-id id ... . id)
```
The same as `(define header body ...)`, except that uses of `main-id` in the header are annotated with the `racket/contract:contract` property (as above).

```scheme
(define/subexpression-pos-prop header body ...)
```

```scheme
header = main-id
  | (main-id id ...)
  | (main-id id ... . id)
```

The same as `(define header body ...)`, except that uses of `main-id` in the header are annotated with the `racket/contract:contract` property (as above) and arguments are annotated with the `racket/contract:positive-position` property.

### 8.7.4 Utilities for Building New Combinators

```scheme
(contract-stronger? c1 c2) \rightarrow boolean?
```

```scheme
  c1 : contract?
  c2 : contract?
```

Returns `#t` if the contract `c1` accepts either fewer or the same set of values that `c2` does.

Chaperone contracts and flat contracts that are the same (i.e., where `c1` is `equal?` to `c2`) are considered to always be stronger than each other.

This function is conservative, so it may return `#f` when `c1` does, in fact, accept fewer values.

Examples:

```scheme
> (contract-stronger? integer? integer?)
#t
> (contract-stronger? (between/c 25 75) (between/c 0 100))
#t
> (contract-stronger? (between/c 0 100) (between/c 25 75))
#f
> (contract-stronger? (between/c -10 0) (between/c 0 10))
#f
> (contract-stronger? (\(x\) (and (real? x) (\(<=\) x 0)))
  (\(x\) (and (real? x) (\(<=\) x 100)))))
#f
```

```scheme
(contract-equivalent? c1 c2) \rightarrow boolean?
```

```scheme
  c1 : contract?
  c2 : contract?
```

687
Returns #t if the contract c1 accepts the same set of values that c2 does.

Chaperone contracts and flat contracts that are the same (i.e., where c1 is equal? to c2) are considered to always be equivalent to each other.

This function is conservative, so it may return #f when c1 does, in fact, accept the same set of values that c2 does.

Examples:

> (contract-equivalent? integer? integer?)
#t

> (contract-equivalent? (non-empty-listof integer?)
  (cons/c integer? (listof integer?)))
#t

> (contract-equivalent? (λ (x) (and (real? x) (and (number? x) (>= (sqr x) 0))))
  (λ (x) (and (real? x) (real? x)))))
#f

Added in version 6.90.0.30 of package base.

(contract-first-order-passes? contract v) → boolean?
contract : contract?
v : any/c

Returns a boolean indicating whether the first-order tests of contract pass for v.

If it returns #f, the contract is guaranteed not to hold for that value; if it returns #t, the contract may or may not hold. If the contract is a first-order contract, a result of #t guarantees that the contract holds.

See also contract-first-order-okay-to-give-up? and contract-first-order-try-less-hard.

(contract-first-order c) → (-> any/c boolean?)
c : contract?

Produces the first-order test used by or/c to match values to higher-order contracts.

8.8  Contract Utilities

(contract? v) → boolean?
v : any/c

Returns #t if its argument is a contract (i.e., constructed with one of the combinators described in this section or a value that can be used as a contract) and #f otherwise.
(chaperone-contract? v) → boolean?
   v : any/c

Returns #t if its argument is a chaperone contract, i.e., one that guarantees that it returns a value which passes chaperone-of? when compared to the original, uncontracted value.

(impersonator-contract? v) → boolean?
   v : any/c

Returns #t if its argument is an impersonator contract, i.e., a contract that is neither a chaperone contract nor a flat contract.

(flat-contract? v) → boolean?
   v : any/c

Returns #t when its argument is a contract that can be checked immediately (unlike, say, a function contract).

For example, flat-contract constructs flat contracts from predicates, and symbols, booleans, numbers, and other ordinary Racket values (that are defined as contracts) are also flat contracts.

(list-contract? v) → boolean?
   v : any/c

Recognizes certain contract? values that accept list?s.

A list contract is one that insists that its argument is a list?, meaning that the value cannot be cyclic and must either be the empty list or a pair constructed with cons and another list.

Added in version 6.0.1.13 of package base.

(contract-name c) → any/c
   c : contract?

Produces the name used to describe the contract in error messages.

(value-contract v) → (or/c contract? #f)
   v : has-contract?

Returns the contract attached to v, if recorded. Otherwise it returns #f.

To support value-contract and value-contract in your own contract combinators, use prop:contracted or impersonator-prop:contracted.
(has-contract? v) → boolean?
v : any/c

Returns #t if v is a value that has a recorded contract attached to it.

(value-blame v) → (or/c blame? #f)

v : has-blame?

Returns the blame object for the contract attached to v, if recorded. Otherwise it returns #f.

To support value-contract and value-blame in your own contract combinators, use prop:blame or impersonator-prop:blame.

Added in version 6.0.1.12 of package base.

(has-blame? v) → boolean?

v : any/c

Returns #t if v is a value that has a contract with blame information attached to it.

Added in version 6.0.1.12 of package base.

(contract-late-neg-projection c)
→ (-> blame? (-> any/c (or/c #f any/c) any/c))
c : contract?

Produces the projection defining a contract’s behavior.

The first argument, blame? object encapsulates information about the contract checking, mostly used to create a meaningful error message if a contract violation is detected. The resulting function’s first argument is the value that should have the contract and its second argument is a missing party for the blame object, to be passed to raise-contract-error.

If possible, use this function instead of contract-val-first-projection or contract-projection.

(contract-projection c) → (-> blame? (-> any/c any/c))
c : contract?

Produces a projection defining a contract’s behavior. This projection is a curried function of two arguments: the first application accepts a blame object, and the second accepts a value to protect with the contract.

If possible, use contract-late-neg-projection instead.
(contract-val-first-projection c)
→ (-> blame? (-> any/c (-> any/c any/c)))
c : contract?

Produces a projection defining a contract’s behavior. This projection is similar to the result of \texttt{contract-late-neg-projection} except with an extra layer of currying.

If possible, use \texttt{contract-late-neg-projection} instead.

(make-none/c sexp-name) → contract?
sexp-name : any/c

Makes a contract that accepts no values, and reports the name \textit{sexp-name} when signaling a contract violation.

(recursive-contract contract-expr recursive-contract-option ...)  
(recursive-contract contract-expr type recursive-contract-option ...)

recursive-contract-option = #:list-contract?
| #:extra-delay

type = #:impersonator
| #:chaperone
| #:flat

Delays the evaluation of its argument until the contract is checked, making recursive contracts possible. If \textit{type} is not given, an impersonator contract is created.

If the \texttt{recursive-contract-option #:list-contract?} is given, then the result is a \texttt{list-contract?} and the \texttt{contract-expr} must evaluate to a \texttt{list-contract?}.

If the \texttt{recursive-contract-option #:extra-delay} is given, then the \texttt{contract-expr} expression is evaluated only when the first value to be checked against the contract is supplied to the contract. Without it, the \texttt{contract-expr} is evaluated earlier. This option is supported only when \textit{type} is \#:flat.

Examples:

> (define even-length-list/c
   (or/c null?
    (cons/c any/c
     (cons/c any/c
      (recursive-contract even-length-list/c #:flat))))))
> (even-length-list/c '(A B))
#t
> (even-length-list/c '(1 2 3))
#f

Changed in version 6.0.1.13 of package base: Added the #:list-contract? option.
Changed in version 6.7.0.3: Added the #:extra-delay option.

```
(opt/c contract-expr maybe-name)
```

```
maybe-name =
  | #:error-name id
```

This optimizes its argument contract expression by traversing its syntax and, for known contract combinators, fuses them into a single contract combinator that avoids as much allocation overhead as possible. The result is a contract that should behave identically to its argument, except faster.

If the #:error-name argument is present, and contract-expr evaluates to a non-contract expression, then opt/c raises an error using id as the name of the primitive, instead of using the name opt/c.

Examples:

```
> (define/contract (f x)
  (opt/c '(not-a-contract))
x)
opt/c: contract violation
   expected: contract?
   given: '(not-a-contract)
> (define/contract (f x)
  (opt/c '(not-a-contract) #:error-name define/contract)
x)
define/contract: contract violation
   expected: contract?
   given: '(not-a-contract)
```

```
(define-opt/c (id id ...) expr)
```

This defines a recursive contract and simultaneously optimizes it. As long as the defined function terminates, define-opt/c behaves just as if the -opt/c were not present, defining a function on contracts (except that the body expression must return a contract). But, it also optimizes that contract definition, avoiding extra allocation, much like opt/c does.

For example,
(define-contract-struct bt (val left right))

(define-opt/c (bst-between/c lo hi)
  (or/c null?
    (bt/c [val (real-in lo hi)]
      [left (val) (bst-between/c lo val)]
      [right (val) (bst-between/c val hi)]))))

(define bst/c (bst-between/c -inf.0 +inf.0))

defines the bst/c contract that checks the binary search tree invariant. Removing the -opt/c also makes a binary search tree contract, but one that is (approximately) 20 times slower.

Note that in some cases, a call to a function defined by define-opt/c may terminate, even if the corresponding define-based function would not terminate. This is a shortcoming in define-opt/c that we hope to understand and fix at some point, but have no concrete plans currently.

[[contract-continuation-mark-key : continuation-mark-key?]

Key used by continuation marks that are present during contract checking. The value of these marks are the blame objects that correspond to the contract currently being checked.

Added in version 6.4.0.4 of package base.

[(contract-custom-write-property-proc c p mode) → void?]

  c : contract?
  p : output-port?
  mode : (or/c #f #t 0 1)

Prints c to p using the contract’s name.

Added in version 6.1.1.5 of package base.

[(rename-contract contract name) → contract?]

  contract : contract?
  name : any/c

Produces a contract that acts like contract but with the name name.

The resulting contract is a flat contract if contract is a flat contract.

Added in version 6.3 of package base.
(contract-first-order-okay-to-give-up?)

This form returns a boolean that controls the result of first-order contact checks. More specifically, if it returns #t, then a first-order check may return #t even when the entire first-order checks have not happened. If it returns #f then the first order checks must continue until a definitive answer is returned.

This will only return #t in the dynamic extent of or/c or first-or/c’s checking to determine which branch to use.

Added in version 6.3.0.9 of package base.

(contract-first-order-try-less-hard e)

Encourages first-order checks that happen in the dynamic-extent of e to be more likely to give up. That is, makes it more likely that contract-first-order-okay-to-give-up? might return #t.

If not in the dynamic-extent of or/c’s or first-or/c’s checking to determine the branch, then this form has no effect.

Added in version 6.3.0.9 of package base.

(if/c predicate then-contract else-contract) → contract?

predicate : (-> any/c any/c)
then-contract : contract?
else-contract : contract?

Produces a contract that, when applied to a value, first tests the value with predicate; if predicate returns true, the then-contract is applied; otherwise, the else-contract is applied. The resulting contract is a flat contract if both then-contract and else-contract are flat contracts.

For example, the following contract enforces that if a value is a procedure, it is a thunk; otherwise it can be any (non-procedure) value:

(if/c procedure? (-> any) any/c)

Note that the following contract is not equivalent:

(or/c (-> any) any/c) ; wrong!

The last contract is the same as any/c because or/c tries flat contracts before higher-order contracts.

Added in version 6.3 of package base.
failure-result/c : contract?

A contract that describes the failure result arguments of procedures such as hash-ref.

Equivalent to (if/c procedure? (-> any) any/c).

Added in version 6.3 of package base.

(get/build-val-first-projection c)
→ (-> blame? (-> any/c (-> any/c any/c)))
c : contract?

Returns the val-first projection for c.

See make-contract for more details.

Added in version 6.1.1.5 of package base.

(get/build-late-neg-projection c)
→ (-> blame? (-> any/c any/c any/c))
c : contract?

Returns the late-neg projection for c.

If c does not have a late-neg contract, then this function uses the original projection for it and logs a warning to the racket/contract logger.

See make-contract for more details.

Added in version 6.2.900.11 of package base.

8.9 racket/contract/base

(requires racket/contract/base) package: base

The racket/contract/base module provides a subset of the exports of racket/contract module. In particular, it contains everything in the

• §8.1 “Data-structure Contracts”
• §8.2 “Function Contracts”
• §8.6 “Attaching Contracts to Values” and
• §8.8 “Contract Utilities” sections.
Unfortunately, using `racket/contract/base` does not yield a significantly smaller memory footprint than `racket/contract`, but it can still be useful to add contracts to libraries that `racket/contract` uses to implement some of the more sophisticated parts of the contract system.

### 8.10 Collapsible Contracts

```scheme
(require racket/contract/collapsible) package: base
```

Added in version 7.1.0.10 of package `base`.

Collapsible contracts are an optimization in the contract system designed to avoid a particular pathological build up of contract wrappers on higher-order values. The `vectorof`, `vector/c`, and `->` contract combinators support collapsing for vector contracts and function contracts for functions returning a single value.

Intuitively, a collapsible contract is a tree structure. The tree nodes represent higher-order contracts (e.g., `->`) and the tree leaves represent sequences of flat contracts. Two trees can collapse into one tree via the `merge` procedure, which removes unnecessary flat contracts from the leaves.

For more information on the motivation and design of collapsible contracts, see [Feltey18]. For the theoretical foundations, see [Greenberg15].

**Warning**: the features described in this section are experimental and may not be sufficient to implement new collapsible contracts. Implementing new collapsible contracts requires the use of unsafe chaperones and impersonators which are only supported for vector and procedure values. This documentation exists primarily to allow future maintenance of the `racket/contract/collapsible` library. *End Warning*

```scheme
(get/build-collapsible-late-neg-projection c) ➞ (-> blame? (values (-> any/c any/c any/c) collapsible-contract?))
c : contract?
```

Returns the `collapsible-late-neg` projection for `c`.

If `c` does not have a `collapsible-late-neg` projection, then this function uses the original projection for it and constructs a leaf as its collapsible representation.

```scheme
(collapsible-contract-continuation-mark-key : continuation-mark-key?)
```

Key used by continuation marks that are present during collapsible contract checking. The value of these marks are `#t` if the current contract is collapsible.

```scheme
(with-collapsible-contract-continuation-mark body ...)
```

696
Inserts a continuation mark that informs the contract profiler that the current contract is collapsible.

---

**prop:collapsible-contract : struct-type-property?**

Structures implementing this property are usable as collapsible contracts. The value associated with this property should be constructed by calling `build-collapsible-contract-property`.

---

```scheme
(collapsible-contract? v) -> boolean?
```

**v : any/c**

A predicate recognizing structures with the `prop:collapsible-contract` property.

---

```scheme
(merge new-cc new-neg old-cc old-neg) -> collapsible-contract?
```

**new-cc : collapsible-contract?**

**new-neg : any/c**

**old-cc : collapsible-contract?**

**old-neg : any/c**

Combine two collapsible contracts into a single collapsible contract. The `new-neg` and `old-neg` arguments are expected to be blame parties similar to those passed to a late neg projection.

---

```scheme
(collapsible-guard cc val neg-party) -> any/c
```

**cc : collapsible-contract?**

**val : any/c**

**neg-party : any/c**

Similar to a late neg projection, this function guards the value `val` with the collapsible contract `cc`.

---

```scheme
(collapsible-contract-property? v) -> boolean?
```

**v : any/c**

This predicate indicates that a value can be used as the property for `prop:collapsible-contract`.

---

```scheme
(build-collapsible-contract-property
  #:try-merge try-merge
  #:collapsible-guard collapsible-guard)
  -> collapsible-contract-property?
```

**try-merge : (or/c #f (-> collapsible-contract? any/c collapsible-contract?) any/c (or/c #f collapsible-contract?))**
collapsible-guard : (-> collapsible-contract? any/c any/c any/c) = (λ (cc v neg) (error "internal error: contract does not support `collapsible-guard`" cc)) Constructs a collapsible contract property from a merging function and a guard. The try-merge argument is similar to merge, but may return #f instead of a collapsible contract and may be specialized to a particular collapsible contract. The collapsible-guard argument should be specialized to the particular collapsible contract being implemented.

(struct collapsible-ho/c (latest-blame missing-party latest-ctc))
  latest-blame : blame?
  missing-party : any/c
  latest-ctc : contract?

A common parent structure for collapsible contracts for higher-order values. The latest-blame field holds the blame object for the most recent contract attached. Similarly, the missing-party field holds the latest missing party passed to the contract. The latest-contract field stores the most recent contract attached to the value.

(struct collapsible-leaf/c (proj-list contract-list blame-list missing-party-list))
  proj-list : (listof (-> any/c any/c any/c))
  contract-list : (listof contract?)
  blame-list : (listof blame?)
  missing-party-list : (listof any/c)

A structure representing the leaf nodes of a collapsible contract. The proj-list field holds a list of partially applied late neg projections. The contract-list, blame-list, and missing-party-list fields hold a list of contracts, blame objects, and blame missing parties respectively.

impersonator-prop:collapsible : impersonator-property?
(has-impersonator-prop:collapsible? v) → boolean?
  v : any/c
(get-impersonator-prop:collapsible v) → collapsible-property?
  v : any/c

An impersonator property (and its accessors) that should be attached to chaperoned or impersonated values that are guarded with a collapsible contract.

(struct collapsible-property (c-c neg-party ref))
  c-c : collapsible-contract?
  neg-party : any/c
  ref : (or/c #f impersonator?)
The parent struct of properties that should be attached to chaperones or impersonators of values protected with a collapsible contract. The \texttt{c-c} field stores the collapsible contract that is or will in the future be attached to the the value. The \texttt{neg-party} field stores the latest missing blame party passed to the contract on the value. The \texttt{ref} field is mutable and stores a reference to the chaperone or impersonator to which this property is attached. This is necessary to determine whether an unknown chaperone has been attached to a value after it has been protected by a collapsible contract.

\begin{verbatim}
(struct collapsible-count-property collapsible-property (count prev))
  count : natural-number/c
  prev : (or/c collapsible-count-property? any/c)
\end{verbatim}

This property is associated with the \texttt{impersonator-prop:collapsible} property before the value completely enters the collapsible mode. These properties keep track of the number of contracts on a value in the \texttt{count} field, and hold a reference to the previous \texttt{count property} in the \texttt{prev} field or the original value without a contract. This allows the contract system to traverse the chain of attached contracts and merge them into a single collapsible contract to protect the original value.

\begin{verbatim}
(struct collapsible-wrapper-property collapsible-property (checking-wrapper))
  checking-wrapper : impersonator?
\end{verbatim}

This property is used when a value is guarded by a collapsible contract. The \texttt{checking-wrapper} field holds a chaperone or impersonator that dispatches to the collapsible contract stored in this property to perform any necessary contract checks. When the value receives another contract and merging happens, the checking wrapper will remain the same even though the specific collapsible contract attached to the value may change.

### 8.11 Legacy Contracts

\begin{verbatim}
(make-proj-contract name proj first-order) \rightarrow contract?
  name : any/c
\end{verbatim}
proj : (or/c (-> any/c 
  any/c 
  (list/c any/c any/c) 
  contact? 
  (-> any/c any/c)) 
  (-> any/c 
  any/c 
  (list/c any/c any/c) 
  contact? 
  boolean? 
  (-> any/c any/c))) 

first-order : (-> any/c boolean?) 

Builds a contract using an old interface.

Modulo errors, it is equivalent to:

(make-contract 
  #:name name 
  #:first-order first-order 
  #:projection 
  (cond 
    [(procedure-arity-includes? proj 5) 
     (lambda (blame) 
       (proj (blame-positive blame) 
       (blame-negative blame) 
       (list (blame-source blame) (blame-value blame)) 
       (blame-contract blame) 
       (not (blame-swapped? blame)))] 
    [(procedure-arity-includes? proj 4) 
     (lambda (blame) 
       (proj (blame-positive blame) 
       (blame-negative blame) 
       (list (blame-source blame) (blame-value blame)) 
       (blame-contract blame)))]

(raise-contract-error val 
  src 
  pos 
  name 
  fmt 
  arg ...) -> any/c 

val : any/c 
src : any/c 
pos : any/c 
name : any/c
Calls **raise-blame-error** after building a **blame** struct from the **val**, **src.pos**, and **name** arguments. The **fmt** string and following arguments are passed to **format** and used as the string in the error message.

```plaintext
(contract-proc c)
  -> (->* (symbol? symbol? (or/c syntax? (list/c any/c any/c)))
       (boolean?)
       (-> any/c any))

ctr : contract?
```

 Constructs an old-style projection from a contract.

The resulting function accepts the information that is in a **blame** struct and returns a projection function that checks the contract.

### 8.12 Random generation

```plaintext
(contract-random-generate ctc [fuel fail]) -> any/c

ctc : contract?

fuel : 5 = exact-nonnegative-integer?

fail : (or/c #f (-> any) (-> boolean? any)) = #f
```

Attempts to randomly generate a value which will match the contract. The **fuel** argument limits how hard the generator tries to generate a value matching the contract and is a rough limit of the size of the resulting value.

The generator may fail to generate a value, either because some contracts do not have corresponding generators (for example, not all predicates have generators) or because there is not enough fuel. In either case, the function **fail** is invoked. If **fail** accepts an argument, it is called with **#t** when there is no generator for **ctc** and called with **#f** when there is a generator, but the generator ended up returning **contract-random-generate-fail**.

Example:

```plaintext
> (for/list ([i (in-range 10)])
  (contract-random-generate (or/c integer? #f)))
'(#f -176.0 94 #f -269825872.0 #f #f 442389800 562526632.0 #f)
```

Changed in version 6.1.1.5 of package base: Allow **fail** to accept a boolean.
Attempts to get the vals to break their contracts (if any).

Uses value-contract to determine if any of the vals have a contract and, for those that do, uses information about the contract’s shape to poke and prod at the value. For example, if the value is function, it will use the contract to tell it what arguments to supply to the value.

The argument fuel determines how hard contract-exercise tries to break the values. It controls both the number of exercise iterations and the size of the intermediate values generated during the exercises.

The argument shuffle? controls whether contract-exercise randomizes the exercise order or not. If shuffle? is not #f, contract-exercise would shuffle the order of the contracts in each exercise iteration.

Examples:

> (define/contract (returns-false x)
  (-> integer? integer?)
  ; does not obey its contract
  #f)
> (contract-exercise returns-false)
returns-false: broke its own contract
  promised: integer?
  produced: #f
  in: the range of
    (-> integer? integer?)
  contract from: (function returns-false)
  blaming: (function returns-false)
  (assuming the contract is correct)
at: eval:2:0
> (define/contract (calls-its-argument-with-eleven f)
  (-> (-> integer? integer?) boolean?)
  ; f returns an integer, but
  ; we’re supposed to return a boolean
  (f 11))
> (contract-exercise calls-its-argument-with-eleven)
calls-its-argument-with-eleven: broke its own contract
  promised: boolean?
  produced: 39
in: the range of
  (-> (-> integer? integer?) boolean?)
contract from:
  (function calls-its-argument-with-eleven)
blaming: (function calls-its-argument-with-eleven)
  (assuming the contract is correct)
at: eval:4:0

Changed in version 7.0.0.18 of package base: Added the `shuffle?` optional argument.

```
(contract-random-generate/choose c fuel) -> (or/c #f (-> c))
c : contract?
fuel : exact-nonnegative-integer?
```

This function is like `contract-random-generate`, but it is intended to be used with combinators that generate values based on sub-contracts they have. It must be called when `contract-random-generate` (and `contract-exercise`) creates the generators. To be more precise, `contract-random-generate/choose` is available only for the `generate` and `exercise` arguments in `build-contract-property`, `build-chaperone-contract-property` or `build-flat-contract-property` and only during the dynamic extent of the call to `generate` (and `exercise`). That is, after it receives the `c` and `fuel` arguments and before it returns the thunk (or the exerciser).

`contract-random-generate/choose` will never fail, but it might escape back to an enclosing call or to the original call to `contract-random-generate`.

It chooses one of several possible generation strategies, and thus it may not actually use the generator associated with `c`, but might instead use a stashed value that matches `c` that it knows about via `contract-random-generate-stash`.

Added in version 6.1.1.5 of package base.

```
(contract-random-generate-fail? v) -> boolean?
v : any/c
```

An atomic value that is used to indicate that a generator failed to generate a value.

Added in version 6.1.1.5 of package base.

```
(contract-random-generate-env? v) -> boolean?
v : any/c
```

A predicate to recognize `contract-random-generate-fail`.

Added in version 6.1.1.5 of package base.
Recognizes contract generation environments.

Added in version 6.1.5 of package base.

```
(contract-random-generate-stash env c v) → void?
env : contract-random-generate-env?
c : contract?
v : c
```

This should be called with values that the program under test supplies during contract generation. For example, when `(- (-> integer? integer?) integer?)` is generated, it may call its argument function. That argument function may return an integer and, if so, that integer should be saved by calling `contract-random-generate-stash`, so it can be used by other integer generators.

Added in version 6.1.5 of package base.

```
(contract-random-generate-get-current-environment) → contract-random-generate-env?
```

Returns the environment currently being for generation. This function can be called only during the dynamic extent of contract generation. It is intended to be grabbed during the construction of a contract generator and then used with `contract-random-generate-stash` while generation is happening.

Added in version 6.1.5 of package base.
9 Pattern Matching

The match form and related forms support general pattern matching on Racket values. See also §4.8 “Regular Expressions” for information on regular-expression matching on strings, bytes, and streams.

\[
\text{(require racket/match) } \quad \text{package: base}
\]

The bindings documented in this section are provided by the racket/match and racket libraries, but not racket/base.

\[
\text{(match val-expr clause ...)}
\]

clause = \[
\text{[pat body ...+]} \\
\text{[pat (=> id) body ...+]} \\
\text{[pat #:when cond-expr body ...+]}
\]

Finds the first \text{pat} that matches the result of \text{val-expr}, and evaluates the corresponding \text{body}s with bindings introduced by \text{pat} (if any). Bindings introduced by \text{pat} are not available in other parts of \text{pat}. The last \text{body} in the matching clause is evaluated in tail position with respect to the match expression.

To find a match, the clauses are tried in order. If no \text{clause} matches, then the \text{exn:misc:match?} exception is raised.

An optional \text{(#:when cond-expr)} specifies that the pattern should only match if \text{cond-expr} produces a true value. \text{cond-expr} is in the scope of all of the variables bound in \text{pat}. \text{cond-expr} must not mutate the object being matched before calling the failure procedure, otherwise the behavior of matching is unpredictable. See also \text{failure-cont}, which is a lower-level mechanism achieving the same ends.

Examples:

\[
\begin{align*}
> & \text{(define (m x)} \\
> & \text{(match x)} \\
> & \text{[(list a b c)} \\
> & \text{ #:when (= 6 (+ a b c))} \\
> & \text{ 'sum-is-six]} \\
> & \text{[(list a b c) 'sum-is-not-six]])} \\
> & \text{(m '(1 2 3))} \\
> & \text{'sum-is-six} \\
> & \text{(m '(2 3 4))} \\
> & \text{'sum-is-not-six}
\end{align*}
\]

An optional \text{(=> id)} between a \text{pat} and the \text{body}s is bound to a failure procedure of zero arguments. If this procedure is invoked, it escapes back to the pattern matching expression,
and resumes the matching process as if the pattern had failed to match. The bodys must not mutate the object being matched before calling the failure procedure, otherwise the behavior of matching is unpredictable.

Examples:

> (define (m x)
   (match x
     [(list a b c)
      (==> exit)
      (f x exit)]
     [(list a b c) 'sum-is-not-six]))
> (define (f x exit)
   (if (= 6 (apply + x))
     'sum-is-six
     (exit)))
> (m '(1 2 3))
'sum-is-six
> (m '(2 3 4))
'sum-is-not-six

The grammar of pat is as follows, where non-italicized identifiers are recognized symbolically (i.e., not by binding).

\[
\begin{align*}
\text{pat} & ::= \text{id} & \text{match anything, bind identifier} \\
& | (\text{var id}) & \text{match anything, bind identifier} \\
& | _ & \text{match anything} \\
& | \text{literal} & \text{match literal} \\
& | (\text{quote datum}) & \text{match equal? value} \\
& | (\text{list} lvp ... ) & \text{match sequence of lvps} \\
& | (\text{list-rest} lvp ... pat) & \text{match lvp}s consed onto a pat \\
& | (\text{list*} lvp ... pat) & \text{match lvp}s consed onto a pat \\
& | (\text{list-no-order} pat ...) & \text{match pats in any order} \\
& | (\text{list-no-order} pat ... lvp) & \text{match pats in any order} \\
& | (\text{vector} lvp ...) & \text{match vector of pats} \\
& | (\text{hash-table} (pat pat) ...) & \text{match hash table} \\
& | (\text{hash-table} (pat pat) ...+ ooo) & \text{match hash table} \\
& | (\text{cons} pat pat) & \text{match pair of pats} \\
& | (\text{mcons} pat pat) & \text{match mutable pair of pats} \\
& | (\text{box} pat) & \text{match boxed pat} \\
& | (\text{struct-id} pat ...) & \text{match struct-id instance} \\
& | (\text{struct} struct-id (pat ...)) & \text{match struct-id instance} \\
& | (\text{regexp} rx-expr) & \text{match string} \\
& | (\text{regexp} rx-expr pat) & \text{match string, result with pat} \\
& | (\text{pregexp} px-expr) & \text{match string} \\
& | (\text{pregexp} px-expr pat) & \text{match string, result with pat}
\end{align*}
\]
(and pat ...) match when all pats match
(or pat ...) match when any pat matches
(not pat ...) match when no pat matches
(app expr pats ...) match (expr value) output values to pats
(? expr pat ...) match if (expr value) and pats
(quasiquote qp) match a quasipattern
derived-pattern match using extension

In more detail, patterns match as follows:

- \textit{id} (excluding the reserved names \_\_\_, \_\_, \_\_, \ldots \_\_. \ldots \_\_. \ldots k, and \_\_\_k for non-negative integers \_\_k)

or \textit{(var id)} — matches anything, and binds \textit{id} to the matching values. If an \textit{id} is used multiple times within a pattern, the corresponding matches must be the same according to \textit{(match-equality-test)}, except that instances of an \textit{id} in different or \textit{not} sub-patterns are independent. The binding for \textit{id} is not available in other parts of the same pattern.

Examples:
> (match '(1 2 3)
   [(list a b a) (list a b)]
   [(list a b c) (list c b a)]
'(3 2 1)
> (match '(1 (x y z) 1)
   [(list a b a) (list a b)]
   [(list a b c) (list c b a)]
'(1 (x y z))
> (match #f
   [else
    (cond
     [#f 'not-evaluated]
     [else 'also-not-evaluated]]))

• _ — matches anything, without binding any identifiers.

Example:

> (match '(1 2 3)
   [(list _ _ a) a])
3

• #t, #f, string, bytes, number, char, or (quote datum) — matches an equal? constant.

Example:

> (match "yes"
   ["no" #f]
   ["yes" #t])
#t

• (list lvp ...) — matches a list of elements. In the case of (list pat ...), the pattern matches a list with as many element as pats, and each element must match the corresponding pat. In the more general case, each lvp corresponds to a “spliced” list of greedy matches.

For spliced lists, ... and ___ are aliases for zero or more matches. The ...k and ___k forms are also aliases, specifying k or more matches. Pattern variables that precede these splicing operators are bound to lists of matching forms.

Examples:

> (match '(1 2 3)
   [(list a b c) (list c b a)]
'(3 2 1)
> (match '(1 2 3)
   [(list 1 a ...) a])
'(2 3)
• \texttt{\textbf{(list-rest 1vp \ldots pat)}} or \texttt{(list* 1vp \ldots pat)} — similar to a \texttt{list} pattern, but the final \texttt{pat} matches the “rest” of the list after the last \texttt{1vp}. In fact, the matched value can be a non-list chain of pairs (i.e., an “improper list”) if \texttt{pat} matches non-list values.

Examples:

\begin{verbatim}
> (match '(1 2 3 . 4)
  [(list-rest a b c d) d])
'(1 2 3)
4
> (match '(1 2 3 . 4)
  [(list-rest a ... d) (list a d)])
'(1 2 3 4)
\end{verbatim}

• \texttt{(list-no-order pat \ldots)} — similar to a \texttt{list} pattern, but the elements to match each \texttt{pat} can appear in the list in any order.

Example:

\begin{verbatim}
> (match '(1 2 3)
  [(list-no-order 3 2 x) x])
'(1 2 3)
1
\end{verbatim}

• \texttt{(list-no-order pat \ldots 1vp)} — generalizes \texttt{list-no-order} to allow a pattern that matches multiple list elements that are interspersed in any order with matches for the other patterns.

Example:

\begin{verbatim}
> (match '(1 2 3 4 5 6)
  [(list-no-order 6 2 y \ldots) y])
'(1 3 4 5)
\end{verbatim}
• **(vector lvp ...)** — like a list pattern, but matching a vector.

  Example:

  ```lisp
  > (match #(1 (2) (2) (2) 5)
     [(vector 1 (list a) ..3 5) a])
  '(2 2 2)
  ```

• **(hash-table (pat pat) ...)** — similar to list-no-order, but matching against hash table’s key–value pairs.

  Example:

  ```lisp
  > (match #hash(("a" . 1) ("b" . 2))
     [(hash-table ("b" b) ("a" a)) (list b a)])
  '(2 1)
  ```

• **(hash-table (pat pat) ...) + ooo** — Generalizes hash-table to support a final repeating pattern.

  Example:

  ```lisp
  > (match #hash(("a" . 1) ("b" . 2))
     [(hash-table (key val) ...) key])
  "b" "a"
  ```

• **(cons pat1 pat2)** — matches a pair value.

  Example:

  ```lisp
  > (match (cons 1 2)
     [(cons a b) (+ a b)])
  3
  ```

• **(mcons pat1 pat2)** — matches a mutable pair value.

  Example:

  ```lisp
  > (match (mcons 1 2)
     [(cons a b) 'immutable]
     [(mcons a b) 'mutable])
  'mutable
  ```

• **(box pat)** — matches a boxed value.

  Example:

  ```lisp
  > (match #&1
     [(box a) a])
  1
  ```
• \texttt{(struct-id pat \ldots)} or \texttt{(struct struct-id (pat \ldots))} — matches an instance of a structure type named \texttt{struct-id}, where each field in the instance matches the corresponding \texttt{pat}. See also \texttt{struct*}.

Usually, \texttt{struct-id} is defined with \texttt{struct}. More generally, \texttt{struct-id} must be bound to expansion-time information for a structure type (see §5.7 "Structure Type Transformer Binding"), where the information includes at least a predicate binding and field accessor bindings corresponding to the number of field \texttt{pats}. In particular, a module import or a unit import with a signature containing a \texttt{struct} declaration can provide the structure type information.

Examples:

\begin{verbatim}
(struct tree (val left right))

> (match (tree 0 (tree 1 #f #f) #f)
  [(tree a (tree b _ _) _) (list a b)])
'(0 1)
\end{verbatim}

• \texttt{(struct struct-id \_)} — matches any instance of \texttt{struct-id}, without regard to contents of the fields of the instance.

• \texttt{(regexp rx-expr)} — matches a string that matches the regexp pattern produced by \texttt{rx-expr}; see §4.8 "Regular Expressions" for more information about regexps.

Examples:

\begin{verbatim}
> (match "apple"
  [(regexp #rx"p+") 'yes]
  [_ 'no])
'yes

> (match "banana"
  [(regexp #rx"p+") 'yes]
  [_ 'no])
'no
\end{verbatim}

• \texttt{(regexp rx-expr pat)} — extends the \texttt{regexp} form to further constrain the match where the result of \texttt{regexp-match} is matched against \texttt{pat}.

Examples:

\begin{verbatim}
> (match "apple"
  [(regexp #rx"p+(.)" (list _ "l")) 'yes]
  [_ 'no])
'yes

> (match "append"
  [(regexp #rx"p+(.)" (list _ "l")) 'yes]
  [_ 'no])
'no
\end{verbatim}
• (pregexp rx-expr) or (pregexp rx-expr pat) — like the regexp patterns, but if rx-expr produces a string, it is converted to a pattern using pregexp instead of regexp.

• (and pat ...) — matches if all of the pat s match. This pattern is often used as (and id pat) to bind id to the entire value that matches pat. The pat s are matched in the order that they appear.

Example:

> (match '(1 (2 3) 4)
    [(list _ (and a (list _ ...)) _) a])
  '(2 3)

• (or pat ...) — matches if any of the pat s match. Each pat must bind the same set of identifiers.

Example:

> (match '(1 2)
    [(or (list a 1) (list a 2)) a])
  1

• (not pat ...) — matches when none of the pat s match, and binds no identifiers.

Examples:

> (match '(1 2 3)
    [(list (not 4) ...) 'yes]
    [_ 'no])
  'yes
> (match '(1 4 3)
    [(list (not 4) ...) 'yes]
    [_ 'no])
  'no

• (app expr pats ...) — applies expr to the value to be matched; each result of the application is matched against one of the pats, respectively.

Examples:

> (match '(1 2)
    [(app length 2) 'yes])
  'yes
> (match "3.14"
    [(app string->number (? number? pi))
      '(I got ,pi)])
  '(I got 3.14)
> (match '(1 2)
    [(app (lambda (v) (split-at v 1)) '(1) '(2)) 'yes])
'yes
> (match '(1 2 3)
   [(app (λ (ls) (apply values ls)) x y (? odd? z))
      (list 'yes x y z))]
'(yes 1 2 3)

• (?, expr pat ...) — applies expr to the value to be matched, and checks whether
the result is a true value; the additional pats must also match; i.e., ? combines a
predicate application and an and pattern. However, ?, unlike and, guarantees that
expr is matched before any of the pats.
Example:
> (match '(1 3 5)
   [(list (? odd?) ...) 'yes])
'yes

• (quasiquote qp) — introduces a quasipattern, in which identifiers match symbols.
Like the quasiquote expression form, unquote and unquote-splicing escape
back to normal patterns.
Example:
> (match '(1 2 3)
   [('\(1 ,a ,(? odd? b)) (list a b)]
'(2 3)

• derived-pattern — matches a pattern defined by a macro extension via define-
match-expander.

Note that the matching process may destructure the input multiple times, and may evaluate
expressions embedded in patterns such as (app expr pat) in arbitrary order, or multiple
times. Therefore, such expressions must be safe to call multiple times, or in an order other
than they appear in the original program.

9.1 Additional Matching Forms

(match* (val-expr ...+) clause* ...)

clause* = [(pat ...+) body ...+
   | [(pat ...+) (= id) body ...+]
   | [(pat ...+) #:when cond-expr body ...+]

Matches a sequence of values against each clause in order, matching only when all patterns
in a clause match. Each clause must have the same number of patterns as the number of
val-exprs.
Examples:

> (match* (1 2 3)
     (((? number? x) (add1 x)))
4
> (match* (15 17)
     (((? number? a) (? number? b))
      #:when (= (+ a 2) b)
      'diff-by-two)
'diff-by-two

(match/values expr clause clause ...

If expr evaluates to n values, then match all n values against the patterns in clause .... Each clause must contain exactly n patterns. At least one clause is required to determine how many values to expect from expr.

(define/match (head args)
  match*-clause ...

  head  =  id
       |  (head  args)

  args = arg ...
       |  arg .... . rest-id

  arg = arg-id
      |  [arg-id  default-expr]
      |  keyword arg-id
      |  keyword [arg-id  default-expr]

  match*-clause = [(pat  ...+) body  ...+]
                  | [(pat  ...+)  (= id)  body  ...+]
                  | [(pat  ...+) #:when  cond-expr  body  ...+]

Binds id to a procedure that is defined by pattern matching clauses using match*. Each clause takes a sequence of patterns that correspond to the arguments in the function header. The arguments are ordered as they appear in the function header for matching purposes.

Examples:

(define/match (fact n)
  [(0) 1]
  [(n) (* n (fact (sub1 n)))]

> (fact 5)
120
The function header may also contain optional or keyword arguments, may have curried arguments, and may also contain a rest argument.

Examples:

```scheme
(define/match ((f x) #:y [y '(1 2 3)])
  [([((regexp #rx"p+") '(,a 2 3)) a]
   (_ _) #f])

> ((f "ape") #:y '(5 2 3))
5
> ((f "dog"))
#f

(define/match (g x y . rst)
  [([0 0 '() ) #t]
   [(5 5 '(5 5)) #t]
   (_ _ _) #f])

> (g 0 0)
#t
> (g 5 5 5 5)
#t
> (g 1 2)
#f

(match-lambda clause ...)
```

Equivalent to `(lambda (id) (match id clause ...)).

```
(match-lambda* clause ...)
```

Equivalent to `(lambda lst (match lst clause ...)).

```
(match-lambda** clause* ...)
```

Equivalent to `(lambda (args ...) (match* (args ...) clause* ...))`, where the number of `args ...` is computed from the number of patterns appearing in each of the `clause*`.

```
(match-let ([pat expr] ... body ...+)
```

Generalizes `let` to support pattern bindings. Each `expr` is matched against its corresponding `pat` (the match must succeed), and the bindings that `pat` introduces are visible in the `bodys`.

Example:
> (match-let ([[list a b] '(1 2)]
   [(vector x ...) #(1 2 3 4)])
   (list b a x))
'(2 1 (1 2 3 4))

(match-let* ([pat expr] ...) body ...+)

Like match-let, but generalizes let*, so that the bindings of each pat are available in each subsequent expr.

Example:

> (match-let* ([[list a b] '(#(1 2 3 4) 2)]
   [(vector x ...) a])
   x)
'(1 2 3 4)

(match-let-values ([[pat ...] expr] ...) body ...+)

Like match-let, but generalizes let-values.

(match-let*-values ([[pat ...] expr] ...) body ...+)

Like match-let*, but generalizes let*-values.

(match-letrec ([pat expr] ...) body ...+)

Like match-let, but generalizes letrec.

(match-letrec-values ([[pat ...] expr] ...) body ...+)

Like match-let, but generalizes letrec-values.

Added in version 6.1.1.8 of package base.

(match-define pat expr)

Defines the names bound by pat to the values produced by matching against the result of expr.

Examples:

> (match-define (list a b) '(1 2))
> b
2
Like `match-define` but for when `expr` produces multiple values. Like `match/values`, it requires at least one pattern to determine the number of values to expect.

Examples:

```scheme
> (match-define-values (a b) (values 1 2))
> b
2
```

```scheme
(exn:misc:match? v) → boolean?
```

A predicate for the exception raised in the case of a match failure.

```scheme
(failure-cont)
```

Continues matching as if the current pattern failed. Note that unlike use of the `=>` form, this does not escape the current context, and thus should only be used in tail position with respect to the match form.

### 9.2 Extending `match`

```scheme
(define-match-expander id proc-expr)
(define-match-expander id proc-expr proc-expr)
```

Binds `id` to a `match expander`.

The first `proc-expr` sub-expression must evaluate to a transformer that produces a `pat` for `match`. Whenever `id` appears as the beginning of a pattern, this transformer is given, at expansion time, a syntax object corresponding to the entire pattern (including `id`). The pattern is replaced with the result of the transformer.

A transformer produced by a second `proc-expr` sub-expression is used when `id` is used in an expression context. Using the second `proc-expr`, `id` can be given meaning both inside and outside patterns.

Match expanders are not invoked unless `id` appears in the first position in a sequence. Instead, identifiers bound by `define-match-expander` are used as binding identifiers (like any other identifier) when they appear anywhere except the first position in a sequence.

For example, to extend the pattern matcher and destructure syntax lists,
(define (syntax-list? x)
  (and (syntax? x)
       (list? (syntax->list x))))

(define-match-expander syntax-list
  (lambda (stx)
    (syntax-case stx ()
      [(_ elts ...) #'
       (? syntax-list?
         (app syntax->list (list elts ...)))]]))

(define (make-keyword-predicate keyword)
  (lambda (stx)
    (and (identifier? stx)
         (free-identifier=? stx keyword))))

(define or-keyword? (make-keyword-predicate #\'or))
(define and-keyword? (make-keyword-predicate #\'and))

> (match #\'(or 3 4)
  [(syntax-list (? or-keyword?) b c)
   (list "OOORRR!" b c)]
  [(syntax-list (? and-keyword?) b c)
   (list "AAANND!" b c)])
'("OOORRR!" #<syntax:eval:69:0 3> #<syntax:eval:69:0 4>)

> (match #\'(and 5 6)
  [(syntax-list (? or-keyword?) b c)
   (list "OOORRR!" b c)]
  [(syntax-list (? and-keyword?) b c)
   (list "AAANND!" b c)])
'("AAANND!" #<syntax:eval:70:0 5> #<syntax:eval:70:0 6>)

And here is an example showing how define-match-expander-bound identifiers are not treated specially unless they appear in the first position of pattern sequence. Consider this (incorrect) definition of a length function:

(define-match-expander nil
  (λ (stx) #""())
  (λ (stx) #""()))

(define (len l)
  (match l
    [nil 0]
    [(cons hd tl) (+ 1 (len tl))])))

Because there are no parenthesis around nil, match treats the first case as an identifier (which matches everything) instead of a use of the match expander and len always returns 0.

> (len nil)

718
Match expanders accept any syntax pair whose first element is an identifier? bound to the expander. The following example shows a match expander which can be called with an improper syntax list of the form `(expander a b . rest)`.

```scheme
(define-match-expander my-vector
  (λ (stx)
    (syntax-case stx ()
      [(_ pat ...)
        #'(vector pat ...)]
      [(_ pat ... . rest-pat)
        #'(app vector->list (list-rest pat ... rest-pat))])))
```

```scheme
> (match #(1 2 3 4 5)
  [(my-vector a b . rest)
   (list->vector (append rest (list a b)))])
'(3 4 5 1 2)
```

Changed in version 7.7.0.2 of package base: Match expanders now allowed any syntax pair whose first element is an identifier? bound to the expander. The example above did not work with previous versions.

### prop:match-expander : struct-type-property?

A structure type property to identify structure types that act as match expanders like the ones created by `define-match-expander`.

The property value must be an exact non-negative integer or a procedure of one or two arguments. In the former case, the integer designates a field within the structure that should contain a procedure; the integer must be between 0 (inclusive) and the number of non-automatic fields in the structure type (exclusive, not counting supertype fields), and the designated field must also be specified as immutable.

If the property value is a procedure of one argument, then the procedure serves as the transformer for match expansion. If the property value is a procedure of two arguments, then the first argument is the structure whose type has `prop:match-expander` property, and the second argument is a syntax object as for a match expander.

If the property value is a assignment transformer, then the wrapped procedure is extracted with `set!-transformer-procedure` before it is called.

This binding is provided for-syntax.
prop:legacy-match-expander : struct-type-property?

Like prop:match-expander, but for the legacy match syntax.

This binding is provided for-syntax.

\[
\begin{align*}
\text{(match-expander? } v) & \rightarrow \text{ boolean?} \\
v & : \text{any/c} \\
\text{(legacy-match-expander? } v) & \rightarrow \text{ boolean?} \\
v & : \text{any/c}
\end{align*}
\]

Predicates for values which implement the appropriate match expander properties.

\[
\begin{align*}
\text{(syntax-local-match-introduce } stx) & \rightarrow \text{ syntax?} \\
stx & : \text{syntax?}
\end{align*}
\]

For backward compatibility only; equivalent to syntax-local-introduce.

Changed in version 6.90.0.29 of package base: Made equivalent to syntax-local-introduce.

\[
\begin{align*}
\text{(match-equality-test)} & \rightarrow (\text{any/c any/c . } \rightarrow . \text{ any}) \\
\text{(match-equality-test comp-proc)} & \rightarrow \text{ void?} \\
\text{comp-proc} & : (\text{any/c any/c . } \rightarrow . \text{ any})
\end{align*}
\]

A parameter that determines the comparison procedure used to check whether multiple uses of an identifier match the “same” value. The default is equal?.

\[
\begin{align*}
\text{(match/derived } val\text{-expr original-datum clause ...)} \\
\text{(match*/derived (val\text{-expr ...)} original-datum clause* ...)}
\end{align*}
\]

Like match and match* respectively, but includes a sub-expression to be used as the source for all syntax errors within the form. For example, match-lambda expands to match/derived so that errors in the body of the form are reported in terms of match-lambda instead of match.

9.3 Library Extensions

\[
\begin{align*}
(== \text{ val comparator}) \\
(== \text{ val})
\end{align*}
\]

A match expander which checks if the matched value is the same as \text{val} when compared by \text{comparator}. If \text{comparator} is not provided, it defaults to equal?.

Examples:
A match pattern form that matches an instance of a structure type named \textit{struct-id}, where the field \textit{field} in the instance matches the corresponding \textit{pat}. The fields do not include those from super types.

Any field of \textit{struct-id} may be omitted, and such fields can occur in any order.

Examples:

\begin{verbatim}
(struct tree (val left right))
(struct tree* tree (val))

> (match (tree 0 (tree 1 #f #f) #f)
[ (struct* tree* ([val a]
  [left (struct* tree ([right #f] [val b]))]))
  (list a b))
'(0 1)
> (match (tree* 0 #f #f 42)
[(and (struct* tree* ([val a]))
  (struct* tree ([val b])))
  (list a b))
'(42 0)
\end{verbatim}
10  Control Flow

10.1  Multiple Values

See §1.1.3 “Multiple Return Values” for general information about multiple result values. In addition to `call-with-values` (described in this section), the `let-values`, `let*values`, `letrec-values`, and `define-values` forms (among others) create continuations that receive multiple values.

\[(\text{values } v \ ... ) \rightarrow \text{any} \]
\[v : \text{any/c} \]

Returns the given vs. That is, `values` returns its provided arguments.

Examples:

\[
> (\text{values } 1) \\
1 \\
> (\text{values } 1 \ 2 \ 3) \\
1 \\
2 \\
3 \\
> (\text{values}) \\
\]

\[(\text{call-with-values } \text{generator } \text{receiver}) \rightarrow \text{any} \]
\[\text{generator} : (\rightarrow \text{any}) \]
\[\text{receiver} : \text{procedure?} \]

Calls `generator`, and passes the values that `generator` produces as arguments to `receiver`. Thus, `call-with-values` creates a continuation that accepts any number of values that `receiver` can accept. The `receiver` procedure is called in tail position with respect to the `call-with-values` call.

Examples:

\[
> (\text{call-with-values } (\lambda () (\text{values } 1 \ 2)) \ +) \\
3 \\
> (\text{call-with-values } (\lambda () 1) (\lambda (x \ y) (+ x y))) \\
\text{<procedure>}: \text{arity mismatch; the expected number of arguments does not match the given number} \\
\text{expected: } 2 \\
\text{given: } 1 \\
\]
10.2 Exceptions

See §1.1.14 “Exceptions” for information on the Racket exception model. It is based on a proposal by Friedman, Haynes, and Dybvig [Friedman95].

Whenever a primitive error occurs in Racket, an exception is raised. The value that is passed to the current exception handler for a primitive error is always an instance of the \texttt{exn} structure type. Every \texttt{exn} structure value has a \texttt{message} field that is a string, the primitive error message. The default exception handler recognizes exception values with the \texttt{exn?} predicate and passes the error message to the current error display handler (see \texttt{error-display-handler}).

Primitive procedures that accept a procedure argument with a particular required arity (e.g., \texttt{call-with-input-file}, \texttt{call/cc}) check the argument’s arity immediately, raising \texttt{exn:fail:contract} if the arity is incorrect.

10.2.1 Error Message Conventions

Racket’s error message convention is to produce error messages with the following shape:

```
\langle srcloc\rangle: \langle name\rangle: \langle message\rangle;
\langle continued-message\rangle  \ldots
\langle field\rangle: \langle detail\rangle
\ldots
```

The message starts with an optional source location, \langle srcloc\rangle, which is followed by a colon and space when present. The message continues with an optional \langle name\rangle that usually identifies the complaining function, syntactic form, or other entity, but may also refer to an entity being complained about; the \langle name\rangle is also followed by a colon and space when present.

The \langle message\rangle should be relatively short, and it should be largely independent of specific values that triggered the error. More detailed explanation that requires multiple lines should continue with each line indented by a single space, in which case \langle message\rangle should end in a semi-colon (but the semi-colon should be omitted if \langle continued-message\rangle is not present). Message text should be lowercase—using semi-colons to separate sentences if needed, although long explanations may be better deferred to extra fields.

Specific values that triggered the error or other helpful information should appear in separate \langle field\rangle lines, each of which is indented by two spaces. If a \langle detail\rangle is especially long or takes multiple lines, it should start on its own line after the \langle field\rangle label, and each of its lines should be indented by three spaces. Field names should be all lowercase.

A \langle field\rangle name should end with \ldots if the field provides relatively detailed information that might be distracting in common cases but useful in others. For example, when a contract failure is reported for a particular argument of a function, other arguments to the function
might be shown in an “other arguments...” field. The intent is that fields whose names end in ... might be hidden by default in an environment such as DrRacket.

Make \( \langle \text{field} \rangle \) names as short as possible, relying on \( \langle \text{message} \rangle \) or \( \langle \text{continued message} \rangle \) text to clarify the meaning for a field. For example, prefer “given” to “given turtle” as a field name, where \( \langle \text{message} \rangle \) is something like “given turtle is too sleepy” to clarify that “given” refers to a turtle.

10.2.2 Raising Exceptions

\[
\text{(raise } v \text{ [barrier?]}) \rightarrow \text{any}
\]

\[
v : \text{any/c}
\]

\[
\text{barrier? : any/c = } \#t
\]

Raises an exception, where \( v \) represents the exception being raised. The \( v \) argument can be anything; it is passed to the current exception handler.

If \( \text{barrier?} \) is true, then the call to the exception handler is protected by a continuation barrier, so that multiple returns/escapes are impossible. All exceptions raised by \textit{racket} functions effectively use \textit{raise} with a \#t value for \( \text{barrier?} \).

Breaks are disabled from the time the exception is raised until the exception handler obtains control, and the handler itself is \textit{parameterize-breaked} to disable breaks initially; see \texttt{§10.6 “Breaks”} for more information on breaks.

Examples:

\[
> \text{(with-handlers ([number? (lambda (n)
(+ n 5)])
(raise 18 \#t))}
23
\]

\[
> \text{(struct my-exception exn:fail:user ())}
> \text{(with-handlers ([my-exception? (lambda (e)
(#f)])
(+ 5 (raise (my-exception
"failed"
(current-continuation-marks))))
#f
> \text{(raise 'failed \#t)}
\text{uncaught exception: failed}
\]

\[
\text{(error message-sym)} \rightarrow \text{any}
\]

\[
\text{message-sym : symbol?}
\]

\[
\text{(error message-str v ...)} \rightarrow \text{any}
\]
message-str : string?
  v : any/c
(error who-sym format-str v ...) → any
  who-sym : symbol?
  format-str : string?
  v : any/c

Raises the exception exn:fail, which contains an error string. The different forms produce the error string in different ways:

- (error message-sym) creates a message string by concatenating "error: " with the string form of message-sym. Use this form sparingly.

- (error message-str v ...) creates a message string by concatenating message-str with string versions of the vs (as produced by the current error value conversion handler; see error-value->string-handler). A space is inserted before each v. Use this form sparingly, because it does not conform well to Racket’s error message conventions; consider raise-arguments-error, instead.

- (error who-sym format-str v ...) creates a message string equivalent to the string created by
  
  (format (string-append "~s: " format-str) who-sym v ...)

  When possible, use functions such as raise-argument-error, instead, which construct messages that follow Racket’s error message conventions.

In all cases, the constructed message string is passed to make-exn:fail, and the resulting exception is raised.

Examples:

> (error 'failed)
  error: failed
> (error "failed" 23 'pizza (list 1 2 3))
  failed 23 'pizza '(1 2 3)
> (error 'method-a "failed because no argument supplied")
  method-a: failed because no argument supplied

(raise-user-error message-sym) → any
  message-sym : symbol?
(raise-user-error message-str v ...) → any
  message-str : string?
  v : any/c
(raise-user-error who-sym format-str v ...) → any
Like `error`, but constructs an exception with `make-exn:fail:user` instead of `make-exn:fail`. The default error display handler does not show a “stack trace” for `exn:fail:user` exceptions (see §10.5 “Continuation Marks”), so `raise-user-error` should be used for errors that are intended for end users.

```scheme
(raise-argument-error name expected v) → any
  name : symbol?
  expected : string?
  v : any/c

(raise-argument-error name
  expected
  bad-pos
  v ...) → any
  name : symbol?
  expected : string?
  bad-pos : exact-nonnegative-integer?
  v : any/c
```

Creates an `exn:fail:contract` value and raises it as an exception. The `name` argument is used as the source procedure’s name in the error message. The `expected` argument is used as a description of the expected contract (i.e., as a string, but the string is intended to contain a contract expression).

In the first form, `v` is the value received by the procedure that does not have the expected type.

In the second form, the bad argument is indicated by an index `bad-pos` (counting from 0), and all of the original arguments `v` are provided (in order). The resulting error message names the bad argument and also lists the other arguments. If `bad-pos` is not less than the number of `v`s, the `exn:fail:contract` exception is raised.

Examples:

```scheme
> (define (feed-machine bits)
  (if (not (integer? bits))
    (raise-argument-error 'feed-machine "integer?" bits)
    "fed the machine"))
> (feed-machine 'turkey)
feed-machine: contract violation
  expected: integer?
  given: 'turkey
> (define (feed-cow animal)
```
(if (not (eq? animal 'cow))
  (raise-argument-error 'feed-cow "'cow" animal)
  "fed the cow"))
> (feed-cow 'turkey)

feed-cow: contract violation
  expected: 'cow
  given: 'turkey

> (define (feed-animals cow sheep goose cat)
  (if (not (eq? goose 'goose))
    (raise-argument-error 'feed-animals "'goose" 2 cow sheep goose cat)
    "fed the animals"))
> (feed-animals 'cow 'sheep 'dog 'cat)

feed-animals: contract violation
  expected: 'goose
  given: 'dog
  argument position: 3rd
  other arguments...:
    'cow
    'sheep
    'cat

(raise-result-error name expected v) → any
name : symbol?
expected : string?
v : any/c

(raise-result-error name expected bad-pos v ...) → any
name : symbol?
expected : string?
bad-pos : exact-nonnegative-integer?
v : any/c

Like raise-argument-error, but the error message describe v as a “result” instead of an “argument.”

(raise-arguments-error name message field v ...) → any
name : symbol?
message : string?
field : string?
v : any/c
Creates an `exn:fail:contract` value and raises it as an exception. The `name` is used as the source procedure’s name in the error message. The `message` is the error message; if `message` contains newline characters, each extra line should be suitably indented (with one extra space at the start of each line), but it should not end with a newline character. Each `field` must have a corresponding `v`, and the two are rendered on their own line in the error message; each `v` is formatted using the error value conversion handler (see `error-value->string-handler`), unless `v` is a unquoted-printing string, in which case the string content is displayed without using the error value conversion handler.

Example:

```scheme
> (raise-arguments-error 'eat
 "fish is smaller than its given meal"
 "fish" 12
 "meal" 13)
eat: fish is smaller than its given meal
 fish: 12
 meal: 13
```

```
(raise-range-error name
 type-description
 index-prefix
 index
 in-value
 lower-bound
 upper-bound
 [alt-lower-bound]) → any
```

<table>
<thead>
<tr>
<th>name</th>
<th>symbol?</th>
</tr>
</thead>
<tbody>
<tr>
<td>type-description</td>
<td>string?</td>
</tr>
<tr>
<td>index-prefix</td>
<td>string?</td>
</tr>
<tr>
<td>index</td>
<td>exact-integer?</td>
</tr>
<tr>
<td>in-value</td>
<td>any/c</td>
</tr>
<tr>
<td>lower-bound</td>
<td>exact-integer?</td>
</tr>
<tr>
<td>upper-bound</td>
<td>exact-integer?</td>
</tr>
<tr>
<td>alt-lower-bound</td>
<td>(or/c #f exact-integer?) = #f</td>
</tr>
</tbody>
</table>

Creates an `exn:fail:contract` value and raises it as an exception to report an out-of-range error. The `type-description` string describes the value for which the index is meant to select an element, and `index-prefix` is a prefix for the word “index.” The `index` argument is the rejected index. The `in-value` argument is the value for which the index was meant. The `lower-bound` and `upper-bound` arguments specify the valid range of indices, inclusive; if `upper-bound` is below `lower-bound`, the value is characterized as “empty.” If `alt-lower-bound` is not `#f`, and if `index` is between `alt-lower-bound` and `upper-bound`, then the error is report as `index` being less than the “starting” index `lower-bound`. 

728
Since upper-bound is inclusive, a typical value is one less than the size of a collection—for example, \((\text{sub1 (vector-length vec)})\), \((\text{sub1 (length lst)})\), and so on.

Examples:

```
> (raise-range-error 'vector-ref "vector" "starting"
  5 #(1 2 3 4) 0 3)
vector-ref: starting index is out of range
  starting index: 5
  valid range: [0, 3]
  vector: #(1 2 3 4)
> (raise-range-error 'vector-ref "vector" "ending"
  5 #(1 2 3 4) 0 3)
vector-ref: ending index is out of range
  ending index: 5
  valid range: [0, 3]
  vector: #(1 2 3 4)
> (raise-range-error 'vector-ref "vector" "." 3 #() 0 -1)
vector-ref: index is out of range for empty vector
  index: 3
> (raise-range-error 'vector-ref "vector" "." 1 #(1 2 3 4) 2 3 0)
vector-ref: ending index is smaller than starting index
  ending index: 1
  starting index: 2
  valid range: [0, 3]
  vector: #(1 2 3 4)
```

\[
\text{(raise-type-error name expected v)} \rightarrow \text{any}
\]

name : symbol?
expected : string?
v : any/c

\[
\text{(raise-type-error name expected bad-pos v ...) \rightarrow any}
\]

name : symbol?
expected : string?
bad-pos : exact-nonnegative-integer?
v : any/c

Like raise-argument-error, but with Racket’s old formatting conventions, and where expected is used as a “type” description instead of a contract expression. Use raise-argument-error or raise-result-error, instead.

\[
\text{(raise-mismatch-error name message v ...+ \rightarrow any)}
\]
Similar to `raise-arguments-error`, but using Racket's old formatting conventions, with a required `v` immediately after the first `message` string, and with further `message` strings that are spliced into the message without line breaks or space. Use `raise-arguments-error`, instead.

```scheme
(raise-arity-error name arity-v arg-v ...) -> any
name : (or/c symbol? procedure?)
arity-v : (or/c exact-nonnegative-integer?
       arity-at-least?
       (listof
        (or/c exact-nonnegative-integer?
         arity-at-least?)))
arg-v : any/c
```

Creates an `exn:fail:contract:arity` value and raises it as an exception. The `name` is used for the source procedure’s name in the error message.

The `arity-v` value must be a possible result from `procedure-arity`, except that it does not have to be normalized (see `procedure-arity?` for the details of normalized arities); `raise-arity-error` will normalize the arity and use the normalized form in the error message. If `name` is a procedure, its actual arity is ignored.

The `arg-v` arguments are the actual supplied arguments, which are shown in the error message (using the error value conversion handler; see `error-value->string-handler`); also, the number of supplied `arg-v`s is explicitly mentioned in the message.

Example:

```scheme
> (raise-arity-error 'unite (arity-at-least 13) "Virginia" "Maryland")
unite: arity mismatch:
the expected number of arguments does not match the given number
  expected: at least 13
given: 2
arguments...:
  "Virginia"
  "Maryland"
```

```scheme
(raise-arity-mask-error name mask arg-v ...) -> any
name : (or/c symbol? procedure?)
mask : exact-integer?
arg-v : any/c
```
The same as `raise-arity-error`, but using the arity representation described with `procedure-arity-mask`.

Added in version 7.0.0.11 of package `base`.

```
(raise-result-arity-error name
   arity-v
   detail-str
   result-v ...) \rightarrow any
```

Like `raise-arity-error`, but reports a “result” mismatch instead of an “argument” mismatch. The `name` argument can be `#f` to omit an initial source for the error. The `detail-str` argument, if non-`#f`, should be a string that starts with a newline, since it is added near the end of the generated error message.

Example:

```
> (raise-result-arity-error 'let-values 2 "\n in: example"
   'a 2.0 "three")
let-values: result arity mismatch;
   expected number of values not received
   expected: 2
   received: 3
   in: example
   arguments...:
     'a
     2.0
     "three"
```

Added in version 6.90.0.26 of package `base`.

```
(raise-syntax-error name
   message
   [expr
    sub-expr
    extra-sources
    message-suffix]) \rightarrow any
```

731
Creates an `exn:fail:syntax` value and raises it as an exception. Macros use this procedure to report syntax errors.

The `name` argument is usually `#f` when `expr` is provided; it is described in more detail below. The `message` is used as the main body of the error message; if `message` contains newline characters, each new line should be suitably indented (with one space at the start), and it should not end with a newline character.

The optional `expr` argument is the erroneous source syntax object or S-expression (but the expression `#f` cannot be represented by itself; it must be wrapped as a syntax object). The optional `sub-expr` argument is a syntax object or S-expression (again, `#f` cannot represent itself) within `expr` that more precisely locates the error. Both may appear in the generated error-message text if `error-print-source-location` is `#t`. Source location information in the error-message text is similarly extracted from `sub-expr` or `expr` when at least one is a syntax object and `error-print-source-location` is `#t`.

If `sub-expr` is provided and not `#f`, it is used (in syntax form) for the `exprs` field of the generated exception record, else the `expr` is used if provided and not `#f`. In either case, the syntax object is `consed` onto `extra-sources` to produce the `exprs` field, or `extra-sources` is used directly for `exprs` if neither `expr` nor `sub-expr` is provided and not `#f`. The `extra-sources` argument is also used directly for `exprs` in the unusual case that the `sub-expr` or `expr` that would be included in `exprs` cannot be converted to a syntax object (because it contains a cycle).

The form name used in the generated error message is determined through a combination of the `name`, `expr`, and `sub-expr` arguments:

- When `name` is `#f`, and when `expr` is either an identifier or a syntax pair containing an identifier as its first element, then the form name from the error message is the identifier's symbol.
- When `name` is `#f` and when `expr` is not an identifier or a syntax pair containing an identifier as its first element, then the form name in the error message is `"?"`.
- When `name` is a symbol, then the symbol is used as the form name in the generated error message.

The `message-suffix` string is appended to the end of the error message. If not `""`, it should normally start with a newline and two spaces to add extra fields to the message (see §10.2.1 "Error Message Conventions").

Changed in version 6.90.0.18 of package base: Added the `message-suffix` optional argument.

```scheme
(quoter-printing-string? v) → boolean?
 v : any/c
(unquoted-printing-string s) → unquoted-printing-string?
 s : string?
```

732
A call to an exception handler is parameterize-breaked to disable breaks, and it is wrapped with call-with-exception-handler to install an exception handler that reports both the original and newly raised exceptions via the error display handler and then escapes via the error escape handler.

\[
\text{(uncaught-exception-handler)} \rightarrow \text{(any/c . -> . any)}
\]

\[
\text{(uncaught-exception-handler} f) \rightarrow \text{void?}
\]

\[
f : \text{(any/c . -> . any)}
\]
A parameter that determines an uncaught-exception handler used by `raise` when the relevant continuation has no exception handler installed with `call-with-exception-handler` or `with-handlers`. Unlike exception handlers installed with `call-with-exception-handler`, the uncaught-exception handler must not return a value when called by `raise`; if it returns, an exception is raised (to be handled by an exception handler that reports both the original and newly raised exception).

The default uncaught-exception handler prints an error message using the current error display handler (see `error-display-handler`), unless the argument to the handler is an instance of `exn:break:hang-up`. If the argument to the handler is an instance of `exn:break:hang-up` or `exn:break:terminate`, the default uncaught-exception handler then calls the exit handler with 1, which normally exits or escapes. For any argument, the default uncaught-exception handler then escapes by calling the current error escape handler (see `error-escape-handler`). The call to each handler is parameterized to set `error-display-handler` to the default error display handler, and it is parameterize-broke to disable breaks. The call to the error escape handler is further parameterized to set `error-escape-handler` to the default error escape handler; if the error escape handler returns, then the default error escape handler is called.

When the current error display handler is the default handler, then the error-display call is parameterized to install an emergency error display handler that logs an error (see `log-error`) and never fails.

```
(with-handlers ([pred-expr handler-expr] ...) body ...)```

Evaluates each `pred-expr` and `handler-expr` in the order that they are specified, and then evaluates the `body`s with a new exception handler during its dynamic extent.

The new exception handler processes an exception only if one of the `pred-expr` procedures returns a true value when applied to the exception, otherwise the exception handler is invoked from the continuation of the `with-handlers` expression (by raising the exception again). If an exception is handled by one of the `handler-expr` procedures, the result of the entire `with-handlers` expression is the return value of the handler.

When an exception is raised during the evaluation of `body`s, each predicate procedure `pred-expr` is applied to the exception value; if a predicate returns a true value, the corresponding `handler-expr` procedure is invoked with the exception as an argument. The predicates are tried in the order that they are specified.

Before any predicate or handler procedure is invoked, the continuation of the entire `with-handlers` expression is restored, but also parameterize-broke to disable breaks. Thus, breaks are disabled by default during the predicate and handler procedures (see §10.6 “Breaks”), and the exception handler is the one from the continuation of the `with-handlers` expression.

The `exn:fail?` procedure is useful as a handler predicate to catch all error exceptions.
Avoid using `(lambda (x) #t)` as a predicate, because the `exn:break` exception typically should not be caught (unless it will be re-raised to cooperatively break). Beware, also, of catching and discarding exceptions, because discarding an error message can make debugging unnecessarily difficult; instead of discarding an error message, consider logging it via `log-error` or a logging form created by `define-logger`.

Examples:

```scheme
> (with-handlers ([exn:fail:syntax? (λ (e) (displayln "got a syntax error"))])
  (raise-syntax-error #f "a syntax error"))
> (with-handlers ([exn:fail:syntax? (λ (e) (displayln "got a syntax error"))]
                   [exn:fail? (λ (e) (displayln "fallback clause"))])
  (raise-syntax-error #f "a syntax error"))
```

Like `with-handlers`, but if a `handler-expr` procedure is called, breaks are not explicitly disabled, and the handler call is in tail position with respect to the `with-handlers*` form.

### 10.2.4 Configuring Default Handling

```scheme
(error-escape-handler) → (→ any)
(error-escape-handler proc) → void?
proc : (→ any)
```

A parameter for the `error escape handler`, which takes no arguments and escapes from the dynamic context of an exception. The default error escape handler escapes using `(abort-current-continuation (default-continuation-prompt-tag) void)`.

The error escape handler is normally called directly by an exception handler, in a parameterization that sets the error display handler and error escape handler to the default handlers, and it is normally parameterize-breaked to disable breaks. To escape from a run-time error in a different context, use `raise` or `error`.

Due to a continuation barrier around exception-handling calls, an error escape handler cannot invoke a full continuation that was created prior to the exception, but it can abort to a prompt (see `call-with-continuation-prompt`) or invoke an escape continuation (see `call-with-escape-continuation`).
A parameter for the *error display handler*, which is called by the default exception handler with an error message and the exception value. More generally, the handler’s first argument is a string to print as an error message, and the second is a value representing a raised exception.

The default error display handler displays its first argument to the current error port (determined by the `current-error-port` parameter) and extracts a stack trace (see `continuation-mark-set->context`) to display from the second argument if it is an `exn` value but not an `exn:fail:user` value.

To report a run-time error, use `raise` or procedures like `error`, instead of calling the error display handler directly.

A parameter whose value is used as the maximum number of characters used to print a Racket value that is embedded in a primitive error message.

A parameter whose value is used by the default error display handler as the maximum number of lines of context (or “stack trace”) to print; a single “...” line is printed if more lines are available after the first `cnt` lines. A `0` value for `cnt` disables context printing entirely.

A parameter that determines the *error value conversion handler*, which is used to print a Racket value that is embedded in a primitive error message.
The integer argument to the handler specifies the maximum number of characters that should be used to represent the value in the resulting string. The default error value conversion handler prints the value into a string (using the current global port print handler; see global-port-print-handler). If the printed form is too long, the printed form is truncated and the last three characters of the return string are set to “...”.

If the string returned by an error value conversion handler is longer than requested, the string is destructively "truncated" by setting the first extra position in the string to the null character. If a non-string is returned, then the string "...", is used. If a primitive error string needs to be generated before the handler has returned, the default error value conversion handler is used.

Calls to an error value conversion handler are parameterized to re-install the default error value conversion handler, and to enable printing of unreadable values (see print-unreadable).

```scheme
(error-print-source-location) -> boolean?
(error-print-source-location include?) -> void?
  include?: any/c
```

A parameter that controls whether read and syntax error messages include source information, such as the source line and column or the expression. This parameter also controls the error message when a module-defined variable is accessed before its definition is executed; the parameter determines whether the message includes a module name. Only the message field of an exn:fail:read, exn:fail:syntax, or exn:fail:contract:variable structure is affected by the parameter. The default is #t.

### 10.2.5 Built-in Exception Types

```scheme
(struct exn (message continuation-marks)
  #:extra-constructor-name make-exn
  #:transparent)
message : string?
continuation-marks : continuation-mark-set?
```

The base structure type for exceptions. The message field contains an error message, and the continuation-marks field contains the value produced by (current-continuation-marks) immediately before the exception was raised.

Exceptions raised by Racket form a hierarchy under exn:

```
exn
  exn:fail
    exn:fail:contract
```
(struct exn:fail exn ()
   #:extra-constructor-name make-exn:fail
   #:transparent)

Raised for exceptions that represent errors, as opposed to \texttt{exn:break}.

(\begin{verbatim}
(struct exn:fail:contract exn:fail ()
   #:extra-constructor-name make-exn:fail:contract
   #:transparent)
\end{verbatim})

Raised for errors from the inappropriate run-time use of a function or syntactic form.

(\begin{verbatim}
(struct exn:fail:contract:arity exn:fail:contract ()
   #:extra-constructor-name make-exn:fail:contract:arity
   #:transparent)
\end{verbatim})

Raised when a procedure is applied to the wrong number of arguments.

(\begin{verbatim}
   #:extra-constructor-name make-exn:fail:contract:divide-by-zero
   #:transparent)
\end{verbatim})
Raised for division by exact zero.

\[
\begin{aligned}
(\text{struct exn:fail:contract:non-fixnum-result exn:fail:contract (})
\end{aligned}
\]

#extra-constructor-name

\[
\begin{aligned}
\text{make-exn:fail:contract:non-fixnum-result}
\end{aligned}
\]

#:transparent

Raised by functions like \texttt{fx+} when the result would not be a fixnum.

\[
\begin{aligned}
(\text{struct exn:fail:contract:continuation exn:fail:contract (})
\end{aligned}
\]

#extra-constructor-name

\[
\begin{aligned}
\text{make-exn:fail:contract:continuation}
\end{aligned}
\]

#:transparent

Raised when a continuation is applied where the jump would cross a continuation barrier.

\[
\begin{aligned}
(\text{struct exn:fail:contract:variable exn:fail:contract (id})
\end{aligned}
\]

#extra-constructor-name

\[
\begin{aligned}
\text{make-exn:fail:contract:variable}
\end{aligned}
\]

#:transparent

\[
\begin{aligned}
id : \text{symbol}
\end{aligned}
\]

Raised for a reference to a not-yet-defined top-level variable or module-level variable.

\[
\begin{aligned}
(\text{struct exn:fail:syntax exn:fail (exprs})
\end{aligned}
\]

#extra-constructor-name

\[
\begin{aligned}
\text{make-exn:fail:syntax}
\end{aligned}
\]

#:transparent

\[
\begin{aligned}
\text{exprs : (listof syntax?)}
\end{aligned}
\]

Raised for a syntax error that is not a \texttt{read} error. The \texttt{exprs} indicate the relevant source expressions, least-specific to most-specific.

This structure type implements the \texttt{prop:exn:srclocs} property.

\[
\begin{aligned}
(\text{struct exn:fail:syntax:unbound exn:fail:syntax (})
\end{aligned}
\]

#extra-constructor-name

\[
\begin{aligned}
\text{make-exn:fail:syntax:unbound}
\end{aligned}
\]

#:transparent

Raised by \texttt{#\%top or set!} for an unbound identifier within a module.

\[
\begin{aligned}
(\text{struct exn:fail:syntax:missing-module exn:fail:syntax (path})
\end{aligned}
\]

#extra-constructor-name

\[
\begin{aligned}
\text{make-exn:fail:syntax:missing-module}
\end{aligned}
\]

#:transparent

\[
\begin{aligned}
\text{path : module-path?}
\end{aligned}
\]

Raised by the default module name resolver or default load handler to report a module path—a reported in the \texttt{path} field—whose implementation file cannot be found.
The default module name resolver raises this exception only when it is given a syntax object as its second argument, and the default load handler raises this exception only when the value of current-module-path-for-load is a syntax object (in which case both the exprs field and the path field are determined by the syntax object.

This structure type implements the prop:exn:missing-module property.

```scheme
(struct exn:fail:read exn:fail (srclocs)
   #:extra-constructor-name make-exn:fail:read
   #:transparent)
srclocs : (listof srcloc?)
```

Raised for a read error. The srclocs indicate the relevant source expressions.

```scheme
(struct exn:fail:read:eof exn:fail:read ()
   #:extra-constructor-name make-exn:fail:read:eof
   #:transparent)
```

Raised for a read error, specifically when the error is due to an unexpected end-of-file.

```scheme
(struct exn:fail:read:non-char exn:fail:read ()
   #:extra-constructor-name make-exn:fail:read:non-char
   #:transparent)
```

Raised for a read error, specifically when the error is due to an unexpected non-character (i.e., “special”) element in the input stream.

```scheme
(struct exn:fail:filesystem exn:fail ()
   #:extra-constructor-name make-exn:fail:filesystem
   #:transparent)
```

Raised for an error related to the filesystem (such as a file not found).

```scheme
(struct exn:fail:filesystem:exists exn:fail:filesystem ()
   #:extra-constructor-name make-exn:fail:filesystem:exists
   #:transparent)
```

Raised for an error when attempting to create a file that exists already.

```scheme
(struct exn:fail:filesystem:version exn:fail:filesystem ()
   #:extra-constructor-name make-exn:fail:filesystem:version
   #:transparent)
```

Raised for a version-mismatch error when loading an extension.
### exn:fail:filesystem:errno

```scheme
(struct exn:fail:filesystem:errno exn:fail:filesystem (errno)
  #:extra-constructor-name make-exn:fail:filesystem:errno
  #:transparent)
errno : (cons/c exact-integer? (or/c 'posix 'windows 'gai))
```

Raised for a filesystem error for which a system error code is available. The symbol part of an `errno` field indicates the category of the error code: `posix` indicates a C/Posix `errno` value, `windows` indicates a Windows system error code (under Windows, only), and `gai` indicates a `getaddrinfo` error code (which shows up only in `exn:fail:network:errno` exceptions for operations that resolve hostnames, but is allowed in `exn:fail:filesystem:errno` instances for consistency).

### exn:fail:filesystem:missing-module

```scheme
(struct exn:fail:filesystem:missing-module exn:fail:filesystem
  (path)
  #:extra-constructor-name
  make-exn:fail:filesystem:missing-module
  #:transparent)
path : module-path?
```

Raised by the default module name resolver or default load handler to report a module path—a reported in the `path` field—whose implementation file cannot be found.

The default module name resolver raises this exception only when it is not given a syntax object as its second argument, and the default load handler raises this exception only when the value of `current-module-path-for-load` is not a syntax object.

This structure type implements the `prop:exn:missing-module` property.

### exn:fail:network

```scheme
(struct exn:fail:network exn:fail ()
  #:extra-constructor-name make-exn:fail:network
  #:transparent)
```

Raised for TCP and UDP errors.

### exn:fail:network:errno

```scheme
(struct exn:fail:network:errno exn:fail:network (errno)
  #:extra-constructor-name make-exn:fail:network:errno
  #:transparent)
errno : (cons/c exact-integer? (or/c 'posix 'windows 'gai))
```

Raised for a TCP or UDP error for which a system error code is available, where the `errno` field is as for `exn:fail:filesystem:errno`.
Raised for an error due to insufficient memory, in cases where sufficient memory is at least available for raising the exception.

```scheme
(struct exn:fail:unsupported exn:fail ()
 #:extra-constructor-name make-exn:fail:unsupported
 #:transparent)
```

Raised for an error due to an unsupported feature on the current platform or configuration.

```scheme
(struct exn:fail:user exn:fail ()
 #:extra-constructor-name make-exn:fail:user
 #:transparent)
```

Raised for errors that are intended to be seen by end users. In particular, the default error printer does not show the program context when printing the error message.

```scheme
(struct exn:break exn (continuation)
 #:extra-constructor-name make-exn:break
 #:transparent)
```

Raised asynchronously (when enabled) in response to a break request. The continuation field can be used to resume the interrupted computation in the uncaught-exception handler or call-with-exception-handler (but not with-handlers because it escapes from the exception context before evaluating any predicates or handlers).

```scheme
(struct exn:break:hang-up exn:break ()
 #:extra-constructor-name make-exn:break:hang-up
 #:transparent)
```

Raised asynchronously for hang-up breaks. The default uncaught-exception handler reacts to this exception type by calling the exit handler.

```scheme
(struct exn:break:terminate exn:break ()
 #:extra-constructor-name make-exn:break:terminate
 #:transparent)
```

Raised asynchronously for termination-request breaks. The default uncaught-exception handler reacts to this exception type by calling the exit handler.

```
prop:exn:srclocs : struct-type-property?
```

A property that identifies structure types that provide a list of srcloc values. The property is normally attached to structure types used to represent exception information.
The property value must be a procedure that accepts a single value—the structure type instance from which to extract source locations—and returns a list of `srclocs`. Some error display handlers use only the first returned location.

As an example,

```racket
#lang racket

;; We create a structure that supports the
;; prop:exn:srcloc protocol. It carries
;; with it the location of the syntax that
;; is guilty.
(struct exn:fail:he-who-shall-not-be-named exn:fail
  (a-srcloc)
  #:property prop:exn:srclocs
  (lambda (a-struct)
    (match a-struct
      [(exn:fail:he-who-shall-not-be-named msg marks a-srcloc)
       (list a-srcloc)]))

;; We can play with this by creating a form that
;; looks at identifiers, and only flags specific ones.
(define-syntax (skeeterize stx)
  (syntax-case stx ()
    [(_ expr)
      (cond
        [(and (identifier? #\expr)
            (eq? (syntax-e #\expr) 'voldemort))
         (quasisyntax/loc stx
           (raise (exn:fail:he-who-shall-not-be-named
                      "oh dear don't say his name" (current-continuation-marks)
                      (srcloc '#, (syntax-source #\expr)
                        '#, (syntax-line #\expr)
                        '#, (syntax-column #\expr)
                        '#, (syntax-position #\expr)
                        '#, (syntax-span #\expr))))
          [else
            ;; Otherwise, leave the expression alone.
            '#\expr]])]]

(define (f x)
  (* (skeeterize x) x))

(define (g voldemort)
  (* (skeeterize voldemort) voldemort))
```

743
Examples:
(f 7)
(g 7)
;; The error should highlight the use
;; of voldemort in g.

\[(\text{exn:srclocs? } v) \to \text{boolean?}\]
\[v : \text{any/c}\]

Returns `#t` if `v` has the `prop:exn:srclocs` property, `#f` otherwise.

\[(\text{exn:srclocs-accessor } v)\]
\[\to (\text{exn:srclocs? } . \to . \ (\text{listof srcloc}))\]
\[v : \text{exn:srclocs?}\]

Returns the `srcloc`-getting procedure associated with `v`.

\[(\text{struct srcloc (source line column position span)}\]
\[\ #:\text{extra-constructor-name make-srcloc}\]
\[\ #:\text{transparent}\]
\[\text{source} : \text{any/c}\]
\[\text{line} : (\text{or/c \ exact-positive-integer? \ #f})\]
\[\text{column} : (\text{or/c \ exact-nonnegative-integer? \ #f})\]
\[\text{position} : (\text{or/c \ exact-positive-integer? \ #f})\]
\[\text{span} : (\text{or/c \ exact-nonnegative-integer? \ #f})\]

A source location is most frequently represented by a `srcloc` structure. More generally, a source location has the same information as a `srcloc` structure, but potentially represented or accessed differently. For example, source-location information is accessed from a syntax object with functions like `syntax-source` and `syntax-line`, while `datum->syntax` accepts a source location as a list, vector, or another syntax object. For ports, a combination of `object-name` and `port-next-location` provides location information, especially in a port for which counting has been enabled through `port-count-lines!`.

The fields of a `srcloc` instance are as follows:

- **source** — An arbitrary value identifying the source, often a path (see §15.1 “Paths”).
- **line** — The line number (counts from 1) or `#f` (unknown).
- **column** — The column number (counts from 0) or `#f` (unknown).
- **position** — The starting position (counts from 1) or `#f` (unknown).
- **span** — The number of covered positions (counts from 0) or `#f` (unknown).
See §1.4.16 “Printing Compiled Code” for information about the treatment of `srcloc` values that are embedded in compiled code.

\[
\text{(srcloc->string srcloc) \rightarrow (or/c string? #f)}
\]

\[
\text{srcloc : srcloc?}
\]

Formats `srcloc` as a string suitable for error reporting. A path source in `srcloc` is shown relative to the value of `current-directory-for-user`. The result is `#f` if `srcloc` does not contain enough information to format a string.

\[
\text{prop:exn:missing-module : struct-type-property?}
\]

A property that identifies structure types that provide a module path for a load that fails because a module is not found.

The property value must be a procedure that accepts a single value—the structure type instance from which to extract source locations—and returns a module path.

\[
\text{(exn:missing-module? v) \rightarrow boolean?}
\]

\[
v : \text{any/c}
\]

Returns `#t` if `v` has the `prop:exn:missing-module` property, `#f` otherwise.

\[
\text{(exn:missing-module-accessor v)}
\]

\[
\rightarrow \text{(exn:missing-module? . \rightarrow . module-path?)}
\]

\[
v : \text{exn:srclocs?}
\]

Returns the module path-getting procedure associated with `v`.

### 10.2.6 Additional Exception Functions

\[
\text{(require racket/exn)}
\]

The bindings documented in this section are provided by the `racket/exn` library, not `racket/base` or `racket`.

Added in version 6.3 of package `base`.

\[
\text{(exn->string exn) \rightarrow string?}
\]

\[
\text{exn : (or/c exn? any/c)}
\]

Formats `exn` as a string. If `exn` is an `exn?`, collects and returns the output from the current `(error-display-handler)`; otherwise, simply converts `exn` to a string using `(format "~s\n" exn)`. 745
10.3 Delayed Evaluation

```
(require racket/promise)  package: base
```

The bindings documented in this section are provided by the `racket/promise` and `racket` libraries, but not `racket/base`.

A promise encapsulates an expression to be evaluated on demand via `force`. After a promise has been forced, every later force of the promise produces the same result.

```
(promise? v) → boolean?
  v : any/c
```

Returns `#t` if `v` is a promise, `#f` otherwise.

```
(delay body ...+)
```

Creates a promise that, when forced, evaluates the `bodys` to produce its value. The result is then cached, so further uses of `force` produce the cached value immediately. This includes multiple values and exceptions.

```
(lazy body ...+)
```

Like `delay`, if the last `body` produces a promise when forced, then this promise is forced, too, to obtain a value. In other words, this form creates a composable promise, where the computation of its body is “attached” to the computation of the following promise, and a single `force` iterates through the whole chain, tail-calling each step.

Note that the last `body` of this form must produce a single value, but the value can itself be a `delay` promise that returns multiple values.

The `lazy` form is useful for implementing lazy libraries and languages, where tail calls can be wrapped in a promise.

```
(force v) → any
  v : any/c
```

If `v` is a promise, then the promise is forced to obtain a value. If the promise has not been forced before, then the result is recorded in the promise so that future `forces` on the promise produce the same value (or values). If forcing the promise raises an exception, then the exception is similarly recorded so that forcing the promise will raise the same exception every time.

If `v` is forced again before the original call to `force` returns, then the `exn:fail` exception is raised.

If `v` is not a promise, then it is returned as the result.
(promise-forced? promise) → boolean?
promise : promise?

Returns #t if promise has been forced.

(promise-running? promise) → boolean?
promise : promise?

Returns #t if promise is currently being forced. (Note that a promise can be either running or forced but not both.)

### 10.3.1 Additional Promise Kinds

- (delay/name body ...+)

  Creates a “call-by-name” promise that is similar to delay-promises, except that the resulting value is not cached. This kind of promise is essentially a thunk that is wrapped in a way that force recognizes.

  If a delay/name promise forces itself, no exception is raised, the promise is never considered “running” or “forced” in the sense of promise-running? and promise-forced?.

- (promise/name? promise) → boolean?
promise : any/c

  Returns #t if promise is a promise created with delay/name.

  Added in version 6.3 of package base.

- (delay/strict body ...+)

  Creates a “strict” promise: it is evaluated immediately, and the result is wrapped in a promise value. Note that the body can evaluate to multiple values, and forcing the resulting promise will return these values.

- (delay/sync body ...+)

  Produces a promise where an attempt to force the promise by a thread other than one currently running the promise causes the force to block until a result is available. This kind of promise is also a synchronizable event for use with sync; syncing on the promise does not force it, but merely waits until a value is forced by another thread. The synchronization result is #<void>.

  If a promise created by delay/sync is forced on a thread that is already running the promise, an exception is raised in the same way as for promises created with delay.
(delay/thread body/option ...+)

body/option = body
| #:group thread-group-expr

Like delay/sync, but begins the computation immediately on a newly created thread. The thread is created under the thread group specified by thread-group-expr, which defaults to (make-thread-group). A #:group specification can appear at most once.

Exceptions raised by the bodys are caught as usual and raised only when the promise is forced. Unlike delay/sync, if the thread running body terminates without producing a result or exception, force of the promise raises an exception (instead of blocking).

(delay/idle body/option ...+)

body/option = body
| #:wait-for wait-evt-expr
| #:work-while while-evt-expr
| #:tick tick-secs-expr
| #:use use-ratio-expr

Like delay/thread, but with the following differences:

- the computation does not start until the event produced by wait-evt-expr is ready, where the default is (system-idle-evt);
- the computation thread gets to work only when the process is otherwise idle as determined by while-evt-expr, which also defaults to (system-idle-evt);
- the thread is allowed to run only periodically: out of every tick-secs-expr (defaults to 0.2) seconds, the thread is allowed to run use-ratio-expr (defaults to 0.12) of the time proportionally; i.e., the thread runs for (* tick-secs-expr use-ratio-expr) seconds.

If the promise is forced before the computation is done, it runs the rest of the computation immediately without waiting on events or periodically restricting evaluation.

A #:wait-for, #:work-while, #:tick, or #:use specification can appear at most once.

10.4 Continuations

See §1.1.1 “Sub-expression Evaluation and Continuations” and §1.1.11 “Prompts, Delimited Continuations, and Barriers” for general information about continuations. Racket’s support
for prompts and composable continuations most closely resembles Dorai Sitaram’s \% and \fcontrol operator [Sitaram93].

Racket installs a continuation barrier around evaluation in the following contexts, preventing full-continuation jumps into the evaluation context protected by the barrier:

- applying an exception handler, an error escape handler, or an error display handler (see §10.2 “Exceptions”);
- applying a macro transformer (see §12.4 “Syntax Transformers”), evaluating a compile-time expression, or applying a module name resolver (see §14.4.1 “Resolving Module Names”);
- applying a custom-port procedure (see §13.1.9 “Custom Ports”), an event guard procedure (see §11.2.1 “Events”), or a parameter guard procedure (see §11.3.2 “Parameters”);
- applying a security-guard procedure (see §14.6 “Security Guards”);
- applying a will procedure (see §16.3 “Wills and Executors”); or
- evaluating or loading code from the stand-alone Racket command line (see §18.1 “Running Racket or GRacket”).

In addition, extensions of Racket may install barriers in additional contexts. Finally, call-with-continuation-barrier applies a thunk barrier between the application and the current continuation.

```
(call-with-continuation-prompt proc
  [prompt-tag
    handler]
  arg ...)  → any

proc : procedure?
prompt-tag : continuation-prompt-tag?
  = (default-continuation-prompt-tag)
handler : (or/c procedure? #f) = #f
arg : any/c
```

Applies proc to the given args with the current continuation extended by a prompt. The prompt is tagged by prompt-tag, which must be a result from either default-continuation-prompt-tag (the default) or make-continuation-prompt-tag. The call to call-with-continuation-prompt returns the result of proc.

The handler argument specifies a handler procedure to be called in tail position with respect to the call-with-continuation-prompt call when the installed prompt is the target of an abort-current-continuation call with prompt-tag; the remaining arguments of abort-current-continuation are supplied to the handler procedure. If handler is
for, the default handler accepts a single `abort-thunk` argument and calls `(call-with-continuation-prompt abort-thunk prompt-tag #f)'; that is, the default handler re-installs the prompt and continues with a given thunk.

```
(abort-current-continuation prompt-tag v ...) ➔ any
  prompt-tag : any/c
  v : any/c
```

Resets the current continuation to that of the nearest prompt tagged by `prompt-tag` in the current continuation; if no such prompt exists, the `exn:fail:contract:continuation` exception is raised. The v's are delivered as arguments to the target prompt’s handler procedure.

The protocol for v's supplied to an abort is specific to the `prompt-tag`. When `abort-current-continuation` is used with `(default-continuation-prompt-tag)`, generally, a single thunk should be supplied that is suitable for use with the default prompt handler. Similarly, when `call-with-continuation-prompt` is used with `(default-continuation-prompt-tag)`, the associated handler should generally accept a single thunk argument.

Each thread’s continuation starts with a prompt for `(default-continuation-prompt-tag)` that uses the default handler, which accepts a single thunk to apply (with the prompt intact).

```
(make-continuation-prompt-tag) ➔ continuation-prompt-tag?
(make-continuation-prompt-tag name) ➔ continuation-prompt-tag?
  name : symbol?
```

Creates a prompt tag that is not `equal?` to the result of any other value (including prior or future results from `make-continuation-prompt-tag`). The optional name argument, if supplied, specifies the name of the prompt tag for printing or `object-name`.

Changed in version 7.9.0.13 of package base: The name argument gives the name of the prompt tag.

```
(default-continuation-prompt-tag) ➔ continuation-prompt-tag?
```

Returns a constant prompt tag for which a prompt is installed at the start of every thread’s continuation; the handler for each thread’s initial prompt accepts any number of values and returns. The result of `default-continuation-prompt-tag` is the default tag for any procedure that accepts a prompt tag.

```
(call-with-current-continuation proc [prompt-tag]) ➔ any
  proc : (continuation? . ➔ . any)
  prompt-tag : continuation-prompt-tag?
    = (default-continuation-prompt-tag)
```

750
Captures the current continuation up to the nearest prompt tagged by `prompt-tag`; if no such prompt exists, the `exn:fail:contract:continuation` exception is raised. The truncated continuation includes only continuation marks and `dynamic-wind` frames installed since the prompt.

The captured continuation is delivered to `proc`, which is called in tail position with respect to the `call-with-current-continuation` call.

If the continuation argument to `proc` is ever applied, then it removes the portion of the current continuation up to the nearest prompt tagged by `prompt-tag` (not including the prompt; if no such prompt exists, the `exn:fail:contract:continuation` exception is raised), or up to the nearest continuation frame (if any) shared by the current and captured continuations—whichever is first. While removing continuation frames, `dynamic-wind post-thunks` are executed. Finally, the (unshared portion of the) captured continuation is appended to the remaining continuation, applying `dynamic-wind pre-thunks`.

The arguments supplied to an applied procedure become the result values for the restored continuation. In particular, if multiple arguments are supplied, then the continuation receives multiple results.

If, at application time, a continuation barrier would be introduced by replacing the current continuation with the applied one, then the `exn:fail:contract:continuation` exception is raised.

A continuation can be invoked from the thread (see §11.1 "Threads") other than the one where it was captured.

```scheme
(call/cc proc [prompt-tag]) -> any
proc : (continuation? . -> . any)
prompt-tag : continuation-prompt-tag?
  = (default-continuation-prompt-tag)
```

The `call/cc` binding is an alias for `call-with-current-continuation`.

```scheme
(call-with-composable-continuation proc [prompt-tag]) -> any
proc : (continuation? . -> . any)
prompt-tag : continuation-prompt-tag?
  = (default-continuation-prompt-tag)
```

Similar to `call-with-current-continuation`, but applying the resulting continuation procedure does not remove any portion of the current continuation. Instead, application always extends the current continuation with the captured continuation (without installing any prompts other than those captured in the continuation).

When `call-with-composable-continuation` is called, if a continuation barrier appears in the continuation before the closest prompt tagged by `prompt-tag`, the
exn:fail:contract:continuation exception is raised (because attempting to apply the continuation would always fail).

\[(\text{call-with-escape-continuation } \text{proc}) \to \text{any}\\
\text{proc : (continuation? . } \to . \text{ any)}\]

Like \text{call-with-current-continuation}, but \text{proc} is not called in tail position, and the continuation procedure supplied to \text{proc} can only be called during the dynamic extent of the \text{call-with-escape-continuation} call.

A continuation obtained from \text{call-with-escape-continuation} is actually a kind of prompt. Escape continuations are provided mainly for backwards compatibility, since they pre-date general prompts in Racket. In the BC implementation of Racket, \text{call-with-escape-continuation} is implemented more efficiently than \text{call-with-current-continuation}, so \text{call-with-escape-continuation} can sometimes replace \text{call-with-current-continuation} to improve performance in those older Racket variants.

\[(\text{call/ec } \text{proc}) \to \text{any}\\
\text{proc : (continuation? . } \to . \text{ any)}\]

The \text{call/ec} binding is an alias for \text{call-with-escape-continuation}.

\[(\text{call-in-continuation } \text{k } \text{proc}) \to \text{any}\\
\text{k : continuation?}\\
\text{proc : (} \to \text{ any)}\]

Similar to applying the continuation \text{k}, but instead of delivering values to the continuation, \text{proc} is called with \text{k} as the continuation of the call (so the result of \text{proc} is returned to the continuation). If \text{k} is a composable continuation, the continuation of the call to \text{proc} is the current continuation extended with \text{k}.

Examples:

\[
\begin{align*}
> & (\text{+ 1}\\
& \quad (\text{call/cc (lambda (k)}\\
& \quad \quad (\text{call-in-continuation k (lambda () 4)))))
\end{align*}
\]

5

\[
\begin{align*}
> & (\text{+ 1}\\
& \quad (\text{call/cc (lambda (k)}\\
& \quad \quad (\text{let ([n 0]}\\
& \quad \quad \quad (\text{dynamic-wind void}\\
& \quad \quad \quad \quad (\text{lambda ()}\\
& \quad \quad \quad \quad \quad ; n accessed after post thunk\\
& \quad \quad \quad \quad (\text{call-in-continuation k (lambda () n))})\\
& \quad \quad \quad \quad (\text{lambda ()}\\
& \quad \quad \quad \quad \quad (\text{set! n 4))})))))
\end{align*}
\]

752
> (+ 1
   (with-continuation-mark
    'n 4
    (call/cc (lambda (k)
               (with-continuation-mark
                'n 0
                (call-in-continuation
                 k
                 (lambda ()
                    ; 'n mark accessed in continuation
                    (continuation-mark-set-first #f 'n)))))))))

Added in version 7.6.0.17 of package base.

(let/cc k body ...+)

Equivalent to (call/cc (lambda (k) body ...)).

(let/ec k body ...+)

Equivalent to (call/ec (lambda (k) body ...)).

(call-with-continuation-barrier thunk) → any
   thunk : (-> any)

Applies thunk with a continuation barrier between the application and the current continuation. The results of thunk are the results of the call-with-continuation-barrier call.

(continuation-prompt-available? prompt-tag [cont]) → any
   prompt-tag : continuation-prompt-tag?
   cont : continuation? = (call/cc values)

Returns #t if cont, which must be a continuation, includes a prompt tagged by prompt-tag, #f otherwise.

(continuation? v) → boolean?
   v : any/c

Return #t if v is a continuation as produced by call-with-current-continuation, call-with-composable-continuation, or call-with-escape-continuation, #f otherwise.
(continuation-prompt-tag? v) → boolean?
  v : any/c

Returns #t if v is a continuation prompt tag as produced by default-continuation-prompt-tag or make-continuation-prompt-tag.

@dynamic-wind pre-thunk
  value-thunk
  post-thunk) → any
  pre-thunk : (-> any)
  value-thunk : (-> any)
  post-thunk : (-> any)

Applies its three thunk arguments in order. The value of a dynamic-wind expression is the value returned by value-thunk. The pre-thunk procedure is invoked before calling value-thunk and post-thunk is invoked after value-thunk returns. The special properties of dynamic-wind are manifest when control jumps into or out of the value-thunk application (either due to a prompt abort or a continuation invocation): every time control jumps into the value-thunk application, pre-thunk is invoked, and every time control jumps out of value-thunk, post-thunk is invoked. (No special handling is performed for jumps into or out of the pre-thunk and post-thunk applications.)

When dynamic-wind calls pre-thunk for normal evaluation of value-thunk, the continuation of the pre-thunk application calls value-thunk (with dynamic-wind’s special jump handling) and then post-thunk. Similarly, the continuation of the post-thunk application returns the value of the preceding value-thunk application to the continuation of the entire dynamic-wind application.

When pre-thunk is called due to a continuation jump, the continuation of pre-thunk

  • jumps to a more deeply nested pre-thunk, if any, or jumps to the destination continuation; then
  • continues with the context of the pre-thunk’s dynamic-wind call.

Normally, the second part of this continuation is never reached, due to a jump in the first part. However, the second part is relevant because it enables jumps to escape continuations that are contained in the context of the dynamic-wind call. Furthermore, it means that the continuation marks (see §10.5 “Continuation Marks”) and parameterization (see §11.3.2 “Parameters”) for pre-thunk correspond to those of the dynamic-wind call that installed pre-thunk. The pre-thunk call, however, is parameterize-breaked to disable breaks (see also §10.6 “Breaks”).

Similarly, when post-thunk is called due to a continuation jump, the continuation of post-thunk jumps to a less deeply nested post-thunk, if any, or jumps to a pre-thunk protecting the destination, if any, or jumps to the destination continuation, then continues from the...
post-thunk’s dynamic-wind application. As for pre-thunk, the parameterization of the original dynamic-wind call is restored for the call, and the call is parameterize-breaked to disable breaks.

In both cases, the target for a jump is recomputed after each pre-thunk or post-thunk completes. When a prompt-delimited continuation (see §1.1.11 “Prompts, Delimited Continuations, and Barriers”) is captured in a post-thunk, it might be delimited and instantiated in such a way that the target of a jump turns out to be different when the continuation is applied than when the continuation was captured. There may even be no appropriate target, if a relevant prompt or escape continuation is not in the continuation after the restore; in that case, the first step in a pre-thunk or post-thunk’s continuation can raise an exception.

Examples:

```
> (let ([v (let/ec out
  (dynamic-wind
   (lambda () (display "in "))
   (lambda ()
    (display "pre ")
    (display (call/cc out))
    #f)
   (lambda () (display "out "))))])
  (when v (v "post ")))
in pre out in post out
> (let/ec k0
  (let/ec k1
   (dynamic-wind void
    (lambda () (k0 'cancel))
    (lambda () (k1 'cancel-canceled))))
'cancel-canceled
> (let* ([x (make-parameter 0)]
  [l null]
  [add (lambda (a b)
   (set! l (append l (list (cons a b)))))]
  (let ([k (parameterize ([x 5])
   (dynamic-wind
    (lambda () (add 1 (x)))
    (lambda () (parameterize ([x 6])
     (let ([k+e (let/cc k (cons k void))])
      (add 2 (x))
      ((cdr k+e))
      (car k+e))))))
   (lambda () (add 3 (x))))))]
  (parameterize ([x 7])
   (let/cc esc
    ...))
```

755
10.4.1 Additional Control Operators

```
(define (racket/control)
  (require racket/control)
  (package base)

The bindings documented in this section are provided by the racket/control library, not racket/base or racket.

The racket/control library provides various control operators from the research literature on higher-order control operators, plus a few extra convenience forms. These control operators are implemented in terms of call-with-continuation-prompt, call-with-composable-continuation, etc., and they generally work sensibly together. Many are redundant; for example, reset and prompt are aliases.

```
(call/prompt proc [prompt-tag handler] arg ...) -> any
proc : procedure?
prompt-tag : continuation-prompt-tag?
  = (default-continuation-prompt-tag)
handler : (or/c procedure? #f) = #f
arg : any/c
```

The call/prompt binding is an alias for call-with-continuation-prompt.

```
(abort/cc prompt-tag v ...) -> any
prompt-tag : any/c
v : any/c
```

The abort/cc binding is an alias for abort-current-continuation.

```
(call/comp proc [prompt-tag]) -> any
proc : (continuation? . -> . any)
prompt-tag : continuation-prompt-tag?
  = (default-continuation-prompt-tag)
```

The call/comp binding is an alias for call-with-composable-continuation.

```
(abort v ...) -> any
v : any/c
```

Returns the vs to a prompt using the default continuation prompt tag and the default abort handler.

That is, (abort v ...) is equivalent to
(abort-current-continuation
  (default-continuation-prompt-tag)
  (lambda () (values v ...)))

Example:

>> (prompt
   (printf "start here\n")
   (printf "answer is ~a\n" (+ 2 (abort 3))))
start here
3

(/% expr)
(/% expr handler-exprr)
(/% expr handler-exprr #:tag tag-exprr)
(fcontrol v #:tag prompt-tag) → any
  v : any/c
  prompt-tag : (default-continuation-prompt-tag)

Sitaram’s operators [Sitaram93].

The essential reduction rules are:

(/% val proc) => val
(/% E[(fcontrol val)] proc) => (proc val (lambda (x) E[x]))
  ; where E has no %

When handler-exprr is omitted, % is the same as prompt. If prompt-tag is provided, %
uses specific prompt tags like prompt-at.

Examples:

>> (% (+ 2 (fcontrol 5))
   (lambda (v k)
       (k v)))
7

>> (% (+ 2 (fcontrol 5))
   (lambda (v k)
       v))
5

(prompt expr ...+)
(control id expr ...+)
Among the earliest operators for higher-order control [Felleisen88a, Felleisen88, Sitaram90].

The essential reduction rules are:

\[
\begin{align*}
(prompt \ val) & => \ val \\
(prompt \ E[(control \ k \ expr)]) & => (prompt \ ((\lambda (k) \ expr) \\
& \quad (\lambda (v) \ E[v])))
\end{align*}
\]

; where \( E \) has no prompt

Examples:

\[
\begin{align*}
> (prompt \\
& \quad (+ \ 2 \ (control \ k \ (k \ 5)))) \\
\Rightarrow \ 7
\end{align*}
\]

\[
\begin{align*}
> (prompt \\
& \quad (+ \ 2 \ (control \ k \ 5))) \\
\Rightarrow \ 5
\end{align*}
\]

\[
\begin{align*}
> (prompt \\
& \quad (+ \ 2 \ (control \ k \ (+ \ 1 \ (control \ k1 \ (k1 \ 6)))))) \\
\Rightarrow \ 7
\end{align*}
\]

\[
\begin{align*}
> (prompt \\
& \quad (+ \ 2 \ (control \ k \ (+ \ 1 \ (control \ k1 \ (k \ 6)))))) \\
\Rightarrow \ 8
\end{align*}
\]

\[
\begin{align*}
> (prompt \\
& \quad (+ \ 2 \ (control \ k \ (control \ k1 \ (control \ k2 \ (k2 \ 6)))))) \\
\Rightarrow \ 6
\end{align*}
\]

\[
\begin{align*}
& (prompt-at \ prompt-tag-expr \ expr \ ...+) \\
& (control-at \ prompt-tag-expr \ id \ expr \ ...+)
\end{align*}
\]

Like prompt and control, but using specific prompt tags:

\[
\begin{align*}
& (prompt-at \ tag \ val) => \ val \\
& (prompt-at \ tag \ E[(control-at \ tag \ k \ expr)]) => (prompt-at \ tag \\
& \quad ((\lambda (k) \ expr) \\
& \quad (\lambda (v) \ E[v])))
\end{align*}
\]

; where \( E \) has no prompt-at for \( tag \)

\[
\begin{align*}
& (reset \ expr \ ...+) \\
& (shift \ id \ expr \ ...+)
\end{align*}
\]

Danvy and Filinski’s operators [Danvy90].

The essential reduction rules are:
\[(\text{reset } \text{val}) \Rightarrow \text{val}\]
\[(\text{reset } E[(\text{shift } k \text{ expr})]) \Rightarrow (\text{reset } ((\lambda (k) \text{ expr})
  \hspace{1em} (\lambda (v) (\text{reset } E[v])))\])\]

; where \(E\) has no \(\text{reset}\)

The \(\text{reset}\) and \(\text{prompt}\) forms are interchangeable.

\[(\text{reset-at } \text{prompt-tag-expr expr } \ldots+)\]
\[(\text{shift-at } \text{prompt-tag-expr identifier expr } \ldots+)\]

Like \(\text{reset}\) and \(\text{shift}\), but using the specified prompt tags.

\[(\text{prompt0 } \text{expr } \ldots+)\]
\[(\text{reset0 } \text{expr } \ldots+)\]
\[(\text{contro0 } \text{id } \text{expr } \ldots+)\]
\[(\text{shift0 } \text{id } \text{expr } \ldots+)\]

Generalizations of \(\text{prompt}\), etc. [Shan04].

The essential reduction rules are:

\[(\text{prompt0 } \text{val}) \Rightarrow \text{val}\]
\[(\text{prompt0 } E[(\text{contro0 } k \text{ expr})]) \Rightarrow ((\lambda (k) \text{ expr})
  \hspace{1em} (\lambda (v) E[v]))\]
\[(\text{reset0 } \text{val}) \Rightarrow \text{val}\]
\[(\text{reset0 } E[(\text{shift0 } k \text{ expr})]) \Rightarrow ((\lambda (k) \text{ expr})
  \hspace{1em} (\lambda (v) (\text{reset0 } E[v])))\]

The \(\text{reset0}\) and \(\text{prompt0}\) forms are interchangeable. Furthermore, the following reductions apply:

\[(\text{prompt } E[(\text{contro0 } k \text{ expr})]) \Rightarrow (\text{prompt } ((\lambda (k) \text{ expr})
  \hspace{1em} (\lambda (v) E[v])))\]
\[(\text{reset } E[(\text{shift0 } k \text{ expr})]) \Rightarrow (\text{reset } ((\lambda (k) \text{ expr})
  \hspace{1em} (\lambda (v) (\text{reset0 } E[v])))\)]
\[(\text{prompt0 } E[(\text{contro0 } k \text{ expr})]) \Rightarrow (\text{prompt0 } ((\lambda (k) \text{ expr})
  \hspace{1em} (\lambda (v) E[v])))\]
\[(\text{reset0 } E[(\text{shift0 } k \text{ expr})]) \Rightarrow (\text{reset0 } ((\lambda (k) \text{ expr})
  \hspace{1em} (\lambda (v) (\text{reset } E[v])))\)]

That is, both the \(\text{prompt}/\text{reset}\) and \(\text{contro}/\text{shift}\) sites must agree for 0-like behavior, otherwise the non-0 behavior applies.

\[(\text{prompt0-at } \text{prompt-tag-expr expr } \ldots+)\]
\[(\text{reset0-at } \text{prompt-tag-expr expr } \ldots+)\]
\[(\text{contro0-at } \text{prompt-tag-expr id expr } \ldots+)\]
\[(\text{shift0-at } \text{prompt-tag-expr id expr } \ldots+)\]
Variants of prompt0, etc., that accept a prompt tag.

\[ (\text{spawn proc}) \rightarrow \text{any} \]
\[ \text{proc} : ((\text{any/c} . \rightarrow . \text{any}) . \rightarrow . \text{any}) \]

The operators of Hieb and Dybvig [Hieb90].

The essential reduction rules are:

\[ (\text{prompt-at tag obj}) \Rightarrow \text{obj} \]
\[ (\text{spawn proc}) \Rightarrow (\text{prompt tag (proc (lambda (x) (abort tag x))))} \]
\[ (\text{prompt-at tag E[(abort tag proc)]}) \Rightarrow (\text{proc (lambda (x) (prompt-at tag E[x]))}) \]
; where E has no prompt-at for tag

\[ (\text{splitter proc}) \rightarrow \text{any} \]
\[ \text{proc} : (((\rightarrow \text{any}) . \rightarrow . \text{any}) . \rightarrow . \text{any}) \]
\[ ((\text{continuation?} . \rightarrow . \text{any}) . \rightarrow . \text{any}) \]
\[ . \rightarrow . \text{any} \]

The operator of Queinnec and Serpette [Queinnec91].

The essential reduction rules are:

\[ (\text{splitter proc}) \Rightarrow (\text{prompt-at tag} \)
\[ \text{proc (lambda (thunk)} \]
\[ (\text{abort tag thunk}) \)
\[ (\text{lambda (proc)} \)
\[ (\text{control0-at tag k (proc k)}))) \]
\[ (\text{prompt-at tag E[(abort tag thunk)]}) \Rightarrow (\text{thunk}) \]
; where E has no prompt-at for tag
\[ (\text{prompt-at tag E[(control0-at tag k expr)]}) \Rightarrow ((\text{lambda (k) expr}) \]
\[ (\text{lambda (x) E[x]}) \]
; where E has no prompt-at for tag

\[ (\text{new-prompt}) \rightarrow \text{any} \]
\[ (\text{set prompt-expr expr ...+}) \]
\[ (\text{cupto prompt-expr id expr ...+}) \]

The operators of Gunter et al. [Gunter95].

In this library, new-prompt is an alias for make-continuation-prompt-tag, set is an alias for prompt0-at, and cupto is an alias for control0-at.
10.5 Continuation Marks

See §1.1.10 “Continuation Frames and Marks” and §1.1.11 “Prompts, Delimited Continuations, and Barriers” for general information about continuation marks.

The list of continuation marks for a key \( k \) and a continuation \( C \) that extends \( C_0 \) is defined as follows:

- If \( C \) is an empty continuation, then the mark list is \texttt{null}.
- If \( C \)'s first frame contains a mark \( m \) for \( k \), then the mark list for \( C \) is \texttt{(cons m lst)}), where \texttt{lst} is the mark list for \( k \) in \( C_0 \).
- If \( C \)'s first frame does not contain a mark keyed by \( k \), then the mark list for \( C \) is the mark list for \( C_0 \).

The \texttt{with-continuation-mark} form installs a mark on the first frame of the current continuation (see §3.19 “Continuation Marks: with-continuation-mark”). Procedures such as \texttt{current-continuation-marks} allow inspection of marks.

Whenever Racket creates an exception record for a primitive exception, it fills the \texttt{continuation-marks} field with the value of \texttt{(current-continuation-marks)}, thus providing a snapshot of the continuation marks at the time of the exception.

When a continuation procedure returned by \texttt{call-with-current-continuation} or \texttt{call-with-composable-continuation} is invoked, it restores the captured continuation, and also restores the marks in the continuation’s frames to the marks that were present when \texttt{call-with-current-continuation} or \texttt{call-with-composable-continuation} was invoked.

\begin{verbatim}
(continuation-marks cont [prompt-tag]) -> continuation-mark-set?
  cont : (or/c continuation? thread? #f)
  prompt-tag : continuation-prompt-tag?
                 = (default-continuation-prompt-tag)
\end{verbatim}

Returns an opaque value containing the set of continuation marks for all keys in the continuation \( cont \) (or the current continuation of \( cont \) if it is a thread) up to the prompt tagged by \( prompt-tag \). If \( cont \) is \#f, the resulting set of continuation marks is empty. If \( cont \) is an escape continuation (see §1.1.11 “Prompts, Delimited Continuations, and Barriers”), then the current continuation must extend \( cont \), or the \texttt{exn:fail:contract} exception is raised. If \( cont \) was not captured with respect to \( prompt-tag \) and does not include a prompt for \( prompt-tag \), the \texttt{exn:fail:contract} exception is raised. If \( cont \) is a dead thread, the result is an empty set of continuation marks.

\begin{verbatim}
(current-continuation-marks [prompt-tag])
  -> continuation-mark-set?
\end{verbatim}
prompt-tag : continuation-prompt-tag?
     = (default-continuation-prompt-tag)

Returns an opaque value containing the set of continuation marks for all keys in the current
continuation up to prompt-tag. In other words, it produces the same value as

(call-with-current-continuation
     (lambda (k)
          (continuation-marks k prompt-tag))
     prompt-tag)

(continuation-mark-set->list mark-set
     key-v
     [prompt-tag]) → list?
mark-set : (or/c continuation-mark-set? #f)
key-v : any/c
prompt-tag : continuation-prompt-tag?
     = (default-continuation-prompt-tag)

Returns a newly-created list containing the marks for key-v in mark-set, which is a set of
marks returned by current-continuation-marks or #f as a shorthand for (current-
continuation-marks prompt-tag). The result list is truncated at the first point, if any,
where continuation frames were originally separated by a prompt tagged with prompt-tag.
Producing the result takes time proportional to the size of the continuation reflected by mark-
set.

Changed in version 8.0.0.1 of package base: Changed to allow mark-set as #f.

(continuation-mark-set->list* mark-set
     key-list
     [none-v
      prompt-tag]) → (listof vector?)
mark-set : (or/c continuation-mark-set? #f)
key-list : (listof any/c)
none-v : any/c = #f
prompt-tag : continuation-prompt-tag?
     = (default-continuation-prompt-tag)

Returns a newly-created list containing vectors of marks in mark-set for the keys in key-
list, up to prompt-tag, where a #f value for mark-set is equivalent to (current-
continuation-marks prompt-tag). The length of each vector in the result list is the
same as the length of key-list, and a value in a particular vector position is the value for
the corresponding key in key-list. Values for multiple keys appear in a single vector only
when the marks are for the same continuation frame in mark-set. The none-v argument
is used for vector elements to indicate the lack of a value. Producing the result takes time

762
proportional to the size of the continuation reflected by \texttt{mark-set} times the length of \texttt{key-list}.

Changed in version 8.0.0.1 of package \texttt{base}: Changed to allow \texttt{mark-set} as \texttt{#f}.

\begin{verbatim}
(continuation-mark-set->iterator mark-set
 key-list
 [none-v
  prompt-tag])
 \rightarrow (-> (values (or/c vector? #f) procedure?))

mark-set : (or/c continuation-mark-set? #f)
key-list : (listof any/c)
none-v : any/c = #f
prompt-tag : continuation-prompt-tag?
\hspace{1em} = (default-continuation-prompt-tag)
\end{verbatim}

Like \texttt{continuation-mark-set->list*}, but instead of returning a list of values, returns a functional iterator in the form of a procedure that returns one element of the would-be list and a new iterator function for the rest of the would-be list. An iterator procedure returns \texttt{#f} instead of a vector when no more elements are available; in that case, the returned iterator procedure is like the called one, producing no further values. The time required for each step is proportional to the length of \texttt{key-list} times the size of the segment of the continuation reflected by \texttt{mark-set} between frames that have keys in \texttt{key-list}.

Added in version 7.5.0.7 of package \texttt{base}.

Changed in version 8.0.0.1: Changed to allow \texttt{mark-set} as \texttt{#f}.

\begin{verbatim}
(continuation-mark-set-first mark-set
 key-v
 [none-v
  prompt-tag]) \rightarrow any

mark-set : (or/c continuation-mark-set? #f)
key-v : any/c
none-v : any/c = #f
prompt-tag : continuation-prompt-tag?
\hspace{1em} = (default-continuation-prompt-tag)
\end{verbatim}

Returns the first element of the list that would be returned by \texttt{(continuation-mark-set->list \{or mark-set (current-continuation-marks prompt-tag)\} key-v prompt-tag)}, or \texttt{none-v} if the result would be the empty list.

The result is produced in (amortized) constant time. Typically, this result can be computed more quickly using \texttt{continuation-mark-set-first} than using \texttt{continuation-mark-set->list} or by using \texttt{continuation-mark-set->iterator} and iterating just once.

Although \texttt{#f} and \texttt{(current-continuation-marks prompt-tag)} are equivalent for \texttt{mark-set}, providing \texttt{#f} as \texttt{mark-set} can enable shortcuts that make it even faster.
(call-with-immediate-continuation-mark key-v proc [default-v]) → any

key-v : any/c
proc : (any/c . -> . any)
default-v : any/c = #f

Calls proc with the value associated with key-v in the first frame of the current continuation (i.e., a value that would be replaced if the call to call-with-immediate-continuation-mark were replaced with a with-continuation-mark form using key-v as the key expression). If no such value exists in the first frame, default-v is passed to proc. The proc is called in tail position with respect to the call-with-immediate-continuation-mark call.

This function could be implemented with a combination of with-continuation-mark, current-continuation-marks, and continuation-mark-set->list*, as shown below, but call-with-immediate-continuation-mark is implemented more efficiently; it inspects only the first frame of the current continuation.

; Equivalent, but inefficient:
(define (call-with-immediate-continuation-mark key-v proc [default-v #f])
  (define private-key (gensym))
  (with-continuation-mark private-key #t
    (let ([vecs (continuation-mark-set->list* (current-continuation-marks)
                (list key-v private-key)
                default-v)])
      (proc (vector-ref (car vecs) 0)))))

(make-continuation-mark-key) → continuation-mark-key?
(make-continuation-mark-key sym) → continuation-mark-key?
sym : symbol?

Creates a continuation mark key that is not equal? to the result of any other value (including prior and future results from make-continuation-mark-key). The continuation mark key can be used as the key argument for with-continuation-mark or accessor procedures like continuation-mark-set-first. The mark key can be chaperoned or impersonated, unlike other values that are used as the mark key.

The optional sym argument, if provided, is used when printing the continuation mark.

(continuation-mark-key? v) → boolean?
v : any/c
Returns #t if \( v \) is a mark key created by `make-continuation-mark-key`, #f otherwise.

\[
(\text{continuation-mark-set? } v) \rightarrow \text{boolean?
}\]

\( v : \text{any/c} \)

Returns #t if \( v \) is a mark set created by `continuation-marks` or `current-continuation-marks`, #f otherwise.

\[
(\text{continuation-mark-set->context mark-set}) \rightarrow \text{list?}
\]

\( \text{mark-set} : \text{continuation-mark-set?} \)

Returns a list representing an approximate “stack trace” for \( \text{mark-set} \)’s continuation. The list contains pairs, where the car of each pair contains either #f or a symbol for a procedure name, and the cdr of each pair contains either #f or a `srcloc` value for the procedure’s source location (see §13.1.4 “Counting Positions, Lines, and Columns”); the car and cdr are never both #f.

Conceptually, the stack-trace list is the result of `continuation-mark-set->list` with `mark-set` and Racket’s private key for procedure-call marks. The implementation may be different, however, and the results may merely approximate the correct answer. Thus, while the result may contain useful hints to humans about the context of an expression, it is not reliable enough for programmatic use.

A stack trace is extracted from an exception and displayed by the default error display handler (see `error-display-handler`) for exceptions other than `exn:fail:user` (see `raise-user-error` in §10.2.2 “Raising Exceptions”).

Examples:

```
> (define (extract-current-continuation-marks key)
  (continuation-mark-set->list
   (current-continuation-marks)
   key))
> (with-continuation-mark 'key 'mark
  (extract-current-continuation-marks 'key))
'(mark)
> (with-continuation-mark 'key1 'mark1
  (with-continuation-mark 'key2 'mark2
   (list
    (extract-current-continuation-marks 'key1)
    (extract-current-continuation-marks 'key2))))
'((mark1) (mark2))
> (with-continuation-mark 'key 'mark1
  (with-continuation-mark 'key 'mark2 ; replaces previous mark
   (extract-current-continuation-marks 'key)))
'(mark2)
```
> (with-continuation-mark 'key 'mark1
  (list ; continuation extended to evaluate the argument
    (with-continuation-mark 'key 'mark2
      (extract-current-continuation-marks 'key)))))
'(((mark2 mark1))
> (let loop ([n 1000])
  (if (zero? n)
    (extract-current-continuation-marks 'key)
    (with-continuation-mark 'key n
      (loop (sub1 n)))))
'(1)

10.6 Breaks

A break is an asynchronous exception, usually triggered through an external source controlled by the user, or through the break-thread procedure. For example, the user may type Ctrl-C in a terminal to trigger a break. On some platforms, the Racket process may receive SIGINT, SIGHUP, or SIGTERM; the latter two correspond to hang-up and terminate breaks as reflected by exn:break:hang-up and exn:break:terminate, respectively. Multiple breaks may be collapsed into a single exception, and multiple breaks of different kinds may be collapsed to a single “strongest” break, where a hang-up break is stronger than an interrupt break, and a terminate break is stronger than a hang-up break.

A break exception can only occur in a thread while breaks are enabled. When a break is detected and enabled, the exn:break (or exn:break:hang-up or exn:break:terminate) exception is raised in the thread sometime afterward; if breaking is disabled when break-thread is called, the break is suspended until breaking is again enabled for the thread. While a thread has a suspended break, additional breaks are ignored.

Breaks are enabled through the break-enabled parameter-like procedure and through the parameterize-break form, which is analogous to parameterize. The break-enabled procedure does not represent a parameter to be used with parameterize, because changing the break-enabled state of a thread requires an explicit check for breaks, and this check is incompatible with the tail evaluation of a parameterize expression’s body.

Certain procedures, such as semaphore-wait/enable-break, enable breaks temporarily while performing a blocking action. If breaks are enabled for a thread, and if a break is triggered for the thread but not yet delivered as an exn:break exception, then the break is guaranteed to be delivered before breaks can be disabled in the thread. The timing of exn:break exceptions is not guaranteed in any other way.

Before calling a with-handlers predicate or handler, an exception handler, an error display handler, an error escape handler, an error value conversion handler, or a pre-thunk or post-thunk for a dynamic-wind, the call is parameterize-breaked to disable breaks. Furthermore, breaks are disabled during the transitions among handlers related to excep-
tions, during the transitions between pre-thunks and post-thunks for dynamic-wind, and during other transitions for a continuation jump. For example, if breaks are disabled when a continuation is invoked, and if breaks are also disabled in the target continuation, then breaks will remain disabled from the time of the invocation until the target continuation executes unless a relevant dynamic-wind pre-thunk or post-thunk explicitly enables breaks.

If a break is triggered for a thread that is blocked on a nested thread (see call-in-nested-thread), and if breaks are enabled in the blocked thread, the break is implicitly handled by transferring it to the nested thread.

When breaks are enabled, they can occur at any point within execution, which makes certain implementation tasks subtle. For example, assuming breaks are enabled when the following code is executed,

```scheme
(with-handlers ([exn:break? (lambda (x) (void))])
  (semaphore-wait s))
```

then it is not the case that a #<void> result means the semaphore was decremented or a break was received, exclusively. It is possible that both occur: the break may occur after the semaphore is successfully decremented but before a #<void> result is returned by semaphore-wait. A break exception will never damage a semaphore, or any other built-in construct, but many built-in procedures (including semaphore-wait) contain internal sub-expressions that can be interrupted by a break.

In general, it is impossible using only semaphore-wait to implement the guarantee that either the semaphore is decremented or an exception is raised, but not both. Racket therefore supplies semaphore-wait/enable-break (see §11.2.3 “Semaphores”), which does permit the implementation of such an exclusive guarantee:

```scheme
(parameterize-break #f
  (with-handlers ([exn:break? (lambda (x) (void))])
    (semaphore-wait/enable-break s)))
```

In the above expression, a break can occur at any point until breaks are disabled, in which case a break exception is propagated to the enclosing exception handler. Otherwise, the break can only occur within semaphore-wait/enable-break, which guarantees that if a break exception is raised, the semaphore will not have been decremented.

To allow similar implementation patterns over blocking port operations, Racket provides read-bytes-avail!/enable-break, write-bytes-avail/enable-break, and other procedures.

```scheme
(break-enabled) -> boolean?
(break-enabled on?) -> void?
```

on? : any/c
Gets or sets the break enabled state of the current thread. If `on?` is not supplied, the result is `#t` if breaks are currently enabled, `#f` otherwise. If `on?` is supplied as `#f`, breaks are disabled, and if `on?` is a true value, breaks are enabled.

\[(\text{parameterize-break } \text{boolean-expr } \text{body } \ldots+)\]

Evaluates `boolean-expr` to determine whether breaks are initially enabled while evaluating the `bodys` in sequence. The result of the `parameterize-break` expression is the result of the last `expr`.

As with `parameterize`, a fresh thread cell is allocated to hold the break-enabled state of the continuation, and calls to `break-enabled` within the continuation access or modify the new cell. Unlike a parameter, a mutation to the break setting via `break-enabled` is not inherited by new threads (i.e., the thread cell is not preserved).

\[(\text{current-break-parameterization}) \rightarrow \text{break-parameterization?}\]

Analogous to `(current-parameterization)` (see §11.3.2 “Parameters”), it returns a break parameterization (effectively, a thread cell) that holds the current continuation’s break-enable state.

\[(\text{call-with-break-parameterization } \text{break-param} \ \text{thunk}) \rightarrow \text{any}\]

\[\text{break-param} : \text{break-parameterization}\
\text{thunk} : (\rightarrow \text{any})\]

Analogous to `(call-with-parameterization parameterization thunk)` (see §11.3.2 “Parameters”), calls `thunk` in a continuation whose break-enabled state is in `break-param`. The `thunk` is not called in tail position with respect to the `call-with-break-parameterization` call.

\[(\text{break-parameterization? } v) \rightarrow \text{boolean}\]

\[v : \text{any/c}\]

Returns `#t` if `v` is a break parameterization as produced by `current-break-parameterization`, `#f` otherwise.

Added in version 6.1.1.8 of package base.

### 10.7 Exiting

\[(\text{exit } [v]) \rightarrow \text{any}\]

\[v : \text{any/c} = #t\]
Passes $v$ to the current exit handler. If the exit handler does not escape or terminate the thread, \texttt{#<void>} is returned.

\begin{verbatim}
(exit-handler) \rightarrow \texttt{(any/c . \rightarrow . any)}
(exit-handler proc) \rightarrow \texttt{void}?
proc : \texttt{(any/c . \rightarrow . any)}
\end{verbatim}

A parameter that determines the current exit handler. The exit handler is called by \texttt{exit}.

The default exit handler in the Racket executable takes any argument, calls \texttt{plumber-flush-all} on the original plumber, and shuts down the OS-level Racket process. The argument is used as the OS-level exit code if it is an exact integer between 1 and 255 (which normally means “failure”); otherwise, the exit code is 0, (which normally means “success”).

\begin{verbatim}
(executable-yield-handler) \rightarrow \texttt{(byte? . \rightarrow . any)}
(executable-yield-handler proc) \rightarrow \texttt{void}?
proc : \texttt{(byte? . \rightarrow . any)}
\end{verbatim}

A parameter that determines a procedure to be called as the Racket process is about to exit normally. The procedure associated with this parameter is not called when \texttt{exit} (or, more precisely, the default exit handler) is used to exit early. The argument to the handler is the status code that is returned to the system on exit. The default executable-yield handler simply returns \texttt{#<void>}.

The \texttt{scheme/gui/base} library sets this parameter to wait until all frames are closed, timers stopped, and queued events handled in the main eventspace. See \texttt{scheme/gui/base} for more information.

### 10.8 Unreachable Expressions

\begin{verbatim}
(assert-unreachable) \rightarrow \texttt{none/c}
\end{verbatim}

Reports an assertion failure by raising \texttt{exn:fail:contract}, which is useful as a safe counterpart to \texttt{unsafe-assert-unreachable}.

Added in version 8.0.0.11 of package \texttt{base}.

#### 10.8.1 Customized Unreachable Reporting

\begin{verbatim}
(require racket/unreachable) package: base
\end{verbatim}

The bindings documented in this section are provided by the \texttt{racket/unreachable} library, not \texttt{racket/base} or \texttt{racket}.
Added in version 8.0.0.11 of package base.

```
(with-assert-unreachable
  body ...+)
```

Similar to `(assert-unreachable)`, asserts that the `body` forms should not be reached.

Unless the expression is part of a module that includes `(#%declare #:unsafe)`, then it is equivalent to `(let-values () body ...+)`. The intent is that the `body` forms will raise `exn:fail:contract`.

When a `with-assert-unreachable` expression is part of a module with `(#%declare #:unsafe)`, then it is equivalent to `(unsafe-assert-unreachable)`.
11 Concurrency and Parallelism

Racket supports multiple threads of control within a program, thread-local storage, some primitive synchronization mechanisms, and a framework for composing synchronization abstractions. In addition, the racket/future and racket/place libraries provide support for parallelism to improve performance.

11.1 Threads

See §1.1.12 “Threads” for basic information on the Racket thread model. See also §11.4 “Futures” and §11.5 “Places”.

When a thread is created, it is placed into the management of the current custodian and added to the current thread group. A thread can have any number of custodian managers added through thread-resume.

A thread that has not terminated can be garbage collected (see §1.1.6 “Garbage Collection”) if it is unreachable and suspended or if it is unreachable and blocked on only unreachable events through functions such as semaphore-wait, semaphore-wait/enable-break, channel-put, channel-get, sync, sync/enable-break, or thread-wait. Beware, however, of a limitation on place-channel blocking; see the caveat in §11.5 “Places”.

In GRacket, a handler thread for an eventspace is blocked on an internal semaphore when its event queue is empty. Thus, the handler thread is collectible when the eventspace is unreachable and contains no visible windows or running timers.

A thread can be used as a synchronizable event (see §11.2.1 “Events”). A thread is ready for synchronization when thread-wait would not block; the synchronization result of a thread is the thread itself.

11.1.1 Creating Threads

(thread thunk) → thread?
  thunk : (→ any)

Calls thunk with no arguments in a new thread of control. The thread procedure returns immediately with a thread descriptor value. When the invocation of thunk returns, the thread created to invoke thunk terminates.

(thread? v) → thread?
  v : any/c

Returns #t if v is a thread descriptor, #f otherwise.

(current-thread) → thread?

Returns the thread descriptor for the currently executing thread.
(thread/suspend-to-kill thunk) → thread?
  thunk : (→ any)

Like thread, except that “killing” the thread through kill-thread or custodian-shutdown-all merely suspends the thread instead of terminating it.

(call-in-nested-thread thunk [cust]) → any
  thunk : (→ any)
  cust : custodian? = (current-custodian)

Creates a nested thread managed by cust to execute thunk. (The nested thread’s current custodian is inherited from the creating thread, independent of the cust argument.) The current thread blocks until thunk returns, and the result of the call-in-nested-thread call is the result returned by thunk.

The nested thread’s exception handler is initialized to a procedure that jumps to the beginning of the thread and transfers the exception to the original thread. The handler thus terminates the nested thread and re-raises the exception in the original thread.

If the thread created by call-in-nested-thread dies before thunk returns, the exn:fail exception is raised in the original thread. If the original thread is killed before thunk returns, a break is queued for the nested thread.

If a break is queued for the original thread (with break-thread) while the nested thread is running, the break is redirected to the nested thread. If a break is already queued on the original thread when the nested thread is created, the break is moved to the nested thread. If a break remains queued on the nested thread when it completes, the break is moved to the original thread.

If the thread created by call-in-nested-thread dies while itself in a call to call-in-nested-thread, the outer call to call-in-nested-thread waits for the innermost nested thread to complete, and any breaks pending on the inner threads are moved to the original thread.

11.1.2 Suspending, Resuming, and Killing Threads

(thread-suspend thd) → void?
  thd : thread?

Immediately suspends the execution of thd if it is running. If the thread has terminated or is already suspended, thread-suspend has no effect. The thread remains suspended (i.e., it does not execute) until it is resumed with thread-resume. If the current custodian does not solely manage thd (i.e., some custodian of thd is not the current custodian or a subordinate), the exn:fail:contract exception is raised, and the thread is not suspended.
(thread-resume thd [benefactor]) → void?
  thd : thread?
  benefactor : (or/c thread? custodian? #f) = #f

Resumes the execution of thd if it is suspended and has at least one custodian (possibly added through benefactor, as described below). If the thread has terminated, or if the thread is already running and benefactor is not supplied, or if the thread has no custodian and benefactor is not supplied, then thread-resume has no effect. Otherwise, if benefactor is supplied, it triggers up to three additional actions:

- If benefactor is a thread, whenever it is resumed from a suspended state in the future, then thd is also resumed. (Resuming thd may trigger the resumption of other threads that were previously attached to thd through thread-resume.)

- New custodians may be added to thd's set of managers. If benefactor is a thread, then all of the thread's custodians are added to thd. Otherwise, benefactor is a custodian, and it is added to thd (unless the custodian is already shut down). If thd becomes managed by both a custodian and one or more of its subordinates, the redundant subordinates are removed from thd. If thd is suspended and a custodian is added, then thd is resumed only after the addition.

- If benefactor is a thread, whenever it receives a new managing custodian in the future, then thd also receives the custodian. (Adding custodians to thd may trigger adding the custodians to other threads that were previously attached to thd through thread-resume.)

(kill-thread thd) → void?
  thd : thread?

Terminates the specified thread immediately, or suspends the thread if thd was created with thread/suspend-to-kill. Terminating the main thread exits the application. If thd has already terminated, kill-thread does nothing. If the current custodian does not manage thd (and none of its subordinates manages thd), the exn:fail:contract exception is raised, and the thread is not killed or suspended.

Unless otherwise noted, procedures provided by Racket (and GRacket) are kill-safe and suspend-safe; that is, killing or suspending a thread never interferes with the application of procedures in other threads. For example, if a thread is killed while extracting a character from an input port, the character is either completely consumed or not consumed, and other threads can safely use the port.
Registers a break with the specified thread, where \textit{kind} optionally indicates the kind of break to register. If breaking is disabled in \textit{thd}, the break will be ignored until breaks are re-enabled (see §10.6 “Breaks”).

\begin{verbatim}
(sleep [secs]) \rightarrow \text{void}
secs : (\geq_c 0) = 0
\end{verbatim}

Causes the current thread to sleep until at least \textit{secs} seconds have passed after it starts sleeping. A zero value for \textit{secs} simply acts as a hint to allow other threads to execute. The value of \textit{secs} can be a non-integer to request a sleep duration to any precision; the precision of the actual sleep time is unspecified.

\begin{verbatim}
(thread-running? thd) \rightarrow \text{any}
thd : \text{thread}
\end{verbatim}

Returns \#t if \textit{thd} has not terminated and is not suspended, \#f otherwise.

\begin{verbatim}
(thread-dead? thd) \rightarrow \text{any}
thd : \text{thread}
\end{verbatim}

Returns \#t if \textit{thd} has terminated, \#f otherwise.

### 11.1.3 Synchronizing Thread State

\begin{verbatim}
(thread-wait thd) \rightarrow \text{void}
thd : \text{thread}
\end{verbatim}

Blocks execution of the current thread until \textit{thd} has terminated. Note that (\textit{thread-wait (current-thread)}) deadlocks the current thread, but a break can end the deadlock (if breaking is enabled; see §10.6 “Breaks”).

\begin{verbatim}
(thread-dead-evt thd) \rightarrow \text{evt}
thd : \text{thread}
\end{verbatim}

Returns a synchronizable event (see §11.2.1 “Events”) that is ready for synchronization if and only if \textit{thd} has terminated. Unlike using \textit{thd} directly, however, a reference to the event does not prevent \textit{thd} from being garbage collected (see §1.1.6 “Garbage Collection”). For a given \textit{thd}, \textit{thread-dead-evt} always returns the same (i.e., \texttt{eq?}) result. The synchronization result of a thread-dead event is the thread-dead event itself.

\begin{verbatim}
(thread-resume-evt thd) \rightarrow \text{evt}
thd : \text{thread}
\end{verbatim}
Returns a synchronizable event (see §11.2.1 “Events”) that becomes ready for synchronization when \(\text{thd}\) is running. (If \(\text{thd}\) has terminated, the event never becomes ready.) If \(\text{thd}\) runs and is then suspended after a call to \textit{thread-resume-evt}, the result event remains ready; after each suspend of \(\text{thd}\) a fresh event is generated to be returned by \textit{thread-resume-evt}. The result of the event is \(\text{thd}\), but if \(\text{thd}\) is never resumed, then reference to the event does not prevent \(\text{thd}\) from being garbage collected (see §1.1.6 “Garbage Collection”).

\[
\begin{align*}
\text{(thread-suspend-evt } \text{thd} ) & \rightarrow \text{evt} \\
\text{thd} & : \text{thread}?
\end{align*}
\]

Returns a synchronizable event (see §11.2.1 “Events”) that becomes ready for synchronization when \(\text{thd}\) is suspended. (If \(\text{thd}\) has terminated, the event will never unblock.) If \(\text{thd}\) is suspended and then resumes after a call to \textit{thread-suspend-evt}, the result event remains ready; after each resume of \(\text{thd}\) created a fresh event to be returned by \textit{thread-suspend-evt}. The result of the event is \(\text{thd}\), but if \(\text{thd}\) is never resumed, then reference to the event does not prevent \(\text{thd}\) from being garbage collected (see §1.1.6 “Garbage Collection”).

### 11.1.4 Thread Mailboxes

Each thread has a \textit{mailbox} through which it can receive arbitrary messages. In other words, each thread has a built-in asynchronous channel.

\[
\begin{align*}
\text{(thread-send } \text{thd} \ v \ [\text{fail-thunk}] ) & \rightarrow \text{any} \\
\text{thd} & : \text{thread}?
\end{align*}
\]

\[
\begin{align*}
\text{v} & : \text{any/c} \\
\text{fail-thunk} & : (\text{or/c} \ (-\rightarrow \text{any}) \ #f) \\
& = (\text{lambda} () \ (\text{raise-mismatch-error} \ ...))
\end{align*}
\]

Queues \(v\) as a message to \(\text{thd}\) without blocking. If the message is queued, the result is \#<void>. If \(\text{thd}\) stops running—as in \textit{thread-running}—before the message is queued, then \textit{fail-thunk} is called (through a tail call) if it is a procedure to produce the result, or \#f is returned if \textit{fail-thunk} is \#f.

\[
\begin{align*}
\text{(thread-receive) } & \rightarrow \text{any/c}
\end{align*}
\]

Receives and dequeues a message queued for the current thread, if any. If no message is available, \textit{thread-receive} blocks until one is available.

\[
\begin{align*}
\text{(thread-try-receive) } & \rightarrow \text{any/c}
\end{align*}
\]

Receives and dequeues a message queued for the current thread, if any, or returns \#f immediately if no message is available.
(thread-receive-evt) → evt?

Returns a constant synchronizable event (see §11.2.1 "Events") that becomes ready for synchronization when the synchronizing thread has a message to receive. The synchronization result of a thread-receive event is the thread-receive event itself.

(thread-rewind-receive lst) → void?

lst : list?

Pushes the elements of lst back onto the front of the current thread’s queue. The elements are pushed one by one, so that the first available message is the last element of lst.

11.2 Synchronization

Racket’s synchronization toolbox spans four layers:

- synchronizable events — a general framework for synchronization;
- channels — a primitive that can be used, in principle, to build most other kinds of synchronizable events (except the ones that compose events); and
- semaphores — a simple and especially cheap primitive for synchronization.
- future semaphores — a simple synchronization primitive for use with futures.

11.2.1 Events

A synchronizable event (or just event for short) works with the sync procedure to coordinate synchronization among threads. Certain kinds of objects double as events, including ports and threads. Other kinds of objects exist only for their use as events. Racket’s event system is based on Concurrent ML [Reppy99].

At any point in time, an event is either ready for synchronization, or it is not; depending on the kind of event and how it is used by other threads, an event can switch from not ready to ready (or back), at any time. If a thread synchronizes on an event when it is ready, then the event produces a particular synchronization result.

Synchronizing an event may affect the state of the event. For example, when synchronizing a semaphore, then the semaphore’s internal count is decremented, just as with semaphore-wait. For most kinds of events, however (such as a port), synchronizing does not modify the event’s state.
Racket values that act as synchronizable events include asynchronous channels, channels, custodian boxes, log receivers, place channels, ports, semaphores, subprocesses, TCP listeners, threads, and will executors. Libraries can define new synchronizable events, especially though `prop:evt`.

\[
\text{(evt? } v) \rightarrow \text{boolean?}
\]
\[
v : \text{any/c}
\]

Returns \#t if \( v \) is a synchronizable event, \#f otherwise.

Examples:

\[
> \text{(evt? never-evt)}
\]
\[
\#t
\]
\[
> \text{(evt? (make-channel))}
\]
\[
\#t
\]
\[
> \text{(evt? 5)}
\]
\[
\#f
\]

\[
\text{(sync } \text{evt} \ldots) \rightarrow \text{any}
\]
\[
\text{evt} : \text{evt?}
\]

Blocks as long as none of the synchronizable events \textit{evts} are ready, as defined above.

When at least one \textit{evt} is ready, its synchronization result (often \textit{evt} itself) is returned. If multiple \textit{evts} are ready, one of the \textit{evts} is chosen pseudo-randomly for the result; the \texttt{current-evt-pseudo-random-generator} parameter sets the random-number generator that controls this choice.

Examples:

\[
> \text{(define ch (make-channel))}
\]
\[
> \text{(thread \(\lambda\) () (displayln (sync ch)))}
\]
\[
<\text{thread>}
\]
\[
> \text{(channel-put ch 'hellooooo)}
\]
\[
\text{hellooooo}
\]

Changed in version 6.1.0.3 of package \texttt{base}: Allow 0 arguments instead of 1 or more.

\[
\text{(sync/timeout } \text{timeout} \text{ evt} \ldots) \rightarrow \text{any}
\]
\[
\text{timeout} : \text{(or/c } \#f \text{ (and/c real? (not/c negative?))} \rightarrow \text{any})
\]
\[
\text{evt} : \text{evt?}
\]

Like \texttt{sync} if \texttt{timeout} is \#f. If \texttt{timeout} is a real number, then the result is \#f if \texttt{timeout} seconds pass without a successful synchronization. If \texttt{timeout} is a procedure, then it is called in tail position if polling the \textit{evts} discovers no ready events.
A zero value for **timeout** is equivalent to `(lambda () #f)`. In either case, each **evt** is checked at least once before returning #f or calling **timeout**.

See also **alarm-evt** for an alternative timeout mechanism.

Examples:

```scheme
> (sync/timeout 0.5
  (thread (λ () (sleep 1) (displayln "woke up!"))))
#f
> (sync/timeout
  (λ () (displayln "no ready events"))
  never-evt)
no ready events
```

Changed in version 6.1.0.3 of package base: Allow 1 argument instead of 2 or more.

Like **sync**, but breaking is enabled (see §10.6 “Breaks”) while waiting on the **evts**. If breaking is disabled when **sync/enable-break** is called, then either all **evts** remain unchosen or the exn:break exception is raised, but not both.

```scheme
(sync/enable-break evt ...) → any
evt : evt?
```

Like **sync/enable-break**, but with a timeout as for **sync/timeout**.

```scheme
(choice-evt evt ...) → evt?
evt : evt?
```

Creates and returns a single event that combines the **evts**. Supplying the result to **sync** is the same as supplying each **evt** to the same call.

That is, an event returned by **choice-evt** is ready for synchronization when one or more of the **evts** supplied to **choice-evt** are ready for synchronization. If the choice event is chosen, one of its ready **evts** is chosen pseudo-randomly, and the synchronization result is the chosen **evt**'s synchronization result.

Examples:
(define ch1 (make-channel))
(define ch2 (make-channel))
(define either-channel (choice-evt ch1 ch2))
(thread (λ () (displayln (sync either-channel))))
#<thread>
(channel-put (if (> (random) 0.5) ch1 ch2) 'tuturuu)
tuturuu

(wrap-evt evt wrap) → evt?
  evt : evt?
  wrap : (any/c ... . -> . any)

Creates an event that is ready for synchronization when evt is ready for synchronization, but whose synchronization result is determined by applying wrap to the synchronization result of evt. The number of arguments accepted by wrap must match the number of values for the synchronization result of evt.

The call to wrap is parameterize-breaked to disable breaks initially.

Examples:
> (define ch (make-channel))
> (define evt (wrap-evt ch (λ (v) (format "you've got mail: ~a" v))))
> (thread (λ () (displayln (sync evt))))
#<thread>
(channel-put ch "Dear Alice ...")
you've got mail: Dear Alice ...

(handle-evt evt handle) → handle-evt?
  evt : evt?
  handle : (any/c ... . -> . any)

Like wrap-evt, except that handle is called in tail position with respect to the synchronization request—and without breaks explicitly disabled—when it is not wrapped by wrap-evt, chaperone-evt, or another handle-evt.

Examples:
> (define msg-ch (make-channel))
> (define exit-ch (make-channel))
> (thread (λ ()

779
(let loop ([val 0])
  (printf "val = ~a~n" val)
  (sync (handle-evt
         msg-ch
         (λ (val) (loop val)))
         (handle-evt
          exit-ch
          (λ (val) (displayln val)))))))

val = 0
#<thread>
> (channel-put msg-ch 5)
val = 5
> (channel-put msg-ch 7)
val = 7
> (channel-put exit-ch 'done)
done

(guard-evt maker) → evt?
maker : (-> (or/c evt? any/c))

Creates a value that behaves as an event, but that is actually an event maker.

An event guard returned by guard-evt generates an event when guard is used with sync (or whenever it is part of a choice event used with sync, etc.), where the generated event is the result of calling maker. The maker procedure may be called by sync at most once for a given call to sync, but maker may not be called if a ready event is chosen before guard is even considered.

If maker returns a non-event, then maker’s result is replaced with an event that is ready for synchronization and whose synchronization result is guard.

(nack-guard-evt maker) → evt?
maker : (evt? . -> . (or/c evt? any/c))

Like guard-evt, but when maker is called, it is given a NACK (“negative acknowledgment”) event. After starting the call to maker, if the event from maker is not ultimately chosen as the ready event, then the NACK event supplied to maker becomes ready for synchronization with a #<void> value.

The NACK event becomes ready for synchronization when the event is abandoned when either some other event is chosen, the synchronizing thread is dead, or control escapes from the call to sync (even if nack-guard’s maker has not yet returned a value). If the event returned by maker is chosen, then the NACK event never becomes ready for synchronization.

(poll-guard-evt maker) → evt?
maker : (boolean? . -> . (or/c evt? any/c))
Like `guard-evt`, but when `maker` is called, it is provided a boolean value that indicates whether the event will be used for a poll, `#t`, or for a blocking synchronization, `#f`.

If `#t` is supplied to `maker`, if breaks are disabled, if the polling thread is not terminated, and if polling the resulting event produces a synchronization result, then the event will certainly be chosen for its result.

```
(replace-evt evt maker) → evt?
  evt : evt?
  maker : (any/c ... . -> . (or/c evt? any/c))
```

Like `guard-evt`, but `maker` is called only after `evt` becomes ready for synchronization, and the synchronization result of `evt` is passed to `maker`.

The attempt to synchronize on `evt` proceeds concurrently as the attempt to synchronize on the result `guard` from `replace-evt`; despite that concurrency, if `maker` is called, it is called in the thread that is synchronizing on `guard`. Synchronization can succeed for both `evt` and another synchronized with `guard` at the same time; the single-choice guarantee of synchronization applies only to the result of `maker` and other events synchronized with `guard`.

If `maker` returns a non-event, then `maker`'s result is replaced with an event that is ready for synchronization and whose synchronization result is `guard`.

Added in version 6.1.0.3 of package `base`.

```
always-evt : evt?
```

A constant event that is always ready for synchronization, with itself as its synchronization result.

Example:

```
> (sync always-evt)
<always-evt>
```

```
never-evt : evt?
```

A constant event that is never ready for synchronization.

Example:

```
> (sync/timeout 0.1 never-evt)
#f
```

```
(system-idle-evt) → evt?
```
Returns an event that is ready for synchronization when the system is otherwise idle: if the result event were replaced by `never-evt`, no thread in the system would be available to run. In other words, all threads must be suspended or blocked on events with timeouts that have not yet expired. The system-idle event’s synchronization result is `#<void>`. The result of the `system-idle-evt` procedure is always the same event.

Examples:

```scheme
> (define th (thread (λ () (let loop () (loop)))))
woke up!
> (sync/timeout 0.1 (system-idle-evt))
#f
> (kill-thread th)
> (sync (system-idle-evt))
```

```scheme
(alarm-evt msecs) → evt?
msecs : real?
```

Returns a synchronizable event that is not ready for synchronization when `(current-inexact-milliseconds)` would return a value that is less than `msecs`, and it is ready for synchronization when `(current-inexact-milliseconds)` would return a value that is more than `msecs`. The synchronization result of a alarm event is the alarm event itself.

Examples:

```scheme
> (define alarm (alarm-evt (+ (current-inexact-milliseconds) 100)))
> (sync alarm)
#<alarm-evt>
```

```scheme
(handle-evt? evt) → boolean?
evt : evt?
```

Returns `#t` if `evt` was created by `handle-evt` or by `choice-evt` applied to another event for which `handle-evt?` produces `#t`. For any other event, `handle-evt?` produces `#f`.

Examples:

```scheme
> (handle-evt? never-evt)
#f
> (handle-evt? (handle-evt always-evt values))
#t
```

`prop:evt : struct-type-property?`
A structure type property that identifies structure types whose instances can serve as synchronizable events. The property value can be any of the following:

- An event `evt`: In this case, using the structure as an event is equivalent to using `evt`.

- A procedure `proc` of one argument: In this case, the structure is similar to an event generated by `guard-evt`, except that the would-be guard procedure `proc` receives the structure as an argument, instead of no arguments; also, a non-event result from `proc` is replaced with an event that is already ready for synchronization and whose synchronization result is the structure.

- An exact, non-negative integer between 0 (inclusive) and the number of non-automatic fields in the structure type (exclusive, not counting supertype fields): The integer identifies a field in the structure, and the field must be designated as immutable. If the field contains an object or an event-generating procedure of one argument, the event or procedure is used as above. Otherwise, the structure acts as an event that is never ready.

Instances of a structure type with the `prop:input-port` or `prop:output-port` property are also synchronizable events by virtue of being a port. If the structure type has more than one of `prop:evt`, `prop:input-port`, and `prop:output-port`, then the `prop:evt` value (if any) takes precedence for determining the instance’s behavior as an event, and the `prop:input-port` property takes precedence over `prop:output-port` for synchronization.

Examples:

```scheme
> (struct wt (base val)
   #:property prop:evt (struct-field-index base))
> (define sema (make-semaphore))
> (sync/timeout 0 (wt sema #f))
#f
> (semaphore-post sema)
> (sync/timeout 0 (wt sema #f))
#<semaphore>
> (semaphore-post sema)
> (sync/timeout 0 (wt (lambda (self) (wt-val self)) sema))
#<semaphore>
> (semaphore-post sema)
> (define my-wt (wt (lambda (self) (wrap-evt
   (wt-val self)
   (lambda (x) self)))
   sema))
> (sync/timeout 0 my-wt)
#<wt>
```

For working with foreign libraries, a `prop:evt` value can also be a result of `unsafe-poller`, although that possibility is omitted from the safe contract of `prop:evt`.
A parameter that determines the pseudo-random number generator used by `sync` for events created by `choice-evt`.

### 11.2.2 Channels

A **channel** both synchronizes a pair of threads and passes a value from one to the other. Channels are synchronous; both the sender and the receiver must block until the (atomic) transaction is complete. Multiple senders and receivers can access a channel at once, but a single sender and receiver is selected for each transaction.

Channel synchronization is *fair*: if a thread is blocked on a channel and transaction opportunities for the channel occur infinitely often, then the thread eventually participates in a transaction.

In addition to its use with channel-specific procedures, a channel can be used as a synchronizable event (see §11.2.1 “Events”). A channel is ready for synchronization when `channel-get` would not block; the channel’s synchronization result is the same as the `channel-get` result.

For buffered asynchronous channels, see §11.2.4 “Buffered Asynchronous Channels”

```
(channel? v) → boolean?
  v : any/c
```

Returns #t if `v` is a channel, #f otherwise.

```
(make-channel) → channel?
```

Creates and returns a new channel. The channel can be used with `channel-get`, with `channel-try-get`, or as a synchronizable event (see §11.2.1 “Events”) to receive a value through the channel. The channel can be used with `channel-put` or through the result of `channel-put-evt` to send a value through the channel.

```
(channel-get ch) → any
  ch : channel?
```
Blocks until a sender is ready to provide a value through \( ch \). The result is the sent value.

\[
\text{(channel-try-get \( ch \)) \to \text{any}}
\]

\[
\text{\( ch : \text{channel}\?)}
\]

Receives and returns a value from \( ch \) if a sender is immediately ready, otherwise returns \#f.

\[
\text{(channel-put \( ch \) \( v \)) \to \text{void?}}
\]

\[
\text{\( ch : \text{channel}\?)}
\]

\[
\text{\( v : \text{any/c}\)}
\]

Blocks until a receiver is ready to accept the value \( v \) through \( ch \).

\[
\text{(channel-put-evt \( ch \) \( v \)) \to \text{channel-put-evt?}}
\]

\[
\text{\( ch : \text{channel}\?)}
\]

\[
\text{\( v : \text{any/c}\)}
\]

Returns a fresh synchronizable event for use with \texttt{sync}. The event is ready for synchronization when \( \text{(channel-put \( ch \) \( v \))} \) would not block, and the event’s synchronization result is the event itself.

\[
\text{(channel-put-evt? \( v \)) \to \text{boolean?}}
\]

\[
\text{\( v : \text{any/c}\)}
\]

Returns \#t if \( v \) is a channel-put event produced by \texttt{channel-put-evt}, \#f otherwise.

### 11.2.3 Semaphores

A \textit{semaphore} has an internal counter; when this counter is zero, the semaphore can block a thread’s execution (through \texttt{semaphore-wait}) until another thread increments the counter (using \texttt{semaphore-post}). The maximum value for a semaphore’s internal counter is platform-specific, but always at least 10000.

A semaphore’s counter is updated in a single-threaded manner, so that semaphores can be used for reliable synchronization. Semaphore waiting is \textit{fair}: if a thread is blocked on a semaphore and the semaphore’s internal value is non-zero infinitely often, then the thread is eventually unblocked.

In addition to its use with semaphore-specific procedures, a semaphore can be used as a synchronizable event (see §11.2.1 “Events”). A semaphore is ready for synchronization when \texttt{semaphore-wait} would not block. Upon synchronization, the semaphore’s counter is decremented, and the synchronization result of a semaphore is the semaphore itself.

\[
\text{(semaphore? \( v \)) \to \text{boolean?}}
\]

\[
\text{\( v : \text{any/c}\)}
\]
Returns $\#t$ if $v$ is a semaphore, $\#f$ otherwise.

\[
\text{(make-semaphore [init])} \rightarrow \text{semaphore?}
\]

$init : \text{exact-nonnegative-integer?} = 0$

Creates and returns a new semaphore with the counter initially set to $init$. If $init$ is larger than a semaphore’s maximum internal counter value, the $\text{exn:fail}$ exception is raised.

\[
\text{(semaphore-post sema)} \rightarrow \text{void?}
\]

$sema : \text{semaphore?}$

Increments the semaphore’s internal counter and returns $\langle\text{void}\rangle$. If the semaphore’s internal counter has already reached its maximum value, the $\text{exn:fail}$ exception is raised.

\[
\text{(semaphore-wait sema)} \rightarrow \text{void?}
\]

$sema : \text{semaphore?}$

Blocks until the internal counter for semaphore $sema$ is non-zero. When the counter is non-zero, it is decremented and $\text{semaphore-wait}$ returns $\langle\text{void}\rangle$.

\[
\text{(semaphore-try-wait? sema)} \rightarrow \text{boolean?}
\]

$sema : \text{semaphore?}$

Like $\text{semaphore-wait}$, but $\text{semaphore-try-wait?}$ never blocks execution. If $sema$’s internal counter is zero, $\text{semaphore-try-wait?}$ returns $\#f$ immediately without decrementing the counter. If $sema$’s counter is positive, it is decremented and $\#t$ is returned.

\[
\text{(semaphore-wait/enable-break sema)} \rightarrow \text{void?}
\]

$sema : \text{semaphore?}$

Like $\text{semaphore-wait}$, but breaking is enabled (see §10.6 “Breaks”) while waiting on $sema$. If breaking is disabled when $\text{semaphore-wait/enable-break}$ is called, then either the semaphore’s counter is decremented or the $\text{exn:break}$ exception is raised, but not both.

\[
\text{(semaphore-peek-evt sema)} \rightarrow \text{semaphore-peek-evt?}
\]

$sema : \text{semaphore?}$

Creates and returns a new synchronizable event (for use with $\text{sync}$, for example) that is ready for synchronization when $sema$ is ready, but synchronizing the event does not decrement $sema$’s internal count. The synchronization result of a semaphore-peek event is the semaphore-peek event itself.

\[
\text{(semaphore-peek-evt? v)} \rightarrow \text{boolean?}
\]

$v : \text{any/c}$
Returns #t if \( v \) is a semaphore wrapper produced by `semaphore-peek-evt`, #f otherwise.

\[
\text{(call-with-semaphore\ sema \ proc [\text{try-fail-thunk}] \ arg \ldots) \rightarrow \text{any}}
\]

\text{sema : semaphore?}
\text{proc : procedure?}
\text{try-fail-thunk : (or/c (-> any) #f) = #f}
\text{arg : any/c}

Waits on \text{sema} using `semaphore-wait`, calls \text{proc} with all \text{args}, and then posts to \text{sema}.

A continuation barrier blocks full continuation jumps into or out of \text{proc} (see §1.1.11 "Prompts, Delimited Continuations, and Barriers"), but escape jumps are allowed, and \text{sema} is posted on escape. If \text{try-fail-thunk} is provided and is not #f, then \text{semaphore-try-wait?} is called on \text{sema} instead of \text{semaphore-wait}, and \text{try-fail-thunk} is called if the wait fails.

\[
\text{(call-with-semaphore/enable-break\ sema \ proc [\text{try-fail-thunk}] \ arg \ldots) \rightarrow \text{any}}
\]

\text{sema : semaphore?}
\text{proc : procedure?}
\text{try-fail-thunk : (or/c (-> any) #f) = #f}
\text{arg : any/c}

Like \text{call-with-semaphore}, except that \text{semaphore-wait/enable-break} is used with \text{sema} in non-try mode. When \text{try-fail-thunk} is provided and not #f, then breaks are enabled around the use of \text{semaphore-try-wait?} on \text{sema}.

### 11.2.4 Buffered Asynchronous Channels

\[
\text{(require racket/async-channel) \hspace{1em} package: base}
\]

The bindings documented in this section are provided by the `racket/async-channel` library, not `racket/base` or `racket`.

**Creating and Using Asynchronous Channels**

An \textit{asynchronous channel} is like a channel, but it buffers values so that a send operation does not wait on a receive operation.

In addition to its use with procedures that are specific to asynchronous channels, an asynchronous channel can be used as a synchronizable event (see §11.2.1 “Events”). An asynchronous channel is ready for synchronization when \text{async-channel-get} would not block;
the asynchronous channel’s synchronization result is the same as the async-channel-get result.

\[
(\text{async-channel? } v) \rightarrow \text{ boolean?}
\]
\[
v : \text{ any/c}
\]

Returns \#t if \( v \) is an asynchronous channel, \#f otherwise.

\[
(\text{make-async-channel [limit])} \rightarrow \text{ async-channel?}
\]
\[
\text{limit} : (\text{or/c exact-positive-integer? } \#f) = \#f
\]

Returns an asynchronous channel with a buffer limit of \( \text{limit} \) items. A get operation blocks when the channel is empty, and a put operation blocks when the channel has \( \text{limit} \) items already. If \( \text{limit} \) is \#f, the channel buffer has no limit (so a put never blocks).

\[
(\text{async-channel-get ach}) \rightarrow \text{ any/c}
\]
\[
\text{ach} : \text{ async-channel?}
\]

Blocks until at least one value is available in \( \text{ach} \), and then returns the first of the values that were put into \( \text{async-channel} \).

\[
(\text{async-channel-try-get ach}) \rightarrow \text{ any/c}
\]
\[
\text{ach} : \text{ async-channel?}
\]

If at least one value is immediately available in \( \text{ach} \), returns the first of the values that were put into \( \text{ach} \). If \( \text{async-channel} \) is empty, the result is \#f.

\[
(\text{async-channel-put ach v}) \rightarrow \text{ void?}
\]
\[
\text{ach} : \text{ async-channel?}
\]
\[
v : \text{ any/c}
\]

Puts \( v \) into \( \text{ach} \), blocking if \( \text{ach} \)'s buffer is full until space is available.

\[
(\text{async-channel-put-evt ach v}) \rightarrow \text{evt?}
\]
\[
\text{ach} : \text{ async-channel?}
\]
\[
v : \text{ any/c}
\]

Returns a synchronizable event that is ready for synchronization when \( (\text{async-channel-put ach v}) \) would return a value (i.e., when the channel holds fewer values already than its limit); the synchronization result of a asynchronous channel-put event is the asynchronous channel-put event itself.

Examples:
(define (server input-channel output-channel)
  (thread (lambda ()
    (define (get)
      (async-channel-get input-channel))
    (define (put x)
      (async-channel-put output-channel x))
    (define (do-large-computation)
      (sqrt 9))
    (let loop ([data (get)])
      (case data
        [(quit) (void)]
        [(add) (begin
          (put (+ 1 (get)))
          (loop (get)))]
        [(long) (begin
          (put (do-large-computation))
          (loop (get)))])))
  (define to-server (make-async-channel))
  (define from-server (make-async-channel))

  > (server to-server from-server)
  #<thread>
  > (async-channel? to-server)
  #t
  > (printf "Adding 1 to 4\n")
  Adding 1 to 4
  > (async-channel-put to-server 'add)
  > (async-channel-put to-server 4)
  > (printf "Result is ~a\n" (async-channel-get from-server))
  Result is 5
  > (printf "Ask server to do a long computation\n")
  Ask server to do a long computation
  > (async-channel-put to-server 'long)
  > (printf "I can do other stuff\n")
  I can do other stuff
  > (printf "Ok, computation from server is ~a\n"
    (async-channel-get from-server))
  Ok, computation from server is 3
  > (async-channel-put to-server 'quit)

Contracts and Impersonators on Asynchronous Channels

(= contract? (async-channel/c c))

Returns a contract that recognizes asynchronous channels. Values put into or retrieved from
the channel must match \( c \).

If the \( c \) argument is a flat contract or a chaperone contract, then the result will be a chaperone contract. Otherwise, the result will be an impersonator contract.

When an \texttt{async-channel/c} contract is applied to an asynchronous channel, the result is not \texttt{eq?} to the input. The result will be either a chaperone or impersonator of the input depending on the type of contract.

\[
\texttt{(impersonate-async-channel channel get-proc put-proc prop prop-val ...)}
\]
\[
\rightarrow \texttt{(and/c async-channel? impersonator?)}
\]
\[
\texttt{channel : async-channel?}
\]
\[
\texttt{get-proc : (any/c . -> . any/c)}
\]
\[
\texttt{put-proc : (any/c . -> . any/c)}
\]
\[
\texttt{prop : impersonator-property?}
\]
\[
\texttt{prop-val : any}
\]

Returns an impersonator of \texttt{channel}, which redirects the \texttt{async-channel-get} and \texttt{async-channel-put} operations.

The \texttt{get-proc} must accept the value that \texttt{async-channel-get} produces on \texttt{channel}; it must produce a replacement value, which is the result of the get operation on the impersonator.

The \texttt{put-proc} must accept the value passed to \texttt{async-channel-put} called on \texttt{channel}; it must produce a replacement value, which is the value passed to the put procedure called on the original channel.

The \texttt{get-proc} and \texttt{put-proc} procedures are called for all operations that get or put values from the channel, not just \texttt{async-channel-get} and \texttt{async-channel-put}.

Pairs of \texttt{prop} and \texttt{prop-val} (the number of arguments to \texttt{impersonate-async-channel} must be odd) add impersonator properties or override impersonator property values of \texttt{channel}.

\[
\texttt{(chaperone-async-channel channel get-proc put-proc prop prop-val ...)}
\]
\[
\rightarrow \texttt{(and/c async-channel? chaperone?)}
\]
Like `impersonate-async-channel`, but the `get-proc` procedure must produce the same value or a chaperone of the original value, and `put-proc` must produce the same value or a chaperone of the original value.

### 11.3 Thread-Local Storage

Thread cells provides primitive support for thread-local storage. Parameters combine thread cells and continuation marks to support thread-specific, continuation-specific binding.

#### 11.3.1 Thread Cells

A *thread cell* contains a thread-specific value; that is, it contains a specific value for each thread, but it may contain different values for different threads. A thread cell is created with a default value that is used for all existing threads. When the cell’s content is changed with `thread-cell-set!`, the cell’s value changes only for the current thread. Similarly, `thread-cell-ref` obtains the value of the cell that is specific to the current thread.

A thread cell’s value can be *preserved*, which means that when a new thread is created, the cell’s initial value for the new thread is the same as the creating thread’s current value. If a thread cell is non-preserved, then the cell’s initial value for a newly created thread is the default value (which was supplied when the cell was created).

Within the current thread, the current values of all preserved threads cells can be captured through `current-preserved-thread-cell-values`. The captured set of values can be imperatively installed into the current thread through another call to `current-preserved-thread-cell-values`. The capturing and restoring threads can be different.

```scheme
(thread-cell? v) → boolean?
  v : any/c
```

Returns #t if `v` is a thread cell, #f otherwise.

```scheme
(make-thread-cell v [preserved?]) → thread-cell?
  v : any/c
  preserved? : any/c = #f
```
Creates and returns a new thread cell. Initially, \( v \) is the cell’s value for all threads. If \( \texttt{preserved?} \) is true, then the cell’s initial value for a newly created thread is the creating thread’s value for the cell, otherwise the cell’s value is initially \( v \) in all future threads.

\[
(\text{thread-cell-ref } cell) \rightarrow \text{any} \\
\text{cell : thread-cell?}
\]

Returns the current value of \( cell \) for the current thread.

\[
(\text{thread-cell-set! } cell \ v) \rightarrow \text{any} \\
\text{cell : thread-cell?} \\
\text{v : any/c}
\]

Sets the value in \( cell \) to \( v \) for the current thread.

Examples:

\[
> (\text{define cnp (make-thread-cell '(nerve) #f)}) \\
> (\text{define cp (make-thread-cell '(cancer) #t)}) \\
> (\text{thread-cell-ref cnp}) \\
\quad '(\text{nerve}) \\
> (\text{thread-cell-ref cp}) \\
\quad '(\text{cancer}) \\
> (\text{thread-cell-set! cnp '(nerve nerve)}) \\
> (\text{thread-cell-set! cp '(cancer cancer)}) \\
> (\text{thread-cell-ref cnp}) \\
\quad '(\text{nerve nerve}) \\
> (\text{thread-cell-ref cp}) \\
\quad '(\text{cancer cancer}) \\
> (\text{define ch (make-channel)}) \\
> (\text{thread (lambda ()}) \\
\quad (\text{channel-put ch (thread-cell-ref cnp)}) \\
\quad (\text{channel-put ch (thread-cell-ref cp)}) \\
\quad (\text{channel-get ch}) \\
\quad (\text{channel-put ch (thread-cell-ref cp)})) \\
\quad #<\text{thread}> \\
> (\text{channel-get ch}) \\
\quad '(\text{nerve}) \\
> (\text{channel-get ch}) \\
\quad '(\text{cancer cancer}) \\
> (\text{thread-cell-set! cp '(cancer cancer cancer)}) \\
> (\text{thread-cell-ref cp}) \\
\quad '(\text{cancer cancer cancer}) \\
> (\text{channel-put ch 'ok}) \\
> (\text{channel-get ch}) \\
\quad '(\text{cancer cancer})
\]
(current-preserved-thread-cell-values) → thread-cell-values?
(current-preserved-thread-cell-values thread-cell-vals) → void?
thread-cell-vals : thread-cell-values?

When called with no arguments, this procedure produces a thread-cell-vals that represents the current values (in the current thread) for all preserved thread cells.

When called with a thread-cell-vals generated by a previous call to current-preserved-thread-cell-values, the values of all preserved thread cells (in the current thread) are set to the values captured in thread-cell-vals; if a preserved thread cell was created after thread-cell-vals was generated, then the thread cell’s value for the current thread reverts to its initial value.

(thread-cell-values? v) → boolean?
  v : any/c

Returns #t if v is a set of thread cell values produced by current-preserved-thread-cell-values, #f otherwise.

11.3.2 Parameters

See §1.1.13 “Parameters” for basic information on the parameter model. Parameters correspond to preserved thread fluids in Scsh [Gasbichler02].

To parameterize code in a thread- and continuation-friendly manner, use parameterize. The parameterize form introduces a fresh thread cell for the dynamic extent of its body expressions.

When a new thread is created, the parameterization for the new thread’s initial continuation is the parameterization of the creator thread. Since each parameter’s thread cell is preserved, the new thread “inherits” the parameter values of its creating thread. When a continuation is moved from one thread to another, settings introduced with parameterize effectively move with the continuation.

In contrast, direct assignment to a parameter (by calling the parameter procedure with a value) changes the value in a thread cell, and therefore changes the setting only for the current thread. Consequently, as far as the memory manager is concerned, the value originally associated with a parameter through parameterize remains reachable as long the continuation is reachable, even if the parameter is mutated.

(make-parameter v [guard name]) → parameter?
  v : any/c
  guard : (or/c (any/c . -> . any) #f) = #f
  name : symbol? = 'parameter-procedure

Returns a new parameter procedure. The value of the parameter is initialized to \( v \) in all threads.

If \textit{guard} is not \#f, it is used as the parameter’s guard procedure. A guard procedure takes one argument. Whenever the parameter procedure is applied to an argument, the argument is passed on to the guard procedure. The result returned by the guard procedure is used as the new parameter value. A guard procedure can raise an exception to reject a change to the parameter’s value. The \textit{guard} is not applied to the initial \( v \).

The \textit{name} argument is used as the parameter procedure’s name as reported by \texttt{object-name}.

Changed in version 7.4.0.6 of package \texttt{base}: Added the \textit{name} argument.

\begin{verbatim}
\begin{itemize}
\item \texttt{(parameterize ([parameter-expr value-expr] ...)}
\item \texttt{body ...+)}
\end{itemize}
\end{verbatim}

The result of a \texttt{parameterize} expression is the result of the last \texttt{body}. The \texttt{parameter-exprs} determine the parameters to set, and the \texttt{value-exprs} determine the corresponding values to install while evaluating the \texttt{bodys}. The \texttt{parameter-exprs} and \texttt{value-exprs} are evaluated left-to-right (interleaved), and then the parameters are bound in the continuation to preserved thread cells that contain the values of the \texttt{value-exprs}; the result of each \texttt{parameter-expr} is checked with \texttt{parameter?} just before it is bound. The last \texttt{body} is in tail position with respect to the entire \texttt{parameterize} form.

Outside the dynamic extent of a \texttt{parameterize} expression, parameters remain bound to other thread cells. Effectively, therefore, old parameters settings are restored as control exits the \texttt{parameterize} expression.

If a continuation is captured during the evaluation of \texttt{parameterize}, invoking the continuation effectively re-introduces the parameterization, since a parameterization is associated to a continuation via a continuation mark (see \S 10.5 “Continuation Marks”) using a private key.

Examples:

\begin{verbatim}
> (parameterize ([exit-handler (lambda (x) 'no-exit)])
  (exit))
> (define p1 (make-parameter 1))
> (define p2 (make-parameter 2))
> (parameterize ([p1 3]
  [p2 (p1)])
  (cons (p1) (p2)))
'(3 . 1)
> (let ([k (let/cc out
  (parameterize ([p1 2])

\end{verbatim}
(p1 3)
  (cons (let/cc k
      (out k))
    (p1))))]])

(if (procedure? k)
  (k (p1))
  k))
'(1 . 3)
> (define ch (make-channel))
> (parameterize ([p1 0])
  (thread (lambda ()
    (channel-put ch (cons (p1) (p2))))))
#<thread>
> (channel-get ch)
'(0 . 2)
> (define k-ch (make-channel))
> (define (send-k)
  (parameterize ([p1 0])
    (thread (lambda ()
      (let/ec esc
        (channel-put ch
          ((let/cc k
              (channel-put k-ch k)
            (esc)))))))))
> (send-k)
#<thread>
> (thread (lambda ()
            ((channel-get k-ch)
              (let [v (p1)]
                (lambda () v))))))
#<thread>
> (channel-get ch)
1
> (send-k)
#<thread>
> (thread (lambda ()
            ((channel-get k-ch) p1)))
#<thread>
> (channel-get ch)
0

(parameterize* ((parameter-expr value-expr) ...)
  body ...*)

Analogous to let* compared to let, parameterize* is the same as a nested series of single-parameter parameterize forms.
(make-derived-parameter parameter
guard
wrap) → parameter?

parameter : parameter?
guard : (any/c . -> . any)
wrap : (any/c . -> . any)

Returns a parameter procedure that sets or retrieves the same value as parameter, but with:

• guard applied when setting the parameter (before any guard associated with parameter), and
• wrap applied when obtaining the parameter’s value.

See also chaperone-procedure, which can also be used to guard parameter procedures.

(parameter? v) → boolean?

v : any/c

Returns #t if v is a parameter procedure, #f otherwise.

(parameter-procedure=? a b) → boolean?
a : parameter?
b : parameter?

Returns #t if the parameter procedures a and b always modify the same parameter with the same guards (although possibly with different chaperones), #f otherwise.

(current-parameterization) → parameterization?

Returns the current continuation’s parameterization.

call-with-parameterization parameterization thunk → any

parameterization : parameterization?
thunk : (-> any)

Calls thunk (via a tail call) with parameterization as the current parameterization.

(parameterization? v) → boolean?
v : any/c

Returns #t if v is a parameterization returned by current-parameterization, #f otherwise.
11.4 Futures

(requires racket/future) 

The bindings documented in this section are provided by the racket/future and racket libraries, but not racket/base.

The future and touch functions from racket/future provide access to parallelism as supported by the hardware and operating system. In contrast to thread, which provides concurrency for arbitrary computations without parallelism, future provides parallelism for limited computations. A future executes its work in parallel (assuming that support for parallelism is available) until it detects an attempt to perform an operation that is too complex for the system to run safely in parallel. Similarly, work in a future is suspended if it depends in some way on the current continuation, such as raising an exception. A suspended computation for a future is resumed when touch is applied to the future.

“Safe” parallel execution of a future means that all operations provided by the system must be able to enforce contracts and produce results as documented. “Safe” does not preclude concurrent access to mutable data that is visible in the program. For example, a computation in a future might use set! to modify a shared variable, in which case concurrent assignment to the variable can be visible in other futures and threads. Furthermore, guarantees about the visibility of effects and ordering are determined by the operating system and hardware—which rarely support, for example, the guarantee of sequential consistency that is provided for thread-based concurrency; see also §11.7 “Machine Memory Order”. At the same time, operations that seem obviously safe may have a complex enough implementation internally that they cannot run in parallel. See also §20.1 “Parallelism with Futures” in The Racket Guide.

A future never runs in parallel if all of the custodians that allow its creating thread to run are shut down. Such futures can execute through a call to touch, however.

11.4.1 Creating and Touching Futures

(future thunk) → future? 

thunk : (→ any)

(touch f) → any

f : future?

The future procedure returns a future value that encapsulates thunk. The touch function forces the evaluation of the thunk inside the given future, returning the values produced by thunk. After touch forces the evaluation of a thunk, the resulting values are retained by the future in place of thunk, and additional touches of the future return those values.

Between a call to future and touch for a given future, the given thunk may run speculatively in parallel to other computations, as described above.
Example:

```racket
> (let ([f (future (lambda () (+ 1 2)))]
      (list (+ 3 4) (touch f)))
'((7 3)

(futures-enabled?) -> boolean?

Returns whether parallel support for futures is enabled in the current Racket configuration.

(current-future) -> (or/c #f future?)

Returns the descriptor of the future whose thunk execution is the current continuation; that is, if a future descriptor f is returned, (touch f) will produce the result of the current continuation. If a future thunk itself uses touch, future-thunk executions can be nested, in which case the descriptor of the most immediately executing future is returned. If the current continuation does not return to the touch of any future, the result is #f.

(future? v) -> boolean?
v : any/c

Returns #t if v is a future value, #f otherwise.

(would-be-future thunk) -> future?
thunk : (-> any)

Returns a future that never runs in parallel, but that consistently logs all potentially “unsafe” operations during the execution of the future’s thunk (i.e., operations that interfere with parallel execution).

With a normal future, certain circumstances might prevent the logging of unsafe operations. For example, when executed with debug-level logging,

```racket
(touch (future (lambda ()
    (printf "hello1")
    (printf "hello2")
    (printf "hello3")))
```n
might log three messages, one for each printf invocation. However, if the touch is performed before the future has a chance to start running in parallel, the future thunk evaluates in the same manner as any ordinary thunk, and no unsafe operations are logged. Replacing future with would-be-future ensures the logging of all three calls to printf.

(processor-count) -> exact-positive-integer?
Returns the number of parallel computation units (e.g., processors or cores) that are available on the current machine.

```
(for/async (for-clause ...) body ...+)
(for*/async (for-clause ...) body ...+)
```

Like `for` and `for*`, but each iteration of the `body` is executed in a separate `future`, and the futures may be `touched` in any order.

### 11.4.2 Future Semaphores

```
(make-fsemaphore init) → fsemaphore?
init : exact-nonnegative-integer?
```

Creates and returns a new `future semaphore` with the counter initially set to `init`.

A future semaphore is similar to a plain semaphore, but future-semaphore operations can be performed safely in parallel (to synchronize parallel computations). In contrast, operations on plain semaphores are not safe to perform in parallel, and they therefore prevent a computation from continuing in parallel.

Beware of trying to use an fsemaphore to implement a lock. A future may run concurrently and in parallel to other futures, but a future that is not demanded by a Racket thread can be suspended at any time—such as just after it takes a lock and before it releases the lock. If you must share mutable data among futures, lock-free data structures are generally a better fit.

```
(fsemaphore? v) → boolean?
v : any/c
```

Returns `#t` if `v` is an future semaphore value, `#f` otherwise.

```
(fsemaphore-post fsema) → void?
fsema : fsemaphore?
```

Increments the future semaphore’s internal counter and returns `<void>`.

```
(fsemaphore-wait fsema) → void?
fsema : fsemaphore?
```

Blocks until the internal counter for `fsema` is non-zero. When the counter is non-zero, it is decremented and `fsemaphore-wait` returns `<void>`.

```
(fsemaphore-try-wait? fsema) → boolean?
fsema : fsemaphore?
```

Returns `#t` if the internal counter for `fsema` is non-zero, and `#f` otherwise.
Like \texttt{fsemaphore-wait}, but \texttt{fsemaphore-try-wait} never blocks execution. If \texttt{fsema}'s internal counter is zero, \texttt{fsemaphore-try-wait} returns \texttt{#f} immediately without decrementing the counter. If \texttt{fsema}'s counter is positive, it is decremented and \texttt{#t} is returned.

\begin{verbatim}
(define (fsemaphore-count fsema) \rightarrow exact-nonnegative-integer?
  fsema : fsemaphore?
\end{verbatim}

Returns \texttt{fsema}'s current internal counter value.

### 11.4.3 Future Performance Logging

Racket traces use logging (see §15.5 "Logging") extensively to report information about how futures are evaluated. Logging output is useful for debugging the performance of programs that use futures.

Though textual log output can be viewed directly (or retrieved in code via \texttt{trace-futures}), it is much easier to use the graphical profiler tool provided by \texttt{future-visualizer}.

Future events are logged with the topic \texttt{'future}. In addition to its string message, each event logged for a future has a data value that is an instance of a \texttt{future-event} prefab structure:

\begin{verbatim}
(define future-event
  (struct future-event (future-id proc-id action time prim-name user-data)
    #:prefab)
\end{verbatim}

The \texttt{future-id} field is an exact integer that identifies a future, or it is \texttt{#f} when \texttt{action} is \texttt{'missing}. The \texttt{future-id} field is particularly useful for correlating logged events.

The \texttt{proc-id} fields is an exact, non-negative integer that identifies a parallel process. Process 0 is the main Racket process, where all expressions other than future thunks evaluate.

The \texttt{time} field is an inexact number that represents time in the same way as \texttt{current-inexact-milliseconds}.

The \texttt{action} field is a symbol:

- \texttt{'create}: a future was created.
- \texttt{'complete}: a future’s thunk evaluated successfully, so that \texttt{touch} will produce a value for the future immediately.
- \texttt{'start-work} and \texttt{'end-work}: a particular process started and ended working on a particular future.
• 'start-0-work: like 'start-work, but for a future thunk that for some structural reason could not be started in a process other than 0 (e.g., the thunk requires too much local storage to start).

• 'start-overflow-work: like 'start-work, where the future thunk’s work was previously stopped due to an internal stack overflow.

• 'sync: blocking (processes other than 0) or initiation of handing (process 0) for an “unsafe” operation in a future thunk’s evaluation; the operation must run in process 0.

• 'block: like 'sync, but for a part of evaluation that must be delayed until the future is touched, because the evaluation may depend on the current continuation.

• 'touch (never in process 0): like 'sync or 'block, but for a touch operation within a future thunk.

• 'overflow (never in process 0): like 'sync or 'block, but for the case that a process encountered an internal stack overflow while evaluating a future thunk.

• 'result or 'abort: waiting or handling for 'sync, 'block, or 'touch ended with a value or an error, respectively.

• 'suspend (never in process 0): a process blocked by 'sync, 'block, or 'touch abandoned evaluation of a future; some other process may pick up the future later.

• 'touch-pause and 'touch-resume (in process 0, only): waiting in touch for a future whose thunk is being evaluated in another process.

• 'missing: one or more events for the process were lost due to internal buffer limits before they could be reported, and the time-id field reports an upper limit on the time of the missing events; this kind of event is rare.

Assuming no 'missing events, then 'start-work, 'start-0-work, 'start-overflow-work is always paired with 'end-work; 'sync, 'block, and 'touch are always paired with 'result, 'abort, or 'suspend; and 'touch-pause is always paired with 'touch-resume.

In process 0, some event pairs can be nested within other event pairs: 'sync, 'block, or 'touch with 'result or 'abort; 'touch-pause with 'touch-resume; and 'start-work with 'end-work.

A 'block in process 0 is generated when an unsafe operation is handled. This type of event will contain a symbol in the unsafe-op-name field that is the name of the operation. In all other cases, this field contains #f.

The prim-name field will always be #f unless the event occurred on process 0 and its action is either 'block or 'sync. If these conditions are met, prim-name will contain the name of the Racket primitive which required the future to synchronize with the runtime thread (represented as a symbol).
The user-data field may take on a number of different values depending on both the action and prim-name fields:

- 'touch on process 0: contains the integer ID of the future being touched.
- 'sync and prim-name is 'allocate memory: The size (in bytes) of the requested allocation.
- 'sync and prim-name is 'jit_on_demand: The runtime thread is performing a JIT compilation on behalf of the future future-id. The field contains the name of the function being JIT compiled (as a symbol).
- 'create: A new future was created. The field contains the integer ID of the newly created future.

11.5 Places

(require racket/place)  (require racket/place/dynamic)

The bindings documented in this section are provided by the racket/place, racket/place/dynamic, and racket libraries, but not racket/base.

Places enable the development of parallel programs that take advantage of machines with multiple processors, cores, or hardware threads.

A place is a parallel task that is effectively a separate instance of the Racket virtual machine, although all places run within a single operating-system process. Places communicate through place channels, which are endpoints for a two-way buffered communication.

To a first approximation, place channels support only immutable, transparent values as messages. In addition, place channels themselves can be sent across channels to establish new (possibly more direct) lines of communication in addition to any existing lines. Finally, mutable values produced by shared-flvector, make-shared-flvector, shared-fxvector, make-shared-fxvector, shared-bytes, and make-shared-bytes can be sent across place channels; mutation of such values is visible to all places that share the value, because they are allowed in a shared memory space. See place-message-allowed?.

A place channel can be used as a synchronizable event (see §11.2.1 “Events”) to receive a value through the channel. A place channel is ready for synchronization when a message is available on the channel, and the place channel’s synchronization result is the message (which is removed on synchronization). A place can also receive messages with place-channel-get, and messages can be sent with place-channel-put.

Two place channels are equal? if they are endpoints for the same underlying channels while both or neither is a place descriptor. Place channels can be equal? without being eq? after being sent messages through a place channel.
Constraints on messages across a place channel—and therefore on the kinds of data that places share—enable greater parallelism than future, even including separate garbage collection of separate places. At the same time, the setup and communication costs for places can be higher than for futures.

For example, the following expression launches two places, echoes a message to each, and then waits for the places to terminate:

```racket
(let ([pls (for/list ([i (in-range 2)])
                     (dynamic-place "place-worker.rkt" 'place-main))])
  (for ([i (in-range 2)]
        [p pls])
    (place-channel-put p i)
    (printf "\n" (place-channel-get p)))
(map place-wait pls))
```

The "place-worker.rkt" module (in a file that is separate from the above code) must export the place-main function that each place executes, where place-main must accept a single place channel argument:

```racket
#lang racket
(provide place-main)

(define (place-main pch)
  (place-channel-put pch (format "Hello from place ~a\n" (place-channel-get pch)))
)
```

Place channels are subject to garbage collection, like other Racket values, and a thread that is blocked reading from a place channel can be garbage collected if place channel’s writing end becomes unreachable. However, unlike normal channel blocking, if otherwise unreachable threads are mutually blocked on place channels that are reachable only from the same threads, the threads and place channels are all considered reachable, instead of unreachable.

When a place is created, its parameter values are generally set to the initial values of the parameters in the creating place, except that the current values of the following parameters are used: `current-library-collection-paths`, `current-library-collection-links`, and `current-compiled-file-roots`.

A newly created place is registered with the current custodian, so that the place is terminated when the custodian is shut down.

### 11.5.1 Using Places

```racket
(place-enabled?) → boolean?
```
Returns #t if Racket is configured so that `dynamic-place` and `place` create places that can run in parallel, #f if `dynamic-place` and `place` are simulated using `thread`.

\[(\text{place}\?\ v) \rightarrow \text{boolean}\?\]
\[v : \text{any/c}\]

Returns #t if \(v\) is a place descriptor value, #f otherwise. Every place descriptor is also a place channel.

\[(\text{place-channel}\?\ v) \rightarrow \text{boolean}\?\]
\[v : \text{any/c}\]

Returns #t if \(v\) is place channel, #f otherwise.

\[(\text{dynamic-place} \quad \text{module-path} \quad \text{start-name} \quad [\#:\text{at} \quad \text{location} \quad [\#:\text{named} \quad \text{named}]]) \rightarrow \text{place}\?\]
\[\text{module-path} : (\text{or/c} \quad \text{module-path}? \quad \text{path}?)
\[\text{start-name} : \text{symbol}\?\]
\[\text{location} : (\text{or/c} \quad \text{if} \quad \text{place-location}?) = \#f\]
\[\text{named} : \text{any/c} = \#f\]

Creates a place to run the procedure that is identified by `module-path` and `start-name`. The result is a place descriptor value that represents the new parallel task; the place descriptor is returned immediately. The place descriptor value is also a place channel that permits communication with the place.

The module indicated by `module-path` must export a function with the name `start-name`. The function must accept a single argument, which is a place channel that corresponds to the other end of communication for the place descriptor returned by `place`.

If `location` is provided, it must be a place location, such as a distributed places node produced by `create-place-node`.

When the place is created, the initial exit handler terminates the place, using the argument to the exit handler as the place’s completion value. Use `(exit \ v)` to immediately terminate a place with the completion value \(v\). Since a completion value is limited to an exact integer between 0 and 255, any other value for \(v\) is converted to 0.

If the function indicated by `module-path` and `start-name` returns, then the place terminates with the completion value 0.

In the created place, the `current-input-port` parameter is set to an empty input port, while the values of the `current-output-port` and `current-error-port` parameters are connected to the current ports in the creating place. If the output ports in the creating
place are file-stream ports, then the connected ports in the created place share the under-
lying streams, otherwise a thread in the creating place pumps bytes from the created place’s
ports to the current ports in the creating place.

The module-path argument must not be a module path of the form (quote sym) unless
the module is predefined (see module-predefined?).

The dynamic-place binding is protected in the sense of protect-out, so access to this
operation can be prevented by adjusting the code inspector (see §14.10 “Code Inspectors”).

\[
\text{(dynamic-place* module-path start-name}
\begin{array}{ll}
\text{[#:in in} & \text{place?}
\text{:#:out out} & \text{(or/c output-port? #f)}
\text{:#:err err} & \text{(or/c input-port? #f)}
\end{array}
\text{)}
\]

Like dynamic-place, but accepts specific ports to the new place’s ports, and returns a
created port when #f is supplied for a port. The in, out, and err ports are connected
to the current-input-port, current-output-port, and current-error-port ports,
respectively, for the place. Any of the ports can be #f, in which case a file-stream port (for an
operating-system pipe) is created and returned by dynamic-place*. The err argument can
be 'stdout, in which case the same file-stream port or that is supplied as standard output is
also used for standard error. For each port or 'stdout that is provided, no pipe is created
and the corresponding returned value is #f.

The caller of dynamic-place* is responsible for closing all returned ports; none are closed
automatically.

The dynamic-place* procedure returns four values:

- a place descriptor value representing the created place;
- an output port piped to the place’s standard input, or #f if in was a port;
- an input port piped from the place’s standard output, or #f if out was a port;
- an input port piped from the place’s standard error, or #f if err was a port or 'stdout.

The dynamic-place* binding is protected in the same way as dynamic-place.
(place-wait p) → exact-integer?
  p : place?

Returns the completion value of the place indicated by p, blocking until the place has terminated.

If any pumping threads were created to connect a non-file-stream port to the ports in the place for p (see dynamic-place), place-wait returns only when the pumping threads have completed.

(place-dead-evt p) → evt?
  p : place?

Returns a synchronizable event (see §11.2.1 “Events”) that is ready for synchronization if and only if p has terminated. The synchronization result of a place-dead event is the place-dead event itself.

If any pumping threads were created to connect a non-file-stream port to the ports in the place for p (see dynamic-place), the event returned by place-dead-evt may become ready even if a pumping thread is still running.

(place-kill p) → void?
  p : place?

Immediately terminates the place, setting the place’s completion value to 1 if the place does not have a completion value already.

(place-break p [kind]) → void?
  p : place?
  kind : (or/c #f 'hang-up 'terminate) = #f

Sends the main thread of place p a break; see §10.6 “Breaks”.

(place-channel) → place-channel? place-channel?

Returns two place channels. Data sent through the first channel can be received through the second channel, and data sent through the second channel can be received from the first.

Typically, one place channel is used by the current place to send messages to a destination place; the other place channel is sent to the destination place (via an existing place channel).

(place-channel-put pch v) → void
  pch : place-channel?
  v : place-message-allowed?
Sends a message \( v \) on channel \( pch \). Since place channels are asynchronous, \texttt{place-channel-put} calls are non-blocking.

See \texttt{place-message-allowed?} form information on automatic coercions in \( v \), such as converting a mutable string to an immutable string.

\[
\text{(place-channel-get} \ pch) \rightarrow \text{place-message-allowed?}
\]

\[
pch : \text{place-channel?}
\]

Returns a message received on channel \( pch \), blocking until a message is available.

\[
\text{(place-channel-put/get} \ pch \ v) \rightarrow \text{any/c}
\]

\[
pch : \text{place-channel?}
\]

\[
v : \text{any/c}
\]

Sends an immutable message \( v \) on channel \( pch \) and then waits for a message (perhaps a reply) on the same channel.

\[
\text{(place-message-allowed?} \ v) \rightarrow \text{boolean?}
\]

\[
v : \text{any/c}
\]

Returns \#t if \( v \) is allowed as a message on a place channel, \#f otherwise.

If \( \text{place-enabled?} \) returns \#f, then the result is always \#t and no conversions are performed on \( v \) as a message. Otherwise, the following kinds of data are allowed as messages:

- numbers, characters, booleans, keywords, and \#\langle void\rangle;
- symbols, where the \texttt{eq?}ness of uninterned symbols is preserved within a single message, but not across messages;
- strings and byte strings, where mutable strings and byte strings are automatically replaced by immutable variants;
- paths (for any platform);
- pairs, lists, vectors, and immutable prefab structures containing message-allowed values, where a mutable vector is automatically replaced by an immutable vector and where impersonators of vectors and prefab structures are copied;
- hash tables where mutable hash tables are automatically replaced by immutable variants, and where a hash table impersonator is copied;
- place channels, where a place descriptor is automatically replaced by a plain place channel;
- file-stream ports and TCP ports, where the underlying representation (such as a file descriptor, socket, or handle) is duplicated and attached to a fresh port in the receiving place;
• C pointers as created or accessed via \texttt{ffi/unsafe}; and
• values produced by \texttt{shared-flvector}, \texttt{make-shared-flvector}, \texttt{shared-fxvector}, \texttt{make-shared-fxvector}, \texttt{shared-bytes}, and \texttt{make-shared-bytes}.

\begin{verbatim}
prop:place-location : struct-type-property?
(place-location? v) → boolean?
| v : any/c
\end{verbatim}

A structure type property and associated predicate for implementations of \emph{place locations}. The value of \texttt{prop:place-location} must be a procedure of four arguments: the place location itself, a module path, a symbol for the start function exported by the module, and a place name (which can be \texttt{#f} for an anonymous place).

A place location can be passed as the \texttt{#:at} argument to \texttt{dynamic-place}, which in turn simply calls the \texttt{prop:place-location} value of the place location.

A distributed places note created with \texttt{create-place-node} is an example of a place location.

\subsection*{11.5.2 Syntactic Support for Using Places}

The bindings in this section are \textit{not} provided by \texttt{racket/place/dynamic}.

\begin{verbatim}
(place id body ...+)
\end{verbatim}

Creates a place that evaluates \texttt{body} expressions with \texttt{id} bound to a place channel. The \texttt{bodys} close only over \texttt{id} plus the top-level bindings of the enclosing module, because the \texttt{bodys} are lifted to a submodule. The result of \texttt{place} is a place descriptor, like the result of \texttt{dynamic-place}.

The generated submodule has the name \texttt{place-body-n} for an integer \texttt{n}, and the submodule exports a \texttt{main} function that takes a place channel for the new place. The submodule is not intended for use, however, except by the expansion of the \texttt{place} form.

The \texttt{place} binding is protected in the same way as \texttt{dynamic-place}.

\begin{verbatim}
(place* maybe-port ...
    id
    body ...+)
maybe-port =
| #:in in-expr
| #:out out-expr
| #:err err-expr
\end{verbatim}
Like \texttt{place}, but supports optional \#:in, \#:out, and \#:err expressions (at most one of each) to specify ports in the same way and with the same defaults as \texttt{dynamic-place*}. The result of a \texttt{place*} form is also the same as for \texttt{dynamic-place*}.

The \texttt{place*} binding is protected in the same way as \texttt{dynamic-place}.

\[(\texttt{place/context id body ...})\]

Like \texttt{place}, but \texttt{body ...} may have free lexical variables, which are automatically sent to the newly-created place. Note that these variables must have values accepted by \texttt{place-message-allowed?}, otherwise an \texttt{exn:fail:contract} exception is raised.

### 11.5.3 Places Logging

Place events are reported to a logger named \texttt{'place}. In addition to its string message, each event logged for a place has a data value that is an instance of a \texttt{place-event} prefab structure:

\[
\text{(struct place-event (place-id action value time) \\
  #:prefab)}
\]

The \texttt{place-id} field is an exact integer that identifies a place.

The \texttt{time} field is an inexact number that represents time in the same way as \texttt{current-inexact-milliseconds}.

The \texttt{action} field is a symbol:

- \texttt{'create}: a place was created. This event is logged in the creating place, and the event’s \texttt{value} field has the ID for the created place.
- \texttt{'reap}: a place that was previously created in the current place has exited (and that fact has been detected, possibly via \texttt{place-wait}). The event’s \texttt{value} field has the ID for the exited place.
- \texttt{'enter}: a place has started, logged within the started place. The event’s \texttt{value} field has \texttt{#f}.
- \texttt{'exit}: a place is exiting, logged within the exiting place. The event’s \texttt{value} field has \texttt{#f}.
- \texttt{'put}: a place-channel message has been sent. The event’s \texttt{value} field is a positive exact integer that approximates the message’s size.
- \texttt{'get}: a place-channel message has been received. The event’s \texttt{value} field is a positive exact integer that approximates the message’s size.

Changed in version 6.0.0.2 of package \texttt{base}: Added logging via \texttt{'place} and \texttt{place-event}. 809
11.6 Engines

(require racket/engine)  package: base

The bindings documented in this section are provided by the racket/engine library, not racket/base or racket.

An engine is an abstraction that models processes that can be preempted by a timer or other external trigger. They are inspired by the work of Haynes and Friedman [Haynes84].

Engines log their behavior via a logger with the name 'racket/engine. The logger is created when the module is instantiated and uses the result of (current-logger) as its parent. The library adds logs a 'debug level message: when engine-run is called, when the engine timeout expires, and when the engine is stopped (either because it terminated or it reached a safe point to stop). Each log message holds a value of the struct:

(struct engine-info (msec name) #:prefab)

where the msec field holds the result of (current-inexact-milliseconds) at the moment of logging, and the name field holds the name of the procedure passed to engine.

(engine proc) -> engine?
  proc : ((any/c . -> . void?) . -> . any/c)

Returns an engine object to encapsulate a thread that runs only when allowed. The proc procedure should accept one argument, and proc is run in the engine thread when engine-run is called. If engine-run returns due to a timeout, then the engine thread is suspended until a future call to engine-run. Thus, proc only executes during the dynamic extent of a engine-run call.

The argument to proc is a procedure that takes a boolean, and it can be used to disable suspends (in case proc has critical regions where it should not be suspended). A true value passed to the procedure enables suspends, and #f disables suspends. Initially, suspends are allowed.

(engine? v) -> any
  v : any/c

Returns #t if v is an engine produced by engine, #f otherwise.

(engine-run until engine) -> boolean?
  until : (or/c evt? real?)
  engine : engine?

Allows the thread associated with engine to execute for up as long as until milliseconds (if until is a real number) or until is ready (if until is an event). If engine’s procedure disables suspends, then the engine can run arbitrarily long until it re-enables suspends.
The engine-run procedure returns #t if engine’s procedure completes (or if it completed earlier), and the result is available via engine-result. The engine-run procedure returns #f if engine’s procedure does not complete before it is suspended after the until condition is met. If engine’s procedure raises an exception, then it is re-raised by engine-run.

\[(\text{engine-result } engine) \rightarrow \text{any}\]
\[\text{engine : engine?}\]

Returns the result for engine if it has completed with a value (as opposed to an exception), #f otherwise.

\[(\text{engine-kill } engine) \rightarrow \text{void}\]
\[\text{engine : engine?}\]

Forcibly terminates the thread associated with engine if it is still running, leaving the engine result unchanged.

### 11.7 Machine Memory Order

Unlike Racket threads, futures and places can expose the underlying machine’s memory model, including a weak memory ordering. For example, when a future writes to multiple slots in a mutable vector, it’s possible on some platforms for another future to observe the writes in a different order or not at all, unless the futures are explicitly synchronized. Similarly, shared byte strings or fxvectors can expose the machine’s memory model across places.

Racket ensures that a machine’s memory model is not observed in a way that unsafely exposes the implementation of primitive datatypes. For example, it is not possible for one future to see a partially constructed primitive value as a result of reading a vector that is mutated by another future.

The box-cas!, vector-cas!, unsafe-box*-cas!, unsafe-vector*-cas!, and unsafe-struct*-cas! operations all provide a machine-level compare-and-set, so they can be used in ways that are specifically supported by the a machine’s memory model. The (memory-order-acquire) and (memory-order-release) operations similarly constrain machine-level stores and loads. Synchronization operations such as place messages, future touches, and future semaphores imply suitable machine-level acquire and release ordering.

\[(\text{memory-order-acquire}) \rightarrow \text{void}\]
\[\text{(memory-order-release}) \rightarrow \text{void}\]

Those operations implement a machine-level memory fence on platforms where one is needed for synchronization. The memory-order-acquire operation ensures at least a
load–load and load–store fence at the machine level, and the `memory-order-release` operation ensures at least a store–store and store–load fence at the machine level.

Added in version 7.7.0.11 of package `base`. 
12 Macros

See §1.2 "Syntax Model" for general information on how programs are parsed. In particular, the subsection §1.2.3.2 "Expansion Steps" describes how parsing triggers macros, and §1.2.3.5 "Transformer Bindings" describes how macro transformers are called.

12.1 Pattern-Based Syntax Matching

\[
\text{(syntax-case stx-expr (literal-id ...)}
\]
\[
\text{clause ...)}
\]
\[
\text{clause = [pattern result-expr]}
\]
\[
\text{| [pattern fender-expr result-expr]}
\]
\[
\text{pattern = np-pattern}
\]
\[
\text{| (pattern ...)}
\]
\[
\text{| (pattern ...+ . np-pattern)}
\]
\[
\text{| (pattern ... pattern ellipsis pattern ... . np-pattern)}
\]
\[
\text{np-pattern = _
\]
\[
\text{| id}
\]
\[
\text{| #(pattern ...)
\]
\[
\text{| #(pattern ... pattern ellipsis pattern ...)}
\]
\[
\text{| #&pattern}
\]
\[
\text{| #s(key-datum pattern ...)
\]
\[
\text{| #s(key-datum pattern ... pattern ellipsis pattern ...)
\]
\[
\text{| (ellipsis stat-pattern)}
\]
\[
\text{| const
\]
\[
\text{stat-pattern = id}
\]
\[
\text{| (stat-pattern ...)}
\]
\[
\text{| (stat-pattern ...+ . stat-pattern)}
\]
\[
\text{| #(stat-pattern ...)}
\]
\[
\text{| const
\]
\[
\text{ellipsis = ...}
\]

Finds the first pattern that matches the syntax object produced by stx-expr, and for which the corresponding fender-expr (if any) produces a true value; the result is from the corresponding result-expr, which is in tail position for the syntax-case form. If no clause matches, then the exn:fail:syntax exception is raised; the exception is generated by calling raise-syntax-error with #f as the “name” argument, a string with a generic error message, and the result of stx-expr.
A syntax object matches a pattern as follows:

\texttt{-}

A \_ pattern (i.e., an identifier with the same binding as \_) matches any syntax object.

\texttt{id}

An \texttt{id} matches any syntax object when it is not bound to \_ \_ or \_ and does not have the same binding as any \texttt{literal-id}. The \texttt{id} is further bound as \texttt{pattern variable} for the corresponding \texttt{fender-expr} (if any) and \texttt{result-expr}. A pattern-variable binding is a transformer binding; the pattern variable can be referenced only through forms like \texttt{syntax}. The binding’s value is the syntax object that matched the pattern with a depth marker of 0.

An \texttt{id} that has the same binding as a \texttt{literal-id} matches a syntax object that is an identifier with the same binding in the sense of \texttt{free-identifier=?}. The match does not introduce any pattern variables.

\texttt{(pattern ...)}

A (\texttt{pattern ...}) pattern matches a syntax object whose datum form (i.e., without lexical information) is a list with as many elements as sub-patterns in the pattern, and where each syntax object that corresponds to an element of the list matches the corresponding sub-pattern.

Any pattern variables bound by the sub-patterns are bound by the complete pattern; the bindings must all be distinct.

\texttt{(pattern ...+ . np-pattern)}

Like the previous kind of pattern, but matches syntax objects that are not necessarily lists; for \texttt{n} sub-patterns before the final \texttt{np-pattern}, the syntax object’s datum must be a pair such that \texttt{n-1 cdrs} produce pairs. The final \texttt{np-pattern} is matched against the syntax object corresponding to the \texttt{nth cdr} (or the \texttt{datum->syntax} coercion of the datum using the nearest enclosing syntax object’s lexical context and source location).

\texttt{(pattern ... pattern ellipsis pattern ...)}
Like the \( \text{(pattern ...)} \) kind of pattern, but matching a syntax object with any number (zero or more) elements that match the sub-pattern followed by \text{ellipsis} in the corresponding position relative to other sub-patterns.

For each pattern variable bound by the sub-pattern followed by \text{ellipsis}, the larger pattern binds the same pattern variable to a list of values, one for each element of the syntax object matched to the sub-pattern, with an incremented depth marker. (The sub-pattern itself may contain \text{ellipsis}, leading to a pattern variables bound to lists of lists of syntax objects with a depth marker of 2, and so on.)

\[
\text{(pattern ... pattern ellipsis pattern ... . np-pattern)}
\]

Like the previous kind of pattern, but with a final \text{np-pattern} as for \( \text{(pattern ...+ . np-pattern)} \). The final \text{np-pattern} never matches a syntax object whose datum is a pair.

\[
\#(\text{pattern ...})
\]

Like a \( \text{(pattern ...)} \) pattern, but matching a vector syntax object whose elements match the corresponding sub-patterns.

\[
\#(\text{pattern ... pattern ellipsis pattern ...})
\]

Like a \( \text{(pattern ... pattern ellipsis pattern ...)} \) pattern, but matching a vector syntax object whose elements match the corresponding sub-patterns.

\[
\#*\text{pattern}
\]

Matches a box syntax object whose content matches the \text{pattern}.

\[
\#s(\text{key-datum pattern ...})
\]

Like a \( \text{(pattern ...)} \) pattern, but matching a prefab structure syntax object whose fields match the corresponding sub-patterns. The \text{key-datum} must correspond to a valid first argument to \text{make-prefab-struct}.

\[
\#s(\text{key-datum pattern ... pattern ellipsis pattern ...})
\]

Like a \( \text{(pattern ... pattern ellipsis pattern ...)} \) pattern, but matching a prefab structure syntax object whose elements match the corresponding sub-patterns.
Matches the same as `stat-pattern`, which is like a `pattern`, but identifiers with the binding... are treated the same as other `ids`.

A `const` is any datum that does not match one of the preceding forms; a syntax object matches a `const` pattern when its datum is `equal?` to the quoted `const`.

If `stx-expr` produces a non-syntax object, then its result is converted to a syntax object using `datum->syntax` and the lexical context and source location of the `stx-expr`.

If `stx-expr` produces a syntax object that is tainted or armed, then any syntax object bound by a `pattern` are tainted—unless the binding corresponds to the whole syntax object produced by `stx-expr`, in which case it remains tainted or armed.

Examples:

```
> (require (for-syntax racket/base))
> (define-syntax (swap stx)
  (syntax-case stx ()
    [(_ a b) #'(let ([t a])
                  (set! a b)
                  (set! b t))]))
> (let ([x 5] [y 10])
  (swap x y)
  (list x y))
'(10 5)
> (syntax-case #'(ops 1 2 3 => +) (=>)
  [(_ x ... => op) #'(op x ...)])
#<syntax:eval:619:0 (+ 1 2 3)>
> (syntax-case #'(let ([x 5] [y 9] [z 12])
                   (+ x y z))
  (let)
  [(let ([var expr] ...) body ...)
   (list #'(var ...) #'(expr ...))])
'(#<syntax:eval:620:0 (x y z)> #<syntax:eval:620:0 (5 9 12)>)
```

```
(syntax-case* stx-expr (literal-id ...) id-compare-expr clause ...)
```
Like syntax-case, but \textit{id-compare-expr} must produce a procedure that accepts two arguments. A \textit{literal-id} in a \textit{pattern} matches an identifier for which the procedure returns true when given the identifier to match (as the first argument) and the identifier in the \textit{pattern} (as the second argument).

In other words, syntax-case is like \textit{syntax-case*} with an \textit{id-compare-expr} that produces \textit{free-identifier=}?.

\begin{verbatim}
(with-syntax ([pattern stx-expr] ...) 
  body ...+)
\end{verbatim}

Similar to syntax-case, in that it matches a \textit{pattern} to a syntax object. Unlike syntax-case, all \textit{patterns} are matched, each to the result of a corresponding \textit{stx-expr}, and the pattern variables from all matches (which must be distinct) are bound with a single \textit{body} sequence. The result of the \textit{with-syntax} form is the result of the last \textit{body}, which is in tail position with respect to the \textit{with-syntax} form.

If any \textit{pattern} fails to match the corresponding \textit{stx-expr}, the \textit{exn:fail:syntax} exception is raised.

A \textit{with-syntax} form is roughly equivalent to the following \textit{syntax-case} form:

\begin{verbatim}
(syntax-case (list stx-expr ...) () 
  [(pattern ...) (let () body ...+)])
\end{verbatim}

However, if any individual \textit{stx-expr} produces a non-syntax object, then it is converted to one using \textit{datum->syntax} and the lexical context and source location of the individual \textit{stx-expr}.

Examples:

\begin{verbatim}
> (define-syntax (hello stx)
  (syntax-case stx ()
    [(_ name place)
      (with-syntax ([print-name #'(printf "~a\n "name))
        [print-place #'(printf "~a\n "place))])
      #'(begin
        (define (name times)
          (printf "Hello\n")
          (for ([i (in-range 0 times)])
            print-name))
        (define (place times)
          (printf "From\n")
          (for ([i (in-range 0 times)])
            print-place)))))

> (hello jon utah)
> (jon 2)
\end{verbatim}
Hello
jon
jon
> (utah 2)
From
utah
utah
> (define-syntax (math stx)
  (define (make+1 expression)
    (with-syntax ([e expression])
      #'(+ e 1)))
  (syntax-case stx ()
    [(_ numbers ...)
      (with-syntax ([[(added ...)]
                      (map make+1
                           (syntax->list #'(numbers ...)))]
                      #'(begin
                          (printf "got ~a\n" added)
                          ...)))]))
> (math 3 1 4 1 5 9)
got 4
got 2
got 5
got 2
got 6
got 10

(syntax template)
template = id
   | (head-template ...)
   | (head-template ...+ . template)
   | #(head-template ...)
   | #&template
   | #s(key-datum head-template ...)
   | (~? template template)
   | (ellipsis stat-template)
   | const

head-template = template
   | head-template ellipsis ...
   | (~@ . template)
   | (~@ head-template head-template)
   | (~@ head-template)

stat-template = like template, but without ...,~, and ~@

ellipsis = ...

Constructs a syntax object based on a template, which can include pattern variables bound by syntax-case or with-syntax.

A template produces a single syntax object. A head-template produces a sequence of zero or more syntax objects. A stat-template is like a template, except that ...,~, and ~@ are interpreted as constants instead of template forms.

A template produces a syntax object as follows:

| id

If id is bound as a pattern variable, then id as a template produces the pattern variable’s match result. Unless the id is a sub-template that is replicated by ellipsis in a larger template, the pattern variable’s value must be a syntax object with a depth marker of 0 (as opposed to a list of matches).

More generally, if the pattern variable’s value has a depth marker n, then it can only appear within a template where it is replicated by at least n ellipses. In that case, the template will be replicated enough times to use each match result at least once.

If id is not bound as a pattern variable, then id as a template produces (quote-syntax id).
Produces a syntax object whose datum is a list, and where the elements of the
list correspond to syntax objects produced by the `head-templates`.

```(head-template ... . template)```

Like the previous form, but the result is not necessarily a list; instead, the place
of the empty list in the resulting syntax object’s datum is taken by the syntax
object produced by `template`.

```
#(head-template ...)```

Like the `(head-template ...)` form, but producing a syntax object whose
datum is a vector instead of a list.

```#& template```

Produces a syntax object whose datum is a box holding the syntax object pro-
duced by `template`.

```
#s(key-datum head-template ...)```

Like the `(head-template ...)` form, but producing a syntax object whose
datum is a prefab structure instead of a list. The `key-datum` must correspond
to a valid first argument of `make-prefab-struct`.

```(~? template1 template2)```

Produces the result of `template1` if `template1` has no pattern variables with
“missing values”; otherwise, produces the result of `template2`.

A pattern variable bound by `syntax-case` never has a missing value, but pat-
tern variables bound by `syntax-parse` (for example, `~or` or `~optional pat-
terns`) can.

Examples:

```> (syntax-parse #'(m 1 2 3)
    [(_ (~optional (~seq #:op op:expr)) arg:expr ...)
      #'((~? op +) arg ...)])
  #<syntax:eval:3:0 (+ 1 2 3)>```
'(syntax-parse #'(m #:op max 1 2 3)  
  [[(_ (~optional (~seq #:op op:expr)) arg:expr ...)  
    #'((~? op +) arg ...)])  
  #<syntax:eval:4:0 (max 1 2 3)>}

(ellipsis stat-template)

Produces the same result as stat-template, which is like a template, but ...

\(\ldots, \sim?, \sim\emptyset\) are treated like an id (with no pattern binding).

const

A const template is any form that does not match the preceding cases, and it
produces the result (quote-syntax const).

A head-template produces a sequence of syntax objects; that sequence is “inlined” into
the result of the enclosing template. The result of a head-template is defined as follows:

template

Produces one syntax object, according to the rules for template above.

head-template ellipsis ...+

Generates a sequence of syntax objects by “mapping” the head-template over
the values of its pattern variables. The number of iterations depends on the
values of the pattern variables referenced within the sub-template.

To be more precise: Let outer be inner followed by one ellipse. A pattern
variable is an iteration pattern variable for outer if occurs at a depth equal
to its depth marker. There must be at least one; otherwise, an error is raised.
If there are multiple iteration variables, then all of their values must be lists
of the same length. The result for outer is produced by mapping the inner
template over the iteration pattern variable values and decreasing their effective
depth markers by 1 within inner. The outer result is formed by appending
the inner results.

Consequently, if a pattern variable occurs at a depth greater than its depth
marker, it is used as an iteration pattern variable for the innermost ellipses but
not the outermost. A pattern variable must not occur at a depth less than its
depth marker; otherwise, an error is raised.
\[ (~\varnothing . \ template) \]

Produces the sequence of elements in the syntax list produced by \texttt{template}. If \texttt{template} does not produce a proper syntax list, an exception is raised.

Examples:

\[
> (\text{with-syntax} \ [(\text{key ...}) #'(\text{a\ 'b\ 'c})
\quad [(\text{val ...}) #'(1 2 3)])
\quad #'(\text{hash} (~\varnothing \ \text{key\ val}) ...))
#<\text{syntax:eval:2:0 (\text{hash (quote a) 1 (quote b) 2 (quote c) 3)})>
> (\text{with-syntax} \ [(\text{xs #'(2 3 4)})
\quad #'(\text{list} 1 (~\varnothing . \ xs) 5))
#<\text{syntax:eval:3:0 (list 1 2 3 4 5)}>\]

\[ (~? \ \text{head-template1 head-template2}) \]

Produces the result of \texttt{head-template1} if none of its pattern variables have “missing values”; otherwise produces the result of \texttt{head-template2}.

\[ (~? \ \text{head-template}) \]

Produces the result of \texttt{head-template} if none of its pattern variables have “missing values”; otherwise produces nothing.

Equivalent to \((~? \ \text{head-template} (\text{~@}))\).

A \texttt{(syntax template)} form is normally abbreviated as \#\texttt{template}; see also \texttt{§1.3.8 “Reading Quotes”} If \texttt{template} contains no pattern variables, then \#\texttt{template} is equivalent to \texttt{(quote-syntax template)}.

A \texttt{(quasisyntax template)} form is normally abbreviated as \#\texttt{template}; see also \texttt{§1.3.8 “Reading Quotes”} If \texttt{template} contains no pattern variables, then \#\texttt{template} is equivalent to \texttt{(quote-syntax template)}.

Changed in version 6.90.0.25 of package \texttt{base}: Added \texttt{~@} and \texttt{~?}.

\[ (\text{quasisyntax template}) \]

Like \texttt{syntax}, but \texttt{(unsyntax expr)} and \texttt{(unsyntax-splicing expr)} escape to an expression within the \texttt{template}.

The \texttt{expr} must produce a syntax object (or syntax list) to be substituted in place of the unsyntax or unsyntax-splicing form within the quasiquoting template, just like unquote and unquote-splicing within quasiquote, except that a hash table value position is not an escape position for quasisyntax. (If the escaped expression does not generate
a syntax object, it is converted to one in the same way as for the right-hand side of with-
syntax.) Nested quasisyntaxes introduce quasiquoting layers in the same way as nested
quasiquotes.

Also analogous to quasiquote, the reader converts #\ to quasisyntax, #\ to unsyntax,
and #\@ to unsyntax-splicing. See also §1.3.8 “Reading Quotes”.

```lisp
(unsyntax expr)
```

Illegal as an expression form. The unsyntax form is for use only with a quasisyntax
template.

```lisp
(unsyntax-splicing expr)
```

Illegal as an expression form. The unsyntax-splicing form is for use only with a qua-
sisyntax template.

```lisp
(syntax/loc stx-expr template)
```

```lisp
  stx-expr : syntax?
```

Like syntax, except that the immediate resulting syntax object takes its source-location
information from the result of `stx-expr` (which must produce a syntax object).

Only the source location of the immediate result—the “outermost” syntax object—is ad-
justed. The source location is not adjusted if both the source and position of `stx-expr` are
#f. The source location is adjusted only if the resulting syntax object comes from the tem-
plate itself rather than the value of a syntax pattern variable. For example, if `x` is a syntax
pattern variable, then `(syntax/loc stx-expr x)` does not use the location of `stx-expr`.

Changed in version 6.90.0.25 of package base: Previously, syntax/loc did not enforce the contract on `stx-expr`
if `template` was just a pattern variable.

```lisp
(quasisyntax/loc stx-expr template)
```

```lisp
  stx-expr : syntax?
```

Like quasisyntax, but with source-location assignment like syntax/loc.

```lisp
(quote-syntax/prune id)
```

Like quote-syntax, but the lexical context of `id` is pruned via `identifier-prune-
lexical-context` to including binding only for the symbolic name of `id` and for `'#%top`.
Use this form to quote an identifier when its lexical information will not be transferred to
other syntax objects (except maybe to `'#%top for a top-level binding).
(syntax-rules (literal-id ...) [(id . pattern) template] ...)

Equivalent to

(lambda (stx)
  (syntax-case stx (literal-id ...)
    [(generated-id . pattern) (syntax-protect #'template)] ...))

where each generated-id binds no identifier in the corresponding template.

(syntax-id-rules (literal-id ...) [pattern template] ...)

Equivalent to

(make-set!-transformer
  (lambda (stx)
    (syntax-case stx (literal-id ...)
      [pattern (syntax-protect #'template)] ...)))

(define-syntax-rule (id . pattern) template)

Equivalent to

(define-syntax id
  (syntax-rules ()
    [(id . pattern) template]))

but with syntax errors potentially phrased in terms of pattern.

... The ... transformer binding prohibits ... from being used as an expression. This binding is useful only in syntax patterns and templates (or other unrelated expression forms that treat it specially like \texttt{->}), where it indicates repetitions of a pattern or template. See syntax-case and syntax.

_ The _ transformer binding prohibits _ from being used as an expression. This binding is useful only in syntax patterns, where it indicates a pattern that matches any syntax object. See syntax-case.
The ~? and ~@[ transformer bindings prohibit these forms from being used as an expression. The bindings are useful only in syntax templates. See syntax.

Added in version 6.90.0.25 of package base.

```scheme
(syntax-pattern-variable? v) -> boolean?
  v : any/c
```

Returns #t if v is a value that, as a transformer-binding value, makes the bound variable as pattern variable in syntax and other forms. To check whether an identifier is a pattern variable, use syntax-local-value to get the identifier’s transformer value, and then test the value with syntax-pattern-variable?.

The syntax-pattern-variable? procedure is provided for-syntax by racket/base.

### 12.2 Syntax Object Content

```scheme
(syntax? v) -> boolean?
  v : any/c
```

Returns #t if v is a syntax object, #f otherwise. See also §1.2.2 "Syntax Objects".

Examples:

```scheme
> (syntax? #'quinoa)
#t
> (syntax? #'(spelt triticale buckwheat))
#t
> (syntax? (datum->syntax #f 'millet))
#t
> (syntax? "barley")
#f
```

```scheme
(identifier? v) -> boolean?
  v : any/c
```

Returns #t if v is a syntax object and (syntax-e stx) produces a symbol.

Examples:
> (identifier? #'linguine)
#t
> (identifier? #'(if wheat? udon soba))
#f
> (identifier? 'ramen)
#f
> (identifier? 15)
#f

(syntax-source stx) → any/c  
stx : syntax?

Returns the source component of the source location for the syntax object stx, or #f if none is known. The source is represented by an arbitrary value (e.g., one passed to read-syntax), but it is typically a file path string.

(syntax-line stx) → (or/c exact-positive-integer? #f)  
stx : syntax?

Returns the line number (positive exact integer) of the source location for the start of the syntax object in its source, or #f if the line number or source is unknown. The result is #f if and only if (syntax-column stx) produces #f. See also §13.1.4 “Counting Positions, Lines, and Columns”

(syntax-column stx) → (or/c exact-nonnegative-integer? #f)  
stx : syntax?

Returns the column number (non-negative exact integer) of the source location for the start of the syntax object in its source, or #f if the source column is unknown. The result is #f if and only if (syntax-line stx) produces #f. See also §13.1.4 “Counting Positions, Lines, and Columns”

(syntax-position stx) → (or/c exact-positive-integer? #f)  
stx : syntax?

Returns the character position (positive exact integer) of the source location for the start of the syntax object in its source, or #f if the source position is unknown. See also §13.1.4 “Counting Positions, Lines, and Columns”

(syntax-span stx) → (or/c exact-nonnegative-integer? #f)  
stx : syntax?

Returns the span (non-negative exact integer) in characters of the source location for syntax object in its source, or #f if the span is unknown.
(syntax-original? stx) → boolean?
stx : syntax?

Returns #t if stx has the property that read-syntax attaches to the syntax objects that they generate (see §12.7 "Syntax Object Properties"), and if stx’s lexical information does not include any macro-introduction scopes (which indicate that the object was introduced by a syntax transformer; see §1.2.2 "Syntax Objects"). The result is #f otherwise.

This predicate can be used to distinguish syntax objects in an expanded expression that were directly present in the original expression, as opposed to syntax objects inserted by macros.

The (hidden) property to represent original syntax is dropped for a syntax object that is marshaled as part of compiled code; see also current-compile.

(syntax-source-module stx [source?])
→ (or/c module-path-index? symbol? path? resolved-module-path? #f)
stx : syntax?
source? : any/c = #f

Returns an indication of the module whose source contains stx, or #f if no source module for stx can be inferred from its lexical context. If source? is #f, then result is a module path index or symbol (see §14.4.2 "Compiled Modules and References") or a resolved module path; if source? is true, the result is a path or symbol corresponding to the loaded module’s source in the sense of current-module-declare-source.

Note that syntax-source-module does not consult the source location of stx. The result is based on the lexical information of stx.

(syntax-e stx) → any/c
stx : syntax?

Unwraps the immediate datum structure from a syntax object, leaving nested syntax structure (if any) in place. The result of (syntax-e stx) is one of the following:

- a symbol
- a syntax pair (described below)
- the empty list
- an immutable vector containing syntax objects
- an immutable box containing syntax objects
- an immutable hash table containing syntax object values (but not necessarily syntax object keys)
• an immutable prefab structure containing syntax objects
• some other kind of datum—usually a number, boolean, or string—that is interned when `datum-intern-literal` would convert the value

Examples:

```lisp
> (syntax-e #
'a)
'a
> (syntax-e #'(x . y))
'(#<syntax:eval:11:0 x> . #<syntax:eval:11:0 y>)
> (syntax-e #'(1 2 (+ 3 4)))
'#(#<syntax:eval:12:0 1> #<syntax:eval:12:0 2> #<syntax:eval:12:0 (+ 3 4)>)
> (syntax-e #"hello world")
"hello world"
> (syntax-e #"hash((imperial . "yellow") (festival . "green")))
'(imperial . #<syntax:eval:14:0 "yellow">)
> (syntax-e #'(point 3 4))
'(#<syntax:eval:15:0 point> #<syntax:eval:15:0 3> #<syntax:eval:15:0 4>)
> (syntax-e #3)
3
> (syntax-e "three")
"three"
> (syntax-e #\t)
#t
```

A syntax pair is a pair containing a syntax object as its first element, and either the empty list, a syntax pair, or a syntax object as its second element.

A syntax object that is the result of `read-syntax` reflects the use of delimited . in the input by creating a syntax object for every pair of parentheses in the source, and by creating a pair-valued syntax object only for parentheses in the source. See §1.3.6 “Reading Pairs and Lists” for more information.

If `stx` is tainted or armed, then any syntax object in the result of `(syntax-e stx)` is tainted, and multiple calls to `syntax-e` may return values that are not `eq?`. For a `stx` that is not armed, the results from multiple calls to `syntax-e` of `stx` are `eq?`.

```lisp
(syntax->list stx) → (or/c list? #f)
```

```
  stx : syntax?
```

Returns a list of syntax objects or `#f`. The result is a list of syntax objects when `(syntax->datum stx)` would produce a list. In other words, syntax pairs in `(syntax-e stx)` are flattened.
If \texttt{stx} is tainted or armed, then any syntax object in the result of \texttt{(syntax->list stx)} is tainted.

Examples:

\begin{verbatim}
> (syntax->list #'())
'()
> (syntax->list #'(1 (+ 3 4) 5 6))
'(#+syntax:eval:20:0 1>
  #<syntax:eval:20:0 (+ 3 4)>
  #<syntax:eval:20:0 5>
  #<syntax:eval:20:0 6>)
> (syntax->list #'a)
#f
\end{verbatim}

\texttt{(syntax->datum stx)} \rightarrow \texttt{any/c}

\texttt{stx : syntax?}

Returns a datum by stripping the lexical information, source-location information, properties, and tamper status from \texttt{stx}. Inside of pairs, (immutable) vectors, (immutable) boxes, immutable hash table values (not keys), and immutable prefab structures, syntax objects are recursively stripped.

The stripping operation does not mutate \texttt{stx}; it creates new pairs, vectors, boxes, hash tables, and prefab structures as needed to strip lexical and source-location information recursively.

Examples:

\begin{verbatim}
> (syntax->datum #'a)
'a
> (syntax->datum #'(x . y))
'(x . y)
> (syntax->datum #'(1 2 (+ 3 4)))
'(#1 2 (+ 3 4))
> (syntax->datum '#"hello world")
'"hello world"
> (syntax->datum '#hash((imperial . "yellow") (festival . "green")))
'hash((festival . "green") (imperial . "yellow"))
> (syntax->datum '#(point 3 4))
'(#(point 3 4))
> (syntax->datum #'3)
3
> (syntax->datum #'"three")
"three"
\end{verbatim}
> (syntax->datum #'t)
#t

(datum->syntax ctxt v [srcloc prop ignored]) → syntax?
ctxt : (or/c syntax? #f)
v : any/c
srcloc : (or/c syntax? #f)
        (list/c any/c
         (or/c exact-positive-integer? #f)
         (or/c exact-nonnegative-integer? #f)
         (or/c exact-positive-integer? #f)
         (or/c exact-nonnegative-integer? #f))
        (vector/c any/c
         (or/c exact-positive-integer? #f)
         (or/c exact-nonnegative-integer? #f)
         (or/c exact-positive-integer? #f)
         (or/c exact-nonnegative-integer? #f))
        = #f
prop : (or/c syntax? #f) = #f
ignored : (or/c syntax? #f) = #f

Converts the datum v to a syntax object. If v is already a syntax object, then there is no conversion, and v is returned unmodified. The contents of pairs, vectors, and boxes, the fields of prefab structures, and the values of immutable hash tables are recursively converted. The keys of prefab structures and the keys of immutable hash tables are not converted. Mutable vectors and boxes are replaced by immutable vectors and boxes. For any kind of value other than a pair, vector, box, immutable hash table, immutable prefab structure, or syntax object, conversion means wrapping the value with lexical information, source-location information, and properties after the value is interned via datum-intern-literal.

Converted objects in v are given the lexical context information of ctxt and the source-location information of srcloc. The resulting immediate syntax object from conversion is given the properties (see §12.7 “Syntax Object Properties”) of prop (even the hidden ones that would not be visible via syntax-property-symbol-keys); if v is a pair, vector, box, immutable hash table, or immutable prefab structure, recursively converted values are not given properties. If ctxt is tainted or armed, then the resulting syntax object from datum->syntax is tainted. The code inspector of ctxt, if any, is compared to the code inspector of the module for the macro currently being transformed, if any; if both inspectors are available and if one is the same as or inferior to the other, then the result syntax has the same/inferior inspector, otherwise it has no code inspector.

Any of ctxt, srcloc, or prop can be #f, in which case the resulting syntax has no lexical context, source information, and/or new properties.

If srcloc is not #f or a syntax object, it must be a list or vector of five elements:
where source-name is an arbitrary value for the source name; line is an integer for the source line, or #f; column is an integer for the source column, or #f; position is an integer for the source position, or #f; and span is an integer for the source span, or #f. The line and column values must both be numbers or both be #f, otherwise the exn:fail:contract exception is raised.

Graph structure is not preserved by the conversion of v to a syntax object. Instead, v is essentially unfolded into a tree. If v has a cycle through pairs, vectors, boxes, immutable hash tables, and immutable prefab structures, then the exn:fail:contract exception is raised.

The ignored argument is allowed for backward compatibility and has no effect on the returned syntax object.

```scheme
(syntax-binding-set? v) → boolean?
  v : any/c

(syntax-binding-set) → syntax-binding-set?

(syntax-binding-set->syntax binding-set datum) → syntax?
  binding-set : syntax-binding-set?
  datum : any/c
```
(syntax-binding-set-extend
 binding-set
 symbol
 phase
 mpi
 [#:source-symbol source-symbol
 #:source-phase source-phase
 #:nominal-module nominal-mpi
 #:nominal-phase nominal-phase
 #:nominal-symbol nominal-symbol
 #:nominal-require-phase nominal-require-phase
 #:inspector inspector])
→ syntax-binding-set?
binding-set : syntax-binding-set?
symbol : symbol?
phase : (or/c exact-integer? #f)
mpi : module-path-index?
source-symbol : symbol? = symbol
source-phase : (or/c exact-integer? #f) = phase
nominal-mpi : module-path-index? = mpi
nominal-phase : (or/c exact-integer? #f) = source-phase
nominal-symbol : symbol? = source-symbol
nominal-require-phase : (or/c exact-integer? #f) = 0
inspector : (or/c inspector? #f) = #f

A syntax binding set supports explicit construction of binding information for a syntax object. Start by creating an empty binding set with syntax-binding-set, add bindings with syntax-binding-set-extend, and create a syntax object that has the bindings as its lexical information using syntax-binding-set->syntax.

The first three arguments to syntax-binding-set-extend establish a binding of symbol at phase to an identifier that is defined in the module referenced by mpi. Supply source-symbol to make the binding of symbol refer to a different provided variable from mpi, and so on; the optional arguments correspond to the results of identifier-binding.

Added in version 7.0.0.12 of package base.

(datum-intern-literal v) → any/c
v : any/c

Converts some values to be consistent with an interned result produced by the default reader in read-syntax mode.

If v is a number, character, string, byte string, or regular expression, then the result is a value that is equal? to v and eq? to a potential result of the default reader. (Note that mutable strings and byte strings are interned as immutable strings and byte strings.)

832
If \( v \) is an uninterned or an unreadable symbol, the result is still \( v \), since an interned symbol would not be `equal?` to \( v \).

The conversion process does not traverse compound values. For example, if \( v \) is a pair containing strings, then the strings within \( v \) are not interned.

If \( v1 \) and \( v2 \) are `equal?` but not `eq?`, then it is possible that `(datum-intern-literal v1)` will return \( v1 \) and—sometime after \( v1 \) becomes unreachable as determined by the garbage collector (see §1.1.6 “Garbage Collection”)—`(datum-intern-literal v2)` can still return \( v2 \). In other words, `datum-intern-literal` may adopt a given value as an interned representative, but if a former representative becomes otherwise unreachable, then `datum-intern-literal` may adopt a new representative.

```
(syntax-shift-phase-level stx shift) → syntax?
  stx : syntax?
  shift : (or/c exact-integer? #f)
```

Returns a syntax object that is like \( stx \), but with all of its top-level and module bindings shifted by \( shift \) phase levels. If \( shift \) is #f, then only bindings at phase level 0 are shifted to the label phase level. If \( shift \) is 0, then the result is \( stx \).

```
(generate-temporaries stx-pair) → (listof identifier?)
  stx-pair : (or syntax? list?)
```

Returns a list of identifiers that are distinct from all other identifiers. The list contains as many identifiers as \( stx-pair \) contains elements. The \( stx-pair \) argument must be a syntax pair that can be flattened into a list. The elements of \( stx-pair \) can be anything, but string, symbol, keyword (possibly wrapped as syntax), and identifier elements will be embedded in the corresponding generated name, which is useful for debugging purposes.

The generated identifiers are built with interned symbols (not `gensyms`); see also §1.4.16 “Printing Compiled Code”

Examples:

```
> (generate-temporaries '(a b c d))
'('#<syntax a1> #<syntax b2> #<syntax c3> #<syntax d4>)
> (generate-temporaries #'(1 2 3 4))
'(#<syntax temp5> #<syntax temp6> #<syntax temp7> #<syntax temp8>)
> (define-syntax (set!-values stx)
  (syntax-case stx ()
    [(_ (id ...) expr)
     (with-syntax [[(temp ...) (generate-temporaries #'(id ...)])]
      #'(let-values [[(temp ...) expr]]
        (set! id temp) ... (void)))]))
```
(identifier-prune-lexical-context id-stx [syms]) → identifier?

id-stx : identifier?
syms : (listof symbol?) = (list (syntax-e id-stx))

Returns an identifier with the same binding as id-stx, but without possibly lexical information from id-stx that does not apply to the symbols in syms, where even further extension of the lexical information drops information for other symbols. In particular, transferring the lexical context via datum->syntax from the result of this function to a symbol other than one in syms may produce an identifier with no binding.

Currently, the result is always id-stx exactly. Pruning was intended primarily as a kind of optimization in a previous version of Racket, but it is less useful and difficult to implement efficiently in the current macro expander.

See also quote-syntax/prune.

Changed in version 6.5 of package base: Always return id-stx.

(identifier-prune-to-source-module id-stx) → identifier?

id-stx : identifier?

Returns an identifier with its lexical context minimized to that needed for syntax-source-module. The minimized lexical context does not include any bindings.

(syntax-recertify new-stx
  old-stx
  inspector
  key) → syntax?

new-stx : syntax?
old-stx : syntax?
inspector : inspector?
key : any/c

For backward compatibility only; returns new-stx.

(syntax-debug-info stx [phase all-bindings?] ) → hash?

stx : syntax?
phase : (or/c exact-integer? #f) = (syntax-local-phase-level)
all-bindings? : any/c = #f

Produces a hash table that describes the lexical information of stx (not counting components when (syntax-e stx) would return a compound value). The result can include—but is not limited to—the following keys:
• 'name — the result of (syntax-e stx), if it is a symbol.

• 'context — a list of vectors, where each vector represents a scope attached to stx. Each vector starts with a number that is distinct for every scope. A symbol afterward provides a hint at the scope’s origin: 'module for a module scope, 'macro for a macro-introduction scope, 'use-site for a macro use-site scope, or 'local for a local binding form. In the case of a 'module scope that corresponds to the inside edge, the module’s name and a phase (since an inside-edge scope is generated for each phase) are shown.

• 'bindings — a list of bindings, each represented by a hash table. A binding table can include—but is not limited to—the following keys:
  – 'name — the symbolic name for the binding.
  – 'context — the scopes, as a list of vectors, for the binding.
  – 'local — a symbol representing a local binding; when this key is present, 'module is absent.
  – 'module — an encoding of a import from another module; when this key is present, 'local is absent.
  – 'free-identifier=? — a hash table of debugging information from an identifier for which the binding is an alias.

• 'fallbacks — a list of hash tables like the one produced by syntax-debug-info for cross-namespace binding fallbacks.

Added in version 6.3 of package base.

### 12.3 Syntax Object Bindings

```
(bound-identifier=? a-id b-id [phase-level]) -> boolean?
a-id : syntax?
b-id : syntax?
phase-level : (or/c exact-integer? #f)
  = (syntax-local-phase-level)
```

Returns #t if the identifier a-id would bind b-id (or vice versa) if the identifiers were substituted in a suitable expression context at the phase level indicated by phase-level, #f otherwise. A #f value for phase-level corresponds to the label phase level.

Examples:

```
> (define-syntax (check stx)
  (syntax-case stx ()
```

835
([, x y)
  (if (bound-identifier=? #\x #\y)
      #\'(let ([y 'wrong]) (let ([x 'binds] y))
      #\'(let ([y 'no-binds] (let ([x 'wrong]) y)))))
>
(check a a)
'binds
>
(check a b)
'no-binds
>
(define-syntax-rule (check-a x) (check a x))
>
(check-a a)
'no-binds

(free-identifier=? a-id b-id
  [a-phase-level]
  b-phase-level)) boolean?

   a-id : identifier?
   b-id : identifier?
   a-phase-level : (or/c exact-integer? #f)
     = (syntax-local-phase-level)
   b-phase-level : (or/c exact-integer? #f) = a-phase-level

Returns #t if a-id and b-id access the same local binding, module binding, or top-level
binding—perhaps via rename transformers—at the phase levels indicated by a-phase-
level and b-phase-level, respectively. A #f value for a-phase-level or b-phase-
level corresponds to the label phase level.

“Same module binding” means that the identifiers refer to the same original definition site,
and not necessarily to the same require or provide site. Due to renaming in require and
provide, or due to a transformer binding to a rename transformer, the identifiers may return
distinct results with syntax-e.

Examples:
>
(define-syntax (check stx)
  (syntax-case stx ()
    [(, x)
      (if (free-identifier=? #\car #\x)
        #\'(list 'same: x)
        #\'(list 'different: x))])
>
(check car)
'(same: #<procedure:car>)
>
(check mcar)
'(different: #<procedure:mcar>)
>
(let ([car list])
  (check car))

836
'(different: #<procedure:list>)
> (require (rename-in racket/base [car kar]))
> (check kar)
'(same: #<procedure:car>)

(free-transformer-identifier=? a-id b-id) → boolean?
   a-id : identifier?
   b-id : identifier?

Same as (free-identifier=? a-id b-id (add1 (syntax-local-phase-level))).

(free-template-identifier=? a-id b-id) → boolean?
   a-id : identifier?
   b-id : identifier?

Same as (free-identifier=? a-id b-id (sub1 (syntax-local-phase-level))).

(free-label-identifier=? a-id b-id) → boolean?
   a-id : identifier?
   b-id : identifier?

Same as (free-identifier=? a-id b-id #f).

(check-duplicate-identifier ids) → (or/c identifier? #f)
   ids : (listof identifier?)

Compares each identifier in ids with every other identifier in the list with bound-identifier=?. If any comparison returns #t, one of the duplicate identifiers is returned (the first one in ids that is a duplicate), otherwise the result is #f.

(identifier-binding id-stx
   [phase-level
    top-level-symbol?])

→ (or/c 'lexical
   #f
   (list/c module-path-index?
            symbol?
            module-path-index?
            symbol?
            exact-nonnegative-integer?
            (or/c exact-integer? #f)
            (or/c exact-integer? #f))
   (list/c symbol?))
id-stx : identifier?
phase-level : (or/c exact-integer? #f)
    = (syntax-local-phase-level)
top-level-symbol? : any/c = #f

Returns one of three (if top-level-symbol? is #f) or four (if top-level-symbol? is true) kinds of values, depending on the binding of id-stx at the phase level indicated by phase-level (where a #f value for phase-level corresponds to the label phase level):

- The result is 'lexical if id-stx has a local binding.
- The result is a list of seven items when id-stx has a module binding: (list from-mod from-sym nominal-from-mod nominal-from-sym from-phase import-phase nominal-export-phase).
  - from-mod is a module path index (see §14.4.2 “Compiled Modules and References”) that indicates the defining module. It is the “self” module path index if the binding refers to a definition in the enclosing module of id-stx.
  - from-sym is a symbol for the identifier’s name at its definition site in the originating module. This can be different from the local name returned by syntax->datum for several reasons: the identifier is renamed on import, it is renamed on export, or it is implicitly renamed because the binding site was generated by a macro invocation. In that last case, it may be an unreadable symbol, and it may be different from the result of syntax->datum on the identifier in the original source definition.
  - nominal-from-mod is a module path index (see §14.4.2 “Compiled Modules and References”) that indicates the binding’s module as it appears locally in the source around id-stx: it indicates a module required into the context of id-stx to provide its binding, or it is the same “self” as from-mod for a binding that refers to a definition in the enclosing module of id-stx. It can be different from from-mod due to a re-export in nominal-from-mod of some imported identifier. If the same binding is imported in multiple ways, an arbitrary representative is chosen.
  - nominal-from-sym is a symbol for the binding’s identifier as it appears locally in the source around id-stx: it is the identifier’s name as exported by nominal-from-mod, or it is the source identifier’s symbol for a definition within the enclosing module of id-stx. It can be different from from-sym due to a renaming provide, even if from-mod and nominal-from-mod are the same, or due to a definition that was introduced by a macro expansion.
  - from-phase is an exact non-negative integer representing the originating phase. For example, it is 1 if the definition is for-syntax.
  - import-phase is 0 if the binding import of nominal-from-mode is from a definition or a plain require, 1 if it is from a for-syntax import, etc.
– *nominal-export-phase* is the phase level of the export from *nominal-from-mod* for an imported binding or the phase level of the definition for a binding from the enclosing module of *id-stx*.

- The result is `(list top-sym)` if *id-stx* has a top-level binding and `top-level-symbol?` is true. The *top-sym* can different from the name returned by `syntax->datum` when the binding definition was generated by a macro invocation.

- The result is `#f` if *id-stx* has a top-level binding and `top-level-symbol?` is `#f` or if *id-stx* is unbound. An unbound identifier is typically treated the same as an identifier whose top-level binding is a variable.

If *id-stx* is bound to a rename-transformer, the result from `identifier-binding` is for the identifier in the transformer, so that `identifier-binding` is consistent with `free-identifier=?`.

Changed in version 6.6.0.4 of package `base`: Added the `top-level-symbol?` argument to report information on top-level bindings.

```
(identifier-transformer-binding id-stx [rt-phase-level])
```

→ `(or/c 'lexical #f
   (or/c exact-integer? #f)
   (or/c exact-integer? #f)))

**id-stx**: identifier?

**rt-phase-level**: `(or/c exact-integer? #f)`

= `(syntax-local-phase-level)`

Same as `(identifier-binding id-stx (and rt-phase-level (add1 rt-phase-level)))`.

```
(identifier-template-binding id-stx)
```

→ `(or/c 'lexical #f
   (or/c exact-integer? #f)
   (or/c exact-integer? #f)))

839
id-stx : identifier?

Same as (identifier-binding id-stx (sub1 (syntax-local-phase-level))).

(identifier-label-binding id-stx)
→ (or/c 'lexical
   #f
   (listof module-path-index?
       symbol?
       module-path-index?
       symbol?
       exact-nonnegative-integer?
       (or/c exact-integer? #f)
       (or/c exact-integer? #f)))

id-stx : identifier?

Same as (identifier-binding id-stx #f).

(identifier-binding-symbol id-stx
 [phase-level]) → symbol?

id-stx : identifier?
phase-level : (or/c exact-integer? #f)
= (syntax-local-phase-level)

Like identifier-binding, but produces a symbol that corresponds to the binding. The symbol result is the same for any identifiers that are free-identifier=?, but the result may also be the same for identifiers that are not free-identifier=? (i.e., different symbols imply different bindings, but the same symbol does not imply the same binding).

When identifier-binding would produce a list, then the second element of that list is the result that identifier-binding-symbol produces.

12.4 Syntax Transformers

(set!-transformer? v) → boolean?

v : any/c

Returns #t if v is a value created by make-set!-transformer or an instance of a structure type with the prop: set!-transformer property, #f otherwise.

(make-set!-transformer proc) → set!-transformer?

proc : (syntax? . -> . syntax?)
Creates an assignment transformer that cooperates with set!. If the result of \texttt{make-set!-transformer} is bound to \texttt{id} as a transformer binding, then \texttt{proc} is applied as a transformer when \texttt{id} is used in an expression position, or when it is used as the target of a \texttt{set!} assignment as \texttt{(set! id expr)}. When the identifier appears as a \texttt{set!} target, the entire \texttt{set!} expression is provided to the transformer.

Example:

```scheme
> (let ([x 1]
       [y 2])
  (let-syntax ([x (make-set!-transformer
                   (lambda (stx)
                     (syntax-case stx (set!)
                       ; Redirect mutation of x to y
                       ; [(set! id v) #'(set! y v)]
                       ; Normal use of x really gets x
                       [id (identifier? '#id) #'x])])
   (begin
    (set! x 3)
    (list x y))))
'(1 3)
```

\texttt{(set!-transformer-procedure transformer)}

\rightarrow (syntax? . -> . syntax?)

\texttt{transformer : set!-transformer?}

Returns the procedure that was passed to \texttt{make-set!-transformer} to create \texttt{transformer} or that is identified by the \texttt{prop:set!-transformer} property of \texttt{transformer}.

\texttt{prop:set!-transformer : struct-type-property?}

A structure type property to identify structure types that act as assignment transformers like the ones created by \texttt{make-set!-transformer}.

The property value must be an exact integer or procedure of one or two arguments. In the former case, the integer designates a field within the structure that should contain a procedure; the integer must be between 0 (inclusive) and the number of non-automatic fields in the structure type (exclusive, not counting supertype fields), and the designated field must also be specified as immutable.

If the property value is a procedure of one argument, then the procedure serves as a syntax transformer and for \texttt{set!} transformations. If the property value is a procedure of two arguments, then the first argument is the structure whose type has \texttt{prop:set!-transformer} property, and the second argument is a syntax object as for a syntax transformer and for \texttt{set!} transformations; \texttt{set!-transformer-procedure} applied to the structure produces a
new function that accepts just the syntax object and calls the procedure associated through the property. Finally, if the property value is an integer, the target identifier is extracted from the structure instance; if the field value is not a procedure of one argument, then a procedure that always calls \texttt{raise-syntax-error} is used, instead.

If a value has both the \texttt{prop:set!-transformer} and \texttt{prop:rename-transformer} properties, then the latter takes precedence. If a structure type has the \texttt{prop:set!-transformer} and \texttt{prop:procedure} properties, then the former takes precedence for the purposes of macro expansion.

\begin{verbatim}
(rename-transformer? v) → boolean?
  v : any/c
\end{verbatim}

Returns \#t if \texttt{v} is a value created by \texttt{make-rename-transformer} or an instance of a structure type with the \texttt{prop:rename-transformer} property, \#f otherwise.

Examples:

\begin{verbatim}
> (rename-transformer? (make-rename-transformer #'values))
#t
> (rename-transformer? 'not-a-rename-transformer)
#f
\end{verbatim}

\begin{verbatim}
(make-rename-transformer id-stx) → rename-transformer?
  id-stx : syntax?
\end{verbatim}

Creates a rename transformer that, when used as a transformer binding, acts as a transformer that inserts the identifier \texttt{id-stx} in place of whatever identifier binds the transformer, including in non-application positions, in \texttt{set!} expressions.

Such a transformer could be written manually, but the one created by \texttt{make-rename-transformer} triggers special cooperation with the parser and other syntactic forms when \texttt{id} is bound to the rename transformer:

\begin{itemize}
  \item The parser installs a \texttt{free-identifier=} and \texttt{identifier-binding} equivalence between \texttt{id} and \texttt{id-stx}, as long as \texttt{id-stx} does not have a true value for the \texttt{'not-free-identifier=} syntax property.
  \item A provide of \texttt{id} provides the binding indicated by \texttt{id-stx} instead of \texttt{id}, as long as \texttt{id-stx} does not have a true value for the \texttt{'not-free-identifier=} syntax property and as long as \texttt{id-stx} has a binding.
  \item If provide exports \texttt{id}, it uses a symbol-valued \texttt{'nominal-id} property of \texttt{id-stx} to specify the “nominal source identifier” of the binding as reported by \texttt{identifier-binding}.
\end{itemize}
• If `id-stx` has a true value for the `'not-provide-all-defined` syntax property, then `id` (or its target) is not exported by `all-defined-out`.

• The `syntax-local-value` function recognizes rename-transformer bindings and consult their targets.

Examples:

```lisp
> (define-syntax my-or (make-rename-transformer #'or))
> (my-or #f #t)
#t
> (free-identifier=? #'(my-or #'or)
#t
```

Changed in version 6.3 of package `base`: Removed an optional second argument.

Changed in version 7.4.0.10: Adjust rename-transformer expansion to add a macro-introduction scope, the same as regular macro expansion.

```lisp
(rename-transformer-target transformer) → identifier?

transformer : rename-transformer?
```

Returns the identifier passed to `make-rename-transformer` to create `transformer` or as indicated by a `prop:rename-transformer` property on `transformer`.

Example:

```lisp
> (rename-transformer-target (make-rename-transformer #'or))
#<syntax:eval:8:0 or>
```

`prop:rename-transformer` : struct-type-property?

A structure type property to identify structure types that act as rename transformers like the ones created by `make-rename-transformer`.

The property value must be an exact integer, an identifier syntax object, or a procedure that takes one argument. In the former case, the integer designates a field within the structure that should contain an identifier; the integer must be between 0 (inclusive) and the number of non-automatic fields in the structure type (exclusive, not counting supertype fields), and the designated field must also be specified as immutable.

If the property value is an identifier, the identifier serves as the target for renaming, just like the first argument to `make-rename-transformer`. If the property value is an integer, the target identifier is extracted from the structure instance; if the field value is not an identifier, then an identifier `?` with an empty context is used, instead.
If the property value is a procedure that takes one argument, then the procedure is called to obtain the identifier that the rename transformer will use as a target identifier. The returned identifier should probably have the `not-free-identifier=?` syntax property. If the procedure returns any value that is not an identifier, the `exn:fail:contract` exception is raised.

Examples:

```scheme
; Example of a procedure argument for prop:rename-transformer
> (define-syntax slv-1 'first-transformer-binding)
> (define-syntax slv-2 'second-transformer-binding)
> (begin-for-syntax
  (struct slv-cooperator (redirect-to-first?)
    #:property prop:rename-transformer
    (λ (inst)
      (if (slv-cooperator-redirect-to-first? inst)
          #'slv-1
          #'slv-2))))
> (define-syntax (slv-lookup stx)
  (syntax-case stx ()
    [(_ id)
      #'(syntax-local-value #'id)])
> (define-syntax slv-inst-1 (slv-cooperator #t))
> (define-syntax slv-inst-2 (slv-cooperator #f))
> (slv-lookup slv-inst-1)
'first-transformer-binding
> (slv-lookup slv-inst-2)
'second-transformer-binding
```

Changed in version 6.3 of package base: the property now accepts a procedure of one argument.

```scheme
(local-expand stx
  context-v
  stop-ids
  [intdef-ctx]) → syntax?

stx : any/c
context-v : (or/c 'expression 'top-level 'module 'module-begin list?)
stop-ids : (or/c (listof identifier?) empty #f)
intdef-ctx : (or/c internal-definition-context? = '()
              (listof internal-definition-context?) #f)
```

Expands `stx` in the lexical context of the expression currently being expanded. The `context-v` argument is used as the result of `syntax-local-context` for immediate expansions; a list indicates an internal-definition context, and more information on the form of
the list is below. If \textit{stx} is not already a syntax object, it is coerced with \texttt{(datum->syntax #f stx)} before expansion.

The \texttt{stop-ids} argument controls how far \texttt{local-expand} expands \texttt{stx}:

- If \texttt{stop-ids} is an empty list, then \texttt{stx} is recursively expanded (i.e. expansion proceeds to sub-expressions). The result is guaranteed to be a fully-expanded form, which can include the bindings listed in \texttt{|1.2.3.1 “Fully Expanded Programs|} plus \texttt{%expression} in any expression position.

- If \texttt{stop-ids} is a list containing just \texttt{module*}, then expansion proceeds as if \texttt{stop-ids} were an empty list, except that expansion does not recur to submodules defined with \texttt{module*} (which are left unexpanded in the result).

- If \texttt{stop-ids} is any other list, then \texttt{begin, quote, set!, %plain-lambda, case-lambda, let-values, letrec-values, if, begin0, with-continuation-mark, letrec-syntaxes+values, %plain-app, %expression, %top, and %variable-reference} are implicitly added to \texttt{stop-ids}. Expansion proceeds recursively, stopping when the expander encounters any of the forms in \texttt{stop-ids}, and the result is the partially-expanded form.

When the expander would normally implicitly introduce a \texttt{%app, %datum, or %top} identifier as described in \texttt{|1.2.3.2 “Expansion Steps|} it checks to see if an identifier with the same binding as the one to be introduced appears in \texttt{stop-ids}. If so, the identifier is \texttt{not} introduced; the result of expansion is the bare application, literal data expression, or unbound identifier rather than one wrapped in the respective explicit form.

When \texttt{%plain-module-begin} is not in \texttt{stop-ids}, the \texttt{%plain-module-begin} transformer detects and expands sub-forms (such as \texttt{define-values}) regardless of the identifiers presence in \texttt{stop-ids}.

Expansion does not replace the scopes in a local-variable reference to match the binding identifier.

- If \texttt{stop-ids} is \texttt{#f} instead of a list, then \texttt{stx} is expanded only as long as the outermost form of \texttt{stx} is a macro (i.e. expansion does \texttt{not} proceed to sub-expressions, and it does not replace the scopes in a local-variable reference to match the binding identifier). The \texttt{%app, %datum, and %top} identifiers are never introduced.

Independent of \texttt{stop-ids}, when \texttt{local-expand} encounters an identifier that has a local binding but no binding in the current expansion context, the variable is left as-is (as opposed to triggering an “out of context” syntax error).

When \texttt{context-v} is \texttt{‘module-begin}, and the result of expansion is a \texttt{%plain-module-begin} form, then a \texttt{‘submodule} syntax property is added to each enclosed module form (but not \texttt{module*} forms) in the same way as by module expansion.
If the `intdef-ctx` argument is an internal-definition context, its bindings and bindings from all parent internal-definition contexts are added to the local binding context during the dynamic extent of the call to `local-expand`. Additionally, unless `#f` was provided for the `add-scope?` argument to `syntax-local-make-definition-context` when the internal-definition context was created, its scope (but not the scopes of any parent internal-definition contexts) is added to the lexical information for both `stx` prior to its expansion and the expansion result (because the expansion might introduce bindings or references to internal-definition bindings). If `intdef-ctx` is a list, all bindings from all of the provided internal-definition contexts and their parents are added to the local binding context, and the scope from each context for which `add-scope?` was not `#f` is added in the same way. For backwards compatibility, providing `#f` for `intdef-ctx` is treated the same as providing an empty list.

For a particular internal-definition context, generate a unique value and put it into a list for `context-v`. To allow liberal expansion of `define` forms, the generated value should be an instance of a structure with a true value for `prop:liberal-define-context`. If the internal-definition context is meant to be self-contained, the list for `context-v` should contain only the generated value; if the internal-definition context is meant to splice into an immediately enclosing context, then when `syntax-local-context` produces a list, `cons` the generated value onto that list.

When expressions are expanded via `local-expand` with an internal-definition context `intdef-ctx`, and when the expanded expressions are incorporated into an overall form `new-stx`, then typically `internal-definition-context-track` should be applied to `intdef-ctx` and `new-stx` to provide expansion history to external tools.

This procedure must be called during the dynamic extent of a syntax transformer application by the expander or while a module is visited (see `syntax-transforming?`), otherwise the `exn:fail:contract` exception is raised.

Examples:

```scheme
> (define-syntax-rule (do-print x ...)  
  (printf x ...))
> (define-syntax-rule (hello x)  
  (do-print "hello ~a" x))
> (define-syntax (show stx)  
  (syntax-case stx ()  
    [(_ x)  
      (let ([partly (local-expand #'(hello x)  
        'expression  
        (list #'do-print))]  
       [fully (local-expand #'(hello x)  
        'expression  
        #f)])  
        (printf "partly expanded: ~s\n" (syntax->datum partly))  
        (printf "fully expanded: ~s\n" (syntax->datum fully))
```
partly expanded: (do-print "hello ~a" 1)
fully expanded: (printf "hello ~a" 1)
hello 1

Changed in version 6.0.1.3 of package base: Changed treatment of %top so that it is never introduced as an explicit wrapper.

Changed in version 6.0.90.27: Loosened the contract on the intdef-ctx argument to allow an empty list, which is treated the same way as #f.

\[
\begin{align*}
\text{syntax-local-expand-expression} & \quad \text{stx} \\
\text{[opaque-only?]}
\end{align*}
\]

\[
\rightarrow \begin{cases} 
\text{if opaque-only? \#f syntax?} & \text{syntax?} \\
\text{stx : any/c} & \text{opaque-only? : any/c = \#f} 
\end{cases}
\]

Like local-expand given 'expression and an empty stop list, but with two results: a syntax object for the fully expanded expression, and a syntax object whose content is opaque.

The latter can be used in place of the former (perhaps in a larger expression produced by a macro transformer), and when the macro expander encounters the opaque object, it substitutes the fully expanded expression without re-expanding it; the exn:fail:syntax exception is raised if the expansion context includes scopes that were not present for the original expansion, in which case re-expansion might produce different results. Consistent use of syntax-local-expand-expression and the opaque object thus avoids quadratic expansion times when local expansions are nested.

If opaque-only? is true, then the first result is #f instead of the expanded expression. Obtaining only the second, opaque result is more efficient in some expansion contexts.

Unlike local-expand, syntax-local-expand-expression normally produces an expanded expression that contains no %expression forms. However, if syntax-local-expand-expression is used within an expansion that is triggered by an enclosing local-expand call, then the result of syntax-local-expand-expression can include %expression forms.

This procedure must be called during the dynamic extent of a syntax transformer application by the expander or while a module is visited (see syntax-transforming?), otherwise the exn:fail:contract exception is raised.

Changed in version 6.90.0.13 of package base: Added the opaque-only? argument.
Like `local-expand`, but `stx` is expanded as a transformer expression instead of a run-time expression.

Any lifted expressions—from calls to `syntax-local-lift-expression` during the expansion of `stx`—are captured in the result. If `context-v` is `'top-level`, then lifts are captured in a `begin` form, otherwise lifts are captured in `let-values` forms. If no expressions are lifted during expansion, then no `begin` or `let-values` wrapper is added.

```
(local-expand/capture-lifts stx context-v stop-ids [intdef-ctx lift-ctx]) -> syntax?
```

Like `local-expand`, but `stx` is a syntax object that represents a `begin` expression. Lifted expressions—from calls to `syntax-local-lift-expression` during the expansion of `stx`—appear with their identifiers in `define-values` forms, and the expansion of `stx` is the last expression in the `begin`. The `lift-ctx` value is reported by `syntax-local-lift-context` during local expansion. The lifted expressions are not expanded, but instead left as provided in the `begin` form.

If `context-v` is `'top-level` or `'module`, then `module` forms can appear in the result as added via `syntax-local-lift-module`. If `context-v` is `'module`, then `module*` forms can appear, too.

```
(local-transformer-expand/capture-lifts stx context-v stop-ids [intdef-ctx lift-ctx]) -> syntax?
```
Like `local-expand/capture-lifts`, but `stx` is expanded as a transformer expression instead of a run-time expression. Lifted expressions are reported as `define-values` forms (in the transformer environment).

```
(internal-definition-context? v) -> boolean?
```

Returns `#t` if `v` is an internal-definition context, `#f` otherwise.

```
(syntax-local-make-definition-context [parent-ctx add-scope?])
```

→ internal-definition-context?

```
pARENT-CTX : (or/c internal-definition-context? #f) = #f
ADD-SCOPE? : any/c = #t
```

Creates an opaque internal-definition context value to be used with `local-expand` and other functions. A transformer should create one context for each set of internal definitions to be expanded, and use it when expanding any form whose lexical context should include the definitions. After discovering an internal `define-values` or `define-syntaxes` form, use `syntax-local-bind-syntaxes` to add bindings to the context.

An internal-definition context internally creates a scope to represent the context. Unless `add-scope?` is `#f`, the scope is added to any form that is expanded within the context or that appears as the result of a (partial) expansion within the context.

If `parent-ctx` is not `#f`, then `parent-ctx` is made the parent internal-definition context for the new internal-definition context. Whenever the new context’s bindings are added to the local binding context (e.g. by providing the context to `local-expand`, `syntax-local-bind-syntaxes`, or `syntax-local-value`), then the bindings from `parent-ctx` are also added as well. If `parent-ctx` was also created with a parent internal-definition context, bindings from its parent are also added, and so on recursively. Note that the scopes of parent contexts are not added implicitly, only the bindings, even when the scope of the child context would be implicitly added. If the scopes of parent definition contexts should be added, the parent contexts must be provided explicitly.

Additionally, if the created definition context is intended to be spliced into a surrounding definition context, the surrounding context should always be provided for the `parent-ctx`
argument to ensure the necessary use-site scopes are added to macros expanded in the context. Otherwise, expansion of nested definitions can be inconsistent with the expansion of definitions in the surrounding context.

The scope associated with a new definition context is pruned from quote-syntax forms only when it is created during the dynamic extent of a syntax transformer application or in a begin-for-syntax form (potentially nested) within a module being expanded.

This procedure must be called during the dynamic extent of a syntax transformer application by the expander or while a module is visited (see syntax-transforming?), otherwise the exn:fail:contract exception is raised.

Changed in version 6.3 of package base: Added the add-scope? argument, and made calling internal-definition-context-seal no longer necessary.

```
syntax-local-bind-syntaxes  id-list
  expr
  intdef-ctx
  [extra-intdef-ctxs]] -> void?

id-list : (listof identifier?)
expr : (or/c syntax? #f)
intdef-ctx : internal-definition-context?
extra-intdef-ctxs : (or/c internal-definition-context?
                       (listof internal-definition-context?))
               = '()
```

Binds each identifier in id-list within the internal-definition context represented by intdef-ctx, where intdef-ctx is the result of syntax-local-make-definition-context. The lexical information of each element of extra-intdef-ctxs is also added to each identifier in id-list before binding.

Supply #f for expr when the identifiers correspond to define-values bindings, and supply a compile-time expression when the identifiers correspond to define-syntaxes bindings. In the latter case, the number of values produced by the expression should match the number of identifiers, otherwise the exn:fail:contract:arity exception is raised.

When expr is not #f, it is expanded in an expression context and evaluated in the current transformer environment. In this case, the bindings and lexical information from both intdef-ctx and extra-intdef-ctxs are used to enrich expr’s lexical information and extend the local binding context in the same way as the fourth argument to local-expand. If expr is #f, the value provided for extra-intdef-ctxs is ignored.

This procedure must be called during the dynamic extent of a syntax transformer application by the expander or while a module is visited (see syntax-transforming?), otherwise the exn:fail:contract exception is raised.

Changed in version 6.90.0.27 of package base: Added the extra-intdef-ctxs argument.
(internal-definition-context-binding-identifiers intdef-ctx)
-> (listof identifier?)
intdef-ctx : internal-definition-context?

Returns a list of all binding identifiers registered for intdef-ctx through syntax-local-bind-syntaxes. Each identifier in the returned list includes the internal-definition context’s scope.

Added in version 6.3.0.4 of package base.

(internal-definition-context-introduce intdef-ctx stx [mode]) -> syntax?
intdef-ctx : internal-definition-context?
stx : syntax?
mode : (or/c 'flip 'add 'remove) = 'flip

Flips, adds, or removes (depending on mode) the scope for intdef-ctx for all parts of stx.

Added in version 6.3 of package base.

(internal-definition-context-seal intdef-ctx) -> void?
intdef-ctx : internal-definition-context?

For backward compatibility only; has no effect.

(identifier-remove-from-definition-context id-stx intdef-ctx)
-> identifier?
id-stx : identifier?
intdef-ctx : (or/c internal-definition-context? (listof internal-definition-context?))

Removes all of the scopes of intdef-ctx (or of each element in a list intdef-ctx) from id-stx.

The identifier-remove-from-definition-context function is provided for backward compatibility; the more general internal-definition-context-introduce function is preferred.

Changed in version 6.3 of package base: Simplified the operation to scope removal.

prop:expansion-contexts : struct-type-property?

A structure type property to constrain the use of macro transformers and rename transformers. The property’s value must be a list of symbols, where the allowed symbols are 'expression, 'top-level, 'module, 'module-begin, and 'definition-context. Each
symbol corresponds to an expansion context in the same way as for `local-expand` or as reported by `syntax-local-context`, except that `definition-context` is used (instead of a list) to represent an internal-definition context.

If an identifier is bound to a transformer whose list does not include a symbol for a particular use of the identifier, then the use is adjusted as follows:

- In a `module-begin` context, then the use is wrapped in a `begin` form.
- In a `module`, `top-level`, `internal-definition` or context, if `expression` is present in the list, then the use is wrapped in an `#%expression` form.
- Otherwise, a syntax error is reported.

The `prop:expansion-contexts` property is most useful in combination with `prop:rename-transformer`, since a general transformer procedure can use `syntax-local-context`. Furthermore, a `prop:expansion-contexts` property makes the most sense when a rename transformer’s identifier has the `not-free-identifier=?` property, otherwise a definition of the binding creates a binding alias that effectively routes around the `prop:expansion-contexts` property.

Added in version 6.3 of package base.

```scheme
(syntax-local-value id-stx
   [failure-thunk
    intdef-ctx]) → any
id-stx : identifier?
failure-thunk : (or/c (-> any) #f) = #f
intdef-ctx : (or/c internal-definition-context? (listof internal-definition-context?) #f) = '()
```

Returns the transformer binding value of the identifier `id-stx` in the context of the current expansion. If `intdef-ctx` is not `#f`, bindings from all provided definition contexts are also considered. Unlike the fourth argument to `local-expand`, the scopes associated with the provided definition contexts are not used to enrich `id-stx`’s lexical information.

If `id-stx` is bound to a rename transformer created with `make-rename-transformer`, `syntax-local-value` effectively calls itself with the target of the rename and returns that result, instead of the rename transformer.

If `id-stx` has no transformer binding (via `define-syntax`, `let-syntax`, etc.) in that environment, the result is obtained by applying `failure-thunk` if not `#f`. If `failure-thunk` is `false`, the `exn:fail:contract` exception is raised.

This procedure must be called during the dynamic extent of a syntax transformer application by the expander or while a module is visited (see `syntax-transforming?`), otherwise the `exn:fail:contract` exception is raised.
Examples:

> (define-syntax swiss-cheeses? #t)
> (define-syntax (transformer stx)
  (if (syntax-local-value #'(swiss-cheeses?)
    #'(gruyère emmental raclette)
    #'(roquefort camembert boursin)))
> (transformer)
'(gruyère emmental raclette)

Examples:

> (define-syntax (transformer-2 stx)
  (syntax-local-value #'(something-else (λ () (error "no binding")))))
> (transformer-2)
no binding

Examples:

> (define-syntax nachos #'(printf "nachos\n")
> (define-syntax chips (make-rename-transformer #'nachos))
> (define-syntax (transformer-3 stx)
  (syntax-local-value #'chips))
> (transformer-3)
nachos

Changed in version 6.90.0.27 of package base: Changed `intdef-ctx` to accept a list of internal-definition contexts in addition to a single internal-definition context or `#f`.

```
(syntax-local-value/immediate id-stx
  [failure-thunk
   intdef-ctx]) → any

id-stx : syntax?
failure-thunk : (or/c (→ any) #f) = #f
intdef-ctx : (or/c internal-definition-context? (listof internal-definition-context?) #f) = '()
```

Like `syntax-local-value`, but the result is normally two values. If `id-stx` is bound to a rename transformer, the results are the rename transformer and the identifier in the transformer. If `id-stx` is not bound to a rename transformer, then the results are the value that `syntax-local-value` would produce and `#f`.

If `id-stx` has no transformer binding, then `failure-thunk` is called (and it can return any number of values), or an exception is raised if `failure-thunk` is `#f`. Beware that `provide` on an `id` bound to a rename transformer may export the target of the rename instead of `id`. See `make-rename-transformer` for more information.
Examples:

```lisp
> (define-syntax agent-007 (make-rename-transformer #'james-bond))
> (define-syntax (show-secret-identity stx)
   (syntax-parse stx
    [(_ name:id)
     (define-values [_ orig-name] (syntax-local-value/immediate #'(name))
      #'(name #,orig-name))))
> (show-secret-identity agent-007)
'(agent-007 james-bond)
```

Returns a fresh identifier, and cooperates with the module, `letrec-syntaxes+values`, `define-syntaxes`, `begin-for-syntax`, and top-level expanders to bind the generated identifier to the expression `stx`.

A run-time expression within a module is lifted to the module’s top level, just before the expression whose expansion requests the lift. Similarly, a run-time expression outside of a module is lifted to a top-level definition. A compile-time expression in a `letrec-syntaxes+values` or `define-syntaxes` binding is lifted to a `let` wrapper around the corresponding right-hand side of the binding. A compile-time expression within `begin-for-syntax` is lifted to a `define` declaration just before the requesting expression within the `begin-for-syntax`.

Other syntactic forms can capture lifts by using `local-expand/capture-lifts` or `local-transformer-expand/capture-lifts`.

This procedure must be called during the dynamic extent of a syntax transformer application by the expander or while a module is visited (see `syntax-transforming?`), otherwise the `exn:fail:contract` exception is raised.

In addition, this procedure can be called only when a lift target is available, as indicated by `syntax-transforming-with-lifts?`.

```
(syntax-local-lift-values-expression n stx)
→ (listof identifier?)
  n : exact-nonnegative-integer?
  stx : syntax?
```

Like `syntax-local-lift-expression`, but binds the result to `n` identifiers, and returns a list of the `n` identifiers.

This procedure must be called during the dynamic extent of a syntax transformer application.
by the expander or while a module is visited (see `syntax-transforming?`), otherwise the
`exn:fail:contract` exception is raised.

```scheme
(syntax-local-lift-context) → any/c
```

Returns a value that represents the target for expressions lifted via `syntax-local-lift-expression`. That is, for different transformer calls for which this procedure returns the same value (as determined by `eq?`), lifted expressions for the two transformer are moved to the same place. Thus, the result is useful for caching lift information to avoid redundant lifts.

This procedure must be called during the dynamic extent of a syntax transformer application by the expander or while a module is visited (see `syntax-transforming?`), otherwise the `exn:fail:contract` exception is raised.

```scheme
(syntax-local-lift-module stx) → void?
    stx : syntax?
```

Cooperates with the module form or top-level expansion to add `stx` as a module declaration in the enclosing module or top-level. The `stx` form must start with `module` or `module*`, where the latter is only allowed within the expansion of a module.

The module is not immediately declared when `syntax-local-lift-module` returns. Instead, the module declaration is recorded for processing when expansion returns to the enclosing module body or top-level sequence.

This procedure must be called during the dynamic extent of a syntax transformer application by the expander or while a module is visited (see `syntax-transforming?`), otherwise the `exn:fail:contract` exception is raised.

If the current expression being transformed is not within a `module` form or within a top-level expansion, then the `exn:fail:contract` exception is raised. If `stx` form does not start with `module` or `module*`, or if it starts with `module*` in a top-level context, the `exn:fail:contract` exception is raised.

Added in version 6.3 of package `base`.

```scheme
(syntax-local-lift-module-end-declaration stx) → void?
    stx : syntax?
```

Cooperates with the module form to insert `stx` as a top-level declaration at the end of the module currently being expanded. If the current expression being transformed is in phase level 0 and not in the module top-level, then `stx` is eventually expanded in an expression context. If the current expression being transformed is in a higher phase level (i.e., nested within some number of `begin-for-syntaxes` within a module top-level), then the lifted declaration is placed at the very end of the module (under a suitable number of `begin-for-syntaxes`), instead of merely the end of the enclosing `begin-for-syntax`.
This procedure must be called during the dynamic extent of a syntax transformer application by the expander or while a module is visited (see `syntax-transforming?`), otherwise the `exn:fail:contract` exception is raised.

If the current expression being transformed is not within a module form (see `syntax-transforming-module-expression?`), then the `exn:fail:contract` exception is raised.

```lisp
(syntax-local-lift-require raw-require-spec stx) → syntax?
raw-require-spec : any/c
stx : syntax?
```

Lifts a `%require` form corresponding to `raw-require-spec` (either as a syntax object or datum) to the top-level or to the top of the module currently being expanded or to an enclosing `begin-for-syntax`.

The resulting syntax object is the same as `stx`, except that a fresh scope is added. The same scope is added to the lifted `%require` form, so that the `%require` form can bind uses of imported identifiers in the resulting syntax object (assuming that the lexical information of `stx` includes the binding environment into which the `%require` is lifted).

If `raw-require-spec` and `stx` are part of the input to a transformer, then typically `syntax-local-introduce` should be applied to each before passing them to `syntax-local-lift-require`, and then `syntax-local-introduce` should be applied to the result of `syntax-local-lift-require`. Otherwise, marks added by the macro expander can prevent access to the new imports.

This procedure must be called during the dynamic extent of a syntax transformer application by the expander or while a module is visited (see `syntax-transforming?`), otherwise the `exn:fail:contract` exception is raised.

Changed in version 6.90.0.27 of package `base`: Changed the scope added to inputs from a macro-introduction scope to one that does not affect whether or not the resulting syntax is considered original as reported by `syntax-original?`.

```lisp
(syntax-local-lift-provide raw-provide-spec-stx) → void?
raw-provide-spec-stx : syntax?
```

Lifts a `%provide` form corresponding to `raw-provide-spec-stx` to the top of the module currently being expanded or to an enclosing `begin-for-syntax`.

This procedure must be called during the dynamic extent of a syntax transformer application by the expander or while a module is visited (see `syntax-transforming?`), otherwise the `exn:fail:contract` exception is raised.

If the current expression being transformed is not within a module form (see `syntax-transforming-module-expression?`), then the `exn:fail:contract` exception is raised.
(syntax-local-name) ➞ any/c

Returns an inferred name for the expression position being transformed, or #f if no such name is available. A name is normally a symbol or an identifier. See also §1.2.6 "Inferred Value Names".

This procedure must be called during the dynamic extent of a syntax transformer application by the expander or while a module is visited (see syntax-transforming?), otherwise the exn:fail:contract exception is raised.

(syntax-local-context) ➞ (or/c 'expression 'top-level 'module 'module-begin list?)

Returns an indication of the context for expansion that triggered a syntax transformer call. See §1.2.3.3 "Expansion Context" for more information on contexts.

The symbol results indicate that the expression is being expanded for an expression context, a top-level context, a module context, or a module-begin context.

A list result indicates expansion in an internal-definition context. The identity of the list’s first element (i.e., its eq?ness) reflects the identity of the internal-definition context; in particular two transformer expansions receive the same first value if and only if they are invoked for the same internal-definition context. Later values in the list similarly identify internal-definition contexts that are still being expanded, and that required the expansion of nested internal-definition contexts.

This procedure must be called during the dynamic extent of a syntax transformer application by the expander or while a module is visited (see syntax-transforming?), otherwise the exn:fail:contract exception is raised.

(syntax-local-phase-level) ➞ exact-integer?

During the dynamic extent of a syntax transformer application by the expander, the result is the phase level of the form being expanded. Otherwise, the result is 0.

Examples:

; a macro bound at phase 0
> (define-syntax (print-phase-level stx)
   (printf "phase level: ~a~n" (syntax-local-phase-level))
   #'(void))
> (require (for-meta 2 racket/base))
> (begin-for-syntax
   ; a macro bound at phase 1
   )
(define-syntax (print-phase-level stx)
  (printf "phase level: ~a~n" (syntax-local-phase-level))
  #'(void)))

> (print-phase-level)
phase level: 0

> (begin-for-syntax (print-phase-level))
phase level: 1

(syntax-local-module-exports mod-path)
→ (listof (cons/c (or/c exact-integer? #f) (listof symbol?)))

mod-path : (or/c module-path?
           (syntax/c module-path?))

Returns an association list from phase-level numbers (or #f for the label phase level) to lists of symbols, where the symbols are the names of provided bindings from mod-path at the corresponding phase level.

This procedure must be called during the dynamic extent of a syntax transformer application by the expander or while a module is visited (see syntax-transforming?), otherwise the exn:fail:contract exception is raised.

(syntax-local-submodules) → (listof symbol?)

Returns a list of submodule names that are declared via module (as opposed to module*) in the current expansion context.

This procedure must be called during the dynamic extent of a syntax transformer application by the expander or while a module is visited (see syntax-transforming?), otherwise the exn:fail:contract exception is raised.

(syntax-local-get-shadower id-stx [only-generated?]) → identifier?

id-stx : identifier?
only-generated? : any/c = #f

Adds scopes to id-stx so that it refers to bindings in the current expansion context or could bind any identifier obtained via (syntax-local-get-shadower id-stx) in more nested contexts. If only-generated? is true, the phase-spanning scope of the enclosing module or namespace is omitted from the added scopes, however, which limits the bindings that can be referenced (and therefore avoids certain ambiguous references).

This function is intended for the implementation of syntax-parameterize and local-require.

This procedure must be called during the dynamic extent of a syntax transformer application by the expander or while a module is visited (see syntax-transforming?), otherwise the exn:fail:contract exception is raised.
Changed in version 6.3 of package base: Simplified to the minimal functionality needed for syntax-parameterize and local-require.

\[
\text{syntax-local-make-delta-introducer } \text{id-stx} \rightarrow \text{procedure?}
\]
\[
id-stx : \text{identifier?}
\]

For (limited) backward compatibility only; raises \text{exn:fail:unsupported}.

Changed in version 6.3 of package base: changed to raise \text{exn:fail:supported}.

\[
\text{syntax-local-certifier } [\text{active?}]
\rightarrow [(\text{syntax?}) (\text{any/c} (\text{or/c} \text{procedure?} \text{#f}))]
\rightarrow* \text{. syntax?}
\]
\[
\text{active? : boolean? } = \text{#f}
\]

For backward compatibility only; returns a procedure that returns its first argument.

\[
\text{syntax-transforming?} \rightarrow \text{boolean?}
\]

Returns \#t during the dynamic extent of a syntax transformer application by the expander and while a module is being visited, \#f otherwise.

\[
\text{syntax-transforming-with-lifts?} \rightarrow \text{boolean?}
\]

Returns \#t if \text{syntax-transforming?} produces \#t and a target context is available for lifting expressions (via \text{syntax-local-lift-expression}), \#f otherwise.

Currently, \text{syntax-transforming?} implies \text{syntax-transforming-with-lifts?}.

Added in version 6.3.0.9 of package base.

\[
\text{syntax-transforming-module-expression?} \rightarrow \text{boolean?}
\]

Returns \#t during the dynamic extent of a syntax transformer application by the expander for an expression within a module form, \#f otherwise.

\[
\text{syntax-local-identifier-as-binding } \text{id-stx} \rightarrow \text{identifier?}
\]
\[
id-stx : \text{identifier?}
\]

Returns an identifier like \text{id-stx}, but without use-site scopes that were previously added to the identifier as part of a macro expansion in the current definition context.

In a syntax transformer that runs in a non-expression context and forces the expansion of subforms with \text{local-expand}, use \text{syntax-local-identifier-as-binding} on an identifier from the expansion before moving it into a binding position or comparing it with \text{bound-identifier=?}. Otherwise, the results can be inconsistent with the way that \text{define} works in the same definition context.
This procedure must be called during the dynamic extent of a syntax transformer application by the expander or while a module is visited (see `syntax-transforming?q`), otherwise the `exn:fail:contract` exception is raised.

Added in version 6.3 of package `base`.

```
(syntax-local-introduce stx) \rightarrow syntax?

stx : syntax?
```

Produces a syntax object that is like `stx`, except that the presence of scopes for the current expansion—both the macro-introduction scope and the use-site scope, if any—is flipped on all parts of the syntax object. See §1.2.3.5 “Transformer Bindings” for information on macro-introduction and use-site scopes.

This procedure must be called during the dynamic extent of a syntax transformer application by the expander or while a module is visited (see `syntax-transforming?q`), otherwise the `exn:fail:contract` exception is raised.

Example:

```
> (module example racket
  `(define-syntax (require-math
                   stx)
     `(syntax-local-introduce #'(require racket/math)))
  (require-math)
  pi)

(make-syntax-introducer [as-use-site?])
\rightarrow ((syntax?) ((or/c 'flip 'add 'remove)) . -* . syntax?)

as-use-site? : any/c = #f
```

Produces a procedure that encapsulates a fresh scope and flips, adds, or removes it in a given syntax object. By default, the fresh scope is a macro-introduction scope, but providing a true value for `as-use-site?` creates a scope that is like a use-site scope; the difference is in how the scopes are treated by `syntax-original?q`.

The action of the generated procedure can be `flip` (the default) to flip the presence of a scope in each part of a given syntax object, `add` to add the scope to each regardless of whether it is present already, or `remove` to remove the scope when it is currently present in any part.

Multiple applications of the same `make-syntax-introducer` result procedure use the same scope, and different result procedures use distinct scopes.

Changed in version 6.3 of package `base`: Added the optional `as-use-site?` argument, and added the optional operation argument in the result procedure.
(make-interned-syntax-introducer key)
     ➔ ((syntax?) ((or/c 'flip 'add 'remove)) . ->* . syntax?)
     key : symbol?

Like **make-syntax-introducer**, but the encapsulated scope is interned. Multiple calls to **make-interned-syntax-introducer** with the same key will produce procedures that flip, add, or remove the same scope, even across phases and module instantiations. Furthermore, the scope remains consistent even when embedded in compiled code, so a scope created with **make-interned-syntax-introducer** will retain its identity in syntax objects loaded from compiled code. (In this sense, the relationship between **make-syntax-introducer** and **make-interned-syntax-introducer** is analogous to the relationship between gensym and quote.)

This function is intended for the implementation of separate binding environments within a single phase, for which the scope associated with each environment must be the same across modules.

Unlike **make-syntax-introducer**, the scope added by a procedure created with **make-interned-syntax-introducer** is always treated like a use-site scope, not a macro-introduction scope, so it does not affect originalness as reported by **syntax-original?**.

Added in version 6.90.0.28 of package base.

(make-syntax-delta-introducer ext-stx base-stx [phase-level])
     ➔ ((syntax?) ((or/c 'flip 'add 'remove)) . ->* . syntax?)
     ext-stx : identifier?
     base-stx : (or/c syntax? #f)
     phase-level : (or/c #f exact-integer?)
     = (syntax-local-phase-level)

Produces a procedure that behaves like the result of **make-syntax-introducer**, but using a set of scopes from ext-stx and with a default action of 'add.

- If the scopes of base-stx are a subset of the scopes of ext-stx, then the result of **make-syntax-delta-introducer** adds, removes, or flips scopes that are in the set for ext-stx and not in the set for base-stx.

- If the scopes of base-stx are not a subset of the scopes of ext-stx, but if it has a binding, then the set of scopes associated with the binding id subtracted from the set of scopes for ext-stx, and the result of **make-syntax-delta-introducer** adds, removes, or flips that difference.

A #f value for base-stx is equivalent to a syntax object with no scopes.

861
This procedure is potentially useful when some \textit{m-id} has a transformer binding that records some \textit{orig-id}, and a use of \textit{m-id} introduces a binding of \textit{orig-id}. In that case, the scopes one the use of \textit{m-id} added since the binding of \textit{m-id} should be transferred to the binding instance of \textit{orig-id}, so that it captures uses with the same lexical context as the use of \textit{m-id}.

If \texttt{ext-stx} is tainted or armed, then an identifier result from the created procedure is tainted.

\begin{verbatim}
(syntax-local-transforming-module-provides?) \rightarrow boolean?
\end{verbatim}

Returns \texttt{#t} while a provide transformer is running (see \texttt{make-provide-transformer}) or while an \texttt{expand} sub-form of \texttt{#%provide} is expanded, \texttt{#f} otherwise.

\begin{verbatim}
(syntax-local-module-defined-identifiers) \rightarrow (and/c hash? immutable?)
\end{verbatim}

Can be called only while \texttt{syntax-local-transforming-module-provides?} returns \texttt{#t}.

It returns a hash table mapping a phase-level number (such as \texttt{0}) to a list of all definitions at that phase level within the module being expanded. This information is used for implementing provide sub-forms like \texttt{all-defined-out}.

Beware that the phase-level keys are absolute relative to the enclosing module, and not relative to the current transformer phase level as reported by \texttt{syntax-local-phase-level}.

\begin{verbatim}
(syntax-local-module-required-identifiers mod-path phase-level) \rightarrow (or/c (listof (cons/c (or/c exact-integer? #f) (listof identifier?))) #f)
mod-path : (or/c module-path? #f)
phase-level : (or/c exact-integer? #f #t)
\end{verbatim}

Can be called only while \texttt{syntax-local-transforming-module-provides?} returns \texttt{#t}.

It returns an association list mapping phase levels to lists of identifiers. Each list of identifiers includes all bindings imported (into the module being expanded) using the module path \texttt{mod-path}, or all modules if \texttt{mod-path} is \texttt{#f}. The association list includes all identifiers imported with a \texttt{phase-level} shift, or all shifts if \texttt{phase-level} is \texttt{#t}. If \texttt{phase-level} is not \texttt{#t}, the result can be \texttt{#f} if no identifiers are exported at that phase.

When an identifier is renamed on import, the result association list includes the identifier by its internal name. Use \texttt{identifier-binding} to obtain more information about the identifier.

Beware that the phase-level keys are absolute relative to the enclosing module, and not relative to the current transformer phase level as reported by \texttt{syntax-local-phase-level}.  

862
prop:liberal-define-context : struct-type-property?
(liberal-define-context? v) → boolean?
v : any/c

An instance of a structure type with a true value for the prop:liberal-define-context property can be used as an element of an internal-definition context representation in the result of syntax-local-context or the second argument of local-expand. Such a value indicates that the context supports liberal expansion of define forms into potentially multiple define-values and define-syntaxes forms. The 'module and 'module-body contexts implicitly allow liberal expansion.

The liberal-define-context? predicate returns #t if v is an instance of a structure with a true value for the prop:liberal-define-context property, #f otherwise.

12.4.1 require Transformers

(require racket/require-transform) package: base

The bindings documented in this section are provided by the racket/require-transform library, not racket/base or racket.

A transformer binding whose value is a structure with the prop:require-transformer property implements a derived require-spec for require as a require transformer.

A require transformer is called with the syntax object representing its use as a require-spec within a require form, and the result must be two lists: a list of imports and a list of import-sources.

If the derived form contains a sub-form that is a require-spec, then it can call expand-import to transform the sub-require-spec to lists of imports and import sources.

See also define-require-syntax, which supports macro-style require transformers.

(expand-import require-spec)
→ (listof import?) (listof import-source?)
require-spec : syntax?

Expands the given require-spec to lists of imports and import sources. The latter specifies modules to be instantiated or visited, so the modules that it represents should be a superset of the modules represented in the former list (so that a module will be instantiated or visited even if all of imports are eventually filtered from the former list).

(make-require-transformer proc) → require-transformer?
proc : (syntax? . -> . (values
(listof import?)
(listof import-source?)))
Creates a require transformer using the given procedure as the transformer. Often used in combination with \texttt{expand-import}.

Examples:

\begin{verbatim}
> (require (for-syntax racket/require-transform))
> (define-syntax printing
  (make-require-transformer
   (lambda (stx)
     (syntax-case stx ()
       [(_ path)
        (printf "Importing: \n"
                 a
                 n" #'path)
        (expand-import #'path)])))))
> (require (printing racket/match))
Importing: #<syntax:eval:37:0 racket/match>
\end{verbatim}

\textbf{prop:require-transformer} : \texttt{struct-type-property}?

A property to identify require transformers. The property value must be a procedure that takes the structure and returns a transformer procedure; the returned transformer procedure takes a syntax object and returns import and import-source lists.

\begin{verbatim}
(define require-transformer? meaningful? v)
\end{verbatim}

\texttt{v} : \texttt{any/c}

Returns \texttt{#t} if \texttt{v} has the \texttt{prop:require-transformer} property, \texttt{#f} otherwise.

\begin{verbatim}
(struct import (local-id
  src-sym
  src-mod-path
  mode
  req-mode
  orig-mode
  orig-stx)
#:extra-constructor-name make-import)
local-id : \texttt{identifier}?
src-sym : \texttt{symbol}?
src-mod-path : (or/c module-path? (syntax/c module-path?))
mode : (or/c exact-integer? #f)
req-mode : (or/c exact-integer? #f)
orig-mode : (or/c exact-integer? #f)
orig-stx : \texttt{syntax}?
\end{verbatim}

A structure representing a single imported identifier:
• **local-id** — the identifier to be bound within the importing module.

• **src-sym** — the external name of the binding as exported from its source module.

• **src-mod-path** — a module path (relative to the importing module) for the source of the imported binding.

• **mode** — the phase level of the binding in the importing module.

• **req-mode** — the phase level shift of the import relative to the exporting module.

• **orig-mode** — the phase level of the binding as exported by the exporting module.

• **orig-stx** — a syntax object for the source of the import, used for error reporting.

```scheme
(struct import-source (mod-path-stx mode)
#:extra-constructor-name make-import-source)

mod-path-stx : (syntax/c module-path?)
mode : (or/c exact-integer? #f)
```

A structure representing an imported module, which must be instantiated or visited even if no binding is imported into a module.

• **mod-path-stx** — a module path (relative to the importing module) for the source of the imported binding.

• **mode** — the phase level shift of the import.

```scheme
(current-require-module-path)  ->  (or/c #f module-path-index?)
(current-require-module-path module-path)  ->  void?
  module-path : (or/c #f module-path-index?)
```

A parameter that determines how relative `require`-level module paths are expanded to `#%require`-level module paths by `convert-relative-module-path` (which is used implicitly by all built-in `require` sub-forms).

When the value of `current-require-module-path` is `#f`, relative module paths are left as-is, which means that the `require` context determines the resolution of the module path.

The `require` form parameterizes `current-require-module-path` as `#f` while invoking sub-form transformers, while `relative-in` parameterizes to a given module path.

```scheme
(convert-relative-module-path module-path)
  ->  (or/c module-path?
      (syntax/c module-path?))
module-path : (or/c module-path?
      (syntax/c module-path?))
```

865
Converts \texttt{module-path} according to \texttt{current-require-module-path}.

If \texttt{module-path} is not relative or if the value of \texttt{current-require-module-path} is \#f, then \texttt{module-path} is returned. Otherwise, \texttt{module-path} is converted to an absolute module path that is equivalent to \texttt{module-path} relative to the value of \texttt{current-require-module-path}.

\begin{verbatim}
(syntax-local-require-certifier)
  → ((syntax?) (or/c #f (syntax? . -> . syntax?)
          . ->* . syntax?)
\end{verbatim}

For backward compatibility only; returns a procedure that returns its first argument.

### 12.4.2 provide Transformers

\begin{verbatim}
(require racket/provide-transform) package: base
\end{verbatim}

The bindings documented in this section are provided by the \texttt{racket/provide-transform} library, not \texttt{racket/base} or \texttt{racket}.

A transformer binding whose value is a structure with the \texttt{prop:provide-transformer} property implements a derived \texttt{provide-spec} for \texttt{provide} as a \texttt{provide transformer}. A provide transformer is applied as part of the last phase of a module’s expansion, after all other declarations and expressions within the module are expanded.

A transformer binding whose value is a structure with the \texttt{prop:provide-pre-transformer} property implements a derived \texttt{provide-spec} for \texttt{provide} as a \texttt{provide pre-transformer}. A provide pre-transformer is applied as part of the first phase of a module’s expansion. Since it is used in the first phase, a provide pre-transformer can use functions such as \texttt{syntax-local-lift-expression} to introduce expressions and definitions in the enclosing module.

An identifier can have a transformer binding to a value that acts both as a provide transformer and provide pre-transformer. The result of a provide pre-transformer is \texttt{not} automatically re-expanded, so a provide pre-transformer can usefully expand to itself in that case.

A transformer is called with the syntax object representing its use as a \texttt{provide-spec} within a provide form and a list of symbols representing the export modes specified by enclosing \texttt{provide-specs}. The result of a provide transformer must be a list of \texttt{exports}, while the result of a provide pre-transformer is a syntax object to be used as a \texttt{provide-spec} in the last phase of module expansion.

If a derived form contains a sub-form that is a \texttt{provide-spec}, then it can call \texttt{expand-export} or \texttt{pre-expand-export} to transform the sub-\texttt{provide-spec} sub-form.

See also \texttt{define-provide-syntax}, which supports macro-style provide transformers.
\[ (\text{expand-exports} \ provide-spec \ modes) \rightarrow (\text{listof export?}) \]

\begin{align*}
provide-spec & : \text{syntax?} \\
modes & : (\text{listof (or/c exact-integer? #f)})
\end{align*}

Expands the given \textit{provide-spec} to a list of exports. The \textit{modes} list controls the expansion of sub-\textit{provide-specs}; for example, an identifier refers to a binding in the phase level of the enclosing \textit{provide} form, unless the \textit{modes} list specifies otherwise. Normally, \textit{modes} is either empty or contains a single element.

\[ (\text{pre-expand-exports} \ provide-spec \ modes) \rightarrow \text{syntax?} \]

\begin{align*}
provide-spec & : \text{syntax?} \\
modes & : (\text{listof (or/c exact-integer? #f)})
\end{align*}

Expands the given \textit{provide-spec} at the level of provide pre-transformers. The \textit{modes} argument is the same as for \textit{expand-exports}.

\[ (\text{make-provide-transformer} \ proc) \rightarrow \text{provide-transformer?} \]

\begin{align*}
proc & : (\text{syntax?} \ (\text{listof (or/c exact-integer? #f)}) \\
& \hspace{1cm} . \rightarrow . (\text{listof export?}))
\end{align*}

\[ (\text{make-provide-pre-transformer} \ proc \ pre-proc) \rightarrow (\text{and/c provide-transformer? provide-pre-transformer?}) \]

\begin{align*}
proc & : (\text{syntax?} \ (\text{listof (or/c exact-integer? #f)}) \\
& \hspace{1cm} . \rightarrow . (\text{listof export?})) \\
pre-proc & : (\text{syntax?} \ (\text{listof (or/c exact-integer? #f)}) \\
& \hspace{1cm} . \rightarrow . \text{syntax?})
\end{align*}

Creates a provide transformer (i.e., a structure with the \textit{prop:provide-transformer} property) using the given procedure as the transformer. If a \textit{pre-proc} is provided, then the result is also a provide pre-transformer. Often used in combination with \textit{expand-exports} and/or \textit{pre-expand-exports}.

\[ (\text{make-provide-pre-transformer} \ pre-proc) \rightarrow \text{provide-pre-transformer?} \]

\begin{align*}
pre-proc & : (\text{syntax?} \ (\text{listof (or/c exact-integer? #f)}) \\
& \hspace{1cm} . \rightarrow . \text{syntax?})
\end{align*}

Like \textit{make-provide-transformer}, but for a value that is a provide pre-transformer, only. Often used in combination with \textit{pre-expand-exports}.

Examples:

\begin{verbatim}
> (module m racket
  (require
   (for-syntax racket/provide-transform syntax/parser syntax/stx))
\end{verbatim}
(define-syntax wrapped-out
  (make-provide-pre-transformer
    (lambda (stx modes)
      (syntax-parse stx
        [(_ f ...)
         #:with (wrapped-f ...)
         (stx-map
           syntax-local-lift-expression
           #'(lambda args
             (printf "applying ~a, args:
" ~a
" " f args)
             (apply f args)) ...) )
        (pre-expand-export
          #'(rename-out [wrapped-f f] ...) modes)))))

(provide (wrapped-out + -))

> (require 'm)
> (- 1 (+ 2 3))
applying +, args: (2 3)
applying -, args: (1 5)
-4

prop:provide-transformer : struct-type-property?

A property to identify provide transformers. The property value must be a procedure that
takes the structure and returns a transformer procedure; the returned transformer procedure
takes a syntax object and mode list and returns an export list.

prop:provide-pre-transformer : struct-type-property?

A property to identify provide pre-transformers. The property value must be a procedure that
takes the structure and returns a transformer procedure; the returned transformer procedure
takes a syntax object and mode list and returns a syntax object.

(provide-transformer? v) → boolean?
  v : any/c

Returns #t if v has the prop:provide-transformer property, #f otherwise.

(provide-pre-transformer? v) → boolean?
  v : any/c

Returns #t if v has the prop:provide-pre-transformer property, #f otherwise.
A structure representing a single exported identifier:

- `local-id` — the identifier that is bound within the exporting module.
- `out-sym` — the external name of the binding.
- `mode` — the phase level of the binding in the exporting module.
- `protect?` — indicates whether the identifier should be protected (see §14.10 “Code Inspectors”).
- `orig-stx` — a syntax object for the source of the export, used for error reporting.

For backward compatibility only; returns a procedure that returns its first argument.

### 12.4.3 Keyword-Argument Conversion Introspection

```scheme
(require racket/keyword-transform) package:base
```

The bindings documented in this section are provided by the `racket/keyword-transform` library, not `racket/base` or `racket`.

```scheme
(syntax-procedure-alias-property stx)
  → (or/c #f
        (letrec ([val? (recursive-contract
                        (or/c (cons/c identifier? identifier?)
                               (cons/c val? val?))])
             (cons/c val? val?))])
  stx : syntax?
```
Reports the value of a syntax property that can be attached to an identifier by the expansion of a keyword-application form. See lambda for more information about the property.

The property value is normally a pair consisting of the original identifier and an identifier that appears in the expansion. Property-value merging via syntax-track-origin can make the value a pair of such values, and so on.

### 12.5 Syntax Parameters

(require racket/stxparam)  

The bindings documented in this section are provided by the racket/stxparam library, not racket/base or racket.

(define-syntax-parameter id expr)

Binds id as syntax to a syntax parameter. The expr is an expression in the transformer environment that serves as the default value for the syntax parameter. The value is typically obtained by a transformer using syntax-parameter-value.

The id can be used with syntax-parameterize or syntax-parameter-value (in a transformer). If expr produces a procedure of one argument or a make-set!-transformer result, then id can be used as a macro. If expr produces a make-rename-transformer result, then id can be used as a macro that expands to a use of the target identifier, but syntax-local-value of id does not produce the target’s value.

Examples:

> (define-syntax-parameter current-class #f)
> (define-syntax-parameter yield (make-rename-transformer #'abort))
> (define-syntax-parameter define/public
     (λ (stx)
       (raise-syntax-error #f "use of a class keyword not in a class" stx))))
> (begin-for-syntax (displayln (syntax-parameter-value #'current-class)))
(syntax-parameterize ([id expr] ...) body-expr ...)  

Each id must be bound to a syntax parameter using define-syntax-parameter. Each expr is an expression in the transformer environment. During the expansion of the body-exprs, the value of each expr is bound to the corresponding id.

If an expr produces a procedure of one argument or a make-set!-transformer result, then its id can be used as a macro during the expansion of the body-exprs. If expr produces a make-rename-transformer result, then id can be used as a macro that expands to a use of the target identifier, but syntax-local-value of id does not produce the target’s value.

Examples:

> (define-syntax-parameter abort (syntax-rules ()))
> (define-syntax forever
  (syntax-rules ()
    [(forever body ...)
     (call/cc (lambda (abort-k)
                   (syntax-parameterize
                     ([abort (syntax-rules () [(_) (abort-k)])]
                      (let loop () body ... (loop))))))])
> (define-syntax-parameter it (syntax-rules ()))
> (define-syntax aif
  (syntax-rules ()
    [(aif test then else)
     (let ([t test])
      (syntax-parameterize ([it (syntax-id-rules () [_ t])]
                           (if t then else)))]))

(define-rename-transformer-parameter id expr)

Binds id as syntax to a syntax parameter that must be bound to a make-rename-transformer result and, unlike define-syntax-parameter, syntax-local-value of id does produce the target’s value, including inside of syntax-parameterize.

Examples:

> (define-syntax (test stx)
  (syntax-case stx ()
    [(_ t)
```scheme
> (define-syntax one 1)
> (define-syntax two 2)
> (define-syntax-parameter not-num
      (make-rename-transformer #'one))
> (test not-num)
#<procedure:syntax-parameter>
> (define-rename-transformer-parameter num
      (make-rename-transformer #'one))
> (test num)
1
> (syntax-parameterize ([num (make-rename-transformer #'two)])
      (test num))
2
```

Added in version 6.3.0.14 of package base.

### 12.5.1 Syntax Parameter Inspection

```scheme
(require racket/stxparam-exptime) package:base

(syntax-parameter-value id-stx) -> any
  id-stx : syntax?
```

This procedure is intended for use in a transformer environment, where `id-stx` is an identifier bound in the normal environment to a syntax parameter. The result is the current value of the syntax parameter, as adjusted by `syntax-parameterize` form.

This binding is provided for-syntax by `racket/stxparam`, since it is normally used in a transformer. It is provided normally by `racket/stxparam-exptime`.

```scheme
(make-parameter-rename-transformer id-stx) -> any
  id-stx : syntax?
```

This procedure is intended for use in a transformer, where `id-stx` is an identifier bound to a syntax parameter. The result is a transformer that behaves as `id-stx`, but that cannot be used with `syntax-parameterize` or `syntax-parameter-value`.

Using `make-parameter-rename-transformer` is analogous to defining a procedure that calls a parameter. Such a procedure can be exported to others to allow access to the parameter value, but not to change the parameter value. Similarly, `make-parameter-rename-transformer` allows a syntax parameter to be used as a macro, but not changed.

The result of `make-parameter-rename-transformer` is not treated specially by `syntax-local-value`, unlike the result of `make-rename-transformer`.
This binding is provided for-synt__by racket/stxparam, since it is normally used in a transformer. It is provided normally by racket/stxparam-exptime.

12.6 Local Binding with Splicing Body

(require racket/splicing)  package: base

The bindings documented in this section are provided by the racket/splicing library, not racket/base or racket.

splicing-let
splicing-letrec
splicing-let-values
splicing-letrec-values
splicing-let-syntax
splicing-letrec-syntax
splicing-let-syntaxes
splicing-letrec-syntaxes
splicing-letrec-syntaxes+values
splicing-local
splicing-parameterize

Like let, letrec, let-values, letrec-values, let-syntax, letrec-syntax, let-syntaxes, letrec-syntaxes, letrec-syntaxes+values, local, and parameterize, except that in a definition context, the body forms are spliced into the enclosing definition context (in the same way as for begin).

Examples:

> (splicing-let-syntax ([one (lambda (stx) #'1)]))
  (define o one))
> o
1
> one
one: undefined;
  cannot reference an identifier before its definition
  in module: top-level

When a splicing binding form occurs in a top-level context or module context, its local bindings are treated similarly to definitions. In particular, syntax bindings are evaluated every time the module is visited, instead of only once during compilation as in let-syntax, etc.

Example:
If a definition within a splicing form is intended to be local to the splicing body, then the identifier should have a true value for the 'definition-intended-as-local syntax property. For example, splicing-let itself adds the property to locally-bound identifiers as it expands to a sequence of definitions, so that nesting splicing-let within a splicing form works as expected (without any ambiguous bindings).

Changed in version 6.12.0.2 of package base: Added splicing-parameterize.

| splicing-syntax-parameterize |

Like syntax-parameterize, except that in a definition context, the body forms are spliced into the enclosing definition context (in the same way as for begin). In a definition context, the body of splicing-syntax-parameterize can be empty.

Note that require transformers and provide transformers are not affected by syntax parameterization. While all uses of require and provide will be spliced into the enclosing context, derived import or export specifications will expand as if they had not been inside of the splicing-syntax-parameterize.

Additionally, submodules defined with module* that specify #f in place of a module path are affected by syntax parameterization, but other submodules (those defined with module or module* with a module path) are not.

Examples:

```
> (define-syntax-parameter place (lambda (stx) #"Kansas")
> (define-syntax-rule (where) '(at ,(place)))
> (where)
'(at "Kansas")
> (splicing-syntax-parameterize ([place (lambda (stx) #"Oz")])
 (define here (where)))
> here
'(at "Oz")
```

Changed in version 6.11.0.1 of package base: Modified to syntax parameterize module* submodules that specify #f in place of a module path.
12.7 Syntax Object Properties

Every syntax object has an associated syntax property list, which can be queried or extended with syntax-property. A property is set as preserved or not; a preserved property is maintained for a syntax object in a compiled form that is marshaled to a byte string or ".zo" file, and other properties are discarded when marshaling.

In read-syntax, the reader attaches a preserved 'paren-shape property to any pair or vector syntax object generated from parsing a pair [ and ] or { and }; the property value is #\[ in the former case, and #\{ in the latter case. The syntax form copies any 'paren-shape property from the source of a template to corresponding generated syntax.

Both the syntax input to a transformer and the syntax result of a transformer may have associated properties. The two sets of properties are merged by the syntax expander: each property in the original and not present in the result is copied to the result, and the values of properties present in both are combined with cons (result value first, original value second) and the consed value is preserved if either of the values were preserved.

Before performing the merge, however, the syntax expander automatically adds a property to the original syntax object using the key 'origin. If the source syntax has no 'origin property, it is set to the empty list. Then, still before the merge, the identifier that triggered the macro expansion (as syntax) is consed onto the 'origin property so far. The 'origin property thus records (in reverse order) the sequence of macro expansions that produced an expanded expression. Usually, the 'origin value is a list of identifiers, but a transformer might return syntax that has already been expanded, in which case an 'origin list can contain other lists after a merge. The syntax-track-origin procedure implements this tracking. The 'origin property is added as non-preserved.

Besides 'origin tracking for general macro expansion, Racket adds properties to expanded syntax (often using syntax-track-origin) to record additional expansion details:

- When a begin form is spliced into a sequence with internal definitions (see §1.2.3.8 “Internal Definitions”), syntax-track-origin is applied to every spliced element from the begin body. The second argument to syntax-track-origin is the begin form, and the third argument is the begin keyword (extracted from the spliced form).

- When an internal define-values or define-syntaxes form is converted into a letrec-syntaxes+values form (see §1.2.3.8 “Internal Definitions”), syntax-track-origin is applied to each generated binding clause. The second argument to syntax-track-origin is the converted form, and the third argument is the define-values or define-syntaxes keyword form the converted form.

- When a letrec-syntaxes+values expression is fully expanded, syntax bindings disappear, and the result is either a letrec-values form (if the unexpanded form contained non-syntax bindings), or only the body of the letrec-syntaxes+values form (wrapped with begin if the body contained multiple expressions). To record the
disappeared syntax bindings, a property is added to the expansion result: an immutable list of identifiers from the disappeared bindings, as a 'disappeared-binding property.

- When a subtyping struct form is expanded, the identifier used to reference the base type does not appear in the expansion. Therefore, the struct transformer adds the identifier to the expansion result as a 'disappeared-use property.

- When a rename transformer is used to replace a set! target, syntax-track-origin is used on the target identifier (the same as when the identifier is used as an expression).

- When a reference to an unexported or protected identifier from a module is discovered, the 'protected property is added to the identifier with a #t value.

- When read-syntax generates a syntax object, it attaches a property to the object (using a private key) to mark the object as originating from a read. The syntax-original? predicate looks for the property to recognize such syntax objects. (See §12.2 “Syntax Object Content” for more information.)

See also Check Syntax for one client of the 'disappeared-use and 'disappeared-binding properties.

See §12.9.1 “Information on Expanded Modules” for information about properties generated by the expansion of a module declaration. See lambda and §1.2.6 “Inferred Value Names” for information about properties recognized when compiling a procedure. See current-compile for information on properties and byte codes.

```scheme
(syntax-property stx key v [preserved?]) → syntax?
  stx : syntax?
  key : (if preserved? (and/c symbol? symbol-interned?) any/c)
  v : any/c
  preserved? : any/c = (eq? key 'paren-shape)
(syntax-property stx key) → any
  stx : syntax?
  key : any/c
```

The three- or four-argument form extends stx by associating an arbitrary property value v with the key key; the result is a new syntax object with the association (while stx itself is unchanged). The property is added as preserved if preserved? is true, in which case key must be an interned symbol, and v should be a value as described below that can be saved in marshaled bytecode.

The two-argument form returns an arbitrary property value associated to stx with the key key, or #f if no value is associated to stx for key.

To support marshaling to bytecode, a value for a preserved syntax property must be a non-cyclic value that is either
• a pair containing allowed preserved-property values;
• a vector (unmarshaled as immutable) containing allowed preserved-property values;
• a box (unmarshaled as immutable) containing allowed preserved-property values;
• an immutable prefab structure containing allowed preserved-property values;
• an immutable hash table whose keys and values are allowed preserved-property values;
• a syntax object; or
• an empty list, symbol, number, character, string, byte string, or regexp value.

Any other value for a preserved property triggers an exception at an attempt to marshal the owning syntax object to bytecode form.

Changed in version 6.4.0.14 of package base: Added the preserved? argument.

```
(syntax-property-remove stx key) → syntax?
stx : syntax?
key : any/c
```

Returns a syntax object like stx, but without a property (if any) for key.

Added in version 6.90.0.20 of package base.

```
(syntax-property-preserved? stx key) → boolean?
stx : syntax?
key : (and/c symbol? symbol-interned?)
```

Returns #t if stx has a preserved property value for key, #f otherwise.

Added in version 6.4.0.14 of package base.

```
(syntax-property-symbol-keys stx) → list?
stx : syntax?
```

Returns a list of all symbols that as keys have associated properties in stx. Uninterned symbols (see §4.7 "Symbols") are not included in the result list.

```
(syntax-track-origin new-stx
    orig-stx
    id-stx) → any
new-stx : syntax?
orig-stx : syntax?
id-stx : identifier?
```
Adds properties to new-stx in the same way that macro expansion adds properties to a transformer result. In particular, it merges the properties of orig-stx into new-stx, first adding id-stx as an 'origin property, and it returns the property-extended syntax object. Use the syntax-track-origin procedure in a macro transformer that discards syntax (corresponding to orig-stx with a keyword id-stx) leaving some other syntax in its place (corresponding to new-stx).

For example, the expression

\[(or \ x \ y)\]

expands to

\[(let ([or-part x]) (if or-part or-part (or y)))\]

which, in turn, expands to

\[(let-values ([((or-part) x)] (if or-part or-part y))\]

The syntax object for the final expression will have an 'origin property whose value is \((list \ (quote-syntax \ let) \ (quote-syntax \ or))\).

### 12.8 Syntax Taints

The tamper status of a syntax object is either tainted, armed, or clean:

- A tainted identifier is rejected by the macro expander for use as either a binding or expression. If a syntax object is tainted, then any syntax object in the result of \(\text{syntax-e stx}\) is tainted, and \(\text{datum->syntax}\) with \(\text{stx}\) as its first argument produces a tainted syntax object.

  Other derived operations, such as pattern matching in \(\text{syntax-case}\), also taint syntax objects when extracting them from a tainted syntax object.

- An armed syntax object has a set of dye packs, which creates taints if the armed syntax object is used without first disarming the dye packs. In particular, if a syntax object is armed, \(\text{syntax-e, datum->syntax}\), \(\text{quote-syntax}\), and derived operations effectively treat the syntax object as tainted. The macro expander, in contrast, disarms dye packs before pulling apart syntax objects.

  Each dye pack, which is added to a syntax object with the \(\text{syntax-arm}\) function, is keyed by an inspector. A dye pack can be disarmed using \(\text{syntax-disarm}\) with an inspector that is the same as or a superior of the dye pack’s inspector.

- A clean syntax object has no immediate taints or dye packs, although it may contain syntax objects that are tainted or armed.
Taints cannot be removed, and attempting to arm a syntax object that is already tainted has no effect on the resulting syntax object.

The macro expander disarms any syntax object that it encounters in an expression position or as a module body. A syntax object is therefore disarmed when it is provided to a syntax transformer. The transformer’s result, however, is rearmed by copying to it any dye packs that were originally attached to the transformer’s input. The rearming process obeys the following rules:

- If the result has a `taint-mode` property (see §12.7 “Syntax Object Properties”) that is `opaque`, then dye packs are attached to the immediate syntax object.
- If the result has a `taint-mode` property that is `none`, then no dye pack is attached to the syntax object. The `none` mode is rarely appropriate.
- If the result has a `taint-mode` property that is `transparent`, then the dye packs are propagated recursively to syntax object that corresponds to elements of the syntax object’s datum as a list (or, more precisely, to the `car` of the datum as reached by any number of `cdr`s), and the immediate syntax object loses its lexical context; If the immediate syntax object is already armed, then recursive propagation taints the elements. Recursive propagation uses syntax properties and shapes, as for the immediate rearming.
- If the result has a `taint-mode` property that is `transparent-binding`, then dye packs are attached in a way similar to `transparent`, but further treating the syntax object corresponding to the second list element as having a `transparent` value for the `taint-mode` property if it does not already have a `taint-mode` property value.
- If the result has no `taint-mode` property value, but its datum is a pair, and if the syntax object corresponding to the `car` of the pair is an identifier bound to `begin`, `begin-for-syntax`, or `#%plain-module-begin`, then dye packs are propagated as if the syntax object had the `transparent` property value.
- If the result has no `taint-mode` property value, but its datum is a pair, and if the syntax object corresponding to the `car` of the pair is an identifier bound to define-values or define-syntaxes, then dye packs are propagated as if the syntax object had the `transparent-binding` property value.

For backward compatibility, a `certify-mode` property is treated the same as a `taint-mode` property if the former is not attached. To avoid accidental transfer of a `taint-mode` or `certify-mode` property value, the expander always removes any `taint-mode` and `certify-mode` property on a syntax object that is passed to a syntax transformer.

\[(syntax-tainted? stx) \rightarrow boolean?\]

\[stx : syntax?\]

Returns `#t` if `stx` is tainted, `#f` otherwise.
(syntax-arm stx [inspector use-mode?]) → syntax?
stx : syntax?
inspector : (or/c inspector? #f) = #f
use-mode? : any/c = #f

Produces a syntax object like stx, but armed with a dye pack that is keyed by inspector.

A #f value for inspector is equivalent to an inspector that depends on the current dynamic context:

• when a syntax transformer is being applied, the declaration-time code inspector of the module in which a syntax transformer was bound;
• when a module is being visited, the module’s declaration-time code inspector;
• (current-code-inspector), otherwise.

If use-mode? is #f, then if stx is tainted or already armed with the key inspector, the result is stx.

If use-mode? is a true value, then a dye pack is not necessarily added directly to stx. Instead, the dye pack is pushed to interior syntax objects in the same way that the expander pushes armings into a syntax transformer’s results when rearming (based on a 'taint-mode syntax property or identifier bindings); see the expander’s rearming rules for more information. To the degree that pushing dye packs into a syntax object must destructure stx, existing taints or dye packs can lead to tainted results rather than armed results.

(syntax-protect stx) → syntax?
stx : syntax?

Equivalent to (syntax-arm stx #f #t).

(syntax-disarm stx inspector) → syntax?
stx : syntax?
inspector : (or/c inspector? #f)

Produces a disarmed version of stx, removing any immediate dye packs that match inspector. An inspector matches when it is either the same as or a super-inspector of the dye pack’s inspector. A #f value for inspector is replaced by a specific inspector in the same way as for syntax-arm.

(syntax-rearm stx from-stx [use-mode?]) → syntax?
stx : syntax?
from-stx : syntax?
use-mode? : any/c = #f
Produces a rearmed or tainted version of \texttt{stx} by adding all immediate taints and dye packs of \texttt{from-stx}.

If \texttt{use-mode?} is a true value, \texttt{stx} is not necessarily tainted or armed directly. Instead, taints or dye packs are pushed to interior syntax objects in the same way as for \texttt{syntax-arm} or rearming by the expander.

\begin{verbatim}
(syntax-taint stx) → syntax?
  stx : syntax?
\end{verbatim}

Returns tainted version of \texttt{stx}—equivalent to \texttt{(datum->syntax (syntax-arm stx) (syntax-e stx) stx stx)}—or \texttt{stx} if it is already tainted.

### 12.9 Expanding Top-Level Forms

\begin{verbatim}
(expand top-level-form) → syntax?
  top-level-form : any/c
\end{verbatim}

Expands all non-primitive syntax in \texttt{top-level-form}, and returns a syntax object for the expanded form that contains only core forms, matching the grammar specified by [§1.2.3.1 “Fully Expanded Programs”](#).

Before \texttt{top-level-form} is expanded, its lexical context is enriched with \texttt{namespace-syntax-introduce}, just as for \texttt{eval}. Use \texttt{syntax->datum} to convert the returned syntax object into a printable datum.

Here’s an example of using \texttt{expand} on a module:

\begin{verbatim}
(parameterize ([current-namespace (make-base-namespace)])
  (expand (datum->syntax #f '
    '(module foo scheme
      (define a 3)
      (+ a 4))))))
\end{verbatim}

Here’s an example of using \texttt{expand} on a non-top-level form:

\begin{verbatim}
(define-namespace-anchor anchor)
(parameterize ([current-namespace
  (namespace-anchor->namespace anchor)])
  (expand (datum->syntax #f
    '(delay (+ 1 2))))))
\end{verbatim}
Like (expand stx), except that the argument must be a syntax object, and its lexical context is not enriched before expansion.

(\texttt{(expand-once top-level-form)} \rightarrow \texttt{syntax?})

\texttt{top-level-form : any/c}

Partially expands \texttt{top-level-form} and returns a syntax object for the partially-expanded expression. Due to limitations in the expansion mechanism, some context information may be lost. In particular, calling \texttt{expand-once} on the result may produce a result that is different from expansion via \texttt{expand}.

Before \texttt{top-level-form} is expanded, its lexical context is enriched with \texttt{namespace-syntax-introduce}, as for \texttt{eval}.

(\texttt{(expand-syntax-once stx)} \rightarrow \texttt{syntax?})

\texttt{stx : syntax?}

Like (\texttt{expand-once stx}), except that the argument must be a syntax object, and its lexical context is not enriched before expansion.

(\texttt{(expand-to-top-form top-level-form)} \rightarrow \texttt{syntax?})

\texttt{top-level-form : any/c}

Partially expands \texttt{top-level-form} to reveal the outermost syntactic form. This partial expansion is mainly useful for detecting top-level uses of \texttt{begin}. Unlike the result of \texttt{expand-once}, expanding the result of \texttt{expand-to-top-form} with \texttt{expand} produces the same result as using \texttt{expand} on the original syntax.

Before \texttt{stx-or-sexpr} is expanded, its lexical context is enriched with \texttt{namespace-syntax-introduce}, as for \texttt{eval}.

(\texttt{(expand-syntax-to-top-form stx)} \rightarrow \texttt{syntax?})

\texttt{stx : syntax?}

Like (\texttt{expand-to-top-form stx}), except that the argument must be a syntax object, and its lexical context is not enriched before expansion.

\section*{12.9.1 Information on Expanded Modules}

Information for an expanded \texttt{module} declaration is stored in a set of syntax properties (see §12.7 “Syntax Object Properties” attached to the syntax object:
• 'module-variable-provides — a list of provided items, where each item is one of the following:

  - symbol — represents a locally defined variable that is provided with its defined name.

  - (cons provided-sym defined-sym) — represents a locally defined variable that is provided with renaming; the first symbol is the exported name, and the second symbol is the defined name.

  - (list* module-path-index provided-sym defined-sym) — represents a re-exported and possibly re-named variable from the specified module; module-path-index is either a module path index or symbol (see §14.4.2 "Compiled Modules and References"), indicating the source module for the binding. The provided-sym is the external name for the re-export, and defined-sym is the originally defined name in the module specified by module-path-index.

• 'module-syntax-provides — like 'module-variable-provides, but for syntax exports instead of variable exports.

• 'module-indirect-provides — a list of symbols for variables that are defined in the module but not exported; they may be exported indirectly through macro expansions. Definitions of macro-generated identifiers create uninterned symbols in this list. The order of identifiers in the list corresponds to an order for access from bytecode.

• 'module-indirect-for-meta-provides — similar to 'module-indirect-provides: an association list from a phase level to a list of symbols for variables that are defined in the module at phases higher than 0 and not exported.

  Added in version 6.5.0.5 of package base.

• 'module-body-context — a syntax object whose lexical information corresponds to the inside of the module, so it includes the expansion’s outside-edge scope and its inside-edge scope: that is, the syntax object simulates an identifier that is present in the original module body and inaccessible to manipulation by any macro, so that its lexical information includes bindings for the module’s imports and definitions.

  Added in version 6.4.0.1 of package base.

• 'module-body-inside-context — a syntax object whose lexical information corresponds to an identifier that starts with no lexical context and is moved into the macro, so that it includes only the expansions’s inside-edge scope.

  Added in version 6.4.0.1 of package base.

• 'module-body-context-simple? — a boolean, where #t indicates that the bindings of the module’s body (as recorded in the lexical information of the value of the 'module-body-inside-context property) can be directly reconstructed from modules directly imported into the module, including imported for-syntax, for-meta, and for-template.

  Added in version 6.4.0.1 of package base.
12.10 Serializing Syntax

```scheme
(syntax-serialize
 stx
 #:preserve-property-keys preserve-property-keys
 #:provides-namespace provides-namespace
 #:base-module-path-index base-module-path-index)
```

→ any/c

stx : syntax?

preserve-property-keys : (listof symbol?)

provides-namespace : (or/c namespace? #f)

= (current-namespace)

base-module-path-index : (or/c module-path-index? #f) = #f

Converts `stx` to a serialized form that is suitable for use with `sEXP->FASL` or `serialize`. Although `stx` could be serialized with `(compile '(quote-syntax stx))` and then writing the compiled form, `syntax-serialize` provides more control over serialization:

- The `preserve-property-keys` lists syntax-property keys to whose values should be preserved in serialization, even if the property value was not added as preserved with `syntax-property` (so it would be discarded in compiled form). The values associated with the properties to preserve must be serializable in the sense required by `syntax-property` for a preserved property.

- The `provides-namespace` argument constrains how much the serialized syntax object can rely on `bulk bindings`, which are shared binding tables provided by exporting modules. If `provides-namespace` is `#f`, then complete binding information is recorded in the syntax object’s serialized form, and no bulk bindings will be needed from the namespace at deserialization. Otherwise, bulk bindings will be used only for modules declared in `provides-namespace` (i.e., the deserialize-time namespace will have the same module declarations as `provides-namespace`); note that supplying a namespace with no module bindings is equivalent to supplying `#f`.

- The `base-module-path-index` argument specifies a module path index to which binding information in `stx` is relative. For example, if a syntax object originates from `quote-syntax` in the body of a module, then `base-module-path-index` could usefully be the enclosing module’s module path index as produced by `(variable-reference->module-path-index (#%variable-reference))` within the module. On deserialization, a different module path index can be supplied to substitute in place of `base-module-path-index`, which shifts any binding that is relative to the serialize-time module’s identity to be relative to the module identity supplied at deserialize time. If `base-module-path-index` is `#f`, then no shifting is supported at deserialize time, and any `base-module-path-index` supplied at that time is ignored.
A serialized syntax object is otherwise similar to compiled code: it is version-specific, and deserialization will require a sufficiently powerful code inspector.

Added in version 8.0.0.13 of package base.

```
(syntax-deserialize
  v
  [#:base-module-path-index base-module-path-index])
→ syntax?
  v : any/c
  base-module-path-index : (or/c module-path-index? #f) = #f
```

Converts the result of syntax-serialize back to a syntax object. See syntax-serialize for more information.

Added in version 8.0.0.13 of package base.

### 12.11 File Inclusion

```
(require racket/include)  package:base
```

The bindings documented in this section are provided by the racket/include and racket libraries, but not racket/base.

```
.include path-spec
  path-spec = string
    | (file string)
    | (lib string ...+)
```

Inlines the syntax in the file designated by path-spec in place of the include expression.

A path-spec resembles a subset of the mod-path forms for require, but it specifies a file whose content need not be a module. That is, string refers to a file using a platform-independent relative path, (file string) refers to a file using platform-specific notation, and (lib string ...) refers to a file within a collection.

If path-spec specifies a relative path, the path is resolved relative to the source for the include expression, if that source is a complete path string. If the source is not a complete path string, then path-spec is resolved relative to (current-load-relative-directory) if it is not #f, or relative to (current-directory) otherwise.

The included syntax is given the lexical context of the include expression, while the included syntax’s source location refers to its actual source.

```
.include-at/relative-to context source path-spec
```

885
Like `include`, except that the lexical context of `context` is used for the included syntax, and a relative `path-spec` is resolved with respect to the source of `source`. The `context` and `source` elements are otherwise discarded by expansion.

```scheme
(include/reader path-spec reader-expr)
```

Like `include`, except that the procedure produced by the expression `reader-expr` is used to read the included file, instead of `read-syntax`.

The `reader-expr` is evaluated at expansion time in the transformer environment. Since it serves as a replacement for `read-syntax`, the expression’s value should be a procedure that consumes two inputs—a string representing the source and an input port—and produces a syntax object or `eof`. The procedure will be called repeatedly until it produces `eof`.

The syntax objects returned by the procedure should have source location information, but usually no lexical context; any lexical context in the syntax objects will be ignored.

```scheme
(include-at/relative-to/reader context source path-spec reader-expr)
```

Combines `include-at/relative-to` and `include/reader`.

### 12.12 Syntax Utilities

```scheme
(require racket/syntax)
```

The bindings documented in this section are provided by the `racket/syntax` library, not `racket/base` or `racket`.

#### 12.12.1 Creating formatted identifiers

```scheme
(format-id lctx fmt v ... [#:source src #:props props #:cert ignored #:subs? subs? #:subs-intro subs-introducer]) → identifier?
```

- `lctx`: (or/c syntax? #f)
- `fmt`: string?
- `v`: (or/c string? symbol? identifier? keyword? char? number?)
- `src`: (or/c syntax? #f) = #f
- `props`: (or/c syntax? #f) = #f
Like `format`, but produces an identifier using `lctx` for the lexical context, `src` for the source location, and `props` for the properties. An argument supplied with `#:cert` is ignored. (See `datum->syntax`.)

The format string must use only one placeholder. Identifiers in the argument list are automatically converted to symbols.

Examples:

```scheme
(define-syntax (make-pred stx)
  (syntax-case stx ()
    [(make-pred name)
      (format-id #\'name "~a?" (syntax-e #\'name))]))

(make-pred pair)
<procedure:pair?>

(make-pred none-such)
none-such?: undefined;
cannot reference an identifier before its definition in module: top-level

(define-syntax (better-make-pred stx)
  (syntax-case stx ()
    [(better-make-pred name)
      (format-id #\'name #:source #\'name
                   "~a?" (syntax-e #\'name))]))

(better-make-pred none-such)
none-such?: undefined;
cannot reference an identifier before its definition in module: top-level
```

(Scribble doesn’t show it, but the DrRacket pinpoints the location of the second error but not of the first.)

If `subs?` is `#t`, then a `sub-range-binders` syntax property is added to the result that records the position of each identifier in the `vs`. The `subs-intro` procedure is applied to each identifier, and its result is included in the sub-range binder record. This property value overrides a `sub-range-binders` property copied from `props`.

Example:

```scheme
(syntax-property (format-id #\'here "~a/~a-
        ~a" #\'point 2 #\'y #\:subs? #t)
        'sub-range-binders)
```
'(##<syntax point/2-y> 8 1 0.5 0.5 #<syntax:eval:8:0 y> 0 1 0.5 0.5)
  ##(##<syntax point/2-y> 0 5 0.5 0.5 #<syntax:eval:8:0 point> 0 5 0.5 0.5))

Changed in version 7.4.0.5 of package base: Added the #:subs? and #:subs-intro arguments.

(format-symbol fmt v ...) → symbol?
fmt : string?
v : (or/c string? symbol? identifier? keyword? char? number?)

Like format, but produces a symbol. The format string must use only placeholders. Identifiers in the argument list are automatically converted to symbols.

Example:
> (format-symbol "make-~a" 'triple)
'make-triple

12.12.2 Pattern variables

(define/with-syntax pattern stx-expr)

stx-expr : syntax?

Definition form of with-syntax. That is, it matches the syntax object result of stx-expr against pattern and creates pattern variable definitions for the pattern variables of pattern.

Examples:
> (define/with-syntax (px ...) '#(a b c))
> (define/with-syntax (tmp ...) (generate-temporaries '#(px ...)))
> '#([[tmp px] ...])
#<syntax:eval:12:0 (((a9 a) (b10 b) (c11 c)))>
> (define/with-syntax name #'Alice)
> #'(hello name)
#<syntax:eval:14:0 (hello Alice)>

12.12.3 Error reporting

(current-syntax-context) → (or/c syntax? false/c)
(current-syntax-context stx) → void?
stx : (or/c syntax? false/c)
The current contextual syntax object, defaulting to #f. It determines the special form name that prefixes syntax errors created by wrong-syntax.

\[\text{(wrong-syntax stx format-string v ...)} \rightarrow \text{any}\]
\[\text{stx : syntax?}\]
\[\text{format-string : string?}\]
\[\text{v : any/c}\]

 Raises a syntax error using the result of (current-syntax-context) as the “major” syntax object and the provided stx as the specific syntax object. (The latter, stx, is usually the one highlighted by DrRacket.) The error message is constructed using the format string and arguments, and it is prefixed with the special form name as described under current-syntax-context.

Examples:

\[> (\text{wrong-syntax #'here "expected ~s" 'there})\]
\[eval:15:0: ?: expected there\]
\[at: here\]

\[> (\text{parameterize ([(current-syntax-context #'(look over here)])}}\]
\[\text{(wrong-syntax #'here "expected ~s" 'there})\]
\[eval:16:0: look: expected there\]
\[at: here\]
\[in: (look over here)\]

A macro using wrong-syntax might set the syntax context at the very beginning of its transformation as follows:

\[(\text{define-syntax (my-macro stx)}\]
\[\text{(parameterize ([current-syntax-context #'(look over here)])}}\]
\[\text{(syntax-case stx () \)}\]
\[\text{____)})\]

Then any calls to wrong-syntax during the macro’s transformation will refer to my-macro (more precisely, the name that referred to my-macro where the macro was used, which may be different due to renaming, prefixing, etc).

12.12.4 Recording disappeared uses

\[(\text{current-recorded-disappeared-uses)}\]
\[\rightarrow \text{(or/c (listof identifier?) false/c)}\]
\[\text{(current-recorded-disappeared-uses ids)} \rightarrow \text{void?}\]
\[\text{ids : (or/c (listof identifier?) false/c)}\]

889
Parameter for tracking disappeared uses. Tracking is “enabled” when the parameter has a non-false value. This is done automatically by forms like with-disappeared-uses.

\[
\text{(with-disappeared-uses body-expr ... stx-expr)}
\]

\[stx-expr: \text{syntax?}\]

Evaluates the body-exprs and stx-expr, catching identifiers looked up using syntax-local-value/record. Adds the caught identifiers to the \disappeared-use syntax property of the syntax object produced by stx-expr.

Changed in version 6.5.0.7 of package base: Added the option to include body-exprs.

\[
\text{(syntax-local-value/record id predicate)} \rightarrow \text{any/c}
\]

\[id: \text{identifier?}
\]

\[predicate: (\rightarrow \text{any/c boolean?})\]

Looks up id in the syntactic environment (as syntax-local-value). If the lookup succeeds and returns a value satisfying the predicate, the value is returned and id is recorded as a disappeared use by calling record-disappeared-uses. If the lookup fails or if the value does not satisfy the predicate, #f is returned and the identifier is not recorded as a disappeared use.

\[
\text{(record-disappeared-uses id [intro?])} \rightarrow \text{void?}
\]

\[id: (\text{or/c identifier? (listof identifier?)})
\]

\[intro?: \text{boolean?} = (\text{syntax-transforming?})\]

Add id to (current-recorded-disappeared-uses). If id is a list, perform the same operation on all the identifiers. If intro? is true, then syntax-local-introduce is first called on the identifiers.

If not used within the extent of a with-disappeared-uses form or similar, has no effect.

Changed in version 6.5.0.7 of package base: Added the option to pass a single identifier instead of requiring a list. Changed in version 7.2.0.11: Added the intro? argument.

### 12.12.5 Miscellaneous utilities

\[
\text{(generate-temporary [name-base])} \rightarrow \text{identifier?}
\]

\[name-base: \text{any/c} = 'g\]

Generates one fresh identifier. Singular form of generate-temporaries. If name-base is supplied, it is used as the basis for the identifier’s name.
(internal-definition-context-apply intdef-ctx stx) → syntax?

intdef-ctx : internal-definition-context?
stx : syntax?

Equivalent to (internal-definition-context-introduce intdef-ctx stx 'add). The internal-definition-context-apply function is provided for backwards compatibility; the more general internal-definition-context-introduce function is preferred.

(syntax-local-eval stx [intdef-ctx]) → any
stx : syntax?
intdef-ctx : (or/c internal-definition-context? (listof internal-definition-context?) #f) = '()

Evaluates stx as an expression in the current transformer environment (that is, at phase level 1). If intdef-ctx is not #f, the value provided for intdef-ctx is used to enrich stx's lexical information and extend the local binding context in the same way as the fourth argument to local-expand.

Examples:

> (define-syntax (show-me stx)
  (syntax-case stx ()
    [(show-me expr)
      (begin
        (printf "at compile time produces ~s\n" (syntax-local-eval #'expr))
        #'(printf "at run time produces ~s\n" expr))]))

> (show-me (+ 2 5))
at compile time produces 7
at run time produces 7
> (define-for-syntax fruit 'apple)
> (define fruit 'pear)
> (show-me fruit)
at compile time produces apple
at run time produces pear

Changed in version 6.90.0.27 of package base: Changed intdef-ctx to accept a list of internal-definition contexts in addition to a single internal-definition context or #f.

(with-syntax* ([pattern stx-expr] ...) body ...*)
**stx-expr**: `syntax?`

Similar to `with-syntax`, but the pattern variables of each `pattern` are bound in the `stx-expr`s of subsequent clauses as well as the `body`s, and the `patterns` need not bind distinct pattern variables; later bindings shadow earlier bindings.

Example:

```
> (with-syntax* ([[(x y) (list #'val1 #'val2)]
                [nest #'((x) (y))]])
  #'nest)
#<syntax:eval:22:0 ((val1) (val2))>
```
13 Input and Output

13.1 Ports

Ports produce and/or consume bytes. An input port produces bytes, while an output port consumes bytes (and some ports are both input ports and output ports). When an input port is provided to a character-based operation, the bytes are decoded to a character, and character-based output operations similarly encode the character to bytes; see §13.1.1 “Encodings and Locales”. In addition to bytes and characters encoded as bytes, some ports can produce and/or consume arbitrary values as special results.

When a port corresponds to a file, network connection, or some other system resource, it must be explicitly closed via close-input-port or close-output-port (or indirectly via custodian-shutdown-all) to release low-level resources associated with the port. For any kind of port, after it is closed, attempting to read from or write to the port raises exn:fail.

Data produced by a input port can be read or peeked. When data is read, it is considered consumed and removed from the port’s stream. When data is peeked, it remains in the port’s stream to be returned again by the next read or peek. Previously peeked data can be committed, which causes the data to be removed from the port as for a read in a way that can be synchronized with other attempts to peek or read through a synchronizable event. Both read and peek operations are normally blocking, in the sense that the read or peek operation does not complete until data is available from the port; non-blocking variants of read and peek operations are also available.

The global variable eof is bound to the end-of-file value, and eof-object? returns #t only when applied to this value. Reading from a port produces an end-of-file result when the port has no more data, but some ports may also return end-of-file mid-stream. For example, a port connected to a Unix terminal returns an end-of-file when the user types control-D; if the user provides more input, the port returns additional bytes after the end-of-file.

Every port has a name, as reported by object-name. The name can be any value, and it is used mostly for error-reporting purposes. The read-syntax procedure uses the name of an input port as the default source location for the syntax objects that it produces.

A port can be used as a synchronizable event. An input port is ready for synchronization when read-byte would not block, and an output port is ready for synchronization when write-bytes-avail would not block or when the port contains buffered characters and write-bytes-avail* can flush part of the buffer (although write-bytes-avail might block). A value that can act as both an input port and an output port acts as an input port for a synchronizable event. The synchronization result of a port is the port itself.
13.1.1 Encodings and Locales

When a port is provided to a character-based operation, such as `read-char` or `read`, the port’s bytes are read and interpreted as a UTF-8 encoding of characters. Thus, reading a single character may require reading multiple bytes, and a procedure like `char-ready?` may need to peek several bytes into the stream to determine whether a character is available. In the case of a byte stream that does not correspond to a valid UTF-8 encoding, functions such as `read-char` may need to peek one byte ahead in the stream to discover that the stream is not a valid encoding.

When an input port produces a sequence of bytes that is not a valid UTF-8 encoding in a character-reading context, then bytes that constitute an invalid sequence are converted to the character `#�`. Specifically, bytes 255 and 254 are always converted to `#�`, bytes in the range 192 to 253 produce `#�` when they are not followed by bytes that form a valid UTF-8 encoding, and bytes in the range 128 to 191 are converted to `#�` when they are not part of a valid encoding that was started by a preceding byte in the range 192 to 253. To put it another way, when reading a sequence of bytes as characters, a minimal set of bytes are changed to the encoding of `#�` so that the entire sequence of bytes is a valid UTF-8 encoding.

See §4.5 “Byte Strings” for procedures that facilitate conversions using UTF-8 or other encodings. See also `reencode-input-port` and `reencode-output-port` for obtaining a UTF-8-based port from one that uses a different encoding of characters.

A locale captures information about a user’s language-specific interpretation of character sequences. In particular, a locale determines how strings are “alphabetized,” how a lowercase character is converted to an uppercase character, and how strings are compared without regard to case. String operations such as `string-ci=?` are not sensitive to the current locale, but operations such as `string-locale-ci=?` (see §4.4 “Strings”) produce results consistent with the current locale.

A locale also designates a particular encoding of code-point sequences into byte sequences. Racket generally ignores this aspect of the locale, with a few notable exceptions: command-line arguments passed to Racket as byte strings are converted to character strings using the locale’s encoding; command-line strings passed as byte strings to other processes (through `subprocess`) are converted to byte strings using the locale’s encoding; environment variables are converted to and from strings using the locale’s encoding; filesystem paths are converted to and from strings (for display purposes) using the locale’s encoding; and, finally, Racket provides functions such as `string->bytes/locale` to specifically invoke a locale-specific encoding.

A Unix user selects a locale by setting environment variables, such as `LC_ALL`. On Windows and Mac OS, the operating system provides other mechanisms for setting the locale. Within Racket, the current locale can be changed by setting the `current-locale` parameter. The locale name within Racket is a string, and the available locale names depend on the platform and its configuration, but the “” locale means the current user’s default locale;
on Windows and Mac OS, the encoding for "" is always UTF-8, and locale-sensitive operations use the operating system's native interface. (In particular, setting the LC_ALL and LC_CTYPE environment variables does not affect the locale "" on Mac OS. Use getenv and current-locale to explicitly install the environment-specified locale, if desired.) Setting the current locale to #f makes locale-sensitive operations locale-insensitive, which means using the Unicode mapping for case operations and using UTF-8 for encoding.

```
(current-locale) → (or/c string? #f)
(current-locale locale) → void?
 locale : (or/c string? #f)
```

A parameter that determines the current locale for procedures such as string-locale-ci=?.

When locale sensitivity is disabled by setting the parameter to #f, strings are compared, etc., in a fully portable manner, which is the same as the standard procedures. Otherwise, strings are interpreted according to a locale setting (in the sense of the C library's setlocale). The "" locale is always an alias for the current machine's default locale, and it is the default. The "C" locale is also always available; setting the locale to "C" is the same as disabling locale sensitivity with #f only when string operations are restricted to the first 128 characters. Other locale names are platform-specific.

String or character printing with write is not affected by the parameter, and neither are symbol case or regular expressions (see §4.8 "Regular Expressions").

### 13.1.2 Managing Ports

```
(input-port? v) → boolean?
 v : any/c
```

Returns #t if v is an input port, #f otherwise.

```
(output-port? v) → boolean?
 v : any/c
```

Returns #t if v is an output port, #f otherwise.

```
(port? v) → boolean?
 v : any/c
```

Returns #t if either (input-port? v) or (output-port? v) is #t, #f otherwise.

```
(close-input-port in) → void?
 in : input-port?
```
Closes the input port \textit{in}. For some kinds of ports, closing the port releases lower-level resources, such as a file handle. If the port is already closed, \texttt{close-input-port} has no effect.

\begin{verbatim}
(close-output-port out) \rightarrow \texttt{void}
out : \texttt{output-port}
\end{verbatim}

Closes the output port \textit{out}. For some kinds of ports, closing the port releases lower-level resources, such as a file handle. Also, if the port is buffered, closing may first flush the port before closing it, and this flushing process can block. If the port is already closed, \texttt{close-output-port} has no effect.

\begin{verbatim}
(port-closed? port) \rightarrow \texttt{boolean}
port : \texttt{port}
\end{verbatim}

Returns \texttt{#t} if the input or output port \textit{port} is closed, \texttt{#f} otherwise.

\begin{verbatim}
(port-closed-evt port) \rightarrow \texttt{evt}
port : \texttt{port}
\end{verbatim}

Return a synchronizable event that becomes ready for synchronization when \textit{port} is closed. The synchronization result of a port-closed event is the port-closed event itself.

\begin{verbatim}
(current-input-port) \rightarrow \texttt{input-port}
(current-input-port in) \rightarrow \texttt{void}
in : \texttt{input-port}
\end{verbatim}

A parameter that determines a default input port for many operations, such as \texttt{read}.

\begin{verbatim}
(current-output-port) \rightarrow \texttt{output-port}
(current-output-port out) \rightarrow \texttt{void}
out : \texttt{output-port}
\end{verbatim}

A parameter that determines a default output port for many operations, such as \texttt{write}.

\begin{verbatim}
(current-error-port) \rightarrow \texttt{output-port}
(current-error-port out) \rightarrow \texttt{void}
out : \texttt{output-port}
\end{verbatim}

A parameter that determines an output port that is typically used for errors and logging. For example, the default error display handler writes to this port.

\begin{verbatim}
(file-stream-port? v) \rightarrow \texttt{boolean}
v : \texttt{any/c}
\end{verbatim}
Returns #t if \( v \) is a file-stream port (see §13.1.5 “File Ports”), #f otherwise.

Changed in version 7.2.0.5 of package base: Extended file-stream-port? to any value, instead of restricting the domain to ports

\[
(\text{file-stream-port? } v) \to \text{boolean}?
\]
\[
v : \text{any/c}
\]

Returns #t if \( v \) is a port that is attached to an interactive terminal, #f otherwise.

Changed in version 7.2.0.5 of package base: Extended terminal-port? to any value, instead of restricting the domain to ports

\[
(\text{terminal-port? } v) \to \text{boolean}?
\]
\[
v : \text{any/c}
\]

Returns #t if \( port \) is not ready for reading or writing because it is waiting for a peer process to complete a stream construction, #f otherwise.

On Unix and Mac OS, opening a fifo for output creates a peer-waiting port if no reader for the same fifo is already opened. In that case, the output port is not ready for writing until a reader is opened; that is, write operations will block. Use sync if necessary to wait until writing will not block—that is, until the read end of the fifo is opened.

Added in version 7.4.0.5 of package base.

\[
\text{eof} : \text{eof-object}?
\]

A value (distinct from all other values) that represents an end-of-file.

\[
(\text{eof-object? } v) \to \text{boolean}?
\]
\[
v : \text{any/c}
\]

Returns #t if \( v \) is eof, #f otherwise.

### 13.1.3 Port Buffers and Positions

Some ports—especially those that read from and write to files—are internally buffered:

- An input port is typically block-buffered by default, which means that on any read, the buffer is filled with immediately-available bytes to speed up future reads. Thus, if a file is modified between a pair of reads to the file, the second read can produce stale data. Calling file-position to set an input port’s file position flushes its buffer.
• An output port is typically block-buffered by default, though a terminal output port is line-buffered, and the initial error output port is unbuffered. An output buffer is filled with a sequence of written bytes to be committed as a group, either when the buffer is full (in block mode), when a newline is written (in line mode), when the port is closed via close-output-port, or when a flush is explicitly requested via a procedure like flush-output.

If a port supports buffering, its buffer mode can be changed via file-stream-buffer-mode (even if the port is not a file-stream port).

For an input port, peeking always places peeked bytes into the port’s buffer, even when the port’s buffer mode is 'none; furthermore, on some platforms, testing the port for input (via char-ready? or sync) may be implemented with a peek. If an input port’s buffer mode is 'none, then at most one byte is read for read-bytes-avail!, read-bytes-avail!, peek-bytes-avail!* or peek-bytes-avail!: if any bytes are buffered in the port (e.g., to satisfy a previous peek), the procedures may access multiple buffered bytes, but no further bytes are read.

In addition, the initial current output and error ports are automatically flushed when they are terminal ports (see terminal-port?) and when read, read-line, read-bytes, read-string, etc., are performed on the initial standard input port. (More precisely, instead of read, flushing is performed by the default port read handler; see port-read-handler.)

\[
(flush-output \ [out]) \to \text{void}
\]
\[
out : \text{output-port?} = (current-output-port)
\]

Forces all buffered data in the given output port to be physically written. Only file-stream ports, TCP ports, and custom ports (see §13.1.9 “Custom Ports”) use buffers; when called on a port without a buffer, flush-output has no effect.

If flushing a file-stream port or TCP port encounters an error when writing, then all buffered bytes in the port are discarded. Consequently, a further attempt to flush or close the port will not fail.

Changed in version 7.4.0.10 of package base: Consistently, discard buffered bytes on error, including in a TCP output port.

\[
(file-stream-buffer-mode \ port) \to \text{or/c 'none 'line 'block #f)}
\]
\[
port : \text{port?}
\]
\[
(file-stream-buffer-mode \ port \ mode) \to \text{void}
\]
\[
port : \text{port?}
\]
\[
mode : \text{or/c 'none 'line 'block}
\]

Gets or sets the buffer mode for port, if possible. File-stream ports support setting the buffer mode, TCP ports (see §15.3 “Networking”) support setting and getting the buffer mode, and custom ports (see §13.1.9 “Custom Ports”) may support getting and setting buffer modes.
If `mode` is provided, it must be one of `'none`, `'line` (output only), or `'block`, and the port’s buffering is set accordingly. If the port does not support setting the mode, the `exn:fail` exception is raised.

If `mode` is not provided, the current mode is returned, or `#f` is returned if the mode cannot be determined. If `port` is an input port and `mode` is `'line`, the `exn:fail:contract` exception is raised.

```
(file-position port) → exact-nonnegative-integer?
  port : port?
(file-position port pos) → void?
  port : port?
  pos : (or/c exact-nonnegative-integer? eof-object?)
```

Returns or sets the current read/write position of `port`.

Calling `file-position` without a position on a port other than a file-stream port or string port returns the number of bytes that have been read from that port if the position is known (see §13.1.4 “Counting Positions, Lines, and Columns”), otherwise the `exn:fail:filesystem` exception is raised.

For file-stream ports and string ports, the position-setting variant sets the read/write position to `pos` relative to the beginning of the file or (byte) string if `pos` is a number, or to the current end of the file or (byte) string if `pos` is `eof`. In position-setting mode, `file-position` raises the `exn:fail:contract` exception for port kinds other than file-stream ports and string ports. Furthermore, not all file-stream ports support setting the position; if `file-position` is called with a position argument on such a file-stream port, the `exn:fail:filesystem` exception is raised.

When `file-position` sets the position `pos` beyond the current size of an output file or (byte) string, the file/string is enlarged to size `pos` and the new region is filled with 0 bytes; in the case of a file. In the case of a file output port, the file might not be enlarged until more data is written to the file; in that case, beware that writing to a file opened in `'append` mode on Unix and Mac OS will reset the file pointer to the end of a file before each write, which defeats file enlargement via `file-position`. If `pos` is beyond the end of an input file or (byte) string, then reading thereafter returns `eof` without changing the port’s position.

When changing the file position for an output port, the port is first flushed if its buffer is not empty. Similarly, setting the position for an input port clears the port’s buffer (even if the new position is the same as the old position). However, although input and output ports produced by `open-input-output-file` share the file position, setting the position via one port does not flush the other port’s buffer.

```
(file-position* port) → (or/c exact-nonnegative-integer? #f)
  port : port?
```

Like `file-position` on a single argument, but returns `#f` if the position is not known.
(file-truncate port size) → void?
  port : (and/c output-port? file-stream-port?)
  size : exact-nonnegative-integer?

Sets the size of the file written by port to size, assuming that the port is associated to a file whose size can be set.

The new file size can be either larger or smaller than its current size, but “truncate” in this function’s name reflects that it is normally used to decrease the size of a file, since writing to a file or using file-position can extend a file’s size.

13.1.4 Counting Positions, Lines, and Columns

By default, Racket keeps track of the position in a port as the number of bytes that have been read from or written to any port (independent of the read/write position, which is accessed or changed with file-position). Optionally, however, Racket can track the position in terms of characters (after UTF-8 decoding), instead of bytes, and it can track line locations and column locations; this optional tracking must be specifically enabled for a port via port-count-lines! or the port-count-lines-enabled parameter. Position, line, and column locations for a port are used by read-syntax. Position and line locations are numbered from 1; column locations are numbered from 0.

When counting lines, Racket treats linefeed, return, and return-linefeed combinations as a line terminator and as a single position (on all platforms). Each tab advances the column count to one before the next multiple of 8. When a sequence of bytes in the range 128 to 253 forms a UTF-8 encoding of a character, the position/column is incremented once for each byte, and then decremented appropriately when a complete encoding sequence is discovered. See also §13.1 “Ports” for more information on UTF-8 decoding for ports.

A position is known for any port as long as its value can be expressed as a fixnum (which is more than enough tracking for realistic applications in, say, syntax-error reporting). If the position for a port exceeds the value of the largest fixnum, then the position for the port becomes unknown, and line and column tacking is disabled. Return-linefeed combinations are treated as a single character position only when line and column counting is enabled.

Custom ports can define their own counting functions, which are not subject to the rules above, except that the counting functions are invoked only when tracking is specifically enabled with port-count-lines!.

(port-count-lines! port) → void?
  port : port?

Turns on line location and column location counting for a port. Counting can be turned on at any time, though generally it is turned on before any data is read from or written to a port. At
the point that line counting is turned on, \texttt{port\-next\-location} typically starts reporting as
its last result (one more than) the number of characters read since line counting was enabled,
instead of (one more than) bytes read since the port was opened.

When a port is created, if the value of the \texttt{port\-count\-lines\-enabled} parameter is true,
then line counting is automatically enabled for the port. Line counting cannot be disabled
for a port after it is enabled.

\begin{verbatim}
(port\-counts\-lines? port) -> boolean?
  port : port?
\end{verbatim}

Returns \#t if line location and column location counting has been enabled for \texttt{port}, \#f
otherwise.

\begin{verbatim}
(port\-next\-location port)
  -> (or/c exact\-positive\-integer? \#f)
      (or/c exact\-nonnegative\-integer? \#f)
      (or/c exact\-positive\-integer? \#f)
  port : port?
\end{verbatim}

Returns three values: an integer or \#f for the line number of the next read/written item, an
integer or \#f for the next item’s column, and an integer or \#f for the next item’s position.
The next column and position normally increase as bytes are read from or written to the
port, but if line/character counting is enabled for \texttt{port}, the column and position results can
decrease after reading or writing a byte that ends a UTF-8 encoding sequence.

If line counting is not enabled for a port, than the first two results are \#f, and the last result is
one more than the number of bytes read so far. At the point when line counting is enabled, the
first two results typically become non-\#f, and last result starts reporting characters instead
of bytes, typically starting from the point when line counting is enabled.

Even with line counting enabled, a port may return \#f values if it somehow cannot keep
track of lines, columns, or positions.

\begin{verbatim}
(set\-port\-next\-location! port
  line
  column
  position) -> void?
  port : port?
  line : (or/c exact\-positive\-integer? \#f)
  column : (or/c exact\-nonnegative\-integer? \#f)
  position : (or/c exact\-positive\-integer? \#f)
\end{verbatim}

Sets the next line, column, and position for \texttt{port}. If line counting has not been enabled for
\texttt{port} or if \texttt{port} is a custom port that defines its own counting function, then \texttt{set\-port\-next\-location!} has no effect.
A parameter that determines whether line counting is enabled automatically for newly created ports. The default value is \( \#f \).

13.1.5 File Ports

A port created by `open-input-file`, `open-output-file`, `subprocess`, and related functions is a file-stream port. The initial input, output, and error ports in racket are also file-stream ports. The `file-stream-port?` predicate recognizes file-stream ports.

When an input or output file-stream port is created, it is placed into the management of the current custodian (see §14.7 “Custodians”). In the case of an output port, a flush callback is registered with the current plumber to flush the port.

```
(open-input-file path#
[:mode mode-flag
 #:for-module? for-module?]) → input-port?

path : path-string?
mode-flag : (or/c 'binary 'text) = 'binary
for-module? : any/c = #f
```

Opens the file specified by `path` for input. The `mode-flag` argument specifies how the file’s bytes are translated on input:

- `'binary` — bytes are returned from the port exactly as they are read from the file.
- `'text` — return and linefeed bytes (10 and 13) as read from the file are filtered by the port in a platform specific manner:
  - Unix and Mac OS: no filtering occurs.
  - Windows: a return-linefeed combination from a file is returned by the port as a single linefeed; no filtering occurs for return bytes that are not followed by a linefeed, or for a linefeed that is not preceded by a return.

On Windows, `'text` mode works only with regular files; attempting to use `'text` with other kinds of files triggers an `exn:fail:filesystem` exception.

Otherwise, the file specified by `path` need not be a regular file. It might be a device that is connected through the filesystem, such as "aux" on Windows or "/dev/null" on Unix. In all cases, the port is buffered by default.
The port produced by `open-input-file` should be explicitly closed, either through `close-input-port` or indirectly via `custodian-shutdown-all`, to release the OS-level file handle. The input port will not be closed automatically if it is otherwise available for garbage collection (see §1.1.6 “Garbage Collection”); a `will` could be associated with an input port to close it more automatically (see §16.3 “Wills and Executors”).

A path value that is the cleansed version of `path` is used as the name of the opened port.

If opening the file fails due to an error in the filesystem, then `exn:fail:filesystem:errno` exception is raised—as long as `for-module?` is `#f`, `current-module-path-for-load` has a non-`#f` value, or the filesystem error is not recognized as a file-not-found error. Otherwise, when `for-module?` is true, `current-module-path-for-load` has a non-`#f` value, and the filesystem error is recognized as a file-not-found error, then the raised exception is either `exn:fail:syntax:missing-module` (if the value of `current-module-path-for-load` is a syntax object) or `exn:fail:filesystem:missing-module` (otherwise).

Changed in version 6.0.1.6 of package base: Added `#:for-module?`.

Examples:

```scheme
> (with-output-to-file some-file
   (lambda () (printf "hello world")))
> (define in (open-input-file some-file))
> (read-string 11 in)
"hello world"
> (close-input-port in)

(open-output-file path
  [:mode mode-flag
   #:exists exists-flag
   #:permissions permissions]) → output-port?

path : path-string?
mode-flag : (or/c 'binary 'text) = 'binary
exists-flag : (or/c 'error 'append 'update 'can-update
              'replace 'truncate
              'must-truncate 'truncate/replace)
            = 'error
permissions : (integer-in 0 65535) = #o666
```

Opens the file specified by `path` for output. The `mode-flag` argument specifies how bytes written to the port are translated when written to the file:

- `binary` — bytes are written to the file exactly as written to the port.
- `text` — on Windows, a linefeed byte (10) written to the port is translated to a return-linefeed combination in the file; no filtering occurs for returns.
On Windows, `text` mode works only with regular files; attempting to use `text` with other kinds of files triggers an `exn:fail:filesystem` exception.

The `exists-flag` argument specifies how to handle/require files that already exist:

- `'error` — raise `exn:fail:filesystem` if the file exists.
- `'replace` — remove the old file, if it exists, and write a new one.
- `'truncate` — remove all old data, if the file exists.
- `'must-truncate` — remove all old data in an existing file; if the file does not exist, the `exn:fail:filesystem` exception is raised.
- `'truncate/replace` — try `'truncate`; if it fails (perhaps due to file permissions), try `'replace`.
- `'update` — open an existing file without truncating it; if the file does not exist, the `exn:fail:filesystem` exception is raised. Use `file-position` to change the current read/write position.
- `'can-update` — open an existing file without truncating it, or create the file if it does not exist.
- `'append` — append to the end of the file, whether it already exists or not; on Windows, `'append` is equivalent to `'update`, except that the file is not required to exist, and the file position is immediately set to the end of the file after opening it.

When the file specified by `path` is created, `permissions` specifies the permissions of the created file, where an integer representation of permissions is treated the same as for `file-or-directory-permissions`. On Unix and Mac OS, these permissions bits are combined with the process’s umask. On Windows, the only relevant property of `permissions` is whether it has the `#o2` bit set for write permission. Note that a read-only file can be created with `open-output-file`, in which case writing is prohibited only for later attempts to open the file.

The file specified by `path` need not be a regular file. It might be a device that is connected through the filesystem, such as "aux" on Windows or "/dev/null" on Unix. The output port is block-buffered by default, unless the file corresponds to a terminal, in which case it is line-buffered by default. On Unix and Mac OS, if the file is a fifo, then the port will block for writing until a reader for the fifo is available; see also `port-waiting-peer?`.

The port produced by `open-output-file` should be explicitly closed, either though `close-output-port` or indirectly via `custodian-shutdown-all`, to release the OS-level file handle. The output port will not be closed automatically if it is otherwise available for garbage collection (see §1.1.6 "Garbage Collection"); a will could be associated with an output port to close it more automatically (see §16.3 "Wills and Executors").
A path value that is the cleansed version of path is used as the name of the opened port.

If opening the file fails due to an error in the underlying filesystem then exn:fail:filesystem:errno exception is raised.

Examples:

```scheme
> (define out (open-output-file some-file))
> (write "hello world" out)
> (close-output-port out)
```

Changed in version 6.9.0.6 of package base: On Unix and Mac OS, make 'truncate/replace replace on a permission error. On Windows, make 'replace always replace instead truncating like 'truncate/replace.

Changed in version 7.4.0.5: Changed handling of a fifo on Unix and Mac OS to make the port block for output until the fifo has a reader.

Changed in version 8.1.0.3: Added the #:permissions argument.

```scheme
(call-with-input-file path
  proc
  [#:exists exists-flag
    #:permissions permissions])
```

Like open-output-file, but producing two values: an input port and an output port. The two ports are connected in that they share the underlying file descriptor. This procedure is intended for use with special devices that can be opened by only one process, such as "COM1" in Windows. For regular files, sharing the file descriptor can be confusing. For example, using one port does not automatically flush the other port’s buffer, and reading or writing in one port moves the file position (if any) for the other port. For regular files, use separate open-input-file and open-output-file calls to avoid confusion.

```scheme
(call-with-input-file path
  proc
  [#:mode mode-flag]) → any
```

Calls open-input-file with the path and mode-flag arguments, and passes the resulting port to proc. The result of proc is the result of the call-with-input-file call, but the newly opened port is closed when proc returns.
Examples:
> (with-output-to-file some-file
   (lambda () (printf "text in a file")))
> (call-with-input-file some-file
   (lambda (in) (read-string 14 in)))
"text in a file"

(call-with-output-file path proc
 [#:mode mode-flag
 #:exists exists-flag
 #:permissions permissions]) → any

path : path-string?
proc : (output-port? . -> . any)
mode-flag : (or/c 'binary 'text) = 'binary
exists-flag : (or/c 'error 'append 'update 'replace 'truncate 'truncate/replace)
             = 'error
permissions : (integer-in 0 65535) = #o666

Analogous to call-with-input-file, but passing path, mode-flag, exists-flag, and permissions to open-output-file.

Examples:
> (call-with-output-file some-file
   (lambda (out)
     (write 'hello out)))
> (call-with-input-file some-file
   (lambda (in)
     (read-string 5 in)))
"hello"

Changed in version 8.1.0.3 of package base: Added the #:permissions argument.

(call-with-input-file* path proc
 [#:mode mode-flag]) → any

path : path-string?
proc : (input-port? . -> . any)
mode-flag : (or/c 'binary 'text) = 'binary

Like call-with-input-file, but the newly opened port is closed whenever control escapes the dynamic extent of the call-with-input-file* call, whether through proc's return, a continuation application, or a prompt-based abort.
Like `call-with-output-file`, but the newly opened port is closed whenever control escapes the dynamic extent of the `call-with-output-file*` call, whether through `proc`'s return, a continuation application, or a prompt-based abort.

Changed in version 8.1.0.3 of package base: Added the `#:permissions` argument.

Like `call-with-input-file*`, but instead of passing the newly opened port to the given procedure argument, the port is installed as the current input port (see `current-input-port`) using `parameterize` around the call to `thunk`.

Examples:

```
> (with-output-to-file some-file
  (lambda () (printf "hello")))
"hello"
```

```
> (with-input-from-file some-file
  (lambda () (read-string 5)))
"hello"
```

```
(with-output-to-file path
  thunk
  [#:mode mode-flag
   #:exists exists-flag
   #:permissions permissions]) → any

path : path-string?
thunk : (-> any)
mode-flag : (or/c 'binary 'text) = 'binary
exists-flag : (or/c 'error 'append 'update
  'replace 'truncate 'truncate/replace)
  = 'error
permissions : (integer-in 0 65535) = #o666
```
mode-flag : (or/c 'binary 'text) = 'binary
exists-flag : (or/c 'error 'append 'update
             'replace 'truncate 'truncate/replace) = 'error
permissions : (integer-in 0 65535) = #o666

Like call-with-output-file*, but instead of passing the newly opened port to the given
procedure argument, the port is installed as the current output port (see current-output-
port) using parameterize around the call to thunk.

Examples:
> (with-output-to-file some-file
   (lambda () (printf "hello")))
> (with-input-from-file some-file
   (lambda () (read-string 5)))
"hello"

Changed in version 8.1.0.3 of package base: Added the #:permissions argument.

(port-try-file-lock? port mode) → boolean?
  port : file-stream-port?
  mode : (or/c 'shared 'exclusive)

Attempts to acquire a lock on the file using the current platform’s facilities for file locking.
Multiple processes can acquire a 'shared lock on a file, but at most one process can hold
an 'exclusive lock, and 'shared and 'exclusive locks are mutually exclusive. When
mode is 'shared, then port must be an input port; when mode is 'exclusive, then port
must be an output port.

The result is #t if the requested lock is acquired, #f otherwise. When a lock is acquired,
it is held until either it is released with port-file-unlock or the port is closed (perhaps
because the process terminates).

Depending on the platform, locks may be merely advisory (i.e., locks affect only the ability
of processes to acquire locks) or they may correspond to mandatory locks that prevent reads
and writes to the locked file. Specifically, locks are mandatory on Windows and advisory on
other platforms. Multiple tries for a 'shared lock on a single port can succeed; on Unix and
Mac OS, a single port-file-unlock release the lock, while on other Windows, a port-
file-unlock is needed for each successful port-try-file-lock?. On Unix and Mac
OS, multiple tries for a 'exclusive lock can succeed and a single port-file-unlock
releases the lock, while on Windows, a try for an 'exclusive lock fails for a given port if
the port already holds the lock.

A lock acquired for an input port from open-input-output-file can be released through
port-file-unlock on the corresponding output port, and vice versa. If the output port

908
from open-input-output-file holds an 'exclusive lock, the corresponding input port can still acquire a 'shared lock, even multiple times; on Windows, a port-file-unlock is needed for each successful lock try, while a single port-file-unlock balances the lock tries on Unix and Mac OS. A 'shared lock on an input port can be upgraded to an 'exclusive lock through the corresponding output port on Unix and Mac OS, in which case a single port-file-unlock (on either port) releases the lock, while such upgrades are not allowed on Windows.

Locking is normally supported only for file ports, and attempting to acquire a lock with other kinds of file-stream ports raises an exn:fail:filesystem exception.

```scheme
(port-file-unlock port) → void?
  port : file-stream-port?
```

Releases a lock held by the current process on the file of port.

```scheme
(port-file-identity port) → exact-positive-integer?
  port : file-stream-port?
```

Returns a number that represents the identity of the device and file read or written by port. For two ports whose open times overlap, the result of port-file-identity is the same for both ports if and only if the ports access the same device and file. For ports whose open times do not overlap, no guarantee can be provided for the port identities (even if the ports actually access the same file)—except as can be inferred through relationships with other ports. If port is closed, the exn:fail exception is raised. On Windows 95, 98, and Me, if port is connected to a pipe instead of a file, the exn:fail:filesystem exception is raised.

Examples:

```scheme
> (define file1 (open-output-file some-file))
> (define file2 (open-output-file some-other-file))
> (port-file-identity file1)
43404906198806298628
> (port-file-identity file2)
43404906203101265924
> (close-output-port file1)
> (close-output-port file2)
```

13.1.6 String Ports

A string port reads or writes from a byte string. An input string port can be created from either a byte string or a string; in the latter case, the string is effectively converted to a byte string using string->bytes/utf-8. An output string port collects output into a byte string, but get-output-string conveniently converts the accumulated bytes to a string.
Input and output string ports do not need to be explicitly closed. The `file-position` procedure works for string ports in position-setting mode.

\[
\text{(string-port? } p) \rightarrow \text{ boolean?}
\]

\[
p : \text{port?}
\]

Returns `#t` if `p` is a string port, `#f` otherwise.

Added in version 6.0.1.6 of package base.

\[
\text{(open-input-bytes } bstr \ [\text{name}]) \rightarrow (\text{and/c input-port? string-port?})
\]

\[
bstr : \text{bytes?}
\]

\[
\text{name} : \text{any/c = 'string}
\]

Creates an input string port that reads characters from `bstr` (see §4.5 “Byte Strings”). Modifying `bstr` afterward does not affect the byte stream produced by the port. The optional `name` argument is used as the name for the returned port.

Examples:

\[
> (define sp (open-input-bytes "(apples 42 day)"))
\]

\[
> (define sexp1 (read sp))
\]

\[
> (first sexp1)
\]

\[
'\text{apples}
\]

\[
> (\text{rest sexp1})
\]

\[
'(42 day)
\]

\[
> (\text{read-line (open-input-bytes}
\]

\[
\#"\text{the cow jumped over the moon\nthe little dog\n})
\]

\[
\"\text{the cow jumped over the moon}"
\]

\[
\text{(open-input-string } str \ [\text{name}]) \rightarrow (\text{and/c input-port? string-port?})
\]

\[
\text{str} : \text{string?}
\]

\[
\text{name} : \text{any/c = 'string}
\]

Creates an input string port that reads bytes from the UTF-8 encoding (see §13.1.1 “Encodings and Locales”) of `str`. The optional `name` argument is used as the name for the returned port.

Examples:

\[
> (define sp (open-input-string "((\lambda (x) x))")
\]

\[
> (read sp)
\]

\[
'((\lambda (x) x))
\]

\[
> (define names (open-input-string "Günter Harder\nFrédéric Paulin\n")
\]

\[
910
\]
(open-output-bytes [name]) → (and/c output-port? string-port?)

name : any/c = 'string

Creates an output string port that accumulates the output into a byte string. The optional
name argument is used as the name for the returned port.

Examples:

> (define op1 (open-output-bytes))
> (write '(((1 2 3) ("Tom" "Dick") (quote a) (quote b) (quote c))) op1)
> (get-output-bytes op1)
"(((1 2 3) ("Tom\" "Dick\") (quote a) (quote b) (quote c)))"
> (define op2 (open-output-bytes))
> (write "Hi " op2)
> (write "there" op2)
> (get-output-bytes op2)
"\"Hi \"\"there\"
> (define op3 (open-output-bytes))
> (write-bytes="#" op3)
3
> (get-output-bytes op3)
"#Hi there"

(open-output-string [name]) → (and/c output-port? string-port?)

name : any/c = 'string

The same as open-output-bytes.

Examples:

> (define op1 (open-output-string))
> (write '(((1 2 3) ("Tom" "Dick") (quote a) (quote b) (quote c))) op1)
> (get-output-string op1)
"(((1 2 3) ("Tom\" "Dick\") (quote a) (quote b) (quote c)))"
> (define op2 (open-output-string))
> (write "$Hi "$ op2)
> (write "there" op2)
> (get-output-string op2)
"Hi ""there"
> (define op3 (open-output-string))
> (write-string "Hi " op3)
 3
> (write-string "there" op3)
 5
> (get-output-string op3)
"Hi there"

(get-output-bytes out
  [reset?
   start-pos
   end-pos)] → bytes?
out : (and/c output-port? string-port?)
reset? : any/c = #f
start-pos : exact-nonnegative-integer? = 0
end-pos : exact-nonnegative-integer? = #f

Returns the bytes accumulated in the string port out so far in a freshly allocated byte string (including any bytes written after the port’s current position, if any). The out port must be an output string port produced by open-output-bytes (or open-output-string) or a structure whose prop:output-port property refers to such an output port (transitively).

If reset? is true, then all bytes are removed from the port, and the port’s position is reset to 0; if reset? is #f, then all bytes remain in the port for further accumulation (so they are returned for later calls to get-output-bytes or get-output-string), and the port’s position is unchanged.

The start-pos and end-pos arguments specify the range of bytes in the port to return; supplying start-pos and end-pos is the same as using subbytes on the result of get-output-bytes, but supplying them to get-output-bytes can avoid an allocation. The end-pos argument can be #f, which corresponds to not passing a second argument to subbytes.

Examples:

> (define op (open-output-bytes))
> (write '((1 2 3) ("Tom" "Dick") ('a 'b 'c)) op)
> (get-output-bytes op)
"((1 2 3) ("Tom\" "Dick\") ((quote a) (quote b) (quote c)))"
> (get-output-bytes op #f 3 16)
"#f 3 16"
> (get-output-bytes op #t)
"((1 2 3) ("Tom\" "Dick\") ((quote a) (quote b) (quote c)))"
> (get-output-bytes op)
""

912
(get-output-string out) → string?
  out : (and/c output-port? string-port?)

Returns (bytes->string/utf-8 (get-output-bytes out) #\uFFFD).

Examples:

> (define i (open-input-string "hello world"))
> (define o (open-output-string))
> (write (read i) o)
> (get-output-string o)
"hello"

13.1.7 Pipes

A Racket pipe is internal to Racket, and not related to OS-level pipes for communicating between different processes.

(make-pipe [limit input-name output-name])
  → input-port? output-port?
  limit : exact-positive-integer? = #f
  input-name : any/c = 'pipe
  output-name : any/c = 'pipe

Returns two port values: the first port is an input port and the second is an output port. Data written to the output port is read from the input port, with no intermediate buffering. Unlike some other kinds of ports, pipe ports do not need to be explicitly closed to be reclaimed by garbage collection.

If limit is #f, the new pipe holds an unlimited number of unread bytes (i.e., limited only by the available memory). If limit is a positive number, then the pipe will hold at most limit unread/unpeeked bytes; writing to the pipe’s output port thereafter will block until a read or peek from the input port makes more space available. (Peeks effectively extend the port’s capacity until the peeked bytes are read.)

The optional input-name and output-name are used as the names for the returned input and output ports, respectively.

(pipe-content-length pipe-port) → exact-nonnegative-integer?
  pipe-port : port?

Returns the number of bytes contained in a pipe, where pipe-port is either of the pipe’s ports produced by make-pipe. The pipe’s content length counts all bytes that have been written to the pipe and not yet read (though possibly peeked).
13.1.8 Structures as Ports

\[ \text{prop:input-port} : \text{struct-type-property?} \]

\[ \text{prop:output-port} : \text{struct-type-property?} \]

The \text{prop:input-port} and \text{prop:output-port} structure type properties identify structure types whose instances can serve as input and output ports, respectively.

Each property value can be either of the following:

- An input port (for \text{prop:input-port}) or output port (for \text{prop:output-port}): In this case, using the structure as port is equivalent to using the given input or output port.

- An exact, non-negative integer between 0 (inclusive) and the number of non-automatic fields in the structure type (exclusive, not counting supertype fields): The integer identifies a field in the structure, and the field must be designated as immutable. If the field contains an input port (for \text{prop:input-port}) or output port (for \text{prop:output-port}), the port is used. Otherwise, an empty string input port is used for \text{prop:input-port}, and a port that discards all data is used for \text{prop:output-port}.

Some procedures, such as \text{file-position}, work on both input and output ports. When given an instance of a structure type with both the \text{prop:input-port} and \text{prop:output-port} properties, the instance is used as an input port.

13.1.9 Custom Ports

The \text{make-input-port} and \text{make-output-port} procedures create custom ports with arbitrary control procedures (much like implementing a device driver). Custom ports are mainly useful to obtain fine control over the action of committing bytes as read or written.

```
(make-input-port name read-in peek close [get-progress-evt commit get-location count-lines! init-position buffer-mode]) \rightarrow \text{input-port?}
```
name : any/c
read-in : (or/c
  (bytes?
    . -> . (or/c exact-nonnegative-integer?
      eof-object?
      procedure?
      evt?)
  )
)
input-port?)
peek : (or/c
  (bytes? exact-nonnegative-integer? (or/c evt? #f)
    . -> . (or/c exact-nonnegative-integer?
      eof-object?
      procedure?
      evt?
      #f)
  )
)
input-port?)
close : (-> any)
get-progress-evt : (or/c (-> evt?) #f) = #f
commit : (or/c (exact-positive-integer? evt? evt? . -> . any)
  #f)
  = #f
get-location : (or/c
  (->
    (values (or/c exact-positive-integer? #f)
      (or/c exact-nonnegative-integer? #f)
      (or/c exact-positive-integer? #f)))
  #f)
  = #f
count-lines! : (-> any) = void
init-position : (or/c exact-positive-integer?
  port?
  #f
  (-> (or/c exact-positive-integer? #f)))
  = 1
buffer-mode : (or/c (case-> ((or/c 'block 'none) . -> . any)
  (-> (or/c 'block 'none #f)))
  #f)
  = #f

Creates an input port, which is immediately open for reading. If close procedure has no side effects, then the port need not be explicitly closed. See also make-input-port/read-to-peek.

The arguments implement the port as follows:
- **name** — the name for the input port.
- **read-in** — either an input port, in which case reads are redirected to the given port, or a procedure that takes a single argument: a mutable byte string to receive read bytes. The procedure’s result is one of the following:
  - the number of bytes read, as an exact, non-negative integer;
  - *eof*;
  - a procedure of arity four (representing a “special” result, as discussed further below), but a procedure result is allowed only when *peek* is not *#f*;
  - a pipe input port that supplies bytes to be used as long as the pipe has content (see *pipe-content-length*) or until *read-in* or *peek* is called again; or
  - a synchronizable event (see §11.2.1 “Events”) other than a pipe input port or procedure of arity four; the event becomes ready when the read is complete (roughly): the event’s value can be one of the above four results or another event like itself; in the last case, a reading process loops with *sync* until it gets a non-event result.

The *read-in* procedure must not block indefinitely. If no bytes are immediately available for reading, the *read-in* must return 0 or an event, and preferably an event (to avoid busy waits). The *read-in* should not return 0 (or an event whose value is 0) when data is available in the port, otherwise polling the port will behave incorrectly. An event result from an event can also break polling.

If the result of a *read-in* call is not one of the above values, the *exn:fail:contract* exception is raised. If a returned integer is larger than the supplied byte string’s length, the *exn:fail:contract* exception is raised. If *peek* is *#f* and a procedure for a special result is returned, the *exn:fail:contract* exception is raised.

The *read-in* procedure can report an error by raising an exception, but only if no bytes are read. Similarly, no bytes should be read if *eof*, an event, or a procedure is returned. In other words, no bytes should be lost due to spurious exceptions or non-byte data.

A port’s reading procedure may be called in multiple threads simultaneously (if the port is accessible in multiple threads), and the port is responsible for its own internal synchronization. Note that improper implementation of such synchronization mechanisms might cause a non-blocking read procedure to block indefinitely.

If the result is a pipe input port, then previous *get-progress-evt* calls whose event is not yet ready must have been the pipe input port itself. Furthermore, *get-progress-evt* must continue to return the pipe as long as it contains data, or until the *read-in* or *peek-in* procedure is called again (instead of using the pipe, for whatever reason). If *read-in* or *peek-in* is called, any previously associated pipe (as returned by a previous call) is disassociated from the port and is not in use by any other thread as a result of the previous association.

If *peek*, *get-progress-evt*, and *commit* are all provided and non-*#f*, then the following is an acceptable implementation of *read-in*:
(lambda (bstr)
  (let* ([progress-evt (get-progress-evt)]
         [v (peek bstr 0 progress-evt)])
    (cond
      [(sync/timeout 0 progress-evt) 0] ; try again
      [(evt? v) (wrap-evt v (lambda (x) 0))] ; sync, try again
      [(and (number? v) (zero? v)) 0] ; try again
      [else
       (if (commit (if (number? v) v 1)
                 progress-evt
                 always-evt)
          v ; got a result
          0)]))) ; try again

An implementor may choose not to implement the peek, get-progress-evt, and commit procedures, however, and even an implementor who does supply them may provide a different read-in that uses a fast path for non-blocking reads.

In an input port is provided for read-in, then an input port must also be provided for peek.

• peek — either #f, an input port (in which case peeks are redirected to the given port), or a procedure that takes three arguments:
  – a mutable byte string to receive peeked bytes;
  – a non-negative number of bytes (or specials) to skip before peeking; and
  – either #f or a progress event produced by get-progress-evt.

The results and conventions for peek are mostly the same as for read-in. The main difference is in the handling of the progress event, if it is not #f. If the given progress event becomes ready, the peek must abort any skip attempts and not peek any values. In particular, peek must not peek any values if the progress event is initially ready. If the port has been closed, the progress event should be ready, in which case peek should complete (instead of failing because the port is closed).

Unlike read-in, peek should produce #f (or an event whose value is #f) if no bytes were peeked because the progress event became ready. Like read-in, a 0 result indicates that another attempt is likely to succeed, so 0 is inappropriate when the progress event is ready. Also like read-in, peek must not block indefinitely.

The skip count provided to peek is a number of bytes (or specials) that must remain present in the port—in addition to the peek results—when the peek results are reported. If a progress event is supplied, then the peek is effectively canceled when another process reads data before the given number can be skipped. If a progress event is not supplied and data is read, then the peek must effectively restart with the original skip count.

The system does not check that multiple peeks return consistent results, or that peeking and reading produce consistent results.
If `peek` is `#f`, then peeking for the port is implemented automatically in terms of reads, but with several limitations. First, the automatic implementation is not thread-safe. Second, the automatic implementation cannot handle special results (non-byte and non-eof), so `read-in` cannot return a procedure for a special when `peek` is `#f`. Finally, the automatic peek implementation is incompatible with progress events, so if `peek` is `#f`, then `get-progress-evts` and `commit` must be `#f`. See also `make-input-port/read-to-peek`, which implements peeking in terms of `read-in` without these constraints.

In an input port is provided for `peek`, then an input port must also be provided for `read-in`.

- **close** — a procedure of zero arguments that is called to close the port. The port is not considered closed until the closing procedure returns. The port’s procedures will never be used again via the port after it is closed. However, the closing procedure can be called simultaneously in multiple threads (if the port is accessible in multiple threads), and it may be called during a call to the other procedures in another thread; in the latter case, any outstanding reads and peeks should be terminated with an error.

- **get-progress-evts** — either `#f` (the default), or a procedure that takes no arguments and returns an event. The event must become ready only after data is next read from the port or the port is closed. If the port is already closed, the event must be ready. After the event becomes ready, it must remain so. See the description of `read-in` for information about the allowed results of this function when `read-in` returns a pipe input port. See also `semaphore-peek-evts`, which is sometimes useful for implementing `get-progress-evts`.

If `get-progress-evts` is `#f`, then `port-provides-progress-evts?` applied to the port will produce `#f`, and the port will not be a valid argument to `port-progress-evts`.

The result event will not be exposed directly by `port-progress-evts`. Instead, it will be wrapped in an event for which `progress-evts?` returns true.

- **commit** — either `#f` (the default), or a procedure that takes three arguments:
  - an exact, positive integer `k`;
  - a progress event produced by `get-progress-evts`;
  - an event, `done`, that is either a channel-put event, channel, semaphore, semaphore-peek event, always event, or never event.

A `commit` corresponds to removing data from the stream that was previously peeked, but only if no other process removed data first. (The removed data does not need to be reported, because it has been peeked already.) More precisely, assuming that `k` bytes, specials, and mid-stream `eofs` have been previously peeked or skipped at the start of the port’s stream, `commit` must satisfy the following constraints:

- It must return only when the commit is complete or when the given progress event becomes ready.
– It must commit only if \( k_p \) is positive.

– If it commits, then it must do so with either \( k_r \) items or \( k_p \) items, whichever is smaller, and only if \( k_p \) is positive.

– It must never choose done in a synchronization after the given progress event is ready, or after done has been synchronized once.

– It must not treat any data as read from the port unless done is chosen in a synchronization.

– It must not block indefinitely if done is ready; it must return soon after the read completes or soon after the given progress event is ready, whichever is first.

– It can report an error by raising an exception, but only if no data has been committed. In other words, no data should be lost due to an exception, including a break exception.

– It must return a true value if data has been committed, \( #f \) otherwise. When it returns a value, the given progress event must be ready (perhaps because data has just been committed).

– It should return a byte string as a true result when line counting is enabled and get-location is \( #f \) (so that line counting is implemented the default way); the result byte string represents the data that was committed for the purposes of character and line counting. If any other true result is returned when a byte string is expected, it is treated like a byte string where each byte corresponds to a non-newline character.

– It must raise an exception if no data (including eof) has been peeked from the beginning of the port’s stream, or if it would have to block indefinitely to wait for the given progress event to become ready.

A call to commit is parameterize-breaked to disable breaks.

• get-location — either \( #f \) (the default), or a procedure that takes no arguments and returns three values: the line number for the next item in the port’s stream (a positive number or \( #f \)), the column number for the next item in the port’s stream (a non-negative number or \( #f \)), and the position for the next item in the port’s stream (a positive number or \( #f \)). See also §13.1.4 “Counting Positions, Lines, and Columns”.

This procedure is called to implement port-next-location, but only if line counting is enabled for the port via port-count-lines! (in which case count-lines! is called). The read and read-syntax procedures assume that reading a non-whitespace character increments the column and position by one.

• count-lines! — a procedure of no arguments that is called if and when line counting is enabled for the port. The default procedure is void.

• init-position — normally an exact, positive integer that determines the position of the port’s first item, which is used by file-position or when line counting is not enabled for the port. The default is 1. If init-position is \( #f \), the port is treated as having an unknown position. If init-position is a port, then the given port’s position is always used for the new port’s position. If init-position is a procedure, it is called as needed to obtain the port’s position.
• **buffer-mode** — either `#f` (the default) or a procedure that accepts zero or one arguments. If `buffer-mode` is `#f`, then the resulting port does not support a buffer-mode setting. Otherwise, the procedure is called with one symbol argument ("block" or "none") to set the buffer mode, and it is called with zero arguments to get the current buffer mode. In the latter case, the result must be "block", "none", or `#f` (unknown). See §13.1.3 “Port Buffers and Positions” for more information on buffer modes.

**“Special” results:** When `read-in` or `peek` (or an event produced by one of these) returns a procedure, the procedure is used to obtain a non-byte result. (This non-byte result is not intended to return a character or `eof`: in particular, `read-char` raises an exception if it encounters a special-result procedure, even if the procedure produces a byte.) A special-result procedure must accept four arguments that represent a source location. The first argument is `#f` when the special read is triggered by `read` or `read/recursive`.

The special-value procedure can return an arbitrary value, and it will be called zero or one times (not necessarily before further reads or peeks from the port). See §13.7.2 “Reader-Extension Procedures” for more details on the procedure’s result.

If `read-in` or `peek` returns a special procedure when called by any reading procedure other than `read`, `read-syntax`, `read-char-or-special`, `peek-char-or-special`, `read-byte-or-special`, or `peek-byte-or-special`, then the `exn:fail:contract` exception is raised.

Examples:

```scheme
> (define /dev/null-in
  (make-input-port 'null
    (lambda (s) eof)
    (lambda (skip s progress-evt) eof)
    void
    (lambda () never-evt)
    (lambda (k progress-evt done-evt)
      (error "no successful peeks!")')))```

```scheme
> (read-char /dev/null-in) #<eof>
> (peek-char /dev/null-in) #<eof>
> (read-byte-or-special /dev/null-in) #<eof>
> (peek-byte-or-special /dev/null-in 100) #<eof>
> (define infinite-ones
```

; A port that produces a stream of 1s:
> (define infinite-ones

920
> (make-input-port
  'ones
  (lambda (s)
    (bytes-set! s 0 (char->integer #\1)) 1)
  #f
  void))

; But we can't peek ahead arbitrarily far, because the
; automatic peek must record the skipped bytes, so
; we'd run out of memory.

; An infinite stream of 1s with a specific peek procedure:
> (define infinite-ones
  (let ([one! (lambda (s)
                   (bytes-set! s 0 (char->integer #\1)) 1)]))
    (make-input-port
     'ones
     one!
     (lambda (s skip progress-evt) (one! s))
     void)))

> (read-string 5 infinite-ones)
"11111"

; Now we can peek ahead arbitrarily far:
> (peek-string 5 (expt 2 5000) infinite-ones)
"11111"

; The port doesn't supply procedures to implement progress events:
> (port-provides-progress-evts? infinite-ones)
#f

> (port-progress-evt infinite-ones)
port-progress-evt: port does not provide progress evts

; Non-byte port results:
> (define infinite-voids
  (make-input-port
   'voids
   (lambda (s) (lambda args 'void))
   (lambda (skip s evt) (lambda args 'void))
   void)))

> (read-char infinite-voids)
read-char: non-character in an unsupported context

> (read-char-or-special infinite-voids)
'void

; This port produces 0, 1, 2, 0, 1, 2, etc., but it is not
; thread-safe, because multiple threads might read and change n.
> (define mod3-cycle/one-thread

921
(let* ([n 2]
    [mod! (lambda (s delta)
      (bytes-set! s 0 (+ 48 (modulo (+ n delta) 3)))
     1)])
(make-input-port
 'mod3-cycle/not-thread-safe
 (lambda (s)
   (set! n (modulo (add1 n) 3))
   (mod! s 0))
   (lambda (s skip evt)
     (mod! s skip))
     void)))
> (read-string 5 mod3-cycle/one-thread)
"01201"
> (peek-string 5 (expt 2 5000) mod3-cycle/one-thread)
"20120"
; Same thing, but thread-safe and kill-safe, and with progress
; events. Only the server thread touches the stateful part
; directly. (See the output port examples for a simpler thread-
; safe
; example, but this one is more general.)
> (define (make-mod3-cycle)
    (define read-req-ch (make-channel))
    (define peek-req-ch (make-channel))
    (define progress-req-ch (make-channel))
    (define commit-req-ch (make-channel))
    (define close-req-ch (make-channel))
    (define closed? #f)
    (define n 0)
    (define progress-sema #f)
    (define (mod! s delta)
      (bytes-set! s 0 (+ 48 (modulo (+ n delta) 3)))
     1)
    ; -----------------------------
    ; The server has a list of outstanding commit requests,
    ; and it also must service each port operation (read,
    ; progress-evt, etc.)
    (define (serve commit-reqs response-evts)
      (apply
      sync
      (handle-evt read-req-ch
        (handle-read commit-reqs response-evts))
      (handle-evt progress-req-ch
        (handle-progress commit-reqs response-evts))
      (handle-evt commit-req-ch
        (add-commit commit-reqs response-evts))
      void)))
(handle-evt close-req-ch
  (handle-close commit-reqs response-evts))

(append
  (map (make-handle-response commit-reqs response-evts)
       response-evts)
  (map (make-handle-commit commit-reqs response-evts)
       commit-reqs)))

; Read/peek request: fill in the string and commit
(define ((handle-read commit-reqs response-evts) r)
  (let ([s (car r)]
        [skip (cadr r)]
        [ch (caddr r)]
        [nack (cadddr r)]
        [evt (car (cddddr r))]
        [peek? (cdr (cddddr r))])
    (let ([fail? (and evt
                     (sync/timeout 0 evt))])
      (unless (or closed? fail?)
        (mod! s skip)
        (unless peek?
          (commit! 1)))
      ; Add an event to respond:
      (serve commit-reqs
        (cons (choice-evt
               nack
               (channel-put-evt ch (if closed?
                                   0
                                   (if fail? #f 1))))
               response-evts))))

; Progress request: send a peek evt for the current
; progress-sema
(define ((handle-progress commit-reqs response-evts) r)
  (let ([ch (car r)]
        [nack (cdr r)])
    (unless progress-sema
      (set! progress-sema (make-semaphore (if closed? 1 0))))
    ; Add an event to respond:
    (serve commit-reqs
      (cons (choice-evt
             nack
             (channel-put-evt ch
                               (semaphore-peek-evt progress-sema))
             response-evts))))

; Commit request: add the request to the list
(define ((add-commit commit-reqs response-evts) r)
(serve (cons r commit-reqs response-evts))

; Commit handling: watch out for progress, in which case
; the response is a commit failure; otherwise, try
; to sync for a commit. In either event, remove the
; request from the list
(define ((make-handle-commit commit-reqs response-evts) r)
  (let ([k (car r)]
        [progress-evt (cadr r)]
        [done-evt (caddr r)]
        [ch (cadddr r)]
        [nack (cddddr r)])
    ; Note: we don't check that k is <= the sum of
    ; previous peeks, because the entire stream is actually
    ; known, but we could send an exception in that case.
    (choice-evt
     (handle-evt progress-evt
      (lambda (x)
        (sync nack (channel-put-evt ch #f))
        (serve (remq r commit-reqs) response-evts)))
     ; Only create an event to satisfy done-evt if progress-
     ; isn't already ready.
     ; Afterward, if progress-evt becomes ready, then this
     ; event-making function will be called again, because
     ; the server controls all posts to progress-evt.
     (if (sync/timeout 0 progress-evt)
         never-evt
         (handle-evt done-evt
          (lambda (v)
            (commit! k)
            (sync nack (channel-put-evt ch #t))
            (serve (remq r commit-reqs)
                   response-evts))))))

; Response handling: as soon as the respondee listens,
; remove the response
(define ((make-handle-response commit-reqs response-evts) evt)
  (handle-evt evt
   (lambda (x)
    (serve commit-reqs
             (remq evt response-evts))))))

; Close handling: post the progress sema, if any, and set
; the closed? flag
(define ((handle-close commit-reqs response-evts) r)
  (let ([ch (car r)]
        [nack (cdr r)])
    (...))
(set! closed? #t)
(when progress-sema
  (semaphore-post progress-sema))
(serve commit-reqs
  (cons (choice-evt nack
    (channel-put-evt ch (void))
    response-evts)))

; Helper for reads and post-peek commits:
(define (commit! k)
  (when progress-sema
    (semaphore-post progress-sema)
    (set! progress-sema #f)
    (set! n (+ n k)))
  ; Start the server thread:
  (define server-thread (thread (lambda () (serve null null))))

; -------------------
; Client-side helpers:
(define (req-evt f)
  (nack-guard-evt
    (lambda (nack)
      ; Be sure that the server thread is running:
      (thread-resume server-thread (current-thread))
      ; Create a channel to hold the reply:
      (let ((ch (make-channel)))
        (f ch nack)
        ch))))

(define (read-or-peek-evt s skip evt peek?)
  (req-evt (lambda (ch nack)
    (channel-put read-req-ch
      (list* s skip ch nack evt peek?)))))

; Make the port:
(make-input-port 'mod3-cycle
  ; Each handler for the port just sends
  ; a request to the server
  (lambda (s) (read-or-peek-evt s 0 #f #f))
  (lambda (s skip evt)
    (read-or-peek-evt s skip evt #t))
  (lambda () ; close
    (sync (req-evt
      (lambda (ch nack)
        (channel-put progress-req-ch
          (list* ch nack))))))
  (lambda () ; progress-evt
    (sync (req-evt
      (lambda (ch nack)
        (channel-put progress-req-ch
          (list* ch nack))))))

925
(define mod3-cycle (make-mod3-cycle))

(let ([result1 #f]
       [result2 #f])
  (let ([t1 (thread
             (lambda ()
               (set! result1 (read-string 5 mod3-cycle))))]
       [t2 (thread
             (lambda ()
               (set! result2 (read-string 5 mod3-cycle))))]
       (thread-wait t1)
       (thread-wait t2)
       (string-append result1 "," result2)))
"11120,02020"

(define s (make-bytes 1))
(define progress-evt (port-progress-evt mod3-cycle))
(peek-bytes-avail! s 0 progress-evt mod3-cycle) 1
> s
#"1"
(port-commit-peeked 1 progress-evt (make-semaphore 1) mod3-cycle)
#t
> (sync/timeout 0 progress-evt)
<progress-evt>
> (peek-bytes-avail! s 0 progress-evt mod3-cycle) 0
> (port-commit-peeked 1 progress-evt (make-semaphore 1) mod3-cycle)
#f
> (close-input-port mod3-cycle)
(make-output-port name
evtt
write-out
close
[write-out-special
get-write-evt
get-write-special-evt
get-location
count-lines!
init-position
buffer-mode]) → output-port?

name : any/c
evt : evt?
write-out : (or/c
(bytes? exact-nonnegative-integer?
  exact-nonnegative-integer?
  boolean?
  boolean?
  . -> .
  (or/c exact-nonnegative-integer?
    #f
evt?))
output-port?)
close : (-> any)
write-out-special : (or/c (any/c boolean? boolean? = #f
  . -> .
  (or/c any/c
    #f
evt?))
output-port?
#f)
get-write-evt : (or/c
  = #f
  (bytes? exact-nonnegative-integer?
    exact-nonnegative-integer?
    . -> .
evt?)
  #f)
get-write-special-evt : (or/c
  = #f
  (any/c . -> . evt?)
  #f)
get-location : (or/c
(->
(values (or/c exact-positive-integer? #f)
(or/c exact-nonnegative-integer? #f)
(or/c exact-positive-integer? #f)))
#f)
= #f

count-lines! : (-> any) = void

init-position : (or/c exact-positive-integer?
port?
#f
(-> (or/c exact-positive-integer? #f)))
= 1

buffer-mode : (or/c (case->
((or/c 'block 'line 'none) . -> . any)
(-> (or/c 'block 'line 'none #f)))
#f)
= #f

Creates an output port, which is immediately open for writing. If close procedure has no side effects, then the port need not be explicitly closed. The port can buffer data within its write-out and write-out-special procedures.

- **name** — the name for the output port.
- **evt** — a synchronization event (see §11.2.1 “Events” e.g., a semaphore or another port). The event is used in place of the output port when the port is supplied to synchronization procedures like sync. Thus, the event should be unblocked when the port is ready for writing at least one byte without blocking, or ready to make progress in flushing an internal buffer without blocking. The event must not unblock unless the port is ready for writing; otherwise, the guarantees of sync will be broken for the output port. Use always-evt if writes to the port always succeed without blocking.
- **write-out** — either an output port, which indicates that writes should be redirected to the given port, or a procedure of five arguments:
  - an immutable byte string containing bytes to write;
  - a non-negative exact integer for a starting offset (inclusive) into the byte string;
  - a non-negative exact integer for an ending offset (exclusive) into the byte string;
  - a boolean; #f indicates that the port is allowed to keep the written bytes in a buffer, and that it is allowed to block indefinitely; #t indicates that the write should not block, and that the port should attempt to flush its buffer and completely write new bytes instead of buffering them;
  - a boolean; #t indicates that if the port blocks for a write, then it should enable breaks while blocking (e.g., using sync/enable-break); this argument is always #f if the fourth argument is #t.
The procedure returns one of the following:

- a non-negative exact integer representing the number of bytes written or buffered;
- \( \#f \) if no bytes could be written, perhaps because the internal buffer could not be completely flushed;
- a pipe output port (when buffering is allowed and not when flushing) for buffering bytes as long as the pipe is not full and until \texttt{write-out} or \texttt{write-out-special} is called; or
- a synchronizable event (see §11.2.1 "Events") other than a pipe output port that acts like the result of \texttt{write-bytes-avail-evt} to complete the write.

Since \texttt{write-out} can produce an event, an acceptable implementation of \texttt{write-out} is to pass its first three arguments to the port’s \texttt{get-write-evt}. Some port implementors, however, may choose not to provide \texttt{get-write-evt} (perhaps because writes cannot be made atomic), or may implement \texttt{write-out} to enable a fast path for non-blocking writes or to enable buffering.

From a user’s perspective, the difference between buffered and completely written data is (1) buffered data can be lost in the future due to a failed write, and (2) \texttt{flush-output} forces all buffered data to be completely written. Under no circumstances is buffering required.

If the start and end indices are the same, then the fourth argument to \texttt{write-out} will be \#f, and the write request is actually a flush request for the port’s buffer (if any), and the result should be 0 for a successful flush (or if there is no buffer).

The result should never be 0 if the start and end indices are different, otherwise the \texttt{exn:fail:contract} exception is raised. Similarly, the \texttt{exn:fail:contract} exception is raised if \texttt{write-out} returns a pipe output port when buffering is disallowed or when it is called for flushing. If a returned integer is larger than the supplied byte-string range, the \texttt{exn:fail:contract} exception is raised.

The \#f result should be avoided, unless the next write attempt is likely to work. Otherwise, if data cannot be written, return an event instead.

An event returned by \texttt{write-out} can return \#f or another event like itself, in contrast to events produced by \texttt{write-bytes-avail-evt} or \texttt{get-write-evt}. A writing process loops with \texttt{sync} until it obtains a non-event result.

The \texttt{write-out} procedure is always called with breaks disabled, independent of whether breaks were enabled when the write was requested by a client of the port. If breaks were enabled for a blocking operation, then the fifth argument to \texttt{write-out} will be \#t, which indicates that \texttt{write-out} should re-enable breaks while blocking.

If the writing procedure raises an exception, due to write or commit operations, it must not have committed any bytes (though it may have committed previously buffered bytes).

A port’s writing procedure may be called in multiple threads simultaneously (if the port is accessible in multiple threads). The port is responsible for its own internal
synchronization. Note that improper implementation of such synchronization mecha-
nisms might cause a non-blocking write procedure to block.

- **close** — a procedure of zero arguments that is called to close the port. The port is not considered closed until the closing procedure returns. The port’s procedures will never be used again via the port after it is closed. However, the closing procedure can be called simultaneously in multiple threads (if the port is accessible in multiple threads), and it may be called during a call to the other procedures in another thread; in the latter case, any outstanding writes or flushes should be terminated immediately with an error.

- **write-out-special** — either #f (the default), an output port (which indicates that special writes should be redirected to the given port), or a procedure to handle write-special calls for the port. If #f, then the port does not support special output, and port-writes-special? will return #f when applied to the port.

  If a procedure is supplied, it takes three arguments: the special value to write, a boolean that is #f if the procedure can buffer the special value and block indefinitely, and a boolean that is #t if the procedure should enable breaks while blocking. The result is one of the following:

  - a non-event true value, which indicates that the special is written;
  - #f if the special could not be written, perhaps because an internal buffer could not be completely flushed;
  - a synchronizable event (see §11.2.1 “Events”) that acts like the result of get-write-special-evt to complete the write.

  Since write-out-special can return an event, passing the first argument to an implementation of get-write-special-evt is acceptable as a write-out-special.

  As for write-out, the #f result is discouraged, since it can lead to busy waiting. Also as for write-out, an event produced by write-out-special is allowed to produce #f or another event like itself. The write-out-special procedure is always called with breaks disabled, independent of whether breaks were enabled when the write was requested by a client of the port.

- **get-write-evt** — either #f (the default) or a procedure of three arguments:

  - an immutable byte string containing bytes to write;
  - a non-negative exact integer for a starting offset (inclusive) into the byte string;
  - a non-negative exact integer for an ending offset (exclusive) into the byte string.

  The result is a synchronizable event (see §11.2.1 “Events”) to act as the result of write-bytes_avail-evt for the port (i.e., to complete a write or flush), which becomes available only as data is committed to the port’s underlying device, and whose result is the number of bytes written.
If `get-write-evt` is `#f`, then `port-writes-atomic?` will produce `#f` when applied to the port, and the port will not be a valid argument to procedures such as `write-bytes-avail-evt`. Otherwise, an event returned by `get-write-evt` must not cause data to be written to the port unless the event is chosen in a synchronization, and it must write to the port if the event is chosen (i.e., the write must appear atomic with respect to the synchronization).

If the event’s result integer is larger than the supplied byte-string range, the `exn:fail:contract` exception is raised by a wrapper on the event. If the start and end indices are the same (i.e., no bytes are to be written), then the event should produce `0` when the buffer is completely flushed. (If the port has no buffer, then it is effectively always flushed.)

If the event raises an exception, due to write or commit operations, it must not have committed any new bytes (though it may have committed previously buffered bytes).

Naturally, a port’s events may be used in multiple threads simultaneously (if the port is accessible in multiple threads). The port is responsible for its own internal synchronization.

- `get-write-special-evt` — either `#f` (the default), or a procedure to handle `write-special-evt` calls for the port. This argument must be `#f` if either `write-out-special` or `get-write-evt` is `#f`, and it must be a procedure if both of those arguments are procedures.

  If it is a procedure, it takes one argument: the special value to write. The resulting event (with its constraints) is analogous to the result of `get-write-evt`.

  If the event raises an exception, due to write or commit operations, it must not have committed the special value (though it may have committed previously buffered bytes and values).

- `get-location` — either `#f` (the default), or a procedure that takes no arguments and returns three values: the line number for the next item written to the port’s stream (a positive number or `#f`), the column number for the next item written to the port’s stream (a non-negative number or `#f`), and the position for the next item written to the port’s stream (a positive number or `#f`). See also §13.1.4 "Counting Positions, Lines, and Columns".

  This procedure is called to implement `port-next-location` for the port, but only if line counting is enabled for the port via `port-count-lines!` (in which case `count-lines!` is called).

- `count-lines!` — a procedure of no arguments that is called if and when line counting is enabled for the port. The default procedure is `void`.

- `init-position` — normally an exact, positive integer that determines the position of the port’s first item, which is used by `file-position` or when line counting is not enabled for the port. The default is `1`. If `init-position` is `#f`, the port is treated as having an unknown position. If `init-position` is a port, then the given port’s position is always used for the new port’s position. If `init-position` is a procedure, it is called as needed to obtain the port’s position.
• **buffer-mode** — either #f (the default) or a procedure that accepts zero or one arguments. If **buffer-mode** is #f, then the resulting port does not support a buffer-mode setting. Otherwise, the procedure is called with one symbol argument ("block", 'line', or 'none) to set the buffer mode, and it is called with zero arguments to get the current buffer mode. In the latter case, the result must be 'block, 'line, 'none, or #f (unknown). See §13.1.3 "Port Buffers and Positions" for more information on buffer modes.

Examples:

```
; A port that writes anything to nowhere:
> (define /dev/null-out
  (make-output-port
   'null
   always-evt
   (lambda (s start end non-block? breakable?) (- end start)))
   void
   (lambda (special non-block? breakable?) #t)
   (lambda (s start end) (wrap-evt
     always-evt
     (lambda (x)
       (- end start))))
   (lambda (special) always-evt)))
> (display "hello" /dev/null-out)
5
> (write-special 'hello /dev/null-out)
#t
> (sync (write-bytes-avail-evt="#hello" /dev/null-out))
5
; A port that accumulates bytes as characters in a list,
; but not in a thread-safe way:
> (define accum-list null)
> (define accumulator/not-thread-safe
  (make-output-port
   'accum/not-thread-safe
   always-evt
   (lambda (s start end non-block? breakable?)
    (set! accum-list
     (append accum-list
      (map integer->char
       (map bytes->list (subbytes s start end))))))
    (- end start)))
   void))
> (display "hello" accumulator/not-thread-safe)
> accum-list
```

932
'(#
  \e #\l #\l #\o)
;
; Same as before, but with simple thread-safety:
> (define accum-list null)
> (define accumulator
  (let* ([lock (make-semaphore 1)]
         [lock-peek-evt (semaphore-peek-evt lock)])
    (make-output-port
     'accum
     lock-peek-evt
     (lambda (s start end non-block? breakable?)
       (if (semaphore-try-wait? lock)
           (begin
             (set! accum-list
              (append accum-list
                (map integer->char
                 (bytes->list
                  (subbytes s start end))))))
             (semaphore-post lock)
             (- end start))
           ; Cheap strategy: block until the list is unlocked,
           ; then return 0, so we get called again
           (wrap-evt
            lock-peek-evt
            (lambda (x) 0)))
     void))))
> (display "hello" accumulator)
> accum-list
'(#
  \e #\l #\l #\o)
;
A port that transforms data before sending it on
; to another port. Atomic writes exploit the
; underlying port's ability for atomic writes.
> (define (make-latin-1-capitalize port)
  (define (byte-upcase s start end)
    (list->bytes
     (map (lambda (b) (char->integer
                          (char-upcase
                           (integer->char b))))
      (bytes->list (subbytes s start end))))
  (make-output-port
   'byte-upcase
   ; This port is ready when the original is ready:
   port
   ; Writing procedure:
   (lambda (s start end non-block? breakable?)
    (let ([is (byte-upcase s start end)]
           [is (byte-upcase s start end)])
     (if non-block?
(write-bytes-avail* s port)
(begin
  (display s port)
  (bytes-length s)))))

; Close procedure -- close original port:
(lambda () (close-output-port port))

#f

; Write event:
(and (port-writes-atomic? port)
  (lambda (s start end)
    (write-bytes-avail-evt
      (byte-upcase s start end)
      port))))

> (define orig-port (open-output-string))
> (define cap-port (make-latin-1-capitalize orig-port))
> (display "Hello" cap-port)
"HELLO"
> (sync (write-bytes-avail-evt #"Bye" cap-port))
3
> (get-output-string orig-port)
"HELLOBYE"

13.1.10 More Port Constructors, Procedures, and Events

(require racket/port)  package: base

The bindings documented in this section are provided by the racket/port and racket libraries, but not racket/base.

Port String and List Conversions

(\(port->list\ [r \ in]\) \(\rightarrow\) (listof any/c))
\[r : (input-port? \ . \ . \ any/c) = \text{read}\]
\[in : \text{input-port?} = (current-input-port)\]

Returns a list whose elements are produced by calling \(r\) on \(in\) until it produces \text{eof}.

Examples:

> (define (read-number input-port)
      (define char (read-char input-port))
      (if (eof-object? char)
          char
          (string->number (string char)))))

934
(port->list read-number (open-input-string "12345"))
'(1 2 3 4 5)

(port->string [in #:close? close?]) → string?
  in : input-port? = (current-input-port)
  close? : any/c = #f

Reads all characters from in and returns them as a string. The input port is closed unless close? is #f.

Example:

> (port->string (open-input-string "hello world"))
"hello world"

Changed in version 6.8.0.2 of package base: Added the #:close? argument.

(port->bytes [in #:close? close?]) → bytes?
  in : input-port? = (current-input-port)
  close? : any/c = #f

Reads all bytes from in and returns them as a byte string. The input port is closed unless close? is #f.

Example:

> (port->bytes (open-input-string "hello world"))
#"hello world"

Changed in version 6.8.0.2 of package base: Added the #:close? argument.

(port->lines [in
  #:line-mode line-mode
  #:close? close?]) → (listof string?)
  in : input-port? = (current-input-port)
  line-mode : (or/c 'linefeed 'return 'return-linefeed 'any 'any-one)
    = 'any
  close? : any/c = #f

Read all characters from in, breaking them into lines. The line-mode argument is the same as the second argument to read-line, but the default is 'any instead of 'linefeed. The input port is closed unless close? is #f.

Example:
> (port->lines
  (open-input-string "line 1
line 2
 line 3
line 4"))
'("line 1" "line 2" " line 3" "line 4")

Changed in version 6.8.0.2 of package base: Added the #:close? argument.

(port->bytes-lines [in
  #:line-mode line-mode
  #:close? close?]) → (listof bytes?)

in : input-port? = (current-input-port)
line-mode : (or/c 'linefeed 'return 'return-linefeed 'any 'any-one)
  = 'any
close? : any/c = #f

Like port->lines, but reading bytes and collecting them into lines like read-bytes-line. The input port is closed unless close? is #f.

Example:

> (port->bytes-lines
  (open-input-string "line 1
line 2
 line 3
line 4"))
'(#"line 1" #"line 2" #" line 3" #"line 4")

Changed in version 6.8.0.2 of package base: Added the #:close? argument.

(display-lines lst
  [out
    #:separator separator]) → void?
lst : list?
out : output-port? = (current-output-port)
separator : any/c = #"\n"

Uses display on each element of lst to out, adding separator after each element.

(call-with-output-string proc) → string?
proc : (output-port? . -> . any)

Calls proc with an output port that accumulates all output into a string, and returns the string.

The port passed to proc is like the one created by open-output-string, except that it is wrapped via dup-output-port, so that proc cannot access the port’s content using get-output-string. If control jumps back into proc, the port continues to accumulate new data, and call-with-output-string returns both the old data and newly accumulated data.
(call-with-output-bytes proc) → bytes?
proc : (output-port? . -> . any)

Like call-with-output-string, but returns the accumulated result in a byte string instead of a string. Furthermore, the port’s content is emptied when call-with-output-bytes returns, so that if control jumps back into proc and returns a second time, only the newly accumulated bytes are returned.

(with-output-to-string proc) → string?
proc : (-> any)

Equivalent to

(call-with-output-string
  (lambda (p) (parameterize ([(current-output-port p)] (proc)))))

(with-output-to-bytes proc) → bytes?
proc : (-> any)

Equivalent to

(call-with-output-bytes
  (lambda (p) (parameterize ([(current-output-port p)] (proc)))))

(call-with-input-string str proc) → any
str : string?
proc : (input-port? . -> . any)

Equivalent to (proc (open-input-string str)).

(call-with-input-bytes bstr proc) → any
bstr : bytes?
proc : (input-port? . -> . any)

Equivalent to (proc (open-input-bytes bstr)).

(with-input-from-string str proc) → any
str : string?
proc : (-> any)

Equivalent to
(parameterize ([current-input-port (open-input-string str)])
  (proc))

(with-input-from-bytes bstr proc) → any
  bstr : bytes?
  proc : (→ any)

Equivalent to

(parameterize ([current-input-port (open-input-bytes str)])
  (proc))

Creating Ports

(input-port-append close-at-eof? in ... [#:name name]) → input-port?
  close-at-eof? : any/c
  in : input-port?
  name : any/c = (map object-name in)

Takes any number of input ports and returns an input port. Reading from the input port draws bytes (and special non-byte values) from the given input ports in order. If close-at-eof? is true, then each port is closed when an end-of-file is encountered from the port, or when the result input port is closed. Otherwise, data not read from the returned input port remains available for reading in its original input port.

The name argument determines the name as reported by object-name for the returned input port.

See also merge-input, which interleaves data from multiple input ports as it becomes available.

Changed in version 6.90.0.19 of package base: Added the name argument.

(make-input-port/read-to-peek name read-in fast-peek close
  [get-location count-lines! init-position buffer-position
    buffering? on-consumed]) → input-port?
Similar to make-input-port, but if the given read-in returns an event, the event’s value must be 0. The resulting port’s peek operation is implemented automatically (in terms of read-in) in a way that can handle special non-byte values. The progress-event and commit operations are also implemented automatically. The resulting port is thread-safe, but not kill-safe (i.e., if a thread is terminated or suspended while using the port, the port may become
The \textit{read-in, close, get-location, count-lines!, init-position,} and \textit{buffer-mode} procedures are the same as for \texttt{make-input-port}.

The \texttt{fast-peek} argument can be either \texttt{#f} or a procedure of three arguments: a byte string to receive a peek, a skip count, and a procedure of two arguments. The \texttt{fast-peek} procedure can either implement the requested peek, or it can dispatch to its third argument to implement the peek. The \texttt{fast-peek} is not used when a peek request has an associated progress event.

The \texttt{buffering?} argument determines whether \texttt{read-in} can be called to read more characters than are immediately demanded by the user of the new port. If \texttt{buffer-mode} is not \texttt{#f}, then \texttt{buffering?} determines the initial buffer mode, and \texttt{buffering?} is enabled after a buffering change only if the new mode is \texttt{'block}.

If \texttt{on-consumed} is not \texttt{#f}, it is called when data is read (or committed) from the port, as opposed to merely peeked. The argument to \texttt{on-consumed} is the result value of the port’s reading procedure, so it can be an integer or any result from \texttt{read-in}.

\begin{verbatim}
(make-limited-input-port in limit [close-orig?]) -> input-port?
in : input-port?
limit : exact-nonnegative-integer?
close-orig? : any/c = #t
\end{verbatim}

Returns a port whose content is drawn from \texttt{in}, but where an end-of-file is reported after \texttt{limit} bytes (and non-byte special values) have been read. If \texttt{close-orig?} is true, then the original port is closed if the returned port is closed.

Bytes are consumed from \texttt{in} only when they are consumed from the returned port. In particular, peeking into the returned port peeks into the original port.

If \texttt{in} is used directly while the resulting port is also used, then the \texttt{limit} bytes provided by the port need not be contiguous parts of the original port’s stream.

\begin{verbatim}
(make-pipe-with-specials [limit in-name out-name]) -> input-port? output-port?
limit : exact-nonnegative-integer? = #f
in-name : any/c = 'pipe
out-name : any/c = 'pipe
\end{verbatim}

Returns two ports: an input port and an output port. The ports behave like those returned by \texttt{make-pipe}, except that the ports support non-byte values written with procedures such as \texttt{write-special} and read with procedures such as \texttt{get-byte-or-special}.
The limit argument determines the maximum capacity of the pipe in bytes, but this limit is disabled if special values are written to the pipe before limit is reached. The limit is re-enabled after the special value is read from the pipe.

The optional in-name and out-name arguments determine the names of the result ports.

\[
\text{(combine-output a-out b-out)} \rightarrow \text{output-port?}
\]
\[
a-out : \text{output-port?}
\]
\[
b-out : \text{output-port?}
\]

Accepts two output ports and returns a new output port combining the original ports. When written to, the combined port first writes as many bytes as possible to a-out, and then tries to write the same number of bytes to b-out. If that doesn’t succeed, what is left over is buffered and no further writes can go through until the ports are evened out. The port is ready (for the purposes of synchronization) when each port reports being ready. However, the first port may stop being ready while waiting on the second port to sync, so it cannot be guaranteed that both ports are ready at once. Closing the combined port is done after writing all remaining bytes to b-out.

Added in version 7.7.0.10 of package base.

\[
\text{(merge-input a-in b-in [buffer-limit])} \rightarrow \text{input-port?}
\]
\[
a-in : \text{input-port?}
\]
\[
b-in : \text{input-port?}
\]
\[
buffer-limit : (or/c \text{exact-nonnegative-integer?} \text{#f}) = 4096
\]

Accepts two input ports and returns a new input port. The new port merges the data from two original ports, so data can be read from the new port whenever it is available from either of the two original ports. The data from the original ports are interleaved. When an end-of-file has been read from an original port, it no longer contributes characters to the new port. After an end-of-file has been read from both original ports, the new port returns end-of-file. Closing the merged port does not close the original ports.

The optional buffer-limit argument limits the number of bytes to be buffered from a-in and b-in, so that the merge process does not advance arbitrarily beyond the rate of consumption of the merged data. A #f value disables the limit. As for make-pipe-with-specials, buffer-limit does not apply when a special value is produced by one of the input ports before the limit is reached.

See also input-port-append, which concatenates input streams instead of interleaving them.

\[
\text{(open-output-nowhere [name special-ok?])} \rightarrow \text{output-port?}
\]
\[
name : \text{any/c} = \text{’nowhere}
\]
\[
special-ok? : \text{any/c} = \text{#t}
\]

Creates and returns an output port that discards all output sent to it (without blocking). The
The name argument is used as the port’s name. If the special-ok? argument is true, then the resulting port supports write-special, otherwise it does not.

\[
\text{(peeking-input-port \textit{in} \\
\quad \text{[name skip \\
\quad \#:init-position init-position]}) \rightarrow \text{input-port}}
\]

\text{in} : \text{input-port} \hfill \text{name : any/c = (object-name \textit{in})} \\
\text{skip : exact-nonnegative-integer? = 0} \hfill \text{init-position : exact-positive-integer? = 1}

Returns an input port whose content is determined by peeking into \textit{in}. In other words, the resulting port contains an internal skip count, and each read of the port peeks into \textit{in} with the internal skip count, and then increments the skip count according to the amount of data successfully peeked.

The optional \textit{name} argument is the name of the resulting port. The \textit{skip} argument is the port initial skip count, and it defaults to 0.

The resulting port’s initial position (as reported by file-position) is \((- \text{init-position} 1\)), no matter the position of \textit{in}.

The resulting port supports buffering, and a ‘block buffer mode allows the port to peek further into \textit{in} than requested. The resulting port’s initial buffer mode is ‘block, unless \textit{in} supports buffer mode and its mode is initially ‘none (i.e., the initial buffer mode is taken from \textit{in} when it supports buffering). If \textit{in} supports buffering, adjusting the resulting port’s buffer mode via file-stream-buffer-mode adjusts \textit{in}’s buffer mode.

For example, when you read from a peeking port, you see the same answers as when you read from the original port:

Examples:

\[
> \text{(define an-original-port (open-input-string "123456789"))} \\
> \text{(define a-peeking-port (peeking-input-port an-original-port))} \\
> \text{(file-stream-buffer-mode a-peeking-port 'none)} \\
> \text{(read-string 3 a-peeking-port)} \\
\quad \text{"123"} \\
> \text{(read-string 3 an-original-port)} \\
\quad \text{"123"}
\]

Beware that the read from the original port is invisible to the peeking port, which keeps its own separate internal counter, and thus interleaving reads on the two ports can produce confusing results. Continuing the example before, if we read three more characters from the peeking port, we end up skipping over the 456 in the port (but only because we disabled buffering above):
Example:

> `(read-string 3 a-peeking-port)`

"789"

If we had left the buffer mode of `a-peeking-port` alone, that last `read-string` would have likely produced "456" as a result of buffering bytes from an `original-port` earlier.

Changed in version 6.1.0.3 of package `base`: Enabled buffering and buffer-mode adjustments via `file-stream-buffer-mode`, and set the port’s initial buffer mode to that of `in`.

`=reencode-input-port in encoding [error-bytes close? name convert-newlines? enc-error] -> input-port?`

`in : input-port?`
`encoding : string?`
`error-bytes : (or/c #f bytes?) = #f`
`close? : any/c = #f`
`name : any/c = (object-name in)`
`convert-newlines? : any/c = #f`
`enc-error : (string? input-port? . -> . any) = (lambda (msg port) (error ...))`

Produces an input port that draws bytes from `in`, but converts the byte stream using `(bytes-open-converter encoding-str "UTF-8")`. In addition, if `convert-newlines?` is true, then decoded sequences that correspond to UTF-8 encodings of "\r\n", "\r\u0085", "\r", "\u0085", and "\u2028" are all converted to the UTF-8 encoding of "\n".

If `error-bytes` is provided and not `#f`, then the given byte sequence is used in place of bytes from `in` that trigger conversion errors. Otherwise, if a conversion is encountered, `enc-error` is called, which must raise an exception.

If `close?` is true, then closing the result input port also closes `in`. The `name` argument is used as the name of the result input port.

In non-buffered mode, the resulting input port attempts to draw bytes from `in` only as needed to satisfy requests. Toward that end, the input port assumes that at least `n` bytes must be read to satisfy a request for `n` bytes. (This is true even if the port has already drawn some bytes, as long as those bytes form an incomplete encoding sequence.)
(reencode-output-port  out
  encoding
  [error-bytes
   close?
   name
   newline-bytes
   enc-error])  →  output-port?

out : output-port?
encoding : string?
error-bytes : (or/c #f bytes?) = #f
close? : any/c = #f
name : any/c = (object-name out)
ewline-bytes : (or/c #f bytes?) = #f
enc-error : (string? output-port? . -> . any)
  = (lambda (msg port) (error ...))

Produces an output port that directs bytes to out, but converts its byte stream using (bytes-
open-converter "UTF-8" encoding-str). In addition, if newline-bytes is not #f, then bytes written to the port that are the UTF-8 encoding of "\n" are first converted to newline-bytes (before applying the convert from UTF-8 to encoding-str).

If error-bytes is provided and not #f, then the given byte sequence is used in place of bytes that have been sent to the output port and that trigger conversion errors. Otherwise, enc-error is called, which must raise an exception.

If close? is true, then closing the result output port also closes out. The name argument is used as the name of the result output port.

The resulting port supports buffering, and the initial buffer mode is (or (file-stream-
buffer-mode out) 'block). In 'block mode, the port’s buffer is flushed only when it is full or a flush is requested explicitly. In 'line mode, the buffer is flushed whenever a newline or carriage-return byte is written to the port. In 'none mode, the port’s buffer is flushed after every write. Implicit flushes for 'line or 'none leave bytes in the buffer when they are part of an incomplete encoding sequence.

The resulting output port does not support atomic writes. An explicit flush or special-write
to the output port can hang if the most recently written bytes form an incomplete encoding sequence.

When the port is buffered, a flush callback is registered with the current plumber to flush the buffer.

(dup-input-port in [close?])  →  input-port?
in : input-port?
  close? : any/c = #f

Returns an input port that draws directly from in. Closing the resulting port closes in only
if close? is #t.

The new port is initialized with the port read handler of in, but setting the handler on the result port does not affect reading directly from in.

```scheme
(dup-output-port out [close?]) -> output-port?
  out : output-port?
  close? : any/c = #f
```

Returns an output port that propagates data directly to out. Closing the resulting port closes out only if close? is #t.

The new port is initialized with the port display handler and port write handler of out, but setting the handlers on the result port does not affect writing directly to out.

```scheme
(relocate-input-port in
  line
  column
  position
  [close?]
  #:name name) -> input-port?

  in : input-port?
  line : (or/c exact-positive-integer? #f)
  column : (or/c exact-nonnegative-integer? #f)
  position : exact-positive-integer?
  close? : any/c = #t
  name : (object-name out)
```

Produces an input port that is equivalent to in except in how it reports location information (and possibly its name). The resulting port’s content starts with the remaining content of in, and it starts at the given line, column, and position. A #f for the line or column means that the line and column will always be reported as #f.

The line and column values are used only if line counting is enabled for in and for the resulting port, typically through port-count-lines!. The column value determines the column for the first line (i.e., the one numbered line), and later lines start at column 0. The given position is used even if line counting is not enabled.

When line counting is on for the resulting port, reading from in instead of the resulting port increments location reports from the resulting port. Otherwise, the resulting port’s position does not increment when data is read from in.

If close? is true, then closing the resulting port also closes in. If close? is #f, then closing the resulting port does not close in.

The name argument is used as the name for the resulting port; the default value keeps the same name as in.
(relocate-output-port  out
tline
column
position
[close?]
#:name name)  →  output-port?
out: output-port?
line: (or/c exact-positive-integer? #f)
column: (or/c exact-nonnegative-integer? #f)
position: exact-positive-integer?
close?: any/c = #t
name: (object-name out)

Like relocate-input-port, but for output ports.

(transplant-input-port  in
get-location
init-pos
[close?
count-lines!]
#:name name)  →  input-port?
in: input-port?
get-location: (or/c
  (->
    (values
      (or/c exact-positive-integer? #f)
      (or/c exact-nonnegative-integer? #f)
      (or/c exact-positive-integer? #f))
    #f)
init-pos: exact-positive-integer?
close?: any/c = #t
count-lines!: (-> any) = void
name: (object-name out)

Like relocate-input-port, except that arbitrary position information can be produced (when line counting is enabled) via get-location, which is used as for make-input-port. If get-location is #f, then the port counts lines in the usual way starting from init-pos, independent of locations reported by in.

If count-lines! is supplied, it is called when line counting is enabled for the resulting port. The default is void.
Like `transplant-input-port`, but for output ports.

```scheme
(filter-read-input-port  in
  read-wrap
  peek-wrap
  [close?])  ->  input-port?
in : input-port?
  (or/c exact-nonnegative-integer? eof-object? procedure? evt?))
peek-wrap : (bytes? exact-nonnegative-integer? (or/c evt? #f)
  (or/c exact-nonnegative-integer? eof-object? procedure? evt? #f))
```

```scheme
(transplant-output-port  out
  get-location
  init-pos
  [close?
    count-lines!]
  #:name name)  ->  output-port?

out : output-port?
get-location : (or/c
  (->
    (values
      (or/c exact-positive-integer? #f)
      (or/c exact-nonnegative-integer? #f)
      (or/c exact-positive-integer? #f))
    #f)
  init-pos : exact-positive-integer?
close? : any/c = #t
count-lines! : (-> any) = void
name : (object-name out)
```
close? : any/c = #t

Creates a port that draws from \textit{in}, but each result from the port’s read and peek procedures (in the sense of \texttt{make-input-port}) is filtered by \texttt{read-wrap} and \texttt{peek-wrap}. The filtering procedures each receive both the arguments and results of the read and peek procedures on \textit{in} for each call.

If \textit{close?} is true, then closing the resulting port also closes \textit{in}.

\begin{verbatim}
(special-filter-input-port in proc [close?]) -> input-port?
in : input-port?
proc : (procedure? bytes? . -> . (or/c exact-nonnegative-integer?
  eof-object?
  procedure?
  evt?))

close? : any/c = #t
\end{verbatim}

Produces an input port that is equivalent to \textit{in}, except that when \textit{in} produces a procedure to access a special value, \textit{proc} is applied to the procedure to allow the special value to be replaced with an alternative. The \textit{proc} is called with the special-value procedure and the byte string that was given to the port’s read or peek function (see \texttt{make-input-port}), and the result is used as the read or peek function’s result. The \textit{proc} can modify the byte string to substitute a byte for the special value, but the byte string is guaranteed only to hold at least one byte.

If \textit{close?} is true, then closing the resulting input port also closes \textit{in}.

\textbf{Port Events}

\begin{verbatim}
(eof evt in) -> evt?
in : input-port?
\end{verbatim}

Returns a synchronizable event that is ready when \textit{in} produces an \textit{eof}. If \textit{in} produces a mid-stream \textit{eof}, the \textit{eof} is consumed by the event only if the event is chosen in a synchronization.

If attempting to read from \textit{in} raises an exception during a synchronization attempt, then the exception may be reported during the synchronization attempt, but it will silently discarded if some another event in the same synchronization is selected or if some other event raises an exception first.

Changed in version 7.5.0.3 of package \texttt{base}: Changed handling of read errors so they are propagated to a synchronization attempt, instead of treated as unhandled errors in a background thread.

\begin{verbatim}
(read-bytes-evt k in) -> evt?
k : exact-nonnegative-integer?
in : input-port?
\end{verbatim}

948
Returns a synchronizable event that is ready when \( k \) bytes can be read from \( in \), or when an end-of-file is encountered in \( in \). If \( k \) is 0, then the event is ready immediately with an empty byte string. For non-zero \( k \), if no bytes are available before an end-of-file, the event’s result is `eof`. Otherwise, the event’s result is a byte string of up to \( k \) bytes, which contains as many bytes as are available (up to \( k \)) before an available end-of-file. (The result is a byte string on less than \( k \) bytes only when an end-of-file is encountered.)

Bytes are read from the port if and only if the event is chosen in a synchronization, and the returned bytes always represent contiguous bytes in the port’s stream.

The event can be synchronized multiple times—event concurrently—and each synchronization corresponds to a distinct read request.

The \( in \) must support progress events, and it must not produce a special non-byte value during the read attempt.

Exceptions attempting to read from \( in \) are handled in the same way as by `eof-evt`.

\[
\text{(read-bytes!-evt bstr in)} \rightarrow \text{evt?}
\]
\[
bstr : (\text{and/c bytes? (not/c immutable?)})
\]
\[
in : \text{input-port?}
\]

Like `read-bytes-evt`, except that the read bytes are placed into \( bstr \), and the number of bytes to read corresponds to \( \text{(bytes-length bstr)} \). The event’s result is either `eof` or the number of read bytes.

The \( bstr \) may be mutated any time after the first synchronization attempt on the event and until either the event is selected, a non-#f `progress-evt` is ready, or the current custodian (at the time of synchronization) is shut down. Note that there is no time bound otherwise on when \( bstr \) might be mutated if the event is not selected by a synchronization; nevertheless, multiple synchronization attempts can use the same result from `read-bytes!-evt` as long as there is no intervening read on \( in \) until one of the synchronization attempts selects the event.

Exceptions attempting to read from \( in \) are handled in the same way as by `eof-evt`.

\[
\text{(read-bytes-avail!-evt bstr in)} \rightarrow \text{evt?}
\]
\[
bstr : (\text{and/c bytes? (not/c immutable?)})
\]
\[
in : \text{input-port?}
\]

Like `read-bytes!-evt`, except that the event reads only as many bytes as are immediately available, after at least one byte or one `eof` becomes available.

\[
\text{(read-string-evt k in)} \rightarrow \text{evt?}
\]
\[
k : \text{exact-nonnegative-integer?}
\]
\[
in : \text{input-port?}
\]
Like `read-bytes-evt`, but for character strings instead of byte strings.

```scheme
(read-string!-evt str in) -> evt?
str : (and/c string? (not/c immutable?))
in : input-port?
```

Like `read-bytes!-evt`, but for a character string instead of a byte string.

```scheme
(read-line-evt in [mode]) -> evt?
in : input-port?
mode : (or/c 'linefeed 'return 'return-linefeed 'any 'any-one)
      = 'linefeed
```

Returns a synchronizable event that is ready when a line of characters or end-of-file can be read from `in`. The meaning of `mode` is the same as for `read-line`. The event result is the read line of characters (not including the line separator).

A line is read from the port if and only if the event is chosen in a synchronization, and the returned line always represents contiguous bytes in the port’s stream.

Exceptions attempting to read from `in` are handled in the same way as by `eof-evt`.

```scheme
(read-bytes-line-evt in [mode]) -> evt?
in : input-port?
mode : (or/c 'linefeed 'return 'return-linefeed 'any 'any-one)
      = 'linefeed
```

Like `read-line-evt`, but returns a byte string instead of a string.

```scheme
(peek-bytes-evt k skip progress-evt in) -> evt?
k : exact-nonnegative-integer?
skip : exact-nonnegative-integer?
progress-evt : (or/c progress-evt? #f)
in : input-port?
(peek-bytes!-evt bstr skip progress-evt in) -> evt?
bstr : (and/c bytes? (not/c immutable?))
skip : exact-nonnegative-integer?
progress-evt : (or/c progress-evt? #f)
in : input-port?
(peek-bytes-avail!-evt bstr skip progress-evt in) -> evt?
bstr : (and/c bytes? (not/c immutable?))
skip : exact-nonnegative-integer?
progress-evt : (or/c progress-evt? #f)
in
```

950
Like the read-bytes-evt, etc., functions, but for peeking. The skip argument indicates the number of bytes to skip, and progress-evt indicates an event that effectively cancels the peek (so that the event never becomes ready). The progress-evt argument can be #f, in which case the event is never canceled.

Returns a synchronizable event that is ready when pattern matches the stream of bytes/characters from in; see also regexp-match. The event’s value is the result of the match, in the same form as the result of regexp-match.

If pattern does not require a start-of-stream match, then bytes skipped to complete the match are read and discarded when the event is chosen in a synchronization.

Bytes are read from the port if and only if the event is chosen in a synchronization, and the returned match always represents contiguous bytes in the port’s stream. If not-yet-available bytes from the port might contribute to the match, the event is not ready. Similarly, if pattern begins with a start-of-stream ^ and the pattern does not initially match, then the event cannot become ready until bytes have been read from the port.

The event can be synchronized multiple times—even concurrently—and each synchronization corresponds to a distinct match request.

The in port must support progress events. If in returns a special non-byte value during the match attempt, it is treated like eof.

Exceptions attempting to read from in are handled in the same way as by eof-evt.

Copying Streams

\[
\text{convert-stream from-encoding}
\]
\[
in
\]
\[
to-encoding
\]
\[
out
\]
\[
\rightarrow \text{void?}
\]
from-encoding : string?
in : input-port?
to-encoding : string?
out : output-port?

Reads data from in, converts it using (bytes-open-converter from-encoding to-encoding) and writes the converted bytes to out. The convert-stream procedure returns after reaching eof in in.

If opening the converter fails, the exn:fail exception is raised. Similarly, if a conversion error occurs at any point while reading from in, then exn:fail exception is raised.

(copy-port in out ...+) → void?
in : input-port?
out : output-port?

Reads data from in and writes it back out to out, returning when in produces eof. The copy is efficient, and it is without significant buffer delays (i.e., a byte that becomes available on in is immediately transferred to out, even if future reads on in must block). If in produces a special non-byte value, it is transferred to out using write-special.

This function is often called from a “background” thread to continuously pump data from one stream to another.

If multiple outs are provided, data from in is written to every out. The different outs block output to each other, because each block of data read from in is written completely to one out before moving to the next out. The outs are written in the provided order, so non-blocking ports (e.g., file output ports) should be placed first in the argument list.

### 13.2 Byte and String Input

(read-char [in]) → (or/c char? eof-object?)
in : input-port? = (current-input-port)

Reads a single character from in—which may involve reading several bytes to UTF-8-decode them into a character (see §13.1 “Ports”); a minimal number of bytes are read/peeked to perform the decoding. If no bytes are available before an end-of-file, then eof is returned.

Examples:

> (let ([ip (open-input-string "S2")])
    (print (read-char ip))
    (newline)
    (print (read-char ip))

952
> (let ([ip (open-input-bytes \"316\273\")])
  ; The byte string contains UTF-8-encoded content:
  (print (read-char ip)))
#\l

(read-byte [in]) → (or/c byte? eof-object?)
in : input-port? = (current-input-port)

Reads a single byte from in. If no bytes are available before an end-of-file, then eof is returned.

Examples:

> (let ([ip (open-input-string "a")])
  ; The two values in the following list should be the same.
  (list (read-byte ip) (char->integer #\a)))
'(97 97)
> (let ([ip (open-input-string (string #\λ)])])
  ; This string has a two byte-encoding.
  (list (read-byte ip) (read-byte ip) (read-byte ip)))
'(206 187 #<eof>)

(read-line [in mode]) → (or/c string? eof-object?)
in : input-port? = (current-input-port)
mode : (or/c 'linefeed 'return 'return-linefeed 'any 'any-one)
  = 'linefeed

Returns a string containing the next line of bytes from in.

Characters are read from in until a line separator or an end-of-file is read. The line separator
is not included in the result string (but it is removed from the port’s stream). If no characters
are read before an end-of-file is encountered, eof is returned.

The mode argument determines the line separator(s). It must be one of the following symbols:

- 'linefeed breaks lines on linefeed characters.
- 'return breaks lines on return characters.
• 'return-linefeed breaks lines on return-linefeed combinations. If a return character is not followed by a linefeed character, it is included in the result string; similarly, a linefeed that is not preceded by a return is included in the result string.

• 'any breaks lines on any of a return character, linefeed character, or return-linefeed combination. If a return character is followed by a linefeed character, the two are treated as a combination.

• 'any-one breaks lines on either a return or linefeed character, without recognizing return-linefeed combinations.

Return and linefeed characters are detected after the conversions that are automatically performed when reading a file in text mode. For example, reading a file in text mode on Windows automatically changes return-linefeed combinations to a linefeed. Thus, when a file is opened in text mode, 'linefeed is usually the appropriate read-line mode.

Examples:

> (let ([ip (open-input-string "x\ny\n")])
  (read-line ip))
"x"
> (let ([ip (open-input-string "x\ny\n")])
  (read-line ip 'return))
"x\ny\n"
> (let ([ip (open-input-string "x\ry\r")])
  (read-line ip 'return))
"x"
> (let ([ip (open-input-string "x\r\ny\r\n")])
  (read-line ip 'return-linefeed))
"x"
> (let ([ip (open-input-string "x\r\ny\rz")])
  (list (read-line ip 'any) (read-line ip 'any)))
'("x" "y")
> (let ([ip (open-input-string "x\r\ny\nz")])
  (list (read-line ip 'any-one) (read-line ip 'any-one)))
'("x" "")

(read-bytes-line [in mode]) → (or/c bytes? eof-object?)
in : input-port? = (current-input-port)
mode : (or/c 'linefeed 'return 'return-linefeed 'any 'any-one)
  = 'linefeed

Like read-line, but reads bytes and produces a byte string.

(read-string amt [in]) → (or/c string? eof-object?)
amt : exact-nonnegative-integer?
in : input-port? = (current-input-port)
To read an entire port as a string, use \texttt{port->string}.

Returns a string containing the next \texttt{amt} characters from \texttt{in}.

If \texttt{amt} is 0, then the empty string is returned. Otherwise, if fewer than \texttt{amt} characters are available before an end-of-file is encountered, then the returned string will contain only those characters before the end-of-file; that is, the returned string’s length will be less than \texttt{ amt}. (A temporary string of size \texttt{amt} is allocated while reading the input, even if the size of the result is less than \texttt{amt} characters.) If no characters are available before an end-of-file, then \texttt{eof} is returned.

If an error occurs during reading, some characters may be lost; that is, if \texttt{read-string} successfully reads some characters before encountering an error, the characters are dropped.

Example:

\begin{verbatim}
> (let ([ip (open-input-string "supercalifragilisticexpialidocious")])
 (read-string 5 ip))
"super"
\end{verbatim}

\texttt{(read-bytes \texttt{amt} \texttt{[in]}) \rightarrow (or/c \texttt{bytes?} \texttt{eof-object?})}
\texttt{\texttt{amt} : \texttt{exact-nonnegative-integer?}}
\texttt{\texttt{in} : \texttt{input-port?} = (current-input-port)}

Like \texttt{read-string}, but reads bytes and produces a byte string.

Example:

\begin{verbatim}
> (let ([ip (open-input-bytes
 (bytes 6
 115 101 99 114 101
 116))])
 (define length (read-byte ip))
 (bytes->string/utf-8 (read-bytes length ip)))
"secret"
\end{verbatim}

\texttt{(read-string! \texttt{str} \texttt{in \texttt{start-pos} \texttt{end-pos}})}
\texttt{\rightarrow (or/c \texttt{exact-nonnegative-integer?} \texttt{eof-object?})}
\texttt{\texttt{str} : (and/c \texttt{string?} (not/c immutable?))}
\texttt{\texttt{in} : \texttt{input-port?} = (current-input-port)}
\texttt{\texttt{start-pos} : \texttt{exact-nonnegative-integer?} = 0}
\texttt{\texttt{end-pos} : \texttt{exact-nonnegative-integer?} = (string-length \texttt{str})}

Reads characters from \texttt{in} like \texttt{read-string}, but puts them into \texttt{str} starting from index \texttt{start-pos} (inclusive) up to \texttt{end-pos} (exclusive). Like \texttt{substring}, the
exn:fail:contract exception is raised if start-pos or end-pos is out-of-range for str.

If the difference between start-pos and end-pos is 0, then 0 is returned and str is not modified. If no bytes are available before an end-of-file, then eof is returned. Otherwise, the return value is the number of characters read. If m characters are read and m<end-pos-start-pos, then str is not modified at indices start-pos+m through end-pos.

Example:

```
> (let ([buffer (make-string 10 #\_)]
      [ip (open-input-string "cketRa")])
  (printf "\n" buffer)
  (read-string! buffer ip 2 6)
  (printf "\n" buffer)
  (read-string! buffer ip 0 2)
  (printf "\n" buffer))
"__________"
"__cket____"
"Racket____"
```

(read-bytes! bstr [in start-pos end-pos])
→ (or/c exact-nonnegative-integer? eof-object?)

bstr : bytes?
in : input-port? = (current-input-port)
start-pos : exact-nonnegative-integer? = 0
end-pos : exact-nonnegative-integer? = (bytes-length bstr)

Like read-string!, but reads bytes, puts them into a byte string, and returns the number of bytes read.

Example:

```
> (let ([buffer (make-bytes 10 (char->integer #\_)])
      [ip (open-input-string "cketRa")])
  (printf "\n" buffer)
  (read-bytes! buffer ip 2 6)
  (printf "\n" buffer)
  (read-bytes! buffer ip 0 2)
  (printf "\n" buffer))
"__________"
"__cket____"
"Racket____"
```

(read-bytes-avail! bstr [in start-pos end-pos])
→ (or/c exact-nonnegative-integer? eof-object? procedure?)
Like `read-bytes-avail!`, but returns without blocking after having read the immediately available bytes, and it may return a procedure for a “special” result. The `read-bytes-avail!` procedure blocks only if no bytes (or specials) are yet available. Also unlike `read-bytes!`, `read-bytes-avail!` never drops bytes; if `read-bytes-avail!` successfully reads some bytes and then encounters an error, it suppresses the error (treating it roughly like an end-of-file) and returns the read bytes. (The error will be triggered by future reads.) If an error is encountered before any bytes have been read, an exception is raised.

When `in` produces a special value, as described in §13.1.9 “Custom Ports”, the result is a procedure of four arguments. The four arguments correspond to the location of the special value within the port, as described in §13.1.9 “Custom Ports”. If the procedure is called more than once with valid arguments, the `exn:fail:contract` exception is raised. If `read-bytes-avail!` returns a special-producing procedure, then it does not place characters in `bstr`. Similarly, `read-bytes-avail!` places only as many bytes into `bstr` as are available before a special value in the port’s stream.

```scheme
(read-bytes-avail!* bstr
   [in start-pos end-pos])
→ (or/c exact-nonnegative-integer? eof-object? procedure?)
```

Like `read-bytes-avail!`, but returns 0 immediately if no bytes (or specials) are available for reading and the end-of-file is not reached.

```scheme
(read-bytes-avail!/enable-break bstr
   [in start-pos end-pos])
→ (or/c exact-nonnegative-integer? eof-object? procedure?)
```

Like `read-bytes-avail!`, but breaks are enabled during the read (see also §10.6 “Breaks”). If breaking is disabled when `read-bytes-avail!/enable-break` is called,
and if the `exn:break` exception is raised as a result of the call, then no bytes will have been read from `in`.

```scheme
(peek-string amt skip-bytes-amt [in]) → (or/c string? eof-object?)
amt : exact-nonnegative-integer?
skip-bytes-amt : exact-nonnegative-integer?
in : input-port? = (current-input-port)
```

Similar to `read-string`, except that the returned characters are peeked: preserved in the port for future reads and peeks. (More precisely, undecoded bytes are left for future reads and peeks.) The `skip-bytes-amt` argument indicates a number of bytes (not characters) in the input stream to skip before collecting characters to return; thus, in total, the next `skip-bytes-amt` bytes plus `amt` characters are inspected.

For most kinds of ports, inspecting `skip-bytes-amt` bytes and `amt` characters requires at least `skip-bytes-amt + amt` bytes of memory overhead associated with the port, at least until the bytes/characters are read. No such overhead is required when peeking into a string port (see §13.1.6 “String Ports”), a pipe port (see §13.1.7 “Pipes”), or a custom port with a specific peek procedure (depending on how the peek procedure is implemented; see §13.1.9 “Custom Ports”).

If a port produces `eof` mid-stream, attempts to skip beyond the `eof` for a peek always produce `eof` until the `eof` is read.

```scheme
(peek-bytes amt skip-bytes-amt [in]) → (or/c bytes? eof-object?)
amt : exact-nonnegative-integer?
skip-bytes-amt : exact-nonnegative-integer?
in : input-port? = (current-input-port)
```

Like `peek-string`, but peeks bytes and produces a byte string.

```scheme
(peek-string! str skip-bytes-amt
           [in
              start-pos end-pos])
→ (or/c exact-nonnegative-integer? eof-object?)
str : (and/c string? (not/c immutable?))
skip-bytes-amt : exact-nonnegative-integer?
in : input-port? = (current-input-port)
start-pos : exact-nonnegative-integer? = 0
end-pos : exact-nonnegative-integer? = (string-length str)
```

Like `read-string!`, but for peeking, and with a `skip-bytes-amt` argument like `peek-string`. 
Like `peek-string!`, but peeks bytes, puts them into a byte string, and returns the number of bytes read.

Like `read-bytes-avail!`, but for peeking, and with two extra arguments. The `skip-bytes-amt` argument is as in `peek-bytes`. The `progress` argument must be either `#f` or an event produced by `port-progress-evt` for `in`.

To peek, `peek-bytes-avail!` blocks until finding an end-of-file, at least one byte (or special) past the skipped bytes, or until a non-`#f` `progress` becomes ready. Furthermore, if `progress` is ready before bytes are peeked, no bytes are peeked or skipped, and `progress` may cut short the skipping process if it becomes available during the peek attempt. Furthermore, `progress` is checked even before determining whether the port is still open.

The result of `peek-bytes-avail!` is 0 only in the case that `progress` becomes ready before bytes are peeked.

```scheme
(peek-bytes! bstr
    skip-bytes-amt
    [in
      start-pos
      end-pos])
→ (or/c exact-nonnegative-integer? eof-object?)
  bstr : (and/c bytes? (not/c immutable?))
  skip-bytes-amt : exact-nonnegative-integer?
  in : input-port? = (current-input-port)
  start-pos : exact-nonnegative-integer? = 0
  end-pos : exact-nonnegative-integer? = (bytes-length bstr)
```

```scheme
(peek-bytes-avail! bstr
    skip-bytes-amt
    [progress
      in
      start-pos
      end-pos])
→ (or/c exact-nonnegative-integer? eof-object? procedure?)
  bstr : (and/c bytes? (not/c immutable?))
  skip-bytes-amt : exact-nonnegative-integer?
  progress : (or/c progress-evt? #f) = #f
  in : input-port? = (current-input-port)
  start-pos : exact-nonnegative-integer? = 0
  end-pos : exact-nonnegative-integer? = (bytes-length bstr)
```

959
Like `read-bytes-avail!*`, but for peeking, and with `skip-bytes-amt` and `progress` arguments like `peek-bytes-avail!`. Since this procedure never blocks, it may return before even `skip-bytes-amt` bytes are available from the port.

Like `read-bytes-avail!/enable-break`, but for peeking, and with `skip-bytes-amt` and `progress` arguments like `peek-bytes-avail!`.

Like `read-char`, but if the input port returns a special value (through a value-generating procedure in a custom port, where `source-name` is provided to the procedure; see §13.1.9)
“Custom Ports” and §13.7.3 “Special Comments” for details), then the result of applying 
\textit{special-wrap} to the special value is returned. A \texttt{#f} value for \textit{special-wrap} is treated 
the same as the identity function.

Changed in version 6.8.0.2 of package base: Added the \texttt{special-wrap} and \texttt{source-name} arguments.

\begin{verbatim}
(read-byte-or-special [in
  special-wrap
  source-name])
→ (or/c byte? eof-object? any/c)
in : input-port? = (current-input-port)
special-wrap : (or/c (any/c -> any/c) #f) = #f
source-name : any/c = #f
\end{verbatim}

Like \texttt{read-char-or-special}, but reads and returns a byte instead of a character.

Changed in version 6.8.0.2 of package base: Added the \texttt{special-wrap} and \texttt{source-name} arguments.

\begin{verbatim}
(peek-char [in skip-bytes-amt]) → (or/c char? eof-object?)
in : input-port? = (current-input-port)
skip-bytes-amt : exact-nonnegative-integer? = 0
\end{verbatim}

Like \texttt{read-char}, but peeks instead of reading, and skips \texttt{skip-bytes-amt} bytes (not char-
acters) at the start of the port.

\begin{verbatim}
(peek-byte [in skip-bytes-amt]) → (or/c byte? eof-object?)
in : input-port? = (current-input-port)
skip-bytes-amt : exact-nonnegative-integer? = 0
\end{verbatim}

Like \texttt{peek-char}, but peeks and returns a byte instead of a character.

\begin{verbatim}
(peek-char-or-special [in
  skip-bytes-amt
  special-wrap
  source-name])
→ (or/c char? eof-object? any/c)
in : input-port? = (current-input-port)
skip-bytes-amt : exact-nonnegative-integer? = 0
special-wrap : (or/c (any/c -> any/c) #f 'special) = #f
source-name : any/c = #f
\end{verbatim}

Like \texttt{peek-char}, but if the input port returns a non-byte value after \texttt{skip-bytes-amt} byte 
positions, then the result depends on \texttt{special-wrap}:

\begin{itemize}
  \item If \texttt{special-wrap} is \texttt{#f}, then the special value is returned (as for \texttt{read-char-or-
special}).
\end{itemize}
• If `special-wrap` is a procedure, then it is applied the special value to produce the result (as for `read-char-or-special`).

• If `special-wrap` is `special`, then `special` is returned in place of the special value—without calling the special-value procedure that is returned by the input-port implementation.

Changed in version 6.8.0.2 of package `base`: Added the `special-wrap` and `source-name` arguments.

Changed in version 6.90.0.16: Added `special` as an option for `special-wrap`.

```
(peek-byte-or-special [in
  skip-bytes-amt
  progress
  special-wrap
  source-name])
```

→ `(or/c byte? eof-object? any/c)

Like `peek-char-or-special`, but peeks and returns a byte instead of a character, and it supports a `progress` argument like `peek-bytes-avail!`.

Changed in version 6.8.0.2 of package `base`: Added the `special-wrap` and `source-name` arguments.

Changed in version 6.90.0.16: Added `special` as an option for `special-wrap`.

```
(port-progress-evts [in]) → progress-evt?
```

Returns a synchronizable event (see §11.2.1 “Events”) that becomes ready for synchronization after any subsequent read from `in` or after `in` is closed. After the event becomes ready, it remains ready. The synchronization result of a progress event is the progress event itself.

```
(port-provides-progress-evts? in) → boolean
```

Returns `#t` if `port-progress-evts` can return an event for `in`. All built-in kinds of ports support progress events, but ports created with `make-input-port` (see §13.1.9 “Custom Ports”) may not.
Attempts to commit as read the first \textit{amt} previously peeked bytes, non-byte specials, and \texttt{eofs} from \texttt{in}, or the first \texttt{eof} or special value peeked from \texttt{in}. Mid-stream \texttt{eofs} can be committed, but an \texttt{eof} when the port is exhausted does not necessarily commit, since it does not correspond to data in the stream.

The read commits only if \texttt{progress} does not become ready first (i.e., if no other process reads from \texttt{in} first), and only if \texttt{evt} is chosen by a \texttt{sync} within \texttt{port-commit-peeked} (in which case the event result is ignored); the \texttt{evt} must be either a channel-put event, channel, semaphore, semaphore-peek event, always event, or never event. Suspending the thread that calls \texttt{port-commit-peeked} may or may not prevent the commit from proceeding.

The result from \texttt{port-commit-peeked} is \#t if data has been committed, and \#f otherwise.

If no data has been peeked from \texttt{in} and \texttt{progress} is not ready, then \texttt{exn:fail:contract} exception is raised. If fewer than \texttt{amt} items have been peeked at the current start of \texttt{in}’s stream, then only the peeked items are committed as read. If \texttt{in}’s stream currently starts at an \texttt{eof} or a non-byte special value, then only the \texttt{eof} or special value is committed as read.

If \texttt{progress} is not a result of \texttt{port-progress-evt} applied to \texttt{in}, then \texttt{exn:fail:contract} exception is raised.

\begin{verbatim}
(byte-ready? [in]) → boolean?
  in : input-port? = (current-input-port)
\end{verbatim}

Returns \#t if \texttt{(read-byte in)} would not block (at the time that \texttt{byte-ready?} was called, at least). Equivalent to \texttt{(and (sync/timeout 0 in) \#t)}.

The \texttt{byte-ready?} and \texttt{char-ready?} functions are appropriate for relatively few applications, because ports are meant to support streaming data among concurrent producers and consumers; the fact that a byte or character is not ready in some instant does not necessarily mean that the producer is finished supplying data. (Also, if a port has multiple consumers, data might get consumed between the time that a given process uses \texttt{byte-ready?} to poll the port and the time that it reads data from the port.) Using \texttt{byte-ready?} makes sense if you are implementing your own scheduler or if you know that the port’s implementation and use are particularly constrained.

\begin{verbatim}
(char-ready? [in]) → boolean?
  in : input-port? = (current-input-port)
\end{verbatim}

Returns \#t if \texttt{(read-char in)} would not block (at the time that \texttt{char-ready?} was called, at least). Depending on the initial bytes of the stream, multiple bytes may be needed to form a UTF-8 encoding.

See \texttt{byte-ready?} for a note on how \texttt{byte-ready?} and \texttt{char-ready?} are rarely the right choice.

\begin{verbatim}
(progress-evt? v) → boolean?
\end{verbatim}
v : any/c
(progress-evt? evt in) → boolean?
ev : progress-evt?
in : input-port?

With one argument, returns #t if v is a progress evt for some input port, #f otherwise.

With two arguments, returns #t if evt is a progress event for in, #f otherwise.

13.3 Byte and String Output

(write-char char [out]) → void?
char : char?
out : output-port? = (current-output-port)

Writes a single character to out; more precisely, the bytes that are the UTF-8 encoding of char are written to out.

(write-byte byte [out]) → void?
byte : byte?
out : output-port? = (current-output-port)

Writes a single byte to out.

(newline [out]) → void?
out : output-port? = (current-output-port)

The same as (write-char #\newline out).

(write-string str [out start-pos end-pos])
→ exact-nonnegative-integer?
str : string?
out : output-port? = (current-output-port)
start-pos : exact-nonnegative-integer? = 0
end-pos : exact-nonnegative-integer? = (string-length str)

Writes characters to out from str starting from index start-pos (inclusive) up to end-pos (exclusive). Like substring, the exn:fail:contract exception is raised if start-pos or end-pos is out-of-range for str.

The result is the number of characters written to out, which is always (- end-pos start-pos).

964
If `str` is mutable, mutations after `write-string` returns do not affect the characters written to `out`. (This independence from mutation is not a special property of `write-string`, but instead generally true of output functions.)

```scheme
(write-bytes bstr [out start-pos end-pos])
→ exact-nonnegative-integer?
bstr : bytes?
out : output-port? = (current-output-port)
start-pos : exact-nonnegative-integer? = 0
end-pos : exact-nonnegative-integer? = (bytes-length bstr)
```

Like `write-string`, but writes bytes instead of characters.

```scheme
(write-bytes_avail bstr [out start-pos end-pos])
→ exact-nonnegative-integer?
bstr : bytes?
out : output-port? = (current-output-port)
start-pos : exact-nonnegative-integer? = 0
end-pos : exact-nonnegative-integer? = (bytes-length bstr)
```

Like `write-bytes`, but returns without blocking after writing as many bytes as it can immediately flush. It blocks only if no bytes can be flushed immediately. The result is the number of bytes written and flushed to `out`; if `start-pos` is the same as `end-pos`, then the result can be 0 (indicating a successful flush of any buffered data), otherwise the result is between 1 and `(- end-pos start-pos)`, inclusive.

The `write-bytes-avail` procedure never drops bytes; if `write-bytes-avail` successfully writes some bytes and then encounters an error, it suppresses the error and returns the number of written bytes. (The error will be triggered by future writes.) If an error is encountered before any bytes have been written, an exception is raised.

```scheme
(write-bytes-avail* bstr [out start-pos end-pos])
→ (or/c exact-nonnegative-integer? #f)
bstr : bytes?
out : output-port? = (current-output-port)
start-pos : exact-nonnegative-integer? = 0
end-pos : exact-nonnegative-integer? = (bytes-length bstr)
```

Like `write-bytes-avail`, but never blocks, returns `#f` if the port contains buffered data that cannot be written immediately, and returns 0 if the port’s internal buffer (if any) is flushed but no additional bytes can be written immediately.
Like `write-bytes-avail`, except that breaks are enabled during the write. The procedure
provides a guarantee about the interaction of writing and breaks: if breaking is disabled
when `write-bytes-avail/enable-break` is called, and if the `exn:break` exception is
raised as a result of the call, then no bytes will have been written to `out`. See also §10.6
“Breaks”.

\[
\text{(write-special avail*) } v \ [\text{out}] \rightarrow \text{boolean?}
\]
\[
\begin{align*}
  v &: \text{any/c} \\
  \text{out} &: \text{output-port?} = (\text{current-output-port})
\end{align*}
\]

Writes \( v \) directly to \( out \) if the port supports special writes, or raises `exn:fail:contract`
if the port does not support special write. The result is always \#t, indicating that the write
succeeded.

Similar to `write-bytes-avail`, but instead of writing bytes immediately, it returns a syn-
chronizable event (see §11.2.1 “Events”). The `out` must support atomic writes, as indicated
by `port-writes-atomic?`.

Synchronizing on the object starts a write from `bstr`, and the event becomes ready when
bytes are written (unbuffered) to the port. If `start-pos` and `end-pos` are the same, then
the synchronization result is 0 when the port’s internal buffer (if any) is flushed, otherwise the result is a positive exact integer. If the event is not selected in a synchronization, then no bytes will have been written to out.

```
(write-special-evt v [out]) → evt?
  v : any/c
  out : output-port? = (current-output-port)
```

Similar to write-special, but instead of writing the special value immediately, it returns a synchronizable event (see §11.2.1 “Events”). The out must support atomic writes, as indicated by port-writes-atomic?.

Synchronizing on the object starts a write of the special value, and the event becomes ready when the value is written (unbuffered) to the port. If the event is not selected in a synchronization, then no value will have been written to out.

```
(port-writes-atomic? out) → boolean?
  out : output-port?
```

Returns #t if write-bytes-avail/enable-break can provide an exclusive-or guarantee (break or write, but not both) for out, and if the port can be used with procedures like write-bytes-avail-evt. Racket’s file-stream ports, pipes, string ports, and TCP ports all support atomic writes; ports created with make-output-port (see §13.1.9 “Custom Ports”) may support atomic writes.

```
(port-writes-special? out) → boolean?
  out : output-port?
```

Returns #t if procedures like write-special can write arbitrary values to the port. Racket’s file-stream ports, pipes, string ports, and TCP ports all reject special values, but ports created with make-output-port (see §13.1.9 “Custom Ports”) may support them.

### 13.4 Reading

```
(read [in]) → any
  in : input-port? = (current-input-port)
```

Reads and returns a single datum from in. If in has a handler associated to it via port-read-handler, then the handler is called. Otherwise, the default reader is used, as parameterized by the current-readtable parameter, as well as many other parameters.

See §1.3 “The Reader” for information on the default reader and §1.3.18 “Reading via an Extension” for the protocol of read.
(read-syntax [source-name in] → (or/c syntax? eof-object?))
  source-name : any/c = (object-name (current-input-port))
  in : input-port? = (current-input-port)

Like read, but produces a syntax object with source-location information. The source-name is used as the source field of the syntax object; it can be an arbitrary value, but it should generally be a path for the source file.

See §1.3 “The Reader” for information on the default reader in read-syntax mode and §1.3.18 “Reading via an Extension” for the protocol of read-syntax.

(read/recursive [in start readtable graph?]) → any
  in : input-port? = (current-input-port)
  start : (or/c char? #f) = #f
  readtable : (or/c readtable? #f) = (current-readtable)
  graph? : any/c = #t

Similar to calling read, but normally used during the dynamic extent of read within a reader-extension procedure (see §13.7.2 “Reader-Extension Procedures”). The main effect of using read/recursive instead of read is that graph-structure annotations (see §1.3.17 “Reading Graph Structure”) in the nested read are considered part of the overall read, at least when the graph? argument is true; since the result is wrapped in a placeholder, however, it is not directly inspectable.

If start is provided and not #f, it is effectively prefixed to the beginning of in’s stream for the read. (To prefix multiple characters, use input-port-append.)

The readtable argument is used for top-level parsing to satisfy the read request, including various delimiters of a built-in top-level form (such as parentheses and . for reading a hash table); recursive parsing within the read (e.g., to read the elements of a list) instead uses the current readtable as determined by the current-readtable parameter. A reader macro might call read/recursive with a character and readtable to effectively invoke the readtable’s behavior for the character. If readtable is #f, the default readtable is used for top-level parsing.

When graph? is #f, graph structure annotations in the read datum are local to the datum.

When called within the dynamic extent of read, the read/recursive procedure can produce a special-comment value (see §13.7.3 “Special Comments”) when the input stream’s first non-whitespace content parses as a comment.

See §13.7.1 “Readtables” for an extended example that uses read/recursive.

Changed in version 6.2 of package base: Adjusted use of readtable to more consistently apply to the delimiters of a built-in form.
Analogous to calling `read/recursive`, but the resulting value encapsulates S-expression structure with source-location information. As with `read/recursive`, when `read-syntax/recursive` is used within the dynamic extent of `read-syntax`, the result from `read-syntax/recursive` is either a special-comment value, end-of-file, or opaque graph-structure placeholder (not a syntax object). The placeholder can be embedded in an S-expression or syntax object returned by a reader macro, etc., and it will be replaced with the actual syntax object before the outermost `read-syntax` returns.

Using `read/recursive` within the dynamic extent of `read-syntax` does not allow graph structure for reading to be included in the outer `read-syntax` parsing, and neither does using `read-syntax/recursive` within the dynamic extent of `read`. In those cases, `read/recursive` and `read-syntax/recursive` produce results like `read` and `read-syntax`, except that a special-comment value is returned when the input stream starts with a comment (after whitespace).

See §13.7.1 "Readtables" for an extended example that uses `read-syntax/recursive`.

Changed in version 6.2 of package `base`: Adjusted use of `readtable` in the same way as for `read/recursive`.

```
(read-language [in fail-thunk])
  → (or/c (any/c any/c . -> . any) #f)
  in : input-port? = (current-input-port)
  fail-thunk : (-> any) = (lambda () (error ...))
```

Reads from `in` in the same way as `read`, but stopping as soon as a reader language (or its absence) is determined, and using the current namespace to load a reader module instead of its root namespace (if those are different).

A reader language is specified by `#lang` or `#!` (see §1.3.18 "Reading via an Extension") at the beginning of the input, though possibly after comment forms. The default readable is used by `read-language` (instead of the value of `current-readtable`), and `#reader` forms (which might produce comments) are not allowed before `#lang` or `#!`.

When it finds a `#lang` or `#!` specification, instead of dispatching to a `read` or `read-syntax` function as `read` and `read-syntax` do, `read-language` dispatches to the `get-info` function (if any) exported by the same module. The arguments to `get-info` are the same as

See also §17.3.5 “Source-Handling Configuration” in *The Racket Guide.*
for read as described in §1.3.18 “Reading via an Extension”. The result of the get-info function is the result of read-language if it is a function of two arguments; if get-info produces any other kind of result, the exn:fail:contract exception is raised. If no get-info function is exported, read-language returns #f.

The function produced by get-info reflects information about the expected syntax of the input stream. The first argument to the function serves as a key on such information; acceptable keys and the interpretation of results is up to external tools, such as DrRacket (see the DrRacket documentation). If no information is available for a given key, the result should be the second argument.

Examples:

> (define scribble-manual-info
  (read-language (open-input-string "#lang scribble/manual")))
> (scribble-manual-info 'color-lexer #f)
#proc:scribble-inside-lexer>
> (scribble-manual-info 'something-else #f)
#f

The get-info function itself is applied to five arguments: the input port being read, the module path from which the get-info function was extracted, and the source line (positive exact integer or #f), column (non-negative exact integer or #f), and position (positive exact integer or #f) of the start of the #lang or #! form. The get-info function may further read from the given input port to determine its result, but it should read no further than necessary. The get-info function should not read from the port after returning a function.

If in starts with a reader language specification but the relevant module does not export get-info (but perhaps does export read and read-syntax), then the result of read-language is #f.

If in has a #lang or #! specification, but parsing and resolving the specification raises an exception, the exception is propagated by read-language. Having at least #l or #! (after comments and whitespace) counts as starting a #lang or #! specification.

If in does not specify a reader language with #lang or #!, then fail-thunk is called. The default fail-thunk raises exn:fail:read or exn:fail:read:eof.

(read-case-sensitive) → boolean?
(read-case-sensitive on?) → void?
  on? : any/c

A parameter that controls parsing and printing of symbols. When this parameter’s value is #f, the reader case-folds symbols (e.g., producing ’hi when the input is any one of hi, Hi, HI, or hI). The parameter also affects the way that write prints symbols containing uppercase characters; if the parameter’s value is #f, then symbols are printed with uppercase characters quoted by a \ or |. The parameter’s value is overridden by quoting \ or |
vertical-bar quotes and the `#cs` and `#ci` prefixes; see §1.3.2 “Reading Symbols” for more information. While a module is loaded, the parameter is set to `#t` (see `current-load`).

```
(read-square-bracket-as-paren) → boolean?
(read-square-bracket-as-paren on?) → void?
  on? : any/c
```

A parameter that controls whether `[` and `]` are treated as parentheses. See §1.3.6 “Reading Pairs and Lists” for more information.

```
(read-curly-brace-as-paren) → boolean?
(read-curly-brace-as-paren on?) → void?
  on? : any/c
```

A parameter that controls whether `{` and `}` are treated as parentheses. See §1.3.6 “Reading Pairs and Lists” for more information.

```
(read-square-bracket-with-tag) → boolean?
(read-square-bracket-with-tag on?) → void?
  on? : any/c
```

A parameter that controls whether `[` and `]` are treated as parentheses, but the resulting list tagged with `#%brackets`. See §1.3.6 “Reading Pairs and Lists” for more information.

Added in version 6.3.0.5 of package `base`.

```
(read-curly-brace-with-tag) → boolean?
(read-curly-brace-with-tag on?) → void?
  on? : any/c
```

A parameter that controls whether `{` and `}` are treated as parentheses, but the resulting list tagged with `#%braces`. See §1.3.6 “Reading Pairs and Lists” for more information.

Added in version 6.3.0.5 of package `base`.

```
(read-accept-box) → boolean?
(read-accept-box on?) → void?
  on? : any/c
```

A parameter that controls parsing `#&` input. See §1.3.13 “Reading Boxes” for more information.
A parameter that controls parsing #\sim compiled input. See \cite{1.3:Reader} and \texttt{current-compile} for more information.

\begin{verbatim}
(\texttt{read-accept-bar-quote}) \rightarrow \texttt{boolean}?
(\texttt{read-accept-bar-quote on?}) \rightarrow \texttt{void}?
  \texttt{on?} : \texttt{any/c}
\end{verbatim}

A parameter that controls parsing and printing of | in symbols. See \cite{1.3.2:Symbols} and \cite{1.4:Printer} for more information.

\begin{verbatim}
(\texttt{read-accept-graph}) \rightarrow \texttt{boolean}?
(\texttt{read-accept-graph on?}) \rightarrow \texttt{void}?
  \texttt{on?} : \texttt{any/c}
\end{verbatim}

A parameter value that controls parsing input with sharing. See \cite{1.3.17:GraphStructure} for more information.

\begin{verbatim}
(\texttt{read-decimal-as-inexact}) \rightarrow \texttt{boolean}?
(\texttt{read-decimal-as-inexact on?}) \rightarrow \texttt{void}?
  \texttt{on?} : \texttt{any/c}
\end{verbatim}

A parameter that controls parsing input numbers with a decimal point or exponent (but no explicit exactness tag). See \cite{1.3.3:Numbers} for more information.

\begin{verbatim}
(\texttt{read-single-flonum}) \rightarrow \texttt{boolean}?
(\texttt{read-single-flonum on?}) \rightarrow \texttt{void}?
  \texttt{on?} : \texttt{any/c}
\end{verbatim}

A parameter that controls parsing input numbers that have a \texttt{f}, \texttt{F}, \texttt{s}, or \texttt{S} precision character. See \cite{1.3.3:Numbers} for more information.

Added in version 7.3.0.5 of package \texttt{base}.

\begin{verbatim}
(\texttt{read-accept-dot}) \rightarrow \texttt{boolean}?
(\texttt{read-accept-dot on?}) \rightarrow \texttt{void}?
  \texttt{on?} : \texttt{any/c}
\end{verbatim}

A parameter that controls parsing input with a dot, which is normally used for literal cons cells. See \cite{1.3.6:PairsLists} for more information.

\begin{verbatim}
(\texttt{read-accept-infix-dot}) \rightarrow \texttt{boolean}?
(\texttt{read-accept-infix-dot on?}) \rightarrow \texttt{void}?
  \texttt{on?} : \texttt{any/c}
\end{verbatim}

A parameter that controls parsing input with two dots to trigger infix conversion. See \cite{1.3.6:PairsLists} for more information.
(read-cdot) → boolean?
(read-cdot on?) → void?
  on? : any/c

A parameter that controls parsing input with a dot, in a C structure accessor style. See §1.3.19 “Reading with C-style Infix-Dot Notation” for more information.

Added in version 6.3.0.5 of package base.

(read-accept-quasiquote) → boolean?
(read-accept-quasiquote on?) → void?
  on? : any/c

A parameter that controls parsing input with `#\ or `, which is normally used for quasiquote, unquote, and unquote-splicing abbreviations. See §1.3.8 “Reading Quotes” for more information.

(read-accept-reader) → boolean?
(read-accept-reader on?) → void?
  on? : any/c

A parameter that controls whether `#reader`, `#lang`, or `#!` are allowed for selecting a parser. See §1.3.18 “Reading via an Extension” for more information.

(read-accept-lang) → boolean?
(read-accept-lang on?) → void?
  on? : any/c

A parameter that (along with read-accept-reader) controls whether `#lang` and `#!` are allowed for selecting a parser. See §1.3.18 “Reading via an Extension” for more information.

(current-readtable) → (or/c readtable? #f)
(current-readtable readtable) → void?
  readtable : (or/c readtable? #f)

A parameter whose value determines a readtable that adjusts the parsing of S-expression input, where `#f` implies the default behavior. See §13.7.1 “Readtables” for more information.

(call-with-default-reading-parameterization thunk) → any
  thunk : (-> any)

Calls `thunk` in tail position of a parameterize to set all reader parameters above to their default values.
Using the default parameter values ensures consistency, and it also provides safety when reading from untrusted sources, since the default values disable evaluation of arbitrary code via `#lang` or `#reader`.

```
| (current-reader-guard)     → (any/c . -> . any) |
| (current-reader-guard proc) → void?              |
| proc : (any/c . -> . any)                       |
```

A parameter whose value converts or rejects (by raising an exception) a module-path datum following `#reader`. See §1.3.18 “Reading via an Extension” for more information.

```
| (read-on-demand-source)     → (or/c #f #t (and/c path? complete-path?)) |
| (read-on-demand-source mode) → void?                          |
| mode : (or/c #f #t (and/c path? complete-path?))             |
```

A parameter that enables lazy parsing of compiled code, so that closure bodies and syntax objects are extracted (and validated) from marshaled compiled code on demand. Normally, this parameter is set by the default load handler when `load-on-demand-enabled` is `#t`.

A `#f` value for `read-on-demand-source` disables lazy parsing of compiled code. A `#t` value enables lazy parsing. A path value furthers enable lazy retrieval from disk—instead of keeping unparsed compiled code in memory—when the `PLT_DELAY_FROM_ZO` environment variable is set (to any value) on start-up.

If the file at `mode` as a path changes before the delayed code is parsed when lazy retrieval from disk is enabled, then the on-demand parse most likely will encounter garbage, leading to an exception.

```
| (port-read-handler in)     → (case->                          |
|                            (input-port? . -> . any)             |
|                            (input-port? any/c . -> . any))     |

in : input-port?

(port-read-handler in proc) → void?

in : input-port?

proc : (case->                     |
                            (input-port? . -> . any)             |
                            (input-port? any/c . -> . any))     |
```

Gets or sets the `port read handler` for `in`. The handler called to read from the port when the built-in `read` or `read-syntax` procedure is applied to the port. (The port read handler is not used for `read/recursive` or `read-syntax/recursive`.)

A port read handler is applied to either one argument or two arguments:

- A single argument is supplied when the port is used with `read`; the argument is the
port being read. The return value is the value that was read from the port (or end-of-
file).

- Two arguments are supplied when the port is used with read-syntax; the first argu-
ment is the port being read, and the second argument is a value indicating the source.
The return value is a syntax object that was read from the port (or end-of-file).

The default port read handler reads standard Racket expressions with Racket’s built-in parser
(see §1.3 “The Reader”). It handles a special result from a custom input port (see make-
custom-input-port) by treating it as a single expression, except that special-comment
values (see §13.7.3 “Special Comments”) are treated as whitespace.

The default port read handler itself can be customized through a readtable; see §13.7.1
“Readtables” for more information.

13.5 Writing

\[(\text{write } \text{datum} [\text{out}]) \rightarrow \text{void}\]
| datum : any/c |
| out : output-port? = (current-output-port) |

Writes datum to out, normally in such a way that instances of core datatypes can be read
back in. If out has a handler associated to it via port-write-handler, then the handler
is called. Otherwise, the default printer is used (in write mode), as configured by various
parameters.

See §1.4 “The Printer” for more information about the default printer. In particular, note that
write may require memory proportional to the depth of the value being printed, due to the
initial cycle check.

Examples:

> (write 'hi)
hi
> (write (lambda (n) n))
#<procedure>
> (define o (open-output-string))
> (write "hello" o)
> (get-output-string o)
"\"hello\"

\[(\text{display } \text{datum} [\text{out}]) \rightarrow \text{void}\]
| datum : any/c |
| out : output-port? = (current-output-port) |
Displays `datum` to `out`, similar to `write`, but usually in such a way that byte- and character-based datatypes are written as raw bytes or characters. If `out` has a handler associated to it via `port-display-handler`, then the handler is called. Otherwise, the default printer is used (in `display` mode), as configured by various parameters.

See §1.4 “The Printer” for more information about the default printer. In particular, note that `display` may require memory proportional to the depth of the value being printed, due to the initial cycle check.

```
(print datum [out quote-depth]) → void?
  datum : any/c
  out : output-port? = (current-output-port)
  quote-depth : (or/c 0 1) = 0
```

Prints `datum` to `out`. If `out` has a handler associated to it via `port-print-handler`, then the handler is called. Otherwise, the handler specified by `global-port-print-handler` is called; the default handler uses the default printer in `print` mode.

The optional `quote-depth` argument adjusts printing when the `print-as-expression` parameter is set to `#t`. In that case, `quote-depth` specifies the starting quote depth for printing `datum`.

The rationale for providing `print` is that `display` and `write` both have specific output conventions, and those conventions restrict the ways that an environment can change the behavior of `display` and `write` procedures. No output conventions should be assumed for `print`, so that environments are free to modify the actual output generated by `print` in any way.

```
(writeln datum [out]) → void?
  datum : any/c
  out : output-port? = (current-output-port)
```

The same as `(write datum out)` followed by `(newline out)`.

Added in version 6.1.1.8 of package base.

```
(displayln datum [out]) → void?
  datum : any/c
  out : output-port? = (current-output-port)
```

The same as `(display datum out)` followed by `(newline out)`, which is similar to `println` in Pascal or Java.

```
(println datum [out quote-depth]) → void?
  datum : any/c
  out : output-port? = (current-output-port)
  quote-depth : (or/c 0 1) = 0
```
The same as \((\text{print} \ \text{datum} \ \text{out} \ \text{quote-depth})\) followed by \((\text{newline} \ \text{out})\).

The \texttt{println} function is not equivalent to \texttt{println} in other languages, because \texttt{println} uses printing conventions that are closer to \texttt{write} than to \texttt{display}. For a closer analog to \texttt{println} in other languages, use \texttt{displayln}.

Added in version 6.1.1.8 of package \texttt{base}.

\[
\begin{align*}
\text{fprintf} & \text{ out form } v \ldots \rightarrow \text{void} \\
\text{out} & : \text{output-port} \\
\text{form} & : \text{string} \\
v & : \text{any/c}
\end{align*}
\]

Prints formatted output to \texttt{out}, where \texttt{form} is a string that is printed directly, except for special formatting escapes:

- \texttt{~n} or \texttt{~%} prints a newline character (which is equivalent to \texttt{\n} in a literal format string)
- \texttt{~a} or \texttt{~A} displays the next argument among the \texttt{v}s
- \texttt{~s} or \texttt{~S} writes the next argument among the \texttt{v}s
- \texttt{~v} or \texttt{~V} prints the next argument among the \texttt{v}s
- \texttt{~(c)} where \texttt{(c)} is \texttt{a}, \texttt{A}, \texttt{s}, \texttt{S}, \texttt{v}, or \texttt{V}: truncates default-handler \texttt{display}, \texttt{write}, or \texttt{print} output to (\texttt{error-print-width}) characters, using \texttt{...} as the last three characters if the untruncated output would be longer
- \texttt{~e} or \texttt{~E} outputs the next argument among the \texttt{v}s using the current error value conversion handler (see \texttt{error-value->string-handler}) and current error printing width
- \texttt{~c} or \texttt{~C} \texttt{write-chars} the next argument in \texttt{v}s; if the next argument is not a character, the \texttt{exn:fail:contract} exception is raised
- \texttt{~b} or \texttt{~B} prints the next argument among the \texttt{v}s in binary; if the next argument is not an exact number, the \texttt{exn:fail:contract} exception is raised
- \texttt{~o} or \texttt{~O} prints the next argument among the \texttt{v}s in octal; if the next argument is not an exact number, the \texttt{exn:fail:contract} exception is raised
- \texttt{~x} or \texttt{~X} prints the next argument among the \texttt{v}s in hexadecimal; if the next argument is not an exact number, the \texttt{exn:fail:contract} exception is raised
- \texttt{~} prints a tilde.
- \texttt{~(w)}, where \texttt{(w)} is a whitespace character (see \texttt{char-whitespace?}), skips characters in \texttt{form} until a non-whitespace character is encountered or until a second end-of-line is encountered (whichever happens first). On all platforms, an end-of-line can be \texttt{\#\return{}, \#\newline, or \#\return} followed immediately by \texttt{\#\newline}.
The `form` string must not contain any `~` that is not one of the above escapes, otherwise the `exn:fail:contract` exception is raised. When the format string requires more `vs` than are supplied, the `exn:fail:contract` exception is raised. Similarly, when more `vs` are supplied than are used by the format string, the `exn:fail:contract` exception is raised.

Example:

```scheme
> (printf (current-output-port) "~a as a string is ~s.\n" '3 4) "(3 4)"
(3 4) as a string is "(3 4)".
```

The same as `(printf (current-output-port) form v ...)`.

```scheme
(eprintf form v ...) → void?
form : string?
v : any/c
```

The same as `(printf (current-error-port) form v ...)`.

```scheme
(format form v ...) → string?
form : string?
v : any/c
```

Formats to a string. The result is the same as

```scheme
(let ([o (open-output-string)])
  (fprintf o form v ...)
  (get-output-string o))
```

Example:

```scheme
> (format "~a as a string is ~s.\n" '(3 4) "(3 4)"
"(3 4) as a string is \"(3 4)\".\n"
```

The same as `(printf (current-output-port) form v ...)`.

```scheme
(print-pair-curly-braces) → boolean?
(print-pair-curly-braces on?) → void?
on? : any/c
```
A parameter that controls pair printing. If the value is true, then pairs print using \{ and \} instead of ( and ). The default is \#f.

\[(\text{print-mpair-curly-braces}) \rightarrow \text{boolean}?
(\text{print-mpair-curly-braces on?}) \rightarrow \text{void}?
\quad \text{on?} : \text{any/c}\]

A parameter that controls pair printing. If the value is true, then mutable pairs print using \{ and \} instead of ( and ). The default is \#t.

\[(\text{print-unreadable}) \rightarrow \text{boolean}?
(\text{print-unreadable on?}) \rightarrow \text{void}?
\quad \text{on?} : \text{any/c}\]

A parameter that enables or disables printing of values that have no readable form (using the default reader), including structures that have a custom-write procedure (see \texttt{prop:custom-write}), but not including uninterned symbols and unreadable symbols (which print the same as interned symbols). If the parameter value is \#f, an attempt to print an unreadable value raises \texttt{exn:fail}. The parameter value defaults to \#t. See §1.4 “The Printer” for more information.

\[(\text{print-graph}) \rightarrow \text{boolean}?
(\text{print-graph on?}) \rightarrow \text{void}?
\quad \text{on?} : \text{any/c}\]

A parameter that controls printing data with sharing; defaults to \#f. See §1.4 “The Printer” for more information.

\[(\text{print-struct}) \rightarrow \text{boolean}?
(\text{print-struct on?}) \rightarrow \text{void}?
\quad \text{on?} : \text{any/c}\]

A parameter that controls printing structure values in vector or prefab form; defaults to \#t. See §1.4 “The Printer” for more information. This parameter has no effect on the printing of structures that have a custom-write procedure (see \texttt{prop:custom-write}).

\[(\text{print-box}) \rightarrow \text{boolean}?
(\text{print-box on?}) \rightarrow \text{void}?
\quad \text{on?} : \text{any/c}\]

A parameter that controls printing box values; defaults to \#t. See §1.4.10 “Printing Boxes” for more information.

\[(\text{print-vector-length}) \rightarrow \text{boolean}?
(\text{print-vector-length on?}) \rightarrow \text{void}?
\quad \text{on?} : \text{any/c}\]
A parameter that controls printing vectors; defaults to `#f`. See §1.4.7 “Printing Vectors” for more information.

```scheme
(print-hash-table) → boolean?
(print-hash-table on?) → void?
  on? : any/c
```

A parameter that controls printing hash tables; defaults to `#t`. See §1.4.9 “Printing Hash Tables” for more information.

```scheme
(print-boolean-long-form) → boolean?
(print-boolean-long-form on?) → void?
  on? : any/c
```

A parameter that controls printing of booleans. When the parameter’s value is true, `#t` and `#f` print as `#true` and `#false`, otherwise they print as `#t` and `#f`. The default is `#f`.

```scheme
(print-reader-abbreviations) → boolean?
(print-reader-abbreviations on?) → void?
  on? : any/c
```

A parameter that controls printing of two-element lists that start with `quote`, `quasiquote`, `unquote`, `unquote-splicing`, `syntax`, `quasisyntax`, `unsyntax`, or `unsyntax-splicing`; defaults to `#f`. See §1.4.5 “Printing Pairs and Lists” for more information.

```scheme
(print-as-expression) → boolean?
(print-as-expression on?) → void?
  on? : any/c
```

A parameter that controls printing in `print` mode (as opposed to `write` or `display`); defaults to `#t`. See §1.4 “The Printer” for more information.

```scheme
(print-syntax-width)
  width : (or/c +inf.0 0 (and/c exact-integer? (>/c 3)))
```

A parameter that controls printing of syntax objects. Up to `width` characters are used to show the datum form of a syntax object within `#<syntax...>` (after the syntax object’s source location, if any).

```scheme
(print-value-columns)
  columns : (or/c +inf.0 0 (and/c exact-integer? (>/c 5)))
```
A parameter that contains a recommendation for the number of columns that should be used for printing values via `print`. May or may not be respected by `print` - the current default handler for `print` does not. It is expected that REPLs that use some form of pretty-printing for values respect this parameter.

Added in version 8.0.0.13 of package `base`.

```
(or/c (and/c path? complete-path?)
  (cons/c (and/c path? complete-path?)
    (and/c path? complete-path?))
#f)
```

A parameter that is used when writing compiled code (see §1.4.16 “Printing Compiled Code”) that contains pathname literals, including source-location pathnames for procedure names. When the parameter’s value is a `path`, paths that syntactically extend `path` are converted to relative paths; when the resulting compiled code is read, relative paths are converted back to complete paths using the `current-load-relative-directory` parameter (if it is not `#f`; otherwise, the path is left relative). When the parameter’s value is `(cons rel-to-path base-path)`, then paths that syntactically extend `base-path` are converted as relative to `rel-to-path`; the `rel-to-path` must extend `base-path`, in which case 'up path elements (in the sense of `build-path`) may be used to make a path relative to `rel-to-path`.

```
(or/c (and/c path-string? complete-path?)
  (cons/c (and/c path-string? complete-path?)
    (and/c path-string? complete-path?))
#f)
```

Gets or sets the `port write handler`, `port display handler`, or `port print handler` for `out`. This

981
handler is called to output to the port when write, display, or print (respectively) is applied to the port. Each handler must accept two arguments: the value to be printed and the destination port. The handler’s return value is ignored.

A port print handler optionally accepts a third argument, which corresponds to the optional third argument to print; if a procedure given to port-print-handler does not accept a third argument, it is wrapped with a procedure that discards the optional third argument.

The default port display and write handlers print Racket expressions with Racket’s built-in printer (see §1.4 “The Printer”). The default print handler calls the global port print handler (the value of the global-port-print-handler parameter); the default global port print handler is the same as the default write handler.

```
(global-port-print-handler)
  -> (->* (any/c output-port?) ((or/c 0 1)) any)
  (global-port-print-handler proc) -> void?
    proc : (or/c (->* (any/c output-port?) ((or/c 0 1)) any)
      (any/c output-port? . -> . any))
```

A parameter that determines global port print handler, which is called by the default port print handler (see port-print-handler) to print values into a port. The default value uses the built-in printer (see §1.4 “The Printer”) in print mode.

A global port print handler optionally accepts a third argument, which corresponds to the optional third argument to print. If a procedure given to global-port-print-handler does not accept a third argument, it is wrapped with a procedure that discards the optional third argument.

### 13.6 Pretty Printing

```
(require racket/pretty) package: base
```

The bindings documented in this section are provided by the racket/pretty and racket libraries, but not racket/base.

```
(pretty-print v [port quote-depth #:newline? newline?]) -> void?
  v : any/c
  port : output-port? = (current-output-port)
  quote-depth : (or/c 0 1) = 0
  newline? : boolean? = #t
```

Pretty-prints the value v using the same printed form as the default print mode, but with newlines and whitespace inserted to avoid lines longer than (pretty-print-columns).
as controlled by \(\texttt{(pretty-print-current-style-table)}\). The printed form ends in a newline by default, unless the \(\texttt{newline?}\) argument is supplied with false or the \(\texttt{pretty-print-columns}\) parameter is set to \(\texttt{infinity}\). When \(\texttt{port}\) has line counting enabled (see §13.1.4 "Counting Positions, Lines, and Columns"), then printing is sensitive to the column when printing starts—both for determining an initial line break and indenting subsequent lines.

In addition to the parameters defined in this section, \(\texttt{pretty-print}\) conforms to the \(\texttt{print-graph, print-struct, print-hash-table, print-vector-length, print-box}\), and \(\texttt{print-as-expression}\) parameters.

The pretty printer detects structures that have the \(\texttt{prop:custom-write}\) property and calls the corresponding custom-write procedure. The custom-write procedure can check the parameter \(\texttt{pretty-printing}\) to cooperate with the pretty-printer. Recursive printing to the port automatically uses pretty printing, but if the structure has multiple recursively printed sub-expressions, a custom-write procedure may need to cooperate more to insert explicit newlines. Use \(\texttt{port-next-location}\) to determine the current output column, use \(\texttt{pretty-print-columns}\) to determine the target printing width, and use \(\texttt{pretty-print-newline}\) to insert a newline (so that the function in the \(\texttt{pretty-print-print-line}\) parameter can be called appropriately). Use \(\texttt{make-tentative-pretty-print-output-port}\) to obtain a port for tentative recursive prints (e.g., to check the length of the output).

If the \(\texttt{newline?}\) argument is omitted or supplied with true, the \(\texttt{pretty-print-print-line}\) callback is called with false as the first argument to print the last newline after the printed value. If it is supplied with false, the \(\texttt{pretty-print-print-line}\) callback is not called after the printed value.

Changed in version 6.6.0.3 of package \(\texttt{base}\): Added \(\texttt{newline?}\) argument.

\[
(\texttt{pretty-write v [port #:newline? newline?]}) \rightarrow \texttt{void?}
\]

\(v : \texttt{any/c}\)

\(\texttt{port : output-port?} = (\texttt{current-output-port})\)

\(\texttt{newline? : boolean?} = \texttt{#t}\)

Same as \(\texttt{pretty-print}\), but \(v\) is printed like \(\texttt{write}\) instead of like \(\texttt{print}\).

Changed in version 6.6.0.3 of package \(\texttt{base}\): Added \(\texttt{newline?}\) argument.

\[
(\texttt{pretty-display v [port #:newline? newline?]}) \rightarrow \texttt{void?}
\]

\(v : \texttt{any/c}\)

\(\texttt{port : output-port?} = (\texttt{current-output-port})\)

\(\texttt{newline? : boolean?} = \texttt{#t}\)

Same as \(\texttt{pretty-print}\), but \(v\) is printed like \(\texttt{display}\) instead of like \(\texttt{print}\).

Changed in version 6.6.0.3 of package \(\texttt{base}\): Added \(\texttt{newline?}\) argument.
(pretty-format v [columns #:mode mode]) → string?
  v : any/c
  columns : exact-nonnegative-integer? = (pretty-print-columns)
  mode : (or/c 'print 'write 'display) = 'print

Like pretty-print, except that it returns a string containing the pretty-printed value, rather than sending the output to a port.

The optional argument columns argument is used to parameterize pretty-print-columns.

The keyword argument mode controls whether printing is done like either pretty-print (the default), pretty-write or pretty-display.

Changed in version 6.3 of package base: Added a mode argument.

(pretty-print-handler v) → void?
  v : any/c

Pretty-prints v if v is not #<void>, or prints nothing if v is #<void>. Pass this procedure to current-print to install the pretty printer into the REPL run by read-eval-print-loop.

13.6.1 Basic Pretty-Print Options

(pretty-print-columns)
  → (or/c exact-positive-integer? 'infinity)

(pretty-print-columns width) → void?
  width : (or/c exact-positive-integer? 'infinity)

A parameter that determines the default width for pretty printing.

If the display width is 'infinity, then pretty-printed output is never broken into lines, and a newline is not added to the end of the output.

(pretty-print-depth) → (or/c exact-nonnegative-integer? #f)

(pretty-print-depth depth) → void?
  depth : (or/c exact-nonnegative-integer? #f)

Parameter that controls the default depth for recursive pretty printing. Printing to depth means that elements nested more deeply than depth are replaced with “…”; in particular, a depth of 0 indicates that only simple values are printed. A depth of #f (the default) allows printing to arbitrary depths.
A parameter that determines how exact non-integers are printed. If the parameter’s value is #t, then an exact non-integer with a decimal representation is printed as a decimal number instead of a fraction. The initial value is #f.

A parameter that controls the printing of the symbol whose print name is just a period. If set to a true value, then such a symbol is printed as only the period. If set to a false value, it is printed as a period with vertical bars surrounding it.

A parameter that determines how inexact numbers are printed. If the parameter’s value is #t, then inexact numbers are always printed with a leading #i. The initial value is #f.

### 13.6.2 Per-Symbol Special Printing

A parameter that controls whether or not quote, unquote, unquote-splicing, etc., are abbreviated with ’, ., @, etc. By default, the abbreviations are enabled.

See also pretty-print-remap-stylable.

Returns #t if v is a style table for use with pretty-print-current-style-table, #f otherwise.
A parameter that holds a table of style mappings. See `pretty-print-extend-style-table`.

```scheme
(pretty-print-extend-style-table style-table symbol-list like-symbol-list)
```

Creates a new style table by extending an existing `style-table`, so that the style mapping for each symbol of `like-symbol-list` in the original table is used for the corresponding symbol of `symbol-list` in the new table. The `symbol-list` and `like-symbol-list` lists must have the same length. The `style-table` argument can be `#f`, in which case the default mappings are used from the original table (see below).

The style mapping for a symbol controls the way that whitespace is inserted when printing a list that starts with the symbol. In the absence of any mapping, when a list is broken across multiple lines, each element of the list is printed on its own line, each with the same indentation.

The default style mapping includes mappings for the following symbols, so that the output follows popular code-formatting rules:

```
'lambda 'case-lambda
'define 'define-macro 'define-syntax
'let 'letrec 'let*
'let-syntax 'letrec-syntax
'let-values 'letrec-values 'let*+-values
'let-syntaxes 'letrec-syntaxes
'begin 'begin0 'do
'if 'set! 'set!+-values
'unless 'when
'cond 'case 'and 'or
'module
'syntax-rules 'syntax-case 'letrec-syntaxes+values
'import 'export 'link
'require 'require-for-syntax 'require-for-template 'provide
'public 'private 'override 'rename 'inherit 'field 'init
'shared 'send 'class 'instantiate 'make-object
```

```scheme
(pretty-print-remap-stylable)
→ (any/c . -> . (or/c symbol? #f))
(pretty-print-remap-stylable proc) → void?
proc : (any/c . -> . (or/c symbol? #f))
```
A parameter that controls remapping for styles and for the determination of the reader short-hands.

This procedure is called with each sub-expression that appears as the first element in a sequence. If it returns a symbol, the style table is used, as if that symbol were at the head of the sequence. If it returns #f, the style table is treated normally. Similarly, when determining whether to abbreviate reader macros, this parameter is consulted.

13.6.3 Line-Output Hook

\[
\text{(pretty-print-newline \(\text{port width}\)) \rightarrow \text{void?}}
\]
\[
\text{port : output-port?}
\]
\[
\text{width : exact-nonnegative-integer?}
\]

Calls the procedure associated with the \text{pretty-print-print-line} parameter to print a newline to \text{port}, if \text{port} is the output port that is redirected to the original output port for printing, otherwise a plain newline is printed to \text{port}. The \text{width} argument should be the target column width, typically obtained from \text{pretty-print-columns}.

\[
\text{(pretty-print-print-line \(\text{proc}\)) \rightarrow \text{void?}}
\]
\[
\text{proc : ((or/c exact-nonnegative-integer? #f)}
\]
\[
\text{output-port?}
\]
\[
\text{exact-nonnegative-integer?}
\]
\[
\text{(or/c exact-nonnegative-integer? 'infinity)}
\]
\[
\rightarrow . .
\text{exact-nonnegative-integer?)}
\]

A parameter that determines a procedure for printing the newline separator between lines of a pretty-printed value. The procedure is called with four arguments: a new line number, an output port, the old line’s length, and the number of destination columns. The return value from \text{proc} is the number of extra characters it printed at the beginning of the new line.

The \text{proc} procedure is called before any characters are printed with 0 as the line number and 0 as the old line length. Whenever the pretty-printer starts a new line, \text{proc} is called with the new line’s number (where the first new line is numbered 1) and the just-finished line’s length. The destination-columns argument to \text{proc} is always the total width of the destination printing area, or \text{'infinity} if pretty-printed values are not broken into lines.
If the #:newline? argument was omitted or supplied with a true value, proc is also called after the last character of the value has been printed, with #f as the line number and with the length of the last line.

The default proc procedure prints a newline whenever the line number is not 0 and the column count is not 'infinity, always returning 0. A custom proc procedure can be used to print extra text before each line of pretty-printed output; the number of characters printed before each line should be returned by proc so that the next line break can be chosen correctly.

The destination port supplied to proc is generally not the port supplied to pretty-print or pretty-display (or the current output port), but output to this port is ultimately redirected to the port supplied to pretty-print or pretty-display.

### 13.6.4 Value Output Hook

```scheme
(pretty-print-size-hook)
→ (any/c boolean? output-port?)
  . -> .
  (or/c #f exact-nonnegative-integer?))
(pretty-print-size-hook proc) → void?
proc : (any/c boolean? output-port?)
  . -> .
  (or/c #f exact-nonnegative-integer?))
```

A parameter that determines a sizing hook for pretty-printing.

The sizing hook is applied to each value to be printed. If the hook returns #f, then printing is handled internally by the pretty-printer. Otherwise, the value should be an integer specifying the length of the printed value in characters; the print hook will be called to actually print the value (see pretty-print-print-hook).

The sizing hook receives three arguments. The first argument is the value to print. The second argument is a boolean: #t for printing like display and #f for printing like write. The third argument is the destination port; the port is the one supplied to pretty-print or pretty-display (or the current output port). The sizing hook may be applied to a single value multiple times during pretty-printing.

```scheme
(pretty-print-print-hook)
→ (any/c boolean? output-port? . -> . void?)
(pretty-print-print-hook proc) → void?
proc : (any/c boolean? output-port? . -> . void?)
```

A parameter that determines a print hook for pretty-printing. The print-hook procedure is applied to a value for printing when the sizing hook (see pretty-print-size-hook) returns an integer size for the value.
The print hook receives three arguments. The first argument is the value to print. The second argument is a boolean: #t for printing like display and #f for printing like write. The third argument is the destination port; this port is generally not the port supplied to pretty-print or pretty-display (or the current output port), but output to this port is ultimately redirected to the port supplied to pretty-print or pretty-display.

```
(pretty-print-pre-print-hook)
   -> (any/c output-port? . -> . void)
(pretty-print-pre-print-hook proc) -> void?
   proc : (any/c output-port? . -> . void)
```

A parameter that determines a hook procedure to be called just before an object is printed. The hook receives two arguments: the object and the output port. The port is the one supplied to pretty-print or pretty-display (or the current output port).

```
(pretty-print-post-print-hook)
   -> (any/c output-port? . -> . void)
(pretty-print-post-print-hook proc) -> void?
   proc : (any/c output-port? . -> . void)
```

A parameter that determines a hook procedure to be called just after an object is printed. The hook receives two arguments: the object and the output port. The port is the one supplied to pretty-print or pretty-display (or the current output port).

13.6.5 Additional Custom-Output Support

```
(pretty-printing) -> boolean?
(pretty-printing on?) -> void?
   on? : any/c
```

A parameter that is set to #t when the pretty printer calls a custom-write procedure (see prop:custom-write) for output in a mode that supports line breaks. When pretty printer calls a custom-write procedure merely to detect cycles or to try to print on a single line, it sets this parameter to #f.

```
(make-tentative-pretty-print-output-port out
   width
   overflow-thunk)
   -> output-port?
   out : output-port?
   width : exact-nonnegative-integer?
   overflow-thunk : (-> any)
```

Produces an output port that is suitable for recursive pretty printing without actually producing output. Use such a port to tentatively print when proper output depends on the size
of recursive prints. After printing, determine the size of the tentative output using `file-position`.

The `out` argument should be a pretty-printing port, such as the one supplied to a custom-write procedure when `pretty-printing` is set to true, or another tentative output port. The `width` argument should be a target column width, usually obtained from `pretty-print-columns`, possibly decremented to leave room for a terminator. The `overflow-thunk` procedure is called if more than `width` items are printed to the port or if a newline is printed to the port via `pretty-print-newline`; it can escape from the recursive print through a continuation as a shortcut, but `overflow-thunk` can also return, in which case it is called every time afterward that additional output is written to the port.

After tentative printing, either accept the result with `tentative-pretty-print-port-transfer` or reject it with `tentative-pretty-print-port-cancel`. Failure to accept or cancel properly interferes with graph-structure printing, calls to hook procedures, etc. Explicitly cancel the tentative print even when `overflow-thunk` escapes from a recursive print.

```scheme
(tentative-pretty-print-port-transfer tentative-out orig-out) -> void?
tentative-out : output-port?
orig-out : output-port?
```

Causes the data written to `tentative-out` to be transferred as if written to `orig-out`. The `tentative-out` argument should be a port produced by `make-tentative-pretty-print-output-port`, and `orig-out` should be either a pretty-printing port (provided to a custom-write procedure) or another tentative output port.

```scheme
(tentative-pretty-print-port-cancel tentative-out) -> void?
tentative-out : output-port?
```

 Cancels the content of `tentative-out`, which was produced by `make-tentative-pretty-print-output-port`. The main effect of canceling is that graph-reference definitions are undone, so that a future print of a graph-referenced object includes the defining `#\(\langle n\rangle\)`.

### 13.7 Reader Extension

Racket’s reader can be extended in three ways: through a reader-macro procedure in a readtable (see §13.7.1 “Readtables”), through a `#reader` form (see §1.3.18 “Reading via an Extension”), or through a custom-port byte reader that returns a “special” result procedure (see §13.1.9 “Custom Ports”). All three kinds of `reader extension procedures` accept similar arguments, and their results are treated in the same way by `read` and `read-syntax` (or, more precisely, by the default read handler; see `port-read-handler`).

990
13.7.1 Readtables

The dispatch table in §1.3.1 “Delimiters and Dispatch” corresponds to the default readtable. By creating a new readtable and installing it via the `current-readtable` parameter, the reader’s behavior can be extended.

A readtable is consulted at specific times by the reader:

- when looking for the start of a datum;
- when determining how to parse a datum that starts with #;
- when looking for a delimiter to terminate a symbol or number;
- when looking for an opener (such as ⟨⟩, closer (such as ⟩), or . after the first character parsed as a sequence for a pair, list, vector, or hash table; or
- when looking for an opener after #⟨n⟩ in a vector of specified length ⟨n⟩.

The readtable is ignored at other times. In particular, after parsing a character that is mapped to the default behavior of ;, the readtable is ignored until the comment’s terminating newline is discovered. Similarly, the readtable does not affect string parsing until a closing double-quote is found. Meanwhile, if a character is mapped to the default behavior of ⟨, then it starts sequence that is closed by any character that is mapped to a closing parenthesis ⟩. An apparent exception is that the default parsing of | quotes a symbol until a matching character is found, but the parser is simply using the character that started the quote; it does not consult the readtable.

For many contexts, #f identifies the default readtable. In particular, #f is the initial value for the `current-readtable` parameter, which causes the reader to behave as described in §1.3 “The Reader”.

```scheme
(readtable? v) -> boolean?
  v : any/c
```

Returns #t if v is a readtable, #f otherwise.

```scheme
(make-readtable readable key mode action ...) -> readtable?
  readable : (or/c readable? #f)
  key : (or/c char? #f)
  mode : (or/c 'terminating-macro
            'non-terminating-macro
            'dispatch-macro
            char?)
  action : (or/c procedure?
            readable?
            #f)
```

991
Creates a new readtable that is like readtable (which can be #f to indicate the default readtable), except that the reader’s behavior is modified for each key according to the given mode and action. The ... for make-readtable applies to all three of key, mode, and action; in other words, the total number of arguments to make-readtable must be 1 modulo 3.

The possible combinations for key, mode, and action are as follows:

- **char 'terminating-macro proc** — causes char to be parsed as a delimiter, and an unquoted/uncommented char in the input string triggers a call to the reader macro proc; the activity of proc is described further below. Conceptually, characters like ;, {, and } are mapped to terminating reader macros in the default readtable.

- **char 'non-terminating-macro proc** — like the 'terminating-macro variant, but char is not treated as a delimiter, so it can be used in the middle of an identifier or number. Conceptually, # is mapped to a non-terminating macro in the default readtable.

- **char 'dispatch-macro proc** — like the 'non-terminating-macro variant, but for char only when it follows a # (or, more precisely, when the character follows one that has been mapped to the behavior of # in the default readtable).

- **char like-char readtable** — causes char to be parsed in the same way that like-char is parsed in readtable, where readtable can be #f to indicate the default readtable. (The mapping of char does not apply after #, which is configured separately via 'dispatch-macro.) Mapping a character to the same actions as | in the default reader means that the character starts quoting for symbols, and the same character terminates the quote; in contrast, mapping a character to the same action as a # means that the character starts a string, but the string is still terminated with a closing ". Finally, mapping a character to an action in the default readable means that the character’s behavior is sensitive to parameters that affect the original character; for example, mapping a character to the same action as a curly brace { in the default readable means that the character is disallowed when the read-curly-brace-as-paren parameter is set to #f.

- **#f 'non-terminating-macro proc** — replaces the macro used to parse characters with no specific mapping: i.e., the characters (other than # or |) that can start a symbol or number with the default readable.

If multiple 'dispatch-macro mappings are provided for a single char, all but the last one are ignored. Similarly, if multiple non-'dispatch-macro mappings are provided for a single char, all but the last one are ignored.

A reader macro proc must accept six arguments, and it can optionally accept two arguments. The first two arguments are always the character that triggered the reader macro and the input port for reading. When the reader macro is triggered by read-syntax (or read-syntax/recursive), the procedure is passed four additional arguments that represent a
source location for already-consumed character(s): the source name, a line number or #f, a
column number or #f, and a position or #f. When the reader macro is triggered by read (or
read/recursive), the procedure is passed only two arguments if it accepts two arguments,
otherwise it is passed six arguments where the third is always #f. See §13.7.2 “Reader-
Extension Procedures” for information on the procedure’s results.

A reader macro normally reads characters from the given input port to produce a value to
be used as the “reader macro-expansion” of the consumed characters. The reader macro
might produce a special-comment value (see §13.7.3 “Special Comments”) to cause the
consumed character to be treated as whitespace, and it might use read/recursive or read-
syntax/recursive.

```scheme
(readtable-mapping readtable char)
  (or/c char?)
  → 'terminating-macro
      'non-terminating-macro)
  (or/c #f procedure?)
  (or/c #f procedure?)
readtable : readtable?
char : char?
```

Produces information about the mappings in `readtable` for `char`. The result is three val-
ues:

- either a character (mapping to the same behavior as the character in the default
  readable), 'terminating-macro, or 'non-terminating-macro; this result re-
  ports the main (i.e., non-'dispatch-macro) mapping for `char`. When the result
  is a character, then `char` is mapped to the same behavior as the returned character in
  the default readable.

- either #f or a reader-macro procedure; the result is a procedure when the first result is
  'terminating-macro or 'non-terminating-macro.

- either #f or a reader-macro procedure; the result is a procedure when the character
  has a 'dispatch-macro mapping in `readtable` to override the default dispatch
  behavior.

Note that reader-macro procedures for the default readtable are not directly accessible. To
invoke default behaviors, use `read/recursive` or `read-syntax/recursive` with a char-
acter and the #f readable.

Examples:

```scheme
; Provides raise-read-error and raise-read-eof-error
> (require syntax/readerr)
```

993
> (define (skip-whitespace port)
; Skips whitespace characters, sensitive to the current
; readtable's definition of whitespace
(let ([ch (peek-char port)])
  (unless (eof-object? ch)
    ; Consult current readtable:
    (let-values ([[(like-ch/sym proc dispatch-proc)
                    (readtable-mapping (current-readtable) ch)]])
      ; If like-ch/sym is whitespace, then ch is whitespace
      (when (and (char? like-ch/sym)
                 (char-whitespace? like-ch/sym))
        (read-char port)
        (skip-whitespace port))))))

> (define (skip-comments read-one port src)
; Recursive read, but skip comments and detect EOF
(let loop ()
  (let ([v (read-one)])
    (cond
     [(special-comment? v) (loop)]
     [(eof-object? v)
      (let-values ([[(l c p) (port-next-location port)]])
        (raise-read-eof-error
         "unexpected EOF in tuple" src l c p 1))]
     [else v]])))

> (define (parse port read-one src)
; First, check for empty tuple
(skip-whitespace port)
(if (eq? #\> (peek-char port))
  null
  (let ([elem (read-one)])
    (if (special-comment? elem)
        ; Found a comment, so look for > again
        (parse port read-one src)
        ; Non-empty tuple:
        (cons elem
          (skip-nonempty port read-one src))))))

> (define (parse-nonempty port read-one src)
; Need a comma or closer
(skip-whitespace port)
(case (peek-char port)
  [(#\>) (read-char port)
   ; Done
   null]
  [(#\,) (read-char port)
   ; Read next element and recur
   (cons (skip-comments read-one port src)
     (parse-nonempty port read-one src))]

994
(parse-nonempty port read-one src))]

[else
 ; Either a comment or an error; grab location (in case
 ; of error) and read recursively to detect comments
(let-values ([(l c p) (port-next-location port)]
  [(v) (read-one)])

(cond
  [(special-comment? v)
   ; It was a comment, so try again
   (parse-nonempty port read-one src)]
  [else
   ; Wasn't a comment, comma, or closer; error
   ((if (eof-object? v)
     raise-read-eof-error
     raise-read-error)
   "expected `.` or `>`" src l c p 1)]))])

> (define (make-delims-table)
  ; Table to use for recursive reads to disallow delimiters
  ; (except those in sub-expressions)
  (letrec ([misplaced-delimiter
    (case-lambda
      [(ch port) (misplaced-delimiter ch port #f #f #f #f)]
      [(ch port src line col pos)
       (raise-read-error
        (format "misplaced `~a` in tuple" ch)
        src line col pos 1)]))
    (make-readtable (current-readtable)
      #, 'terminating-macro misplaced-delimiter
      #\> 'terminating-macro misplaced-delimiter))
  )

> (define (wrap l)
  `(make-tuple (list ,@l)))

> (define parse-open-tuple
  (case-lambda
    (parse-open-tuple
      (case-lambda
        [(ch port)
         ; `read` mode
         (wrap (parse port
            (lambda ()
              (read/recursive port #f
                (make-delims-table))))
            (object-name port))))
        [(ch port src line col pos)
         ; `read-syntax` mode
         (datum->syntax #f
           (wrap (parse port
              (lambda ()
                (read/recursive port #f
                  (make-delims-table))))
            (object-name port))))]

995
(lambda ()
  (read-syntax/recursive src port #f
    (make-delims-table)))

src))
  (let-values ([(l c p) (port-next-location port)])
    (list src line col pos (and pos (- p pos))))))))

> (define tuple-readtable
    (make-readtable #f #\< 'terminating-macro parse-open-tuple))

> (parameterize ([current-readtable tuple-readtable])
    (read (open-input-string "<1 , 2 , "a" >"))
  '(make-tuple (list 1 2 "a")))

> (parameterize ([current-readtable tuple-readtable])
    (read (open-input-string
      "< #||# 1 #||# , #||# 2 #||# , #||# "a" #||# >")))
  '(make-tuple (list 1 2 "a")))

> (define tuple-readtable+
    (make-readtable tuple-readtable
      #\* 'terminating-macro (lambda a
        (make-special-comment #f))
      #\_ #\space #f))

> (parameterize ([current-readtable tuple-readtable+])
    (read (open-input-string "< * 1 __,__, 2 __,__, * "a" * >")))
  '(make-tuple (list 1 2 "a")))

### 13.7.2 Reader-Extension Procedures

Calls to reader extension procedures can be triggered through `read`, `read/recursive`, or `read-syntax`. In addition, a special-read procedure can be triggered by calls to `read-char-or-special`, or by the context of `read-bytes-avail!`, `peek-bytes-avail!`, `read-bytes-avail!`, and `peek-bytes-avail!`. Optional arities for reader-macro and special-result procedures allow them to distinguish reads via `read`, etc., from reads via `read-syntax`, etc. (where the source value is `#f` and no other location information is available).

When a reader-extension procedure is called in syntax-reading mode (via `read-syntax`, etc.), it should generally return a syntax object that has no lexical context (e.g., a syntax object created using `datum->syntax` with `#f` as the first argument and with the given location information as the third argument). Another possible result is a special-comment value (see §13.7.3 “Special Comments”). If the procedure’s result is not a syntax object and not a special-comment value, it is converted to one using `datum->syntax`.

When a reader-extension procedure is called in non-syntax-reading modes, it should generally not return a syntax object. If a syntax object is returned, it is converted to a plain value.
using syntax->datum.

In either context, when the result from a reader-extension procedure is a special-comment value (see §13.7.3 “Special Comments”), then read, read-syntax, etc. treat the value as a delimiting comment and otherwise ignore it.

Also, in either context, the result may be copied to prevent mutation to vectors or boxes before the read result is completed, and to support the construction of graphs with cycles. Mutable boxes, vectors, and prefab structures are copied, along with any pairs, boxes, vectors, prefab structures that lead to such mutable values, to placeholders produced by a recursive read (see read/recursive), or to references of a shared value. Graph structure (including cycles) is preserved in the copy.

### 13.7.3 Special Comments

\[
\text{(make-special-comment } v \text{) → special-comment?}
\]

\[
\begin{array}{l}
\text{v : any/c}
\end{array}
\]

Creates a special-comment value that encapsulates \( v \). The read, read-syntax, etc., procedures treat values constructed with `make-special-comment` as delimiting whitespace when returned by a reader-extension procedure (see §13.7.2 “Reader-Extension Procedures”).

\[
\text{(special-comment? } v \text{) → boolean?}
\]

\[
\begin{array}{l}
\text{v : any/c}
\end{array}
\]

Returns `#t` if \( v \) is the result of `make-special-comment`, `#f` otherwise.

\[
\text{(special-comment-value } sc \text{) → any}
\]

\[
\begin{array}{l}
\text{sc : special-comment?}
\end{array}
\]

Returns the value encapsulated by the special-comment value \( sc \). This value is never used directly by a reader, but it might be used by the context of a read-char-or-special, etc., call that detects a special comment.

### 13.8 Printer Extension

\[
\text{gen:custom-write : any/c}
\]

A generic interface (see §5.4 “Generic Interfaces”) that supplies a method, write-proc used by the default printer to display, write, or print instances of the structure type.

A write-proc method takes three arguments: the structure to be printed, the target port, and an argument that is `#t` for write mode, `#f` for display mode, or 0 or 1 indicating the
current quoting depth for print mode. The procedure should print the value to the given port using write, display, print, fprintf, write-special, etc.

The port write handler, port display handler, and print handler are specially configured for a port given to a custom-write procedure. Printing to the port through display, write, or print prints a value recursively with sharing annotations. To avoid a recursive print (i.e., to print without regard to sharing with a value currently being printed), print instead to a string or pipe and transfer the result to the target port using write-string or write-special. To print recursively to a port other than the one given to the custom-write procedure, copy the given port’s write handler, display handler, and print handler to the other port.

The port given to write-proc is not necessarily the actual target port. In particular, to detect cycles, sharing, and quoting modes (in the case of print), the printer invokes a custom-write procedure with a port that records information about recursive prints, and does not retain any other output. This information-gathering phase needs the same objects (in the eq? sense) to be printed as later, so that the recorded information can be correlated with printed values.

Recursive print operations may trigger an escape from a call to write-proc. For example, printing may escape during pretty-printing where a tentative print attempt overflows the line, or it may escape while printing error output that is constrained to a limited width.

The following example definition of a tuple type includes a write-proc procedure that prints the tuple’s list content using angle brackets in write and print mode and no brackets in display mode. Elements of the tuple are printed recursively, so that graph and cycle structure can be represented.

Examples:

```scheme
(define (tuple-print tuple port mode)
  (when mode (write-string "<" port))
  (let ([l (tuple-ref tuple)]
        [recur (case mode
                   [(#t) write]
                   [(#f) display]
                   [else (lambda (p port) (print p port mode))])])
    (unless (zero? (vector-length l))
      (recur (vector-ref l 0) port)
      (for-each (lambda (e)
                 (write-string ", " port)
                 (recur e port))
                (cdr (vector->list l))))
    (when mode (write-string ">") port)))

(struct tuple (ref)
  #:methods gen:custom-write
  [(define write-proc tuple-print)])
```

998
The `make-constructor-style-printer` function can help in the implementation of a `write-proc`, as in this example:

Examples:

```
(require racket/struct)

(struct point (x y)
  #:methods gen:custom-write
  [(define write-proc
    (make-constructor-style-printer
      (lambda (obj) 'point)
      (lambda (obj) (list (point-x obj) (point-y obj))))))]
```

```
> (print (point 1 2))
(point 1 2)
> (write (point 1 2))
#<point: 1 2>
```

A structure type property (see §5.3 “Structure Type Properties”) that supplies a procedure that corresponds to `gen:custom-write`’s `write-proc`. Using the `prop:custom-write` property is discouraged; use the `gen:custom-write` generic interface instead.

```
(custom-write? v) → boolean?
v : any/c
```

Returns `#t` if `v` has the `prop:custom-write` property, `#f` otherwise.

```
(custom-write-accessor v)
→ (custom-write? output-port? (or/c #t #f 0 1) . -> . any)
v : custom-write?
```

Returns the custom-write procedure associated with `v`. 999
A property and associated predicate and accessor. The property value is one of 'self, 'never, 'maybe, or 'always. When a structure has this property in addition to a prop:custom-write property value, then the property value affects printing in print mode; see §1.4 “The Printer”. When a value does not have the prop:custom-print-quotable, it is equivalent to having the 'self property value, which is suitable both for self-quoting forms and printed forms that are unreadable.

13.9 Serialization

(require racket/serialize) package: base

The bindings documented in this section are provided by the racket/serialize library, not racket/base or racket.

(serializable? v) → boolean?
  v : any/c

Returns #t if v appears to be serializable, without checking the content of compound values, and #f otherwise. See serialize for an enumeration of serializable values.

serialize
v
[#:relative-directory relative-to
 #:deserialize-relative-directory deserialize-relative-to]
→ any
  v : serializable?
  relative-to : (or/c (and/c path? complete-path?)
    (cons/c (and/c path? complete-path?)
      (and/c path? complete-path?))
    #f)
  deserialize-relative-to : (or/c (and/c path? complete-path?)
    (cons/c (and/c path? complete-path?)
      (and/c path? complete-path?))
    #f)
    = relative-to

Returns a value that encapsulates the value v. This value includes only readable values, so it can be written to a stream with write or s-exp->fasl, later read from a stream using read or fasl->s-exp, and then converted to a value like the original using deserialize.
Serialization followed by deserialization produces a value with the same graph structure and mutability as the original value, but the serialized value is a plain tree (i.e., no sharing).

The following kinds of values are serializable:

- structures created through `serializable-struct` or `serializable-struct/versions`, or more generally structures with the `prop:serializable` property (see `prop:serializable` for more information);
- prefab structures;
- instances of classes defined with `define-serializable-class` or `define-serializable-class*`;
- booleans, numbers, characters, interned symbols, unreadable symbols, keywords, strings, byte strings, paths (for a specific convention), regexp values, `#<void>`, and the empty list;
- pairs, mutable pairs, vectors, flvectors, fxvectors, boxes, hash tables, and sets;
- `date`, `date*`, `arity-at-least` and `srcloc` structures; and
- module path index values.

Serialization succeeds for a compound value, such as a pair, only if all content of the value is serializable. If a value given to `serialize` is not completely serializable, the `exn:fail:contract` exception is raised.

If `v` contains a cycle (i.e., a collection of objects that are all reachable from each other), then `v` can be serialized only if the cycle includes a mutable value, where a prefab structure counts as mutable only if all of its fields are mutable.

If `relative-to` is not `#f`, then paths to serialize that extend the path in `relative-to` are recorded in relative and platform-independent form. The possible values and treatment of `relative-to` are the same as for `current-write-relative-directory`.

If `deserialize-relative-to` is not `#f`, then any paths to deserializers as extracted via `prop:serializable` are recorded in relative form. Note that `relative-to` and `deserialize-relative-to` are independent, but `deserialize-relative-to` defaults to `relative-to`.

See `deserialize` for information on the format of serialized data.

Changed in version 6.5.0.4 of package `base`: Added keywords and regexp values as serializable.

Changed in version 7.0.0.6: Added the `#:relative-directory` and `#:deserialize-relative-directory` arguments.

```
(deserialize v) -> any
v : any/c
```
Given a value \( \nu \) that was produced by \texttt{serialize}, produces a value like the one given to \texttt{serialize}, including the same graph structure and mutability.

A serialized representation \( \nu \) is a list of six or seven elements:

- An optional list \'(1), ',(2), ',(3), or ',(4)\' that represents the version of the serialization format. If the first element of a representation is not a list, then the version is 0. Version 1 adds support for mutable pairs, version 2 adds support for unreadable symbols, version 3 adds support for \texttt{date*} structures, and version 4 adds support for paths that are meant to be relative to the deserialization directory.

- A non-negative exact integer \( \mathit{s-count} \) that represents the number of distinct structure types represented in the serialized data.

- A list \( \mathit{s-types} \) of length \( \mathit{s-count} \), where each element represents a structure type. Each structure type is encoded as a pair. The \texttt{car} of the pair is \texttt{#f} for a structure whose deserialization information is defined at the top level, otherwise it is a quoted module path, a byte string (to be converted into a platform-specific path using \texttt{bytes->path}) for a module that exports the structure’s deserialization information, or a relative path element list for a module to be resolved with respect to \texttt{current-load-relative-directory} or (as a fallback) \texttt{current-directory}; the list-of-relative-elements form is produced by \texttt{serialize} when the \#:deserialize-relative-directory argument is not \texttt{#f}. The \texttt{cdr} of the pair is the name of a binding (at the top level or exported from a module) for deserialization information, either a symbol or a string representing an unreadable symbol. These two are used with either \texttt{namespace-variable-binding} or \texttt{dynamic-require} to obtain deserialization information. See \texttt{make-deserialize-info} for more information on the binding’s value. See also \texttt{deserialize-module-guard}.

- A non-negative exact integer, \( \mathit{g-count} \) that represents the number of graph points contained in the following list.

- A list \( \mathit{graph} \) of length \( \mathit{g-count} \), where each element represents a serialized value to be referenced during the construction of other serialized values. Each list element is either a box or not:
  - A box represents a value that is part of a cycle, and for deserialization, it must be allocated with \texttt{#f} for each of its fields. The content of the box indicates the shape of the value:
    - a non-negative exact integer \( i \) for an instance of a structure type that is represented by the \( i \)th element of the \( \mathit{s-types} \) list;
    - \'(c for a pair, which fails on deserialization (since pairs are immutable; this case does not appear in output generated by \texttt{serialize});
    - \'(m for a mutable pair;
    - \'(b for a box;
    - a pair whose \texttt{car} is \'(v and whose \texttt{cdr} is a non-negative exact integer \( s \) for a vector of length \( s \);

1002
a list whose first element is 'h and whose remaining elements are symbols that determine the hash-table type:
  · 'equal — (make-hash)
  · 'equal 'weak — (make-weak-hash)
  · 'weak — (make-weak-hasheq)
  · no symbols — (make-hasheq)

'\texttt{date*}' for a \texttt{date*} structure, which fails on deserialization (since dates are immutable; this case does not appear in output generated by serialize);

'\texttt{date}' for a \texttt{date} structure, which fails on deserialization (since dates are immutable; this case does not appear in output generated by serialize);

'\texttt{arity-at-least}' for an \texttt{arity-at-least} structure, which fails on deserialization (since arity-at-least are immutable; this case does not appear in output generated by serialize); or

'\texttt{mpi}' for a module path index, which fails on deserialization (since a module path index is immutable; this case does not appear in output generated by serialize).

'\texttt{srcloc} for a srcloc structure, which fails on deserialization (since srclocs are immutable; this case does not appear in output generated by serialize).

The \#f-filled value will be updated with content specified by the fifth element of the serialization list \texttt{v}.

– A non-box represents a \texttt{serial} value to be constructed immediately, and it is one of the following:

  * a boolean, number, character, interned symbol, or empty list, representing itself.
  * a string, representing an immutable string.
  * a byte string, representing an immutable byte string.
  * a pair whose \texttt{car} is ',? and whose \texttt{cdr} is a non-negative exact integer \texttt{i}; it represents the value constructed for the \texttt{i}th element of \texttt{graph}, where \texttt{i} is less than the position of this element within \texttt{graph}.
  * a pair whose \texttt{car} is a number \texttt{i}; it represents an instance of a structure type that is described by the \texttt{i}th element of the \texttt{s-types} list. The \texttt{cdr} of the pair is a list of serials representing arguments to be provided to the structure type’s deserializer.
  * a pair whose \texttt{car} is ',q and whose \texttt{cdr} is an immutable value; it represents the quoted value.
  * a pair whose \texttt{car} is ',f; it represents an instance of a prefab structure type. The \texttt{cadr} of the pair is a prefab structure type key, and the \texttt{cddr} is a list of serials representing the field values.
  * a pair whose \texttt{car} is ',void, representing \#<void>.
  * a pair whose \texttt{car} is ',su and whose \texttt{cdr} is a character string; it represents an unreadable symbol.
* a pair whose car is 'u and whose cdr is either a byte string or character string; it represents a mutable byte or character string.
* a pair whose car is 'p and whose cdr is a byte string; it represents a path using the serializer’s path convention (deprecated in favor of 'p+).
* a pair whose car is 'p+, whose cadr is a byte string, and whose cddr is one of the possible symbol results of system-path-convention-type; it represents a path using the specified convention.
* a pair whose car is 'p+ and whose cdr is a list of byte strings represents a relative path; it will be converted by deserialization based on current-load-relative-directory, falling back to current-directory.
* a pair whose car is 'c and whose cdr is a pair of serials; it represents an immutable pair.
* a pair whose car is 'c! and whose cdr is a pair of serials; it represents a pair (but formerly represented a mutable pair), and does not appear in output generated by serialize.
* a pair whose car is 'm and whose cdr is a pair of serials; it represents a mutable pair.
* a pair whose car is 'v and whose cdr is a list of serials; it represents an immutable vector.
* a pair whose car is 'v! and whose cdr is a list of serials; it represents a mutable vector.
* a pair whose car is 'v1 and whose cdr is a list of serials; it represents a flvector.
* a pair whose car is 'vx and whose cdr is a list of serials; it represents a fxvector.
* a pair whose car is 'b and whose cdr is a serial; it represents an immutable box.
* a pair whose car is 'b! and whose cdr is a serial; it represents a mutable box.
* a pair whose car is 'h, whose cadr is either '!' or ' ' (mutable or immutable, respectively), whose caddr is a list of symbols (containing 'equal, 'weak, both, or neither) that determines the hash table type, and whose cdddr is a list of pairs, where the car of each pair is a serial for a hash-table key and the cdr is a serial for the corresponding value.
* a pair whose car is 'date* and whose cdr is a list of serials; it represents a date* structure.
* a pair whose car is 'date and whose cdr is a list of serials; it represents a date structure.
* a pair whose car is 'arity-at-least and whose cdr is a serial; it represents an arity-at-least structure.
* a pair whose car is 'mpi and whose cdr is a pair; it represents a module path index that joins the paired values.
* a pair whose car is 'srcloc and whose cdr is a list of serials; it represents a srcloc structure.
• A list of pairs, where the car of each pair is a non-negative exact integer $i$ and the cdr is a serial (as defined in the previous bullet). Each element represents an update to an $i$th element of graph that was specified as a box, and the serial describes how to construct a new value with the same shape as specified by the box. The content of this new value must be transferred into the value created for the box in graph.

• A final serial (as defined in the two bullets back) representing the result of deserial-ize.

The result of deserialize shares no mutable values with the argument to deserialize.

If a value provided to serialize is a simple tree (i.e., no sharing), then the fourth and fifth elements in the serialized representation will be empty.

```
(serialized=? v1 v2) -> boolean?
  v1 : any/c
  v2 : any/c
```

Returns #t if $v1$ and $v2$ represent the same serialization information.

More precisely, it returns the same value that (equal? (deserialize v1) (deserial-ize v2)) would return if

• all structure types whose deserializers are accessed with distinct module paths are actually distinct types;

• all structure types are transparent; and

• all structure instances contain only the constituent values recorded in each of $v1$ and $v2$.

```
(deserialize-module-guard)
  -> (-> module-path? symbol? (or/c void? (cons/c module-path? symbol?)))

(deserialize-module-guard guard) -> void?
  guard : (-> module-path? symbol? (or/c void? (cons/c module-path? symbol?)))
```

A parameter whose value is called by deserialize before dynamically loading a module via dynamic-require. The two arguments provided to the procedure are the same as the arguments to be passed to dynamic-require. The procedure can raise an exception to disallow the dynamic-require.

The procedure can optionally return a pair containing a module-path and symbol. If returned, deserialize will use them as arguments to dynamic-require instead.

Changed in version 6.90.0.30 of package base: Adds optional return values for bindings.
Like `struct`, but instances of the structure type are serializable with `serialize`. This form is allowed only at the top level or in a module’s top level (so that deserialization information can be found later).

Serialization supports cycles involving the created structure type only when all fields are mutable (or when the cycle can be broken through some other mutable value).

In addition to the bindings generated by `struct`, `serializable-struct` binds `deserialize-info:id-v0` to deserialization information. Furthermore, in a module context, it automatically provides this binding in a `deserialize-info` submodule using `module+`.

The `serializable-struct` form enables the construction of structure instances from places where `id` is not accessible, since deserialization must construct instances. Furthermore, `serializable-struct` provides limited access to field mutation, but only for instances generated through the deserialization information bound to `deserialize-info:id-v0`. See `make-deserialize-info` for more information.

Beware that the previous paragraph means that if a serializable struct is exported via `contract-out`, for example, the contracts are not checked during deserialization. Consider using `struct-guard/c` instead.

The `-v0` suffix on the deserialization enables future versioning on the structure type through `serializable-struct/versions`.

When a supertype is supplied as `maybe-super`, compile-time information bound to the supertype identifier must include all of the supertype’s field accessors. If any field mutator is missing, the structure type will be treated as immutable for the purposes of marshaling (so cycles involving only instances of the structure type cannot be handled by the deserializer).

Examples:

```lisp
> (serializable-struct point (x y))
> (point-x (deserialize (serialize (point 1 2))))
1
```

Like `serializable-struct`, but with the supertype syntax and default constructor name `define-struct`.

1006
Like `serializable-struct`, but the generated deserializer binding is `deserialize-info:id-v-vers`. In addition, `deserialize-info:id-v-other-vers` is bound for each `other-vers`. The `vers` and each `other-vers` must be a literal, exact, nonnegative integer.

Each `make-proc-expr` should produce a procedure, and the procedure should accept as many argument as fields in the corresponding version of the structure type, and it produces an instance of `id`. Each `cycle-make-proc-expr` should produce a procedure of no arguments; this procedure should return two values: an instance `x` of `id` (typically with `#f` for all fields) and a procedure that accepts another instance of `id` and copies its field values into `x`.

Examples:

```scheme
> (serializable-struct point (x y) #:mutable #:transparent)
> (define ps (serialize (point 1 2)))
> (deserialize ps)
(point 1 2)
> (define x (point 1 10))
> (set-point-x! x x)
> (define xs (serialize x))
> (deserialize xs)
#0=(point #0# 10)
> (serializable-struct/versions point 1 (x y z)
  (0
   ; Constructor for simple v0 instances:
   (lambda (x y) (point x y 0))
   ; Constructor for v0 instance in a cycle:
   (lambda ()
     (let ((p0 (point #f #f 0)))
       (values
        p0
        (lambda (p)
          (set-point-x! p0 (point-x p))
          (set-point-y! p0 (point-y p)))))))
  #:mutable #:transparent))
> (deserialize (serialize (point 4 5 6)))
(point 4 5 6)
> (deserialize ps)
(point 1 2 0)
```
Like `serializable-struct/versions`, but with the supertype syntax and default constructor name of `define-struct`.

```
(make-deserialize-info make cycle-make) → any
make : procedure?
cycle-make : (-> (values any/c procedure?))
```

Produces a deserialization information record to be used by `deserialize`. This information is normally tied to a particular structure because the structure has a `prop:serializable` property value that points to a top-level variable or module-exported variable that is bound to deserialization information.

The `make` procedure should accept as many arguments as the structure’s serializer put into a vector; normally, this is the number of fields in the structure. It should return an instance of the structure.

The `cycle-make` procedure should accept no arguments, and it should return two values: a structure instance `x` (with dummy field values) and an update procedure. The update procedure takes another structure instance generated by the `make`, and it transfers the field values of this instance into `x`.

```
prop:serializable : property?
```

This property identifies structures and structure types that are serializable. The property value should be constructed with `make-serialize-info`.

```
(make-serialize-info to-vector
deserialize-id
can-cycle?
dir) → any
to-vector : (any/c . -> . vector?)
deserialize-id : (or identifier? symbol?
  (cons/c symbol? module-path-index?)
  (-> any/c))
can-cycle? : any/c
dir : path-string?
```
Produces a value to be associated with a structure type through the prop:serializable property. This value is used by serialize.

The to-vector procedure should accept a structure instance and produce a vector for the instance’s content.

The deserialize-id value indicates a binding for deserialize information, to either a module export or a top-level definition. It must be one of the following:

- If deserialize-id is an identifier, and if (identifier-binding deserialize-id) produces a list, then the third element is used for the exporting module, otherwise the top-level is assumed. Before trying an exporting module directly, its deserialize-info submodule is tried; the module itself is tried if no deserialize-info submodule is available or if the export is not found. In either case, syntax-e is used to obtain the name of an exported identifier or top-level definition.

- If deserialize-id is a symbol, it indicates a top-level variable that is named by the symbol.

- If deserialize-id is a pair, the car must be a symbol to name an exported identifier, and the cdr must be a module path index to specify the exporting module.

- If deserialize-id is a procedure, then it is applied during serialization and its result is used for deserialize-id.

See make-deserialize-info and deserialize for more information.

The can-cycle? argument should be false if instances should not be serialized in such a way that deserialization requires creating a structure instance with dummy field values and then updating the instance later.

The dir argument should be a directory path that is used to resolve a module reference for the binding of deserialize-id. This directory path is used as a last resort when deserialize-id indicates a module that was loaded through a relative path with respect to the top level. Usually, it should be (or (current-load-relative-directory) (current-directory)).

Changed in version 7.0.0.6 of package base: Allow deserialize-id to be a procedure.

Examples:

```scheme
> (struct pie (type)
  #:mutable
  #:property prop:serializable
  (make-serialize-info
    (lambda (this)
      ...))
```

1009
> (define pie-beam
  (make-deserialize-info
   (λ (type)
      (pie type))
   (λ ()
      (define pie-pattern (pie 'transporter-error))
      (values pie-pattern
               (λ (type)
                  (set-pie-type! pie-pattern type))))))

> (define original-pie
  (pie 'apple))
> original-pie
#<pie>
> (define pie-in-transit
  (serialize original-pie))
> pie-in-transit
'((3) 1 ((#f . pie-beam)) 0 () () (0 apple))
> (define beamed-up-pie
  (deserialize pie-in-transit))
> beamed-up-pie
#<pie>
> (pie-type beamed-up-pie)
'apple
> (equal? beamed-up-pie original-pie)
#f

13.10 Fast-Load Serialization

(require racket/fasl)  package: base

The bindings documented in this section are provided by the racket/fasl library, not racket/base or racket.
The `s-exp->fasl` function serializes `v` to a byte string, printing it directly to `out` if `out` is an output port or returning the byte string otherwise. The `fasl->s-exp` function decodes a value from a byte string (supplied either directly or as an input port) that was encoded with `s-exp->fasl`.

The `v` argument must be a value that could be quoted as a literal—that is, a value without syntax objects for which `(compile `\',v)` would work and be readable after `write`—or it can include correlated objects mixed with those values. The byte string produced by `s-exp->fasl` does not use the same format as compiled code, however.

If a value within `v` is not valid as a quoted literal, and if `handle-fail` is not `#f`, then `handle-fail` is called on the nested value, and the result of `handle-fail` is written in that value’s place. The `handle-fail` procedure might raise an exception instead of returning a replacement value. If `handle-fail` is `#f`, then the `exn:fail:contract` exception is raised when an invalid value is encountered.

If `external-lift?` is not `#f`, then it receives each value `v-sub` encountered in `v` by `s-exp->fasl`. If the result of `external-lift?` on `v-sub` is true, then `v-sub` is not encoded in the result, and it instead treated as externally lifted. A deserializing `fasl->s-exp` receives a `external-lifts` vector that has one value for each externally lifted value, in the same order as passed to `external-lift?` on serialization.

Like `(compile `\',v)`, `s-exp->fasl` does not preserve graph structure, support cycles, or handle non-prefab structures. Compose `s-exp->fasl` with `serialize` to preserve graph
structure, handle cyclic data, and encode serializable structures. The `s-exp->fasl` and `fasl->s-exp` functions consult `current-write-relative-directory` and `current-load-relative-directory` (falling back to `current-directory`), respectively, in the same way as bytecode saving and loading to store paths in relative form, and they similarly allow and convert constrained `srcloc` values (see §14.16 “Printing Compiled Code”).

Unless `keep-mutable?` is provided as true to `s-exp->fasl`, then mutable values in `v` are replaced by immutable values when the result is decoded by `fasl->s-exp`. Unless `datum-intern?` is provided as `#f`, then any immutable value produced by `fasl->s-exp` is filtered by `datum-intern-literal`. The defaults make the composition of `s-exp->fasl` and `fasl->s-exp` behave like the composition of `write` and `read`.

If `skip-prefix?` is not `#f`, then a prefix that identifies the stream as a serialization is not written by `s-exp->fasl` or read by `fasl->s-exp`. Omitting a prefix can save a small amount of space, which can useful when serializing small values, but it gives up a sanity check on the `fasl->s-exp` that is often useful.

The byte-string encoding produced by `s-exp->fasl` is independent of the Racket version, except as future Racket versions introduce extensions that are not currently recognized. In particular, the result of `s-exp->fasl` will be valid as input to any future version of `fasl->s-exp` (as long as the `skip-prefix?` arguments are consistent).

Examples:

> `(define fasl (s-exp->fasl (list #("speed") 'racer #\!)))
> fasl
"racket/fasl:\0\24\34\3 \1\23\5speed\16\5racer\r!"
> (fasl->s-exp fasl)
'(#("speed") racer #\!)

Changed in version 6.90.0.21 of package base: Made `s-exp->fasl` format version-independent and added the `#:keep-mutable?` and `#:datum-intern?` arguments.

Changed in version 7.3.0.7: Added support for correlated objects.

Changed in version 7.5.0.3: Added the `#:handle-fail` argument.

Changed in version 7.5.0.9: Added the `#:external-lift?` and `#:external-lifts` arguments.

13.11 Cryptographic Hashing

```racket
(sha1-bytes in [start end]) → bytes?
in : (or/c bytes? input-port?)
start : exact-nonnegative-integer? = 0
end : (or/c #f exact-nonnegative-integer?) = #f
(sha224-bytes in [start end]) → bytes?
in : (or/c bytes? input-port?)
start : exact-nonnegative-integer? = 0
end : (or/c #f exact-nonnegative-integer?) = #f
```
(sha256-bytes in [start end]) → bytes?
  in : (or/c bytes? input-port?)
  start : exact-nonnegative-integer? = 0
  end : (or/c #f exact-nonnegative-integer?) = #f

Computes the SHA-1, SHA-224, or SHA-256 hash of a byte sequence and returns the hash as a byte string with 20 bytes, 28 bytes, or 32 bytes, respectively.

The start and end arguments determine the range of bytes of the input that are used to compute the hash. An end value of #f corresponds to the end of the byte string or an end-of-file position for an input port. When in is a byte string, the start and end values (when non #f) must be no greater than the length of the byte string, and start must be no greater than end. When in is an input port, start must be no greater than end; if in supplies less than start or end bytes before an end-of-file, then start and/or end is effectively changed to the number of supplied bytes (so that an empty or truncated byte sequence is hashed). When in is an input port and end is a number, then at most end bytes are read from the input port.

For security purposes, favor sha224-bytes and sha256-bytes (which are part of the SHA-2 family) over sha1-bytes.

Use bytes->hex-string from file/sha1 to convert a byte string hash to a human-readable string.

Examples:

```
> (sha1-bytes "abc")
"\251\231>6G\6\201j\272>\%qxP\302\234\320\330\235"
> (require file/sha1)
> (bytes->hex-string (sha1-bytes "abc"))
"a9993e364706816aba3e25717850c26c9cd0d89d"
> (bytes->hex-string (sha224-bytes "abc"))
"23097d223405d8228642a477bda255b32aadbce4bda0b3f7e36c9da7"
> (bytes->hex-string (sha224-bytes (open-input-string "xabcy") 1 4))
"23097d223405d8228642a477bda255b32aadbce4bda0b3f7e36c9da7"
```

Added in version 7.0.0.5 of package base.
14 Reflection and Security

14.1 Namespaces

See §1.2.5 “Namespaces” for basic information on the namespace model.

A new namespace is created with procedures like `make-empty-namespace`, and `make-base-namespace`, which return a first-class namespace value. A namespace is used by setting the `current-namespace` parameter value, or by providing the namespace to procedures such as `eval` and `eval-syntax`.

```
(define (namespace? v) (boolean? v)
  v : any/c)
```

Returns `#t` if `v` is a namespace value, `#f` otherwise.

```
(define (make-empty-namespace) (namespace?)
```

Creates a new namespace that is empty, and whose module registry contains only mappings for some internal, predefined modules, such as `'#%kernel`. The namespace’s base phase is the same as the base phase of the current namespace. Attach modules from an existing namespace to the new one with `namespace-attach-module`.

The new namespace is associated with a new `root namespace`, which has the same module registry as the returned namespace and has a base phase of 0. The new root namespace is the same as the returned namespace if both have base phase 0.

```
(define (make-base-empty-namespace) (namespace?)
```

Creates a new empty namespace like `make-empty-namespace`, but with `racket/base` attached. The namespace’s base phase is the same as the phase in which the `make-base-empty-namespace` function was created.

```
(define (make-base-namespace) (namespace?)
```

Creates a new namespace like `make-empty-namespace`, but with `racket/base` attached and required into the top-level environment. The namespace’s base phase is the same as the phase in which the `make-base-namespace` function was created.

```
(define (define-namespace-anchor id)
```

Binds `id` to a namespace anchor that can be used with `namespace-anchor->empty-namespace` and `namespace-anchor->namespace`.

This form can be used only in a top-level context or in a module-context.
(namespace-anchor? v) → boolean?
  v : any/c

Returns #t if v is a namespace-anchor value, #f otherwise.

(namespace-anchor->empty-namespace a) → namespace?
  a : namespace-anchor?

Returns an empty namespace that shares a module registry and root namespace with the source of the anchor, and whose base phase is the phase in which the anchor was created.

If the anchor is from a define-namespace-anchor form in a module context, then the source is the namespace in which the containing module is instantiated. If the anchor is from a define-namespace-anchor form in a top-level content, then the source is the namespace in which the anchor definition was evaluated.

(namespace-anchor->namespace a) → namespace?
  a : namespace-anchor?

Returns a namespace corresponding to the source of the anchor.

If the anchor is from a define-namespace-anchor form in a module context, then the result is a namespace for the module’s body in the anchor’s phase. The result is the same as a namespace obtained via module->namespace, and the module is similarly made available if it is not available already.

If the anchor is from a define-namespace-anchor form in a top-level content, then the result is the namespace in which the anchor definition was evaluated.

(current-namespace) → namespace?
  (current-namespace n) → void?
  n : namespace?

A parameter that determines the current namespace.

(namespace-symbol->identifier sym) → identifier?
  sym : symbol?

Similar to datum->syntax restricted to symbols. The lexical information of the resulting identifier corresponds to the top-level environment of the current namespace; the identifier has no source location or properties.

(namespace-base-phase [namespace]) → exact-integer?
  namespace : namespace? = (current-namespace)

1015
Returns the base phase of \texttt{namespace}.

\begin{verbatim}
\begin{code}
(namespace-module-identifier [where]) \to \texttt{identifier}?
  where : (or/c namespace? exact-integer? #f)
    = (current-namespace)
\end{code}
\end{verbatim}

Returns an identifier whose binding is module in the base phase of \texttt{where} if it is a namespace, or in the \texttt{where} phase level otherwise.

The lexical information of the identifier includes bindings (in the same phase level) for all syntactic forms that appear in fully expanded code (see \S1.2.3.1 “Fully Expanded Programs”), but using the name reported by the second element of \texttt{identifier-binding} for the binding; the lexical information may also include other bindings.

\begin{verbatim}
\begin{code}
(namespace-variable-value \texttt{sym}
  [use-mapping?]
  [failure-thunk]
  [namespace]) \to any

  sym : symbol?
  use-mapping? : any/c = #t
  failure-thunk : (or/c (\to any) #f) = #f
  namespace : namespace? = (current-namespace)
\end{code}
\end{verbatim}

Returns a value for \texttt{sym} in \texttt{namespace}, using \texttt{namespace}’s base phase. The returned value depends on \texttt{use-mapping}?:

- If \texttt{use-mapping}? is true (the default), and if \texttt{sym} maps to a top-level variable or an imported variable (see \S1.2.5 “Namespaces”), then the result is the same as evaluating \texttt{sym} as an expression. If \texttt{sym} maps to syntax or imported syntax, then \texttt{failure-thunk} is called or the \texttt{exn:fail:syntax} exception is raised. If \texttt{sym} is mapped to an undefined variable or an uninitialized module variable, then \texttt{failure-thunk} is called or the \texttt{exn:fail:contract:variable} exception is raised.

- If \texttt{use-mapping}? is \#f, the namespace’s syntax and import mappings are ignored. Instead, the value of the top-level variable named \texttt{sym} in \texttt{namespace} is returned. If the variable is undefined, then \texttt{failure-thunk} is called or the \texttt{exn:fail:contract:variable} exception is raised.

If \texttt{failure-thunk} is not \#f, \texttt{namespace-variable-value} calls \texttt{failure-thunk} to produce the return value in place of raising an \texttt{exn:fail:contract:variable} or \texttt{exn:fail:syntax} exception.

\begin{verbatim}
\begin{code}
(namespace-set-variable-value!
  \texttt{sym}
  \texttt{v}
  [map?]
  [namespace
    as-constant?]) \to void?
\end{code}
\end{verbatim}
Sets the value of `sym` in the top-level environment of `namespace` in the base phase, defining `sym` if it is not already defined.

If `map?` is supplied as true, then the namespace’s identifier mapping is also adjusted (see §1.2.5 “Namespaces”) in the phase level corresponding to the base phase, so that `sym` maps to the variable.

If `as-constant?` is true, then the variable is made a constant (so future assignments are rejected) after `v` is installed as the value.

Changed in version 6.90.0.14 of package `base`: Added the `as-constant?` argument.

```lisp
(namespace-undefine-variable! sym [namespace]) → void?
sym : symbol?
namespace : namespace? = (current-namespace)
```

Removes the `sym` variable, if any, in the top-level environment of `namespace` in its base phase. The namespace’s identifier mapping (see §1.2.5 “Namespaces”) is unaffected.

```lisp
(namespace-mapped-symbols [namespace]) → (listof symbol?)
namespace : namespace? = (current-namespace)
```

Returns a list of all symbols that are mapped to variables, syntax, and imports in `namespace` for the phase level corresponding to the namespace’s base phase.

```lisp
(namespace-require quoted-raw-require-spec [namespace]) → void?
quoted-raw-require-spec : any/c
namespace : namespace? = (current-namespace)
```

Performs the import corresponding to `quoted-raw-require-spec` in the top-level environment of `namespace`, like a top-level `#%require`. The `quoted-raw-require-spec` argument must be either a datum that corresponds to a quoted `raw-require-spec` for `#%require`, which includes module paths, or it can be a resolved module path.

Module paths in `quoted-raw-require-spec` are resolved with respect to `current-load-relative-directory` or `current-directory` (if the former is `#f`), even if the current namespace corresponds to a module body.

Changed in version 6.90.0.16 of package `base`: Added the `namespace` optional argument.

1017
Like `namespace-require` for syntax exported from the module, but exported variables at the namespace’s base phase are treated differently: the export’s current value is copied to a top-level variable in `namespace`.

Changed in version 6.90.0.16 of package base: Added the `namespace` optional argument.

Like `namespace-require`, but for each exported variable at the namespace’s base phase, the export’s value is copied to a corresponding top-level variable that is made immutable. Despite setting the top-level variable, the corresponding identifier is bound as imported.

Changed in version 6.90.0.16 of package base: Added the `namespace` optional argument.

Like `namespace-require`, but only the transformer part of the module is executed relative to `namespace`’s base phase; that is, the module is merely visited, and not instantiated (see §1.2.3.9 “Module Expansion, Phases, and Visits”). If the required module has not been instantiated before, the module’s variables remain undefined.

Changed in version 6.90.0.16 of package base: Added the `namespace` optional argument.

Attaches the instantiated module named by `modname` in `src-namespace` (at its base phase) to the module registry of `dest-namespace`. 

1018
In addition to `modname`, every module that it imports (directly or indirectly) is also recorded in the current namespace’s module registry, and instances at the same phase are also attached to `dest-namespace` (while visits at the module’s phase and instances at higher or lower phases are not attached, nor even made available for on-demand visits). The inspector of the module invocation in `dest-namespace` is the same as inspector of the invocation in `src-namespace`.

If `modname` is not a symbol, the current module name resolver is called to resolve the path, but no module is loaded; the resolved form of `modname` is used as the module name in `dest-namespace`.

If `modname` refers to a submodule or a module with submodules, unless the module was loaded from bytecode (i.e., a "zo" file) independently from submodules within the same top-level module, then declarations for all submodules within the module’s top-level module are also attached to `dest-namespace`.

If `modname` does not refer to an instantiated module in `src-namespace`, or if the name of any module to be attached already has a different declaration or same-phase instance in `dest-namespace`, then the `exn:fail:contract` exception is raised.

If `src-namespace` and `dest-namespace` do not have the same base phase, then the `exn:fail:contract` exception is raised.

Unlike `namespace-require`, `namespace-attach-module` does not instantiate the module, but copies the module instance from the source namespace to the target namespace.

Examples:

> (module food racket/base
  (provide apple)
  (define apple (list "pie")))

> (namespace-require ''food)
> (define ns (current-namespace))
> (parameterize ([current-namespace (make-base-namespace)]
                [dest-namespace ns])
  (namespace-require ''food))

`require: unknown module module name: 'food`

> (parameterize ([current-namespace (make-base-namespace)]
                [dest-namespace ns] [ns 'food])
  (namespace-require 'food)
  (eq? (eval 'apple) apple))

#t

> (parameterize ([current-namespace (make-base-namespace)]
                [dest-namespace ns] [ns 'food])
  (namespace-attach-module-declaration ns 'food)
  (namespace-require 'food)
  (eq? (eval 'apple) apple))

#f
(namespace-attach-module-declaration src-namespace modname [dest-namespace]) → void?

src-namespace : namespace?
modname : module-path?
dest-namespace : namespace? = (current-namespace)

Like namespace-attach-module, but the module specified by modname need only be declared (and not necessarily instantiated) in src-namespace, and the module is merely declared in dest-namespace.

(namespace-unprotect-module inspector modname [namespace]) → void?

inspector : inspector?
modname : module-path?
namespace : namespace? = (current-namespace)

Changes the inspector for the instance of the module referenced by modname in namespace’s module registry so that it is controlled by the current code inspector. The given inspector must currently control the invocation of the module in namespace’s module registry, otherwise the inspector is not changed. See also §14.10 “Code Inspectors”.

(namespace-module-registry namespace) → any

namespace : namespace?

Returns the module registry of the given namespace. This value is useful only for identification via eq?.

(namespace-call-with-registry-lock namespace thunk) → any

namespace : namespace?

thunk : (-> any)

Calls thunk while holding a reentrant lock for the namespace’s module registry.

Namespace functions do not automatically use the registry lock, but it can be used via namespace-call-with-registry-lock among threads that load and instantiate modules to avoid internal race conditions. On-demand instantiation of available modules also takes the lock; see §1.2.3.9 “Module Expansion, Phases, and Visits”.

Added in version 8.1.0.5 of package base.

(module->namespace mod [src-namespace]) → namespace?
mod : (or/c module-path?
    resolved-module-path?
    module-path-index?)

src-namespace : namespace? = (current-namespace)

Returns a namespace that corresponds to the body of an instantiated module in src-namespace’s module registry and in the src-namespace’s base phase, making the module available for on-demand visits at src-namespace’s base phase. The returned namespace has the same module registry as src-namespace. Modifying a binding in the resulting namespace changes the binding seen in modules that require the namespace’s module.

Module paths in a top-level require expression are resolved with respect to the namespace’s module. New provide declarations are not allowed.

If the current code inspector does not control the invocation of the module in src-namespace’s module registry, the exn:fail:contract exception is raised; see also §14.10 “Code Inspectors”.

Bindings in the result namespace cannot be modified if the compile-enforce-module-constants parameter was true when the module was declared, unless the module declaration itself included assignments to the binding via set!.

Changed in version 6.90.0.16 of package base: Added the src-namespace optional argument.

(namespace-syntax-introduce stx [namespace]) → syntax?

stx : syntax?

namespace : namespace? = (current-namespace)

Returns a syntax object like stx, except that namespace’s bindings are included in the syntax object’s lexical information (see §1.2.2 “Syntax Objects”). The additional context is overridden by any existing top-level bindings in the syntax object’s lexical information, or by any existing or future module bindings in the lexical information.

Changed in version 6.90.0.16 of package base: Added the namespace optional argument.

(module-provide-protected? module-path-index
    sym) → boolean?

module-path-index : (or/c symbol? module-path-index?)

sym : symbol?

Returns #f if the module declaration for module-path-index defines sym and exports it unprotected, #t otherwise (which may mean that the symbol corresponds to an unexported definition, a protected export, or an identifier that is not defined at all within the module).

The module-path-index argument can be a symbol; see §14.4.2 “Compiled Modules and References” for more information on module path indices.
Typically, the arguments to `module-provide-protected?` correspond to the first two elements of a list produced by `identifier-binding`.

```scheme
(variable-reference? v) -> boolean?
  v : any/c
```

Return `#t` if `v` is a variable reference produced by `variable-reference`, `#f` otherwise.

```scheme
(variable-reference-constant? varref) -> boolean?
  varref : variable-reference?
```

Returns `#t` if the variable represented by `varref` will retain its current value (i.e., `varref` refers to a variable that cannot be further modified by `set!` or `define`), `#f` otherwise.

```scheme
(variable-reference->empty-namespace varref) -> namespace?
  varref : variable-reference?
```

Returns an empty namespace that shares module declarations and instances with the namespace in which `varref` is instantiated, and with the same phase as `varref`.

```scheme
(variable-reference->namespace varref) -> namespace?
  varref : variable-reference?
```

If `varref` refers to a module-level variable, then the result is a namespace for the module’s body in the referenced variable’s phase; the result is the same as a namespace obtained via `module->namespace`, and the module is similarly made available if it is not available already.

If `varref` refers to a top-level variable, then the result is the namespace in which the referenced variable is defined.

```scheme
(variable-reference->resolved-module-path varref) -> (or/c resolved-module-path? #f)
  varref : variable-reference?
```

If `varref` refers to a module-level variable, the result is a resolved module path naming the module.

If `varref` refers to a top-level variable, then the result is `#f`.

```scheme
(variable-reference->module-path-index varref) -> (or/c module-path-index? #f)
  varref : variable-reference?
```

If `varref` refers to a module-level variable, the result is a module path index naming the module.

If `varref` refers to a top-level variable, then the result is `#f`.  

| 1022 |
(variable-reference->module-source varref)
→ (or/c symbol? (and/c path? complete-path?) #f)
varref : variable-reference?

If varref refers to a module-level variable, the result is a path or symbol naming the module’s source (which is typically, but not always, the same as in the resolved module path). If the relevant module is a submodule, the result corresponds to the enclosing top-level module’s source.

If varref refers to a top-level variable, then the result is #f.

(variable-reference->phase varref) → exact-nonnegative-integer?
varref : variable-reference?

Returns the phase of the variable referenced by varref.

(variable-reference->module-base-phase varref) → exact-integer?
varref : variable-reference?

Returns the phase in which the module is instantiated for the variable referenced by varref, or 0 if the variable for varref is not within a module.

For a variable with a module, the result is less than the result of (variable-reference->phase varref) by n when the variable is bound at phase level n within the module.

(variable-reference->module-declaration-inspector varref)
→ inspector?
varref : variable-reference?

Returns the declaration inspector (see §14.10 “Code Inspectors”) for the module of varref, where varref must refer to an anonymous module variable as produced by (#%variable-reference).

(variable-reference-from-unsafe? varref) → boolean?
varref : variable-reference?

Returns #t if the module of the variable reference itself (not necessarily a referenced variable) is compiled in unsafe mode, #f otherwise.

The variable-reference-from-unsafe? procedure is intended for use as

(variable-reference-from-unsafe? (#%variable-reference))

which the compiler can optimize to a literal #t or #f (since the enclosing module is being compiled in unsafe mode or not).

1023
Currently unsafe mode can be controlled only through the linklet interface, but future changes may make unsafe mode more accessible at the module level.

Added in version 6.12.0.4 of package base.

### 14.2 Evaluation and Compilation

Racket provides programmatic control over evaluation through `eval` and related functions. See §18.6 “Controlling and Inspecting Compilation” for information about extra-linguistic facilities related to the Racket compiler.

\[
\begin{align*}
\text{(current-eval)} &\rightarrow (\text{any/c} \cdot \rightarrow \cdot \text{any}) \\
\text{(current-eval proc)} &\rightarrow \text{void?} \\
\text{proc} &\rightarrow (\text{any/c} \cdot \rightarrow \cdot \text{any})
\end{align*}
\]

A parameter that determines the current evaluation handler. The evaluation handler is a procedure that takes a top-level form and evaluates it, returning the resulting values. The evaluation handler is called by `eval`, `eval-syntax`, the default load handler, and `read-eval-print-loop` to evaluate a top-level form. The handler should evaluate its argument in tail position.

The top-level-form provided to the handler can be a syntax object, a compiled form, a compiled form wrapped as a syntax object, or an arbitrary datum.

The default handler converts an arbitrary datum to a syntax object using `datum->syntax`, and then enriches its lexical information in the same way as `eval`. (If top-level-form is a syntax object, then its lexical information is not enriched.) The default evaluation handler partially expands the form to splice the body of top-level `begin` forms into the top level (see `expand-to-top-form`), and then individually compiles and evaluates each spliced form before continuing to expand, compile, and evaluate later forms.

\[
\begin{align*}
\text{(eval top-level-form [namespace])} &\rightarrow \text{any} \\
\text{top-level-form} &\rightarrow \text{any/c} \\
\text{namespace} &\rightarrow \text{namespace?} = \text{(current-namespace)}
\end{align*}
\]

Calls the current evaluation handler to evaluate top-level-form. The evaluation handler is called in tail position with respect to the `eval` call, and parameterized to set current-namespace to namespace.

If top-level-form is a syntax object whose datum is not a compiled form, then its lexical information is enriched before it is sent to the evaluation handler:

- If top-level-form is a pair whose car is a symbol or identifier, and if applying `namespace-syntax-introduce` to the (datum->syntax-converted) identifier pro-
duces an identifier bound to module in a phase level that corresponds to namespace’s base phase, then only that identifier is enriched.

• For any other top-level-form, namespace-syntax-introduce is applied to the entire syntax object.

For interactive evaluation in the style of read-eval-print-loop and load, wrap each expression with #%top-interaction, which is normally bound to #%top-interaction, before passing it to eval.

\[
\text{(eval-syntax stx [namespace])} \rightarrow \text{any}
\]

\[
stx : \text{syntax}?
\]

\[
\text{namespace : namespace?} = (\text{current-namespace})
\]

Like eval, except that stx must be a syntax object, and its lexical context is not enriched before it is passed to the evaluation handler.

\[
\text{(current-load)}
\]

\[
\rightarrow (\text{path? (or/c #f symbol?) symbol?})
\]

\[
\text{(cons/c (or/c #f symbol?) (non-empty-listof symbol?) )}
\]

\[
. \rightarrow .
\]

\[
\text{any)
}\]

\[
\text{(current-load proc) \rightarrow void?}
\]

\[
\text{proc : (path? (or/c #f symbol?) symbol?)}
\]

\[
\text{(cons/c (or/c #f symbol?) (non-empty-listof symbol?) )}
\]

\[
. \rightarrow .
\]

\[
\text{any)
}\]

A parameter that determines the current load handler to load top-level forms from a file. The load handler is called by load, load-relative, load/cd, and the default compiled-load handler.

A load handler takes two arguments: a path (see §15.1 “Paths”) and an expected module name. The expected module name is a symbol or a list when the call is to load a module declaration in response to a require (in which case the file should contain a module declaration), or #f for any other load.

When loading a module from a stream that starts with a compiled module that contains submodules, the load handler should load only the requested module, where a symbol as the load handler’s indicates the root module and a list indicates a submodule whose path relative to the root module is given by the cdr of the list. The list starts with #f when a submodule should be loaded only if it can be loaded independently (i.e., from compiled form—never
from source); if the submodule cannot be loaded independently, the load handler should return without loading from a file. When the expected module name is a list that starts with a symbol, the root module and any other submodules can be loaded from the given file, which might be from source, and the load handler still should not complain if the expected submodule is not found. When loading modules from a nonexistent source file, the load handler may raise an exception regardless of whether submodules are requested or not.

The default load handler reads forms from the file in `read-syntax` mode with line-counting enabled for the file port, unless the path has a `".zo"` suffix. It also parameterizes each read to set `read-accept-compiled`, `read-accept-reader`, and `read-accept-lang` to `#t`. In addition, if `load-on-demand-enabled` is `#t`, then `read-on-demand-source` is set to the cleansed, absolute form of `path` during the `read-syntax` call. After reading a single form, the form is passed to the current evaluation handler, wrapping the evaluation in a continuation prompt (see `call-with-continuation-prompt`) for the default continuation prompt tag with handler that propagates the abort to the continuation of the `load` call.

If the second argument to the load handler is a symbol, then:

- The `read-syntax` from the file is additionally parameterized as follows (to provide consistent reading of module source):

  ```lisp
  (current-readtable #f)
  (read-case-sensitive #t)
  (read-square-bracket-as-paren #t)
  (read-curly-brace-as-paren #t)
  (read-accept-box #t)
  (read-accept-compiled #t)
  (read-accept-bar-quote #t)
  (read-accept-graph #t)
  (read-decimal-as-inexact #t)
  (read-accept-dot #t)
  (read-accept-infix-dot #t)
  (read-accept-quasiquote #t)
  (read-accept-reader #t)
  (read-accept-lang #t)
  ```

- If the read result is not a `module` form, or if a second `read-syntax` does not produce an end-of-file, then the `exn:fail` exception is raised without evaluating the form that was read from the file. (In previous versions, the module declaration was checked to match the name given as the second argument to the load handler, but this check is no longer performed.)

- The lexical information of the initial `module` identifier is enriched with a binding for `module`, so that the form corresponds to a module declaration independent of the current namespace’s bindings.
If the second argument to the load handler is \#f, then each expression read from the file is wrapped with \#%top-interaction, which is normally bound to \#%top-interaction, before passing it to the evaluation handler.

The return value from the default load handler is the value of the last form from the loaded file, or \#<void> if the file contains no forms. If the given path is a relative path, then it is resolved using the value of \texttt{current-directory}.

\[
\begin{align*}
\text{(load file)} & \rightarrow \text{any} \\
\text{file} & : \text{path-string?}
\end{align*}
\]

Calls the current load handler in tail position. The call is parameterized to set \texttt{current-load-relative-directory} to the directory of \texttt{file}, which is resolved relative to the value of \texttt{current-directory}.

\[
\begin{align*}
\text{(load-relative file)} & \rightarrow \text{any} \\
\text{file} & : \text{path-string?}
\end{align*}
\]

Like \texttt{load/use-compiled}, but when \texttt{file} is a relative path, it is resolved using the value of \texttt{current-load-relative-directory} instead of the value of \texttt{current-directory} if the former is not \#f, otherwise \texttt{current-directory} is used.

\[
\begin{align*}
\text{(load/cd file)} & \rightarrow \text{any} \\
\text{file} & : \text{path-string?}
\end{align*}
\]

Like \texttt{load}, but \texttt{load/cd} sets both \texttt{current-directory} and \texttt{current-load-relative-directory} before calling the load handler.

\[
\begin{align*}
\text{(current-load-extension)} & \rightarrow \text{(path? (or/c \#f symbol?) symbol?) (non-empty-listof symbol?))} \\
\text{proc} : \text{(path? (or/c \#f symbol?) symbol?) (non-empty-listof symbol?))} & \rightarrow \text{any}
\end{align*}
\]

A parameter that determines a \texttt{extension-load handler}, which is called by \texttt{load-extension} and the default compiled-load handler.
An extension-load handler takes the same arguments as a load handler, but the file should be a platform-specific dynamic extension, typically with the file suffix ".so" (Unix), ".dll" (Windows), or ".dylib" (Mac OS). The file is loaded using internal, OS-specific primitives. See Inside: Racket C API for more information on dynamic extensions.

Extensions are supported only when `(system-type 'vm)` returns 'racket.

```
(load-extension file) → any
  file : path-string?
```

Sets `current-load-relative-directory` like `load`, and calls the extension-load handler in tail position.

Extensions are supported only when `(system-type 'vm)` returns 'racket.

```
(load-relative-extension file) → any
  file : path-string?
```

Like `load-extension`, but resolves `file` using `current-load-relative-directory` like `load-relative`.

Extensions are supported only when `(system-type 'vm)` returns 'racket.

```
(current-load/use-compiled)
  → (path? (or/c #f symbol?
          (cons/c (or/c #f symbol?)
                 (non-empty-listof symbol?)))
          . -> . any)

(current-load/use-compiled proc) → void?
  proc : (path? (or/c #f symbol?
                 (cons/c (or/c #f symbol?)
                        (non-empty-listof symbol?)))
          . -> . any)
```

A parameter that determines the current compiled-load handler to load from a file that may have a compiled form. The compiled-load handler is called by `load/use-compiled`.

The protocol for a compiled-load handler is the same as for the load handler (see `current-load`), except that a compiled-load handler is expected to set `current-load-relative-directory` itself. Additionally, the default compiled-load handler does the following:

- When the given path ends with ".rkt", no "rkt" file exists, and when the handler’s second argument is not #f, the default compiled-load handler checks for a ".ss" file.
• The default compiled-load handler checks for the opportunity to load from ".zo" (bytecode) files and, when (system-type 'vm) returns 'racket, for ".so" (native Unix), "dll" (native Windows), or ".dylib" (native Mac OS) files.

• When the default compiled-load handler needs to load from the given path, the given path does not exist, and when the handler’s second argument is not #f, the default compiled-load handler returns without raising an exception.

The check for a compiled file occurs whenever the given path file ends with any extension (e.g., ".rkt" or ".scrbl"), and the check consults the subdirectories indicated by the current-compiled-file-roots and use-compiled-file-paths parameters relative to file, where the former supplies “roots” for compiled files and the latter provides subdirectories. A “root” can be an absolute path, in which case file’s directory is combined with reroot-path and the root as the second argument; if the “root” is a relative path, then the relative path is instead suffixed onto the directory of file. The roots are tried in order, and the subdirectories are checked in order within each root. A ".zo" version of the file (whose name is formed by passing file and ".zo" to path-add-extension) is loaded if it exists directly in one of the indicated subdirectories, or when (system-type 'vm) returns 'racket, then a ".so"/.dll"/.dylib" version of the file is loaded if it exists within a "native" subdirectory of a use-compiled-file-paths directory, in an even deeper subdirectory as named by system-library-subpath. A compiled file is loaded only if it checks out according to (use-compiled-file-check); with the default parameter value of 'modify-seconds, a compiled file is used only if its modification date is not older than the date for file. If both ".zo" and "so"/.dll"/.dylib" files are available when (system-type 'vm) returns 'racket, the "so"/.dll"/.dylib" file is used. If file ends with "rkt", no such file exists, the handler’s second argument is a symbol, and a "ss" file exists, then ".zo" and "so"/.dll"/.dylib" files are used only with names based on file with its suffixed replaced by "ss".

While a "zo", "so", "dll", or "dylib" file is loaded, the current load-relative directory is set to the directory of the original file. If the file to be loaded has the suffix "ss" while the requested file has the suffix "rkt", then the current-module-declare-source parameter is set to the full path of the loaded file, otherwise the current-module-declare-source parameter is set to #f.

If the original file is loaded or a "zo" variant is loaded, the load handler is called to load the file. If any other kind of file is loaded, the extension-load handler is called.

When the default compiled-load handler loads a module from a bytecode (i.e., "zo") file, the handler records the bytecode file path in the current namespace’s module registry. More specifically, the handler records the path for the top-level module of the loaded module, which is an enclosing module if the loaded module is a submodule. Thereafter, loads via the default compiled-load handler for modules within the same top-level module use the recorded file, independent of the file that otherwise would be selected by the compiled-load handler (e.g., even if the use-compiled-file-paths parameter value changes). The default module name resolver transfers bytecode-file information when a module declaration
is attached to a new namespace. This protocol supports independent but consistent loading of submodules from bytecode files.

\[(\text{load/use-compiled file}) \rightarrow \text{any}\]
\[\text{file} : \text{path-string}\]

Calls the current compiled-load handler in tail position.

\[(\text{current-load-relative-directory})\]
\[\rightarrow (\text{or/c (and/c path-string? complete-path?) #f})\]
\[(\text{current-load-relative-directory path}) \rightarrow \text{void}\]
\[\text{path} : (\text{or/c (and/c path-string? complete-path?) #f})\]

A parameter that is set by load, load-relative, load-extension, load-relative-extension, and the default compiled-load handler, and used by load-relative, load-relative-extension, and the default compiled-load handler.

When a new path or string is provided as the parameter’s value, it is immediately expanded (see §15.1 "Paths") and converted to a path. (The directory need not exist.)

\[(\text{use-compiled-file-paths})\]
\[\rightarrow (\text{listof (and/c path? relative-path?)})\]
\[(\text{use-compiled-file-paths paths}) \rightarrow \text{void}\]
\[\text{paths} : (\text{listof (and/c path-string? relative-path?)})\]

A list of relative paths, which defaults to (list (string->path "compiled")). It is used by the compiled-load handler (see current-load/use-compiled).

If the PLT_ZO_PATH environment variable is set on startup, it supplies a path instead of "compiled" to use for the initial parameter value.

Changed in version 7.7.0.9 of package base: Added PLT_ZO_PATH.

\[(\text{current-compiled-file-roots}) \rightarrow (\text{listof (or/c path? 'same)})\]
\[(\text{current-compiled-file-roots paths}) \rightarrow \text{void}\]
\[\text{paths} : (\text{listof (or/c path-string? 'same)})\]

A list of paths and 'same that is is used by the default compiled-load handler (see current-load/use-compiled).

The parameter is normally initialized to (list 'same), but the parameter’s initial value can be adjusted by the installation configuration as reported by (find-compiled-file-roots), and it can be further adjusted by the PLTCOMPILERoots environment variable or the --compiled or -R command-line flag for racket. If the environment variable is defined and not overridden by a command-line flag, it is parsed by first replacing any @(version) with the result of (version), then using path-list-string->path-list with a path list produced by (find-compiled-file-roots) to arrive at the parameter’s initial value.
(find-compiled-file-roots) \rightarrow (listof (or/c path? 'same))

Produces a list of paths and 'same, which is normally used to initialize current-compiled-file-roots. The list is determined by consulting the "config.rtkd" file in the directory reported by (find-config-dir), and it defaults to (list 'same) if not configured there.

See also 'compiled-file-roots in §19 “Installation Configuration and Search Paths”.

Added in version 8.0.0.9 of package base.

(use-compiled-file-check) \rightarrow (or/c 'modify-seconds 'exists)

(check : (or/c 'modify-seconds 'exists))

A parameter that determines how a compiled file is checked against its source to enable use of the compiled file. By default, the file-check mode is 'modify-seconds, which uses a compiled file when its filesystem modification date is at least as new as the source file’s. The 'exists mode causes a compiled file to be used in place of its source as long as the compiled file exists.

If the PLT_COMPILED_FILE_CHECK environment variable is set to modify-seconds or exists, then the environment variable’s value configures the parameter when Racket starts.

Added in version 6.6.0.3 of package base.

(read-eval-print-loop) \rightarrow any

Starts a new REPL using the current input, output, and error ports. The REPL wraps each expression to evaluate with #\top-interaction, which is normally bound to #\top-interaction, and it wraps each evaluation with a continuation prompt using the default continuation prompt tag and prompt handler (see call-with-continuation-prompt). The REPL also wraps the read and print operations with a prompt for the default tag whose handler ignores abort arguments and continues the loop. The read-eval-print-loop procedure does not return until eof is read, at which point it returns #<void>.

The read-eval-print-loop procedure can be configured through the current-prompt-read, current-eval, and current-print parameters.

(current-prompt-read) \rightarrow (-> any)

(current-prompt-read proc) \rightarrow void?

proc : (-> any)

A parameter that determines a prompt read handler, which is a procedure that takes no arguments, displays a prompt string, and returns a top-level form to evaluate. The prompt
read handler is called by `read-eval-print-loop`, and after printing a prompt, the handler typically should call the read interaction handler (as determined by the `current-read-interaction` parameter) with the port produced by the interaction port handler (as determined by the `current-get-interaction-input-port` parameter).

The default prompt read handler prints `>` and returns the result of

```scheme
(let ([in ((current-get-interaction-input-port))])
  ((current-read-interaction) (object-name in) in))
```

If the input and output ports are both terminals (in the sense of `terminal-port?`) and if the output port appears to be counting lines (because `port-next-location` returns a non-#f line and column), then the output port’s line is incremented and its column is reset to 0 via `set-port-next-location!` before returning the read result.

```scheme
(current-get-interaction-input-port) -> (-> input-port?)
(current-get-interaction-input-port proc) -> void?
proc : (-> input-port?)
```

A parameter that determines the `interaction port handler`, which returns a port to use for `read-eval-print-loop` inputs.

The default interaction port handler returns the current input port. In addition, if that port is the initial current input port, the initial current output and error ports are flushed.

The `racket/gui/base` library adjusts this parameter’s value by extending the current value. The extension wraps the result port so that GUI events can be handled when reading from the port blocks.

```scheme
(current-read-interaction) -> (any/c input-port? -> any)
(current-read-interaction proc) -> void?
proc : (any/c input-port? -> any)
```

A parameter that determines the current `read interaction handler`, which is procedure that takes an arbitrary value and an input port and returns an expression read from the input port.

The default read interaction handler accepts `src` and `in` and returns

```scheme
(parameterize ([read-accept-reader #t]
               [read-accept-lang #f])
  (read-syntax src in))
```

```scheme
(current-print) -> (any/c -> any)
(current-print proc) -> void?
proc : (any/c -> any)
```

1032
A parameter that determines the print handler that is called by `read-eval-print-loop` to print the result of an evaluation (and the result is ignored).

The default print handler prints the value to the current output port (as determined by the `current-output-port` parameter) and then outputs a newline, except that it prints nothing when the value is `#<void>`.

```lisp
(current-compile)
  → (any/c boolean? . -> . compiled-expression?)
  (current-compile proc) → void?
  proc : (any/c boolean? . -> . compiled-expression?)
```

A parameter that determines the current compilation handler. The compilation handler is a procedure that takes a top-level form and returns a compiled form; see §1.2.4 “Compilation” for more information on compilation.

The compilation handler is called by `compile`, and indirectly by the default evaluation handler and the default load handler.

The handler’s second argument is `#t` if the compiled form will be used only for immediate evaluation, or `#f` if the compiled form may be saved for later use; the default compilation handler is optimized for the special case of immediate evaluation.

When a compiled form is written to an output port, the written form starts with `#`. See §1.4.16 “Printing Compiled Code” for more information.

For internal testing purposes, when the PLT_VALIDATE_COMPILE environment variable is set, the default compilation handler runs a bytecode validator immediately on its own compilation results (instead of relying only on validation when compiled bytecode is loaded).

```lisp
(compile top-level-form) → compiled-expression?
  top-level-form : any/c
```

Like `eval`, but calls the current compilation handler in tail position with `top-level-form`.

```lisp
(compile-syntax stx) → compiled-expression?
  stx : syntax?
```

Like `eval-syntax`, but calls the current compilation handler in tail position with `stx`.

```lisp
(compiled-expression-recompile ce) → compiled-expression?
  ce : compiled-expression?
```

Recompiles `ce`. If `ce` was compiled as machine-independent and `current-compile-target-machine` is not set to `#f`, then recompiling effectively converts to the current machine format. Otherwise, recompiling effectively re-runs optimization passes to produce an equivalent compiled form with potentially different performance characteristics.
Added in version 6.3 of package base.

\[
(\text{compiled-expression? } v) \rightarrow \text{boolean?}
\]
\[
v : \text{any/c}
\]

Returns \#t if \(v\) is a compiled form, \#f otherwise.

\[
(\text{compile-enforce-module-constants}) \rightarrow \text{boolean?}
\]
\[
(\text{compile-enforce-module-constants on?}) \rightarrow \text{void?}
\]
\[
on? : \text{any/c}
\]

A parameter that determines how a module declaration is compiled.

When constants are enforced, and when the macro-expanded body of a module contains
no \textit{set!} assignment to a particular variable defined within the module, then the variable is
marked as constant when the definition is evaluated. Afterward, the variable’s value cannot
be assigned or undefined through \texttt{module->namespace}, and it cannot be defined by
redeclaring the module.

Enforcing constants allows the compiler to inline some variable values, and it allows the
native-code just-in-time compiler to generate code that skips certain run-time checks.

\[
(\text{compile-allow-set!-undefined}) \rightarrow \text{boolean?}
\]
\[
(\text{compile-allow-set!-undefined allow?}) \rightarrow \text{void?}
\]
\[
allow? : \text{any/c}
\]

A parameter that determines how a \textit{set!} expression is compiled when it mutates a global
variable. If the value of this parameter is a true value, \textit{set!} expressions for global
variables are compiled so that the global variable is set even if it was not previously
defined. Otherwise, \textit{set!} expressions for global variables are compiled to raise the
\texttt{exn:fail:contract:variable} exception if the global variable is not defined at the time
the \textit{set!} is performed. Note that this parameter is used when an expression is \textit{compiled}, not
when it is \textit{evaluated}.

\[
(\text{compile-context-preservation-enabled}) \rightarrow \text{boolean?}
\]
\[
(\text{compile-context-preservation-enabled on?}) \rightarrow \text{void?}
\]
\[
on? : \text{any/c}
\]

A parameter that determines whether compilation should avoid function-call inlining and
other optimizations that may cause information to be lost from stack traces (as reported by
\texttt{continuation-mark-set->context}). The default is \#f, which allows such optimiza-
ations.

\[
(\text{current-compile-target-machine})
\]
\[
\rightarrow (\text{or/c #f (and/c symbol? compile-target-machine?)}))
\]
\[
(\text{current-compile-target-machine target}) \rightarrow \text{void?}
\]
\[
target : (\text{or/c #f (and/c symbol? compile-target-machine?)}))
\]

1034
A parameter that determines the platform and/or virtual machine target for a newly compiled expression.

If the target is #f, the the compiled expression writes in a machine-independent format (usually in ".zo" files). Machine-independent compiled code works for any platform and any Racket virtual machine. When the machine-independent compiled expression is read back in, it is subject to further compilation for the current platform and virtual machine, which can be considerably slower than reading a format that is fully compiled for a platform and virtual machine.

The default is something other than #f, unless machine-independent mode is enabled through the -M/--compile-any command-line flag to stand-alone Racket (or GRacket) or through the PLT_COMPILE_ANY environment variable (set to any value).

Added in version 7.1.0.6 of package base.

```
(compile-target-machine? sym) → boolean?
  sym : symbol?
```

Reports whether sym is a supported compilation target for the currently running Racket.

When (system-type 'vm) reports 'racket, then the only target symbol is 'racket. When (system-type 'vm) reports 'chez-scheme, then a symbol corresponding to the current platform is a target, and other targets may also be supported. The 'target-machine mode of system-type reports the running Racket’s native target machine.

Added in version 7.1.0.6 of package base.

```
(eval-jit-enabled) → boolean?
(eval-jit-enabled on?) → void?
  on? : any/c
```

A parameter that determines whether the native-code just-in-time compiler (JIT) is enabled for code (compiled or not) that is passed to the default evaluation handler. A true parameter value is effective only on platforms for which the JIT is supported and for Racket virtual machines that rely on a JIT.

The default is #t, unless the JIT is not supported by the current platform but is supported on the same virtual machine for other platforms, unless it is disabled through the -j/--no-jit command-line flag to stand-alone Racket (or GRacket), and unless it is disabled through the PLTNOMZJIT environment variable (set to any value).

```
(load-on-demand-enabled) → boolean?
(load-on-demand-enabled on?) → void?
  on? : any/c
```

A parameter that determines whether the default load handler sets read-on-demand-source. See current-load for more information. The default is #t, unless it is disabled.
through the -d/--no-delay command-line flag.

14.3 The racket/load Language

#lang racket/load package: base

The racket/load language supports evaluation where each top-level form in the module body is separately passed to eval in the same way as for load.

The namespace for evaluation shares the module registry with the racket/load module instance, but it has a separate top-level environment, and it is initialized with the bindings of racket. A single namespace is created for each instance of the racket/load module (i.e., multiple modules using the racket/load language share a namespace). The racket/load library exports only #\%module-begin and #\%top-interaction forms that effectively swap in the evaluation namespace and call eval.

For example, the body of a module using racket/load can include module forms, so that running the following module prints 5:

#lang racket/load

(module m racket/base
  (provide x)
  (define x 5))

(module n racket/base
  (require 'm)
  (display x))

(require 'n)

Definitions in a module using racket/load are evaluated in the current namespace, which means that load and eval can see the definitions. For example, running the following module prints 6:

#lang racket/load

(define x 6)
(display (eval 'x))

Since all forms within a racket/load module are evaluated in the top level, bindings cannot be exported from the module using provide. Similarly, since evaluation of the module-body forms is inherently dynamic, compilation of the module provides essentially no benefit. For these reasons, use racket/load for interactive exploration of top-level forms only, and not for constructing larger programs.
14.4 Module Names and Loading

14.4.1 Resolving Module Names

The name of a declared module is represented by a resolved module path, which encapsulates either a symbol or a complete filesystem path (see §15.1 “Paths”). A symbol normally refers to a predefined module or module declared through reflective evaluation (e.g., `eval`). A filesystem path normally refers to a module declaration that was loaded on demand via `require` or other forms.

A module path is a datum that matches the grammar for `module-path` for `require`. A module path is relative to another module.

\[
(\text{resolved-module-path? } v) \rightarrow \text{boolean?}
\]
\[
y : \text{any/c}
\]

Returns `#t` if \(v\) is a resolved module path, `#f` otherwise.

\[
(\text{make-resolved-module-path } \text{path}) \rightarrow \text{resolved-module-path?}
\]
\[
\text{path} : (\text{or/c symbol?} \\
(\text{and/c path? complete-path?}) \\
(\text{cons/c (or/c symbol?} \text{)} \\
(\text{and/c path? complete-path?})) \\
(\text{non-empty-listof symbol?})))
\]

Returns a resolved module path that encapsulates \(\text{path}\), where a list \(\text{path}\) corresponds to a submodule path. If \(\text{path}\) is a path or starts with a path, the path normally should be cleansed (see `cleanse-path`) and simplified (see `simplify-path`, including consulting the file system).

A resolved module path is interned. That is, if two resolved module path values encapsulate paths that are `equal?`, then the resolved module path values are `eq?`.

\[
(\text{resolved-module-path-name } \text{module-path}) \rightarrow (\text{or/c symbol?} \\
(\text{and/c path? complete-path?}) \\
(\text{cons/c (or/c symbol?} \text{)} \\
(\text{and/c path? complete-path?})) \\
(\text{non-empty-listof symbol?})))
\]
\[
\text{module-path} : \text{resolved-module-path?}
\]

Returns the path or symbol encapsulated by a resolved module path. A list result corresponds to a submodule path.

\[
(\text{module-path? } v) \rightarrow \text{boolean?}
\]
\[
y : \text{any/c}
\]
Returns #t if v corresponds to a datum that matches the grammar for module-path for require, #f otherwise. Note that a path (in the sense of path?) is a module path.

\[
\text{(current-module-name-resolver)} \\
\rightarrow \text{(case->)} \\
\text{(resolved-module-path? (or/c #f namespace?) . -> . any)} \\
\text{(module-path?)} \\
\text{(or/c #f resolved-module-path?)} \\
\text{(or/c #f syntax?)} \\
\text{boolean?} \\
\text{. -> .} \\
\text{resolved-module-path?})
\]

A parameter that determines the current module name resolver, which manages the conversion from other kinds of module references to a resolved module path. For example, when the expander encounters (require module-path) where module-path is not an identifier, then the expander passes 'module-path to the module name resolver to obtain a symbol or resolved module path. When such a require appears within a module, the module path resolver is also given the name of the enclosing module, so that a relative reference can be converted to an absolute symbol or resolved module path.

The default module name resolver uses collection-file-path to convert lib and symbolic-shorthand module paths to filesystem paths. The collection-file-path function, in turn, uses the current-library-collection-links and current-library-collection-paths parameters.

A module name resolver takes two and four arguments:

- When given two arguments, the first is a name for a module that is now declared in the current namespace, and the second is optionally a namespace from which the declaration was copied. The module name resolver’s result in this case is ignored.

The current module name resolver is called with two arguments by namespace-attach-module or namespace-attach-module-declaration to notify the resolver that a module declaration was attached to the current namespace (and should not be loaded in the future for the namespace’s module registry). Evaluation of a module declaration also calls the current module name resolver with two arguments,
where the first is the declared module and the second is \#f. No other Racket operation invokes the module name resolver with two arguments, but other tools (such as DrRacket) might call this resolver in this mode to avoid redundant module loads.

- When given four arguments, the first is a module path, equivalent to a quoted \texttt{module-path} for \texttt{require}. The second is name for the source module, if any, to which the path is relative; if the second argument is \#f, the module path is relative to \texttt{(or \texttt{(current-load-relative-directory)} \texttt{(current-directory)})}. The third argument is a syntax object that can be used for error reporting, if it is not \#f. If the last argument is \#t, then the module declaration should be loaded (if it is not already), otherwise the module path should be simply resolved to a name. The result is the resolved name.

For the second case, the standard module name resolver keeps a table per module registry containing loaded module name. If a resolved module path is not in the table, and \#f is not provided as the fourth argument to the module name resolver, then the name is put into the table and the corresponding file is loaded with a variant of \texttt{load/use-compiled} that passes the expected module name to the compiled-load handler.

While loading a file, the default module name resolver sets the \texttt{current-module-declare-name} parameter to the resolved module name (while the compiled-load handler sets \texttt{current-module-declare-source}). Also, the default module name resolver records in a private continuation mark the module being loaded, and it checks whether such a mark already exists; if such a continuation mark does exist in the current continuation, then the \texttt{exn:fail} exception is raised with a message about a dependency cycle.

The default module name resolver cooperates with the default compiled-load handler: on a module-attach notification, bytecode-file information recorded by the compiled-load handler for the source namespace’s module registry is transferred to the target namespace’s module registry.

The default module name resolver also maintains a small, module registry-specific cache that maps \texttt{lib} and symbolic module paths to their resolutions. This cache is consulted before checking parameters such as \texttt{current-library-collection-links} and \texttt{current-library-collection-paths}, so results may “stick” even if those parameter values change. An entry is added to the cache only when the fourth argument to the module name resolver is true (indicating that a module should be loaded) and only when loading succeeds.

Finally, the default module name resolver potentially treats a \texttt{submod} path specially. If the module path as the first element of the \texttt{submod} form refers to non-existent collection, then instead of raising an exception, the default module name resolver synthesizes an uninterned symbol module name for the resulting resolved module path. This special treatment of submodule paths is consistent with the special treatment of nonexistent submodules by the compiled-load handler, so that \texttt{module-declared?} can be used more readily to check for the existence of a submodule.

Module loading is suppressed (i.e., \#f is supplied as a fourth argument to the module name resolver).
resolver) when resolving module paths in syntax objects (see §1.2.2 “Syntax Objects”). When a syntax object is manipulated, the current namespace might not match the original namespace for the syntax object, and the module should not necessarily be loaded in the current namespace.

For historical reasons, the default module name resolver currently accepts three arguments, in addition to two and four. Three arguments are treated the same as four arguments with the fourth argument as #t, except that an error is also logged. Support for three arguments will be removed in a future version.

Changed in version 6.0.1.12 of package base: Added error logging to the default module name resolver when called with three arguments.

Changed in version 7.0.0.17: Added special treatment of submod forms with a nonexistent collection by the default module name resolver.

```
(c current-module-declare-name)
  → (or/c resolved-module-path? #f)
(c current-module-declare-name name) → void?
     name : (or/c resolved-module-path? #f)
```

A parameter that determines a module name that is used when evaluating a module declaration (when the parameter value is not #f). In that case, the id from the module declaration is ignored, and the parameter’s value is used as the name of the declared module.

When declaring submodules, current-module-declare-name determines the name used for the submodule’s root module, while its submodule path relative to the root module is unaffected.

```
(c current-module-declare-source)
  → (or/c symbol? (and/c path? complete-path?) #f)
(c current-module-declare-source src) → void?
     src : (or/c symbol? (and/c path? complete-path?) #f)
```

A parameter that determines source information to be associated with a module when evaluating a module declaration. Source information is used in error messages and reflected by variable-reference->module-source. When the parameter value is #f, the module’s name (as determined by current-module-declare-name) is used as the source name instead of the parameter value.

```
(c current-module-path-for-load)
  → (or/c #f module-path?
              (and/c syntax?
                 (lambda (stx)
                    (module-path? (syntax->datum s)))))
(c current-module-path-for-load path) → void?
```
A parameter that determines a module path used for `exn:fail:syntax:missing-module` and `exn:fail:filesystem:missing-module` exceptions as raised by the default load handler. The parameter is normally set by a module name resolver.

### 14.4.2 Compiled Modules and References

While expanding a module declaration, the expander resolves module paths for imports to load module declarations as necessary and to determine imported bindings, but the compiled form of a module declaration preserves the original module path. Consequently, a compiled module can be moved to another filesystem, where the module name resolver can resolve inter-module references among compiled code.

When a module reference is extracted from compiled form (see `module-compiled-imports`) or from syntax objects in macro expansion (see §12.2 "Syntax Object Content"), the module reference is reported in the form of a module path index. A module path index is a semi-interned (multiple references to the same relative module tend to use the same module path index value, but not always) opaque value that encodes a module path (see `module-path?`) and either a resolved module path or another module path index to which it is relative.

A module path index that uses both `#f` for its path and base module path index represents “self”—i.e., the module declaration that was the source of the module path index—and such a module path index can be used as the root for a chain of module path indexes at compile time. For example, when extracting information about an identifier’s binding within a module, if the identifier is bound by a definition within the same module, the identifier’s source module is reported using the “self” module path index. If the identifier is instead defined in a module that is imported via a module path (as opposed to a literal module name), then the identifier’s source module will be reported using a module path index that contains the required module path and the “self” module path index. A “self” module path index has a submodule path when the module that it refers to is a submodule.

A module path index has state. When it is resolved to a resolved module path, then the resolved module path is stored with the module path index. In particular, when a module is loaded, its root module path index is resolved to match the module’s declaration-time name. This resolved path is forgotten, however, in identifiers that the module contributes to the compiled and marshaled form of other modules. The transient nature of resolved names allows the module code to be loaded with a different resolved name than the name when it was compiled.

Two module path index values are `equal?` when they have `equal?` path and base values.
(even if they have different resolved values).

\[(\text{module-path-index? } v) \rightarrow \text{boolean?}\]
\[v : \text{any/c}\]

Returns \#t if \(v\) is a module path index, \#f otherwise.

\[(\text{module-path-index\-resolve } mpi [load?]) \rightarrow \text{resolved-module-path?}\]
\[mpi : \text{module-path-index?}\]
\[load? : \text{any/c} = \#f\]

Returns a resolved module path for the resolved module name, computing the resolved name (and storing it in \(mpi\)) if it has not been computed before.

Resolving a module path index uses the current module name resolver (see \text{current-module-name-resolver}). Depending on the kind of module paths encapsulated by \(mpi\), the computed resolved name can depend on the value of \text{current-load-relative-directory} or \text{current-directory}. The \text{load?} argument is propagated as the last argument to the module name resolver.

See \text{resolve-module-path-index}.

Changed in version 6.90.0.16 of package \text{base}: Added the \text{load?} optional argument.

\[(\text{module-path-index\-split } mpi) \rightarrow (\text{or/c} \text{module-path?} \#f)\]
\[(\text{or/c} \text{module-path-index?} \text{resolved-module-path?} \#f)\]
\[mpi : \text{module-path-index?}\]

Returns two values: a module path, and a base path—either a module path index, resolved module path, or \#f—to which the first path is relative.

A \#f second result means that the path is relative to an unspecified directory (i.e., its resolution depends on the value of \text{current-load-relative-directory} and/or \text{current-directory}).

A \#f for the first result implies a \#f for the second result, and means that \(mpi\) represents “self” (see above). Such a module path index may have a non-\#f submodule path as reported by \text{module-path-index\-submodule}.

\[(\text{module-path-index\-submodule } mpi) \rightarrow (\text{or/c} \#f \text{(non-empty-listof symbol?)})\]
\[mpi : \text{module-path-index?}\]

Returns a non-empty list of symbols if \(mpi\) is a “self” (see above) module path index that refers to a submodule. The result is always \#f if either result of \(\text{module-path-index\-split } mpi\) is non-\#f.
(module-path-index-join path base [submod]) → module-path-index?
   path : (or/c module-path? #f)
   base : (or/c module-path-index? resolved-module-path? #f)
   submod : (or/c #f (non-empty-listof symbol?)) = #f

Combines path, base, and submod to create a new module path index. The path argument can be #f only if base is also #f. The submod argument can be a list only when path and base are both #f.

(compiled-module-expression? v) → boolean?
   v : any/c

Returns #t if v is a compiled module declaration, #f otherwise. See also current-compile.

(module-compiled-name compiled-module-code)
   → (or/c symbol? (cons/c symbol? (non-empty-listof symbol?)))
   compiled-module-code : compiled-module-expression?
   (module-compiled-name compiled-module-code
      name)
   → compiled-module-expression?
   compiled-module-code : compiled-module-expression?
   name : (or/c symbol? (cons/c symbol? (non-empty-listof symbol?)))

Takes a module declaration in compiled form and either gets the module’s declared name (when name is not provided) or returns a revised module declaration with the given name.

The name is a symbol for a top-level module, or a symbol paired with a list of symbols where the list reflects the submodule path to the module starting with the top-level module’s declared name.

(module-compiled-submodules compiled-module-code
   non-star?)
   → (listof compiled-module-expression?)
   compiled-module-code : compiled-module-expression?
   non-star? : any/c
   (module-compiled-submodules compiled-module-code
      non-star?
      submodules)
   → compiled-module-expression?
   compiled-module-code : compiled-module-expression?
   non-star? : any/c
   submodules : (listof compiled-module-expression?)

Takes a module declaration in compiled form and either gets the module’s submodules (when submodules is not provided) or returns a revised module declaration with the given sub-
modules. The non-star? argument determines whether the result or new submodule list corresponds to module declarations (when non-star? is true) or module* declarations (when non-star? is #f).

\[
\text{(module-compiled-imports compiled-module-code)}
\rightarrow \text{(listof (cons/c (or/c exact-integer? #f) (listof module-path-index?)))}
\]

\[
\text{compiled-module-code : compiled-module-expression?}
\]

Takes a module declaration in compiled form and returns an association list mapping phase level shifts (where #f corresponds to a shift into the label phase level) to module references for the module’s explicit imports.

\[
\text{(module-compiled-exports compiled-module-code [verbosity])}
\rightarrow \text{(listof (cons/c (or/c exact-integer? #f) list?) (listof (cons/c (or/c exact-integer? #f) list?)))}
\]

\[
\text{compiled-module-code : compiled-module-expression?}
\]

\[
\text{verbosity : (or/c #f 'defined-names) = #f}
\]

Returns two association lists mapping phase level values (where #f corresponds to the label phase level) to exports at the corresponding phase. The first association list is for exported variables, and the second is for exported syntax. Beware however, that value bindings re-exported though a rename transformer are in the syntax list instead of the value list.

Each associated list, which is represented by list? in the result contracts above, more precisely matches the contract

\[
\text{(listof (list/c symbol? (listof (or/c module-path-index? (list/c module-path-index? (or/c exact-integer? #f) symbol? (or/c exact-integer? #f)))) ; only if verbosity is 'defined-names: symbol?)}}
\]

For each element of the list, the leading symbol is the name of the export.

The second part—the list of module path index values, etc.—describes the origin of the exported identifier. If the origin list is null, then the exported identifier is defined in the module. If the exported identifier is re-exported, instead, then the origin list provides information on the import that was re-exported. The origin list has more than one element if the binding was imported multiple times from (possibly) different sources.

1044
The last part, a symbol, is included only if \texttt{verbosity} is \texttt{defined-names}. In that case, the included symbol is the name of the definition within its defining module (which may be different than the name that is exported).

For each origin, a module path index by itself means that the binding was imported with a phase level shift of 0 (i.e., a plain require without \texttt{for-meta}, \texttt{for-syntax}, etc.), and imported identifier has the same name as the re-exported name. An origin represented with a list indicates explicitly the import, the import phase level shift (where \texttt{#f} corresponds to a \texttt{for-label} import), the import name of the re-exported binding, and the phase level of the import.

Example:

```scheme
> (module-compiled-exports
  (compile
   '(module banana racket/base
     (require (only-in racket/math pi)
               (for-syntax racket/base))
     (provide pi
              (rename-out [peel wrapper]))
     bush
     cond
              (for-syntax compile-time))
     (define peel pi)
     (define bush (* 2 pi))
     (begin-for-syntax
      (define compile-time (current-seconds))))
' defined-names)
'((0 (bush () bush)
   (pi (#<module-path-index:racket/math>) pi)
   (wrapper () peel))
 (1 (compile-time () compile-time))
'((0 (cond (#<module-path-index:racket/base>) cond)))
```

Changed in version 7.5.0.6 of package \texttt{base}: Added the \texttt{verbosity} argument.

\begin{verbatim}
(module-compiled-indirect-exports compiled-module-code)
\rightarrow (listof (cons/c exact-integer? (listof symbol?)))
\end{verbatim}

\texttt{compiled-module-code} : \texttt{compiled-module-expression?}

Returns an association list mapping phase level values to symbols that represent variables within the module. These definitions are not directly accessible from source, but they are accessible from bytecode, and the order of the symbols in each list corresponds to an order for bytecode access.

Added in version 6.5.0.5 of package \texttt{base}.
(module-compiled-language-info compiled-module-code)
→ (or/c #f (vector/c module-path? symbol? any/c))
compiled-module-code : compiled-module-expression?

Returns information intended to reflect the “language” of the module’s implementation as originally attached to the syntax of the module’s declaration though the ‘module-language’ syntax property. See also module.

If no information is available for the module, the result is #f. Otherwise, the result is (vector mp name val) such that ((dynamic-require mp name val) should return function that takes two arguments. The function’s arguments are a key for reflected information and a default value. Acceptable keys and the interpretation of results is up to external tools, such as DrRacket. If no information is available for a given key, the result should be the given default value.

See also module->language-info and racket/language-info.

(module-compiled-cross-phase-persistent? compiled-module-code)
→ boolean?
compiled-module-code : compiled-module-expression?

Return #t if compiled-module-code represents a cross-phase persistent module, #f otherwise.

14.4.3 Dynamic Module Access

(dynamic-require mod provided [fail-thunk]) → (or/c void? any/c)
mod : (or/c module-path?
resolved-module-path?
module-path-index?)
provided : (or/c symbol? #f 0 void?)
fail-thunk : (-> any) = (lambda () ....)

Dynamically instantiates the module specified by mod in the current namespace’s registry at the namespace’s base phase, if it is not yet instantiated. The current module name resolver may load a module declaration to resolve mod (see current-module-name-resolver); the path is resolved relative to current-load-relative-directory and/or current-directory. Beware that concurrent dynamic-requires in namespaces that share a module registry can create race conditions; see also namespace-call-with-registry-lock.

If provided is #f, then the result is #<void>, and the module is not visited (see §1.2.3.9 “Module Expansion, Phases, and Visits”) or even made available (for on-demand visits) in phases above the base phase.

Because dynamic-require is a procedure, giving a plain S-expression for mod the same way as you would for a require expression likely won’t give you expected results. What you need instead is something that evaluates to an S-expression; using quote is one way to do it.
Examples:

```racket
> (module a racket/base (displayln "hello"))
> (dynamic-require "a #f)
hello
```

When `provided` is a symbol, the value of the module’s export with the given name is returned, and still the module is not visited or made available in higher phases.

Examples:

```racket
> (module b racket/base
  (provide dessert)
  (define dessert "gulab jamun"))
> (dynamic-require "b 'dessert)
"gulab jamun"
```

If the module exports `provided` as syntax, then a use of the binding is expanded and evaluated in a fresh namespace to which the module is attached, which means that the module is visited in the fresh namespace. The expanded syntax must return a single value.

Examples:

```racket
> (module c racket/base
  (require (for-syntax racket/base))
  (provide dessert2)
  (define dessert "nanaimo bar")
  (define-syntax dessert2
    (make-rename-transformer #\'dessert))
> (dynamic-require "c 'dessert2)
"nanaimo bar"
```

If the module has no such exported variable or syntax, then `fail-thunk` is called; the default `fail-thunk` raises `exn:fail:contract`. If the variable named by `provided` is exported protected (see §14.10 "Code Inspectors"), then the `exn:fail:contract` exception is raised.

If `provided` is 0, then the module is instantiated but not visited, the same as when `provided` is `#f`. With 0, however, the module is made available in higher phases.

If `provided` is `#<void>`, then the module is visited but not instantiated (see §1.2.3.9 "Module Expansion, Phases, and Visits"), and the result is `#<void>`.

More examples using different `module-path` grammar expressions are given below:

Example:
> (dynamic-require 'racket/base #f)

Example:
> (dynamic-require (list 'lib "racket/base") #f)

Examples:
> (module a racket/base
  (module b racket/base
    (provide inner-dessert)
    (define inner-dessert "tiramisu")))
> (dynamic-require '(submod 'a b) 'inner-dessert)
"tiramisu"

The last line in the above example could instead have been written as

Example:
> (dynamic-require ((lambda () (list 'submod ''a 'b))) 'inner-dessert)
"tiramisu"

which is equivalent.

| (dynamic-require-for-syntax mod
  provided [fail-thunk]) → any
mod : module-path?
provided : (or/c symbol? #f)
fail-thunk : (-> any) = (lambda () ....)

Like dynamic-require, but in a phase that is 1 more than the namespace’s base phase.

| (module-declared? mod [load?]) → boolean?
mod : (or/c module-path? module-path-index? resolved-module-path?)
load? : any/c = #f

Returns #t if the module indicated by mod is declared (but not necessarily instantiated or visited) in the current namespace, #f otherwise.

If load? is #t and mod is not a resolved module path, the module is loaded in the process of resolving mod (as for dynamic-require and other functions). Checking for the declaration of a submodule does not trigger an exception if the submodule cannot be loaded because it does not exist, either within a root module that does exist or because the root module does not exist.
(module->language-info mod [load?])

-> (or/c #f (vector/c module-path? symbol? any/c))

mod : (or/c module-path? module-path-index?
  resolved-module-path?)

load? : any/c = #f

Returns information intended to reflect the “language” of the implementation of mod. If mod is a resolved module path or load? is #f, the module named by mod must be declared (but not necessarily instantiated or visited) in the current namespace; otherwise, mod may be loaded (as for dynamic-require and other functions). The information returned by module->language-info is the same as would have been returned by module-compiled-language-info applied to the module’s implementation as compiled code.

A module can be declared by using dynamic-require.

Examples:

> (dynamic-require 'racket/dict (void))
> (module->language-info 'racket/dict)
  #f

(module->imports mod)

-> (listof (cons/c (or/c exact-integer? #f)
  (listof module-path-index?)))

mod : (or/c module-path? module-path-index?
  resolved-module-path?)

Like module-compiled-imports, but produces the imports of mod, which must be declared (but not necessarily instantiated or visited) in the current namespace. See module->language-info for an example of declaring an existing module.

Examples:

> (module banana racket/base
  (require (only-in racket/math pi))
  (provide peel)
  (define peel pi)
  (define bush (* 2 pi)))
> (module->imports ''banana)
 '(((0 #<module-path-index:racket/base> #<module-path-index:racket/math>))

(module->exports mod [verbosity])

-> (listof (cons/c (or/c exact-integer? #f) list?))

(listof (cons/c (or/c exact-integer? #f) list?))
**module**: (or/c module-path? module-path-index? resolved-module-path?)

**verbosity**: (or/c #f 'defined-names) = #f

Like `module-compiled-exports`, but produces the exports of `mod`, which must be declared (but not necessarily instantiated or visited) in the current namespace. See `module->language-info` for an example of declaring an existing module.

**Examples**:

```racket
> (module banana racket/base
   (require (only-in racket/math pi))
   (provide (rename-out [peel wrapper]))
   (define peel pi)
   (define bush (* 2 pi)))
> (module->exports ''banana)
'((0 (wrapper ()())))
```

Changed in version 7.5.0.6 of package base: Added the `verbosity` argument.

```racket
(module->indirect-exports mod)
→ (listof (cons/c exact-integer? (listof symbol?)))
mod : (or/c module-path? module-path-index? resolved-module-path?)

Like `module-compiled-indirect-exports`, but produces the exports of `mod`, which must be declared (but not necessarily instantiated or visited) in the current namespace. See `module->language-info` for an example of declaring an existing module.

**Examples**:

```racket
> (module banana racket/base
   (require (only-in racket/math pi))
   (provide peel)
   (define peel pi)
   (define bush (* 2 pi)))
> (module->indirect-exports ''banana)
'((0 bush))
```

Added in version 6.5.0.5 of package base.

```racket
(module-predefined? mod) → boolean?
mod : (or/c module-path? module-path-index? resolved-module-path?)
```

1050
Reports whether mod refers to a module that is predefined for the running Racket instance. Predefined modules always have a symbolic resolved module path, and they may be predefined always or specifically within a particular executable (such as one created by raco exe or create-embedding-executable).

14.5 Impersonators and Chaperones

An impersonator is a wrapper for a value where the wrapper redirects some of the value’s operations. Impersonators apply only to procedures, structures for which an accessor or mutator is available, structure types, hash tables, vectors, boxes, channels, and prompt tags. An impersonator is equal? to the original value, but not eq? to the original value.

A chaperone is a kind of impersonator whose refinement of a value’s operation is restricted to side effects (including, in particular, raising an exception) or chaperoning values supplied to or produced by the operation. For example, a vector chaperone can redirect vector-ref to raise an exception if the accessed vector slot contains a string, or it can cause the result of vector-ref to be a chaperoned variant of the value that is in the accessed vector slot, but it cannot redirect vector-ref to produce a value that is arbitrarily different from the value in the vector slot.

A non-chaperone impersonator, in contrast, can refine an operation to swap one value for any other. An impersonator cannot be applied to an immutable value or refine the access to an immutable field in an instance of a structure type, since arbitrary redirection of an operation amounts to mutation of the impersonated value.

Beware that each of the following operations can be redirected to an arbitrary procedure through an impersonator on the operation’s argument—assuming that the operation is available to the creator of the impersonator:

• a structure-field accessor
• a structure-field mutator
• a structure type property accessor
• application of a procedure
• unbox
• set-box!
• vector-ref
• vector-set!
• hash-ref
• hash-set
Derived operations, such as printing a value, can be redirected through impersonators due to their use of accessor functions. The equal?, equal-hash-code, and equal-secondary-hash-code operations, in contrast, may bypass impersonators (but they are not obliged to).

In addition to redirecting operations that work on a value, an impersonator can include impersonator properties for an impersonated value. An impersonator property is similar to a structure type property, but it applies to impersonators instead of structure types and their instances.

```
(impersonator? v) → boolean?
  v : any/c
```

Returns #t if v is an impersonator created by procedures like impersonate-procedure or impersonate-struct, #f otherwise.

Programs and libraries generally should avoid impersonator? and treat impersonators the same as non-impersonator values. In rare cases, impersonator? may be needed to guard against redirection by an impersonator of an operation to an arbitrary procedure.

A limitation of impersonator? is that it does not recognize an impersonator that is created by instantiating a structure type with the prop:impersonator-of property. The limitation reflects how those impersonators cannot redirect structure access and mutation operations to arbitrary procedures.

```
(chaperone? v) → boolean?
  v : any/c
```

Returns #t if v is a chaperone, #f otherwise.

Programs and libraries generally should avoid chaperone? for the same reason that they should avoid impersonator?. A true value for chaperone? implies a true value of impersonator?.
\[(\text{impersonator-of? } v1 \ v2) \rightarrow \text{boolean?}\]
\[v1 : \text{any/c}\]
\[v2 : \text{any/c}\]

Indicates whether \(v1\) can be considered equivalent modulo impersonators to \(v2\).

Any two values that are \(\text{eq?}\) to one another are also \text{impersonator-of?}. For values that include no impersonators, \(v1\) and \(v2\) are considered impersonators of each other if they are \text{equal?}.

If at least one of \(v1\) or \(v2\) is an impersonator:

- If \(v1\) impersonates \(v1^*\) then \((\text{impersonator-of? } v1 \ v2)\) is \#t if and only if \((\text{impersonator-of? } v1^* \ v2)\) is \#t.

- If \(v2\) is a non-interposing impersonator that impersonates \(v2^*\), i.e., all of its interposition procedures are \#f, then \((\text{impersonator-of? } v1 \ v2)\) is \#t if and only if \((\text{impersonator-of? } v1 \ v2^*)\) is \#t.

- When \(v2\) is an impersonator constructed with at least one non-\#f interposition procedure, but \(v1\) is not an impersonator then \((\text{impersonator-of? } v1 \ v2)\) is \#f.

Otherwise, if neither \(v1\) or \(v2\) is an impersonator, but either of them contains an impersonator as a subpart (e.g., \(v1\) is a list with an impersonator as one of its elements), then \((\text{impersonator-of? } v1 \ v2)\) proceeds by comparing \(v1\) and \(v2\) recursively (as with \text{equal?}), returning true if all subparts are \text{impersonator-of}.

Examples:

\[
\begin{align*}
> & (\text{impersonator-of?} \ (\text{impersonate-procedure} \ add1 \ (\lambda \ (x) \ x)) \ add1) \\
& \#t \\
> & (\text{impersonator-of?} \ (\text{impersonate-procedure} \ add1 \ (\lambda \ (x) \ x)) \ sub1) \\
& \#f \\
> & (\text{impersonator-of?} \ (\text{impersonate-procedure} \ (\text{impersonate-procedure} \ add1 \ (\lambda \ (x) \ x)) \ (\lambda \ (x) \ x)) \ add1) \\
& \#t \\
> & (\text{impersonator-of?} \ (\text{impersonate-procedure} \ add1 \ (\lambda \ (x) \ x)) \\
& \ (\text{impersonate-procedure} \ add1 \ #f)) \\
& \#t \\
> & (\text{impersonator-of?} \ (\text{impersonate-procedure} \ add1 \ (\lambda \ (x) \ x)) \\
& \ (\text{impersonate-procedure} \ add1 \ (\lambda \ (x) \ x))) \\
& \#f
\end{align*}
\]
> (impersonator-of? (list 1 2)
  (list 1 2))
#t

> (impersonator-of? (list (impersonate-procedure add1 (λ (x) x)) sub1)
  (list add1 sub1))
#t

(chaperone-of? v1 v2) → boolean?
  v1 : any/c
  v2 : any/c

Indicates whether v1 can be considered equivalent modulo chaperones to v2.

For values that include no chaperones or other impersonators, v1 and v2 can be considered chaperones of each other if they are equal?, except that corresponding mutable vectors, boxes, strings, byte strings, and mutable structures within v1 and v2 must be eq?.

Otherwise, chaperones and other impersonators within v2 must be intact within v1 analogous to way that impersonator-of? requires that impersonators are preserved. Furthermore, v1 must not have any non-chaperone impersonators whose corresponding value in v2 is not the same impersonator. Note that chaperone-of? implies impersonator-of?, but not vice-versa.

(impersonator-ephemeron v) → ephemeron?
  v : any/c

Produces an ephemeron that can be used to connect the reachability of v (in the sense of garbage collection; see §1.1.6 “Garbage Collection”) with the reachability of any value for which v is an impersonator. That is, the value v will be considered reachable as long as the result ephemeron is reachable in addition to any value that v impersonates (including itself).

(procedure-impersonator*? v) → boolean?
  v : any/c

Returns #t for any procedure impersonator that either was produced by impersonate-procedure* or chaperone-procedure*, or is an impersonator/chaperone of a value that was created with impersonate-procedure* or chaperone-procedure* (possibly transitively).

### 14.5.1 Impersonator Constructors

(impersonate-procedure proc
  wrapper-proc
  prop
  prop-val ...
  ...)


```scheme
→ (and/c procedure? impersonator?)
proc : procedure?
wrapper-proc : (or/c procedure? #f)
prop : impersonator-property?
prop-val : any/c
```

Returns an impersonator procedure that has the same arity, name, and other attributes as `proc`. When the impersonator procedure is applied, the arguments are first passed to `wrapper-proc` (when it is not `#f`), and then the results from `wrapper-proc` are passed to `proc`. The `wrapper-proc` can also supply a procedure that processes the results of `proc`.

The arity of `wrapper-proc` must include the arity of `proc`. The allowed keyword arguments of `wrapper-proc` must be a superset of the allowed keywords of `proc`. The required keyword arguments of `wrapper-proc` must be a subset of the required keywords of `proc`.

For applications without keywords, the result of `wrapper-proc` must be at least the same number of values as supplied to it. Additional results can be supplied—before the values that correspond to the supplied values—in the following pattern:

- An optional procedure, `result-wrapper-proc`, which will be applied to the results of `proc`; followed by
- any number of repetitions of `mark key val` (i.e., three values), where the call `proc` is wrapped to install a continuation mark `key` and `val`.

If `result-wrapper-proc` is produced, it must be a procedure that accepts as many results as produced by `proc`; it must return the same number of results. If `result-wrapper-proc` is not supplied, then `proc` is called in tail position with respect to the call to the impersonator.

For applications that include keyword arguments, `wrapper-proc` must return an additional value before any other values but after `result-wrapper-proc` and `mark key val` sequences (if any). The additional value must be a list of replacements for the keyword arguments that were supplied to the impersonator (i.e., not counting optional arguments that were not supplied). The arguments must be ordered according to the sorted order of the supplied arguments’ keywords.

If `wrapper-proc` is `#f`, then applying the resulting impersonator is the same as applying `proc`. If `wrapper-proc` is `#f` and no `prop` is provided, then `proc` is returned and is not impersonated.

Pairs of `prop` and `prop-val` (the number of arguments to `impersonate-procedure` must be even) add impersonator properties or override impersonator-property values of `proc`.

If any `prop` is `impersonator-prop:application-mark` and if the associated `prop-val` is a pair, then the call to `proc` is wrapped with `with-continuation-mark` using `(car
prop-val) as the mark key and (cdr prop-val) as the mark value. In addition, if the immediate continuation frame of the call to the impersonated procedure includes a value for (car prop-val)—that is, if call-with-immediate-continuation-mark would produce a value for (car prop-val) in the call’s continuation—then the value is also installed as an immediate value for (car prop-val) as a mark during the call to wrapper-proc (which allows tail-calls of impersonators with respect to wrapping impersonators to be detected within wrapper-proc).

Changed in version 6.3.0.5 of package base: Added support for `mark key val results from wrapper-proc`.

Examples:

```scheme
> (define (add15 x) (+ x 15))
> (define add15+print
    (impersonate-procedure add15
      (λ (x)
        (printf "called with ~s\n" x)
        (values (λ (res)
          (printf "returned ~s\n" res)
          res)
          x)))))

> (add15 27)
42
> (add15+print 27)
called with 27
returned 42
42

> (define-values (imp-prop:p1 imp-prop:p1? imp-prop:p1-get)
  (make-impersonator-property 'imp-prop:p1))
> (define-values (imp-prop:p2 imp-prop:p2? imp-prop:p2-get)
  (make-impersonator-property 'imp-prop:p2))
> (define add15.2 (impersonate-procedure add15 #f imp-prop:p1 11))
> (add15.2 2)
17
> (imp-prop:p1? add15.2)
#t
> (imp-prop:p1-get add15.2)
11
> (imp-prop:p2? add15.2)
#f
> (define add15.3 (impersonate-procedure add15.2 #f imp-prop:p2 13))
> (add15.3 3)
18
> (imp-prop:p1? add15.3)
```

1056
(impersonate-procedure*  proc
  wrapper-proc
  prop
  prop-val ...
  ...
)

→ (and/c procedure? impersonator?)
  proc : procedure?
  wrapper-proc : (or/c procedure? #f)
  prop : impersonator-property?
  prop-val : any/c

Like impersonate-procedure, except that wrapper-proc receives an additional argument before all other arguments. The additional argument is the procedure orig-proc that was originally applied.

If the result of impersonate-procedure* is applied directly, then orig-proc is that result. If the result is further impersonated before being applied, however, orig-proc is the further impersonator.

An orig-proc argument might be useful so that wrapper-proc can extract impersonator properties that are overridden by further impersonators, for example.

Added in version 6.1.1.5 of package base.
(impersonate-struct
  v
  [struct-type]
  orig-proc
  redirect-proc ...
  ...
  prop
  prop-val ...
  ...)

v : any/c
struct-type : struct-type? = unspecified
orig-proc : (or/c struct-accessor-procedure?
  struct-mutator-procedure?
  struct-type-property-accessor-procedure?)
redirect-proc : (or/c procedure? #f)
prop : impersonator-property?
prop-val : any/c

Returns an impersonator of \( v \), which redirects certain operations on the impersonated value. The \( \text{orig-proc} \)s indicate the operations to redirect, and the corresponding \( \text{redirect-proc} \)s supply the redirections. The optional \( \text{struct-type} \) argument, when provided, acts as a witness for the representation of \( v \), which must be an instance of \( \text{struct-type} \).

The protocol for a \( \text{redirect-proc} \) depends on the corresponding \( \text{orig-proc} \), where \( \text{self} \) refers to the value to which \( \text{orig-proc} \) is originally applied:

- A structure-field accessor: \( \text{redirect-proc} \) must accept two arguments, \( \text{self} \) and the value \( \text{field-v} \) that \( \text{orig-proc} \) produces for \( v \); it must return a replacement for \( \text{field-v} \). The corresponding field must not be immutable, and either the field’s structure type must be accessible via the current inspector or one of the other \( \text{orig-procs} \) must be a structure-field mutator for the same field.
- A structure-field mutator: \( \text{redirect-proc} \) must accept two arguments, \( \text{self} \) and the value \( \text{field-v} \) supplied to the mutator; it must return a replacement for \( \text{field-v} \) to be propagated to \( \text{orig-proc} \) and \( v \).
- A property accessor: \( \text{redirect-proc} \) uses the same protocol as for a structure-field accessor. The accessor’s property must have been created with ‘\text{can-impersonate} as the second argument to \text{make-struct-type-property}.

When a \( \text{redirect-proc} \) is \#f, the corresponding \( \text{orig-proc} \) is unaffected. Supplying \#f for a \( \text{redirect-proc} \) is useful to allow its \( \text{orig-proc} \) to act as a “witness” of \( v \)’s representation and enable the addition of \( \text{props} \).

Pairs of \( \text{prop} \) and \( \text{prop-val} \) (the number of arguments to \text{impersonate-struct} must be even if \( \text{struct-type} \) is provided, odd otherwise) add impersonator properties or override impersonator-property values of \( v \).
Each orig-proc must indicate a distinct operation. If no struct-type and no orig-procs are supplied, then no props must be supplied. If orig-procs are supplied only with #f redirect-procs and no props are supplied, then v is returned and is not impersonated.

If any orig-proc is itself an impersonator, then a use of the accessor or mutator that orig-proc impersonates is redirected for the resulting impersonated structure to use orig-proc on v before redirect-proc (in the case of accessor) or after redirect-proc (in the case of a mutator).

Changed in version 6.1.1.2 of package base: Changed first argument to an accessor or mutator redirect-proc from v to self.
Changed in version 6.1.1.8: Added optional struct-type argument.

\[
\text{impersonate-vector vec ref-proc set-proc prop prop-val ... ...}
\]
\[
vec : (\text{and/c vector? impersonator?})
\]
\[
\text{ref-proc : (or/c (vector? exact-nonnegative-integer? any/c . -> . any/c) #f)}
\]
\[
\text{set-proc : (or/c (vector? exact-nonnegative-integer? any/c . -> . any/c) #f)}
\]
\[
\text{prop : impersonator-property?}
\]
\[
\text{prop-val : any/c}
\]

Returns an impersonator of vec, which redirects the vector-ref and vector-set! operations.

The ref-proc and set-proc arguments must either both be procedures or both be #f. If they are #f then impersonate-vector does not interpose on vec, but still allows attaching impersonator properties.

If ref-proc is a procedure it must accept vec, an index passed to vector-ref, and the value that vector-ref on vec produces for the given index; it must produce a replacement for the value, which is the result of vector-ref on the impersonator.

If set-proc is a procedure it must accept vec, an index passed to vector-set!, and the value passed to vector-set!; it must produce a replacement for the value, which is used with vector-set! on the original vec to install the value.

Pairs of prop and prop-val (the number of arguments to impersonate-vector must be odd) add impersonator properties or override impersonator-property values of vec.

Changed in version 6.9.0.2 of package base: Added non-interposing vector impersonators.
Like `impersonate-vector`, except that `ref-proc` and `set-proc` each receive an additional vector as argument before other arguments. The additional argument is the original impersonated vector, access to which triggered interposition in the first place.

The additional vector argument might be useful so that `ref-proc` or `set-proc` can extract impersonator properties that are overridden by further impersonators, for example.

Added in version 6.9.0.2 of package `base`.

Returns an impersonator of `box`, which redirects the `unbox` and `set-box!` operations.

The `unbox-proc` must accept `box` and the value that `unbox` produces on `box`; it must produce a replacement value, which is the result of `unbox` on the impersonator.

The `set-proc` must accept `box` and the value passed to `set-box!`; it must produce a replacement value, which is used with `set-box!` on the original `box` to install the value.

Pairs of `prop` and `prop-val` (the number of arguments to `impersonate-box` must be odd) add impersonator properties or override impersonator-property values of `box`. 
(impersonate-hash hash
    ref-proc
    set-proc
    remove-proc
    key-proc
    [clear-proc
      equal-key-proc]
    prop
    prop-val ...
    ...
)  → (and/c hash? impersonator?)
hash : (and/c hash? (not/c immutable?))
ref-proc : (hash? any/c . -> . (values any/c
    (hash? any/c any/c . -> . any/c))
set-proc : (hash? any/c any/c . -> . (values any/c any/c))
remove-proc : (hash? any/c . -> . any/c)
key-proc : (hash? any/c . -> . any/c)
clear-proc : (or/c #f (hash? . -> . any)) = #f
equal-key-proc : (or/c #f (hash? any/c . -> . any/c)) = #f
prop : impersonator-property?
prop-val : any/c

Returns an impersonator of hash, which redirects the hash-ref, hash-set! or hash-set (as applicable), hash-remove or hash-remove! (as applicable), hash-clear or hash-clear! (as applicable and if clear-proc is not #f) operations. When hash-set, hash-remove or hash-clear is used on an impersonator of a hash table, the result is an impersonator with the same redirecting procedures. In addition, operations like hash-iterate-key or hash-map, which extract keys from the table, use key-proc to replace keys extracted from the table. Operations like hash-iterate-value or hash-values implicitly use hash-ref and therefore redirect through ref-proc. The hash-ref-key operation uses both ref-proc and key-proc, the former to lookup the requested key and the latter to extract it.

The ref-proc must accept hash and a key passed to hash-ref. It must return a replacement key as well as a procedure. The returned procedure is called only if the returned key is found in hash via hash-ref, in which case the procedure is called with hash, the previously returned key, and the found value. The returned procedure must itself return a replacement for the found value. The returned procedure is ignored by hash-ref-key.

The set-proc must accept hash, a key passed to hash-set! or hash-set, and the value passed to hash-set! or hash-set; it must produce two values: a replacement for the key and a replacement for the value. The returned key and value are used with hash-set! or hash-set on the original hash to install the value.

The remove-proc must accept hash and a key passed to hash-remove! or hash-remove; it must produce a replacement for the key, which is used with hash-remove! or hash-
remove on the original hash to remove any mapping using the (impersonator-replaced) key.

The key-proc must accept hash and a key that has been extracted from hash (by hash-ref-key, hash-iterate-key, or other operations that use hash-iterate-key internally); it must produce a replacement for the key, which is then reported as a key extracted from the table.

If clear-proc is not #f, it must accept hash as an argument, and its result is ignored. The fact that clear-proc returns (as opposed to raising an exception or otherwise escaping) grants the capability to remove all keys from hash. If clear-proc is #f, then hash-clear or hash-clear! on the impersonator is implemented using hash-iterate-key and hash-remove or hash-remove!.

If equal-key-proc is not #f, it effectively interposes on calls to equal?, equal-hash-code, and equal-secondary-hash-code for the keys of hash. The equal-key-proc must accept as its arguments hash and a key that is either mapped by hash or passed to hash-ref, etc., where the latter has potentially been adjusted by the corresponding ref-proc, etc. The result is a value that is passed to equal?, equal-hash-code, and equal-secondary-hash-code as needed to hash and compare keys. In the case of hash-set! or hash-set, the key that is passed to equal-key-proc is the one stored in the hash table for future lookup.

The hash-iterate-value, hash-map, or hash-for-each functions use a combination of hash-iterate-key and hash-ref. If a key produced by key-proc does not yield a value through hash-ref, then the exn:fail:contract exception is raised.

Pairs of prop and prop-val add impersonator properties or override impersonator-property values of hash.

In the case of an immutable hash table, two impersonated hash tables count as “the same value” (for purposes of impersonator-of?) when their redirection procedures were originally attached to a hash table by the same call to impersonate-hash or chaperone-hash (and potentially propagated by hash-set, hash-remove, or hash-clear), as long as the content of the first hash table is impersonator-of? of the second hash table.

Changed in version 6.3.0.11 of package base: Added the equal-key-proc argument.

```
(impersonate-channel channel
  get-proc
  put-proc
  prop
  prop-val ...
  ...
) → (and/c channel? impersonator?)
channel : channel?
get-proc : (channel? . -> . (values channel? (any/c . -> . any/c)))
put-proc : (channel? any/c . -> . any/c)
```
Returns an impersonator of `channel`, which redirects the `channel-get` and `channel-put` operations.

The `get-proc` generator is called on `channel-get` or any other operation that fetches results from the channel (such as a `sync` on the channel). The `get-proc` must return two values: a channel that is an impersonator of `channel`, and a procedure that is used to check the channel’s contents.

The `put-proc` must accept `channel` and the value passed to `channel-put`; it must produce a replacement value, which is used with `channel-put` on the original `channel` to send the value over the channel.

Pairs of `prop` and `prop-val` (the number of arguments to `impersonate-channel` must be odd) add impersonator properties or override impersonator-property values of `channel`.

```
(prop : impersonator-property?)
(prop-val : any/c)
```

Returns an impersonator of `prompt-tag`, which redirects the `call-with-continuation-prompt` and `abort-current-continuation` operations.

The `handle-proc` must accept the values that the handler of a continuation prompt would take and it must produce replacement values, which will be passed to the handler.

The `abort-proc` must accept the values passed to `abort-current-continuation`; it must produce replacement values, which are aborted to the appropriate prompt.

The `cc-guard-proc` must accept the values produced by `call-with-continuation-prompt` in the case that a non-composable continuation is applied to replace the continuation.
that is delimited by the prompt, but only if \texttt{abort-current-continuation} is not later used to abort the continuation delimited by the prompt (in which case \texttt{abort-proc} is used).

The \texttt{callcc-impersonate-proc} must accept a procedure that guards the result of a continuation captured by \texttt{call-with-current-continuation} with the impersonated prompt tag. The \texttt{callcc-impersonate-proc} is applied (under a continuation barrier) when the captured continuation is applied to refine a guard function (initially \texttt{values}) that is specific to the delimiting prompt; this prompt-specific guard is ultimately composed with any \texttt{cc-guard-proc} that is in effect at the delimiting prompt, and it is not used in the same case that a \texttt{cc-guard-proc} is not used (i.e., when \texttt{abort-current-continuation} is used to abort to the prompt). In the special case where the delimiting prompt at application time is a thread’s built-in initial prompt, \texttt{callcc-impersonate-proc} is ignored (partly on the grounds that the initial prompt’s result is ignored).

Pairs of \texttt{prop} and \texttt{prop-val} (the number of arguments to \texttt{impersonate-prompt-tag} must be odd) add impersonator properties or override impersonator-property values of \texttt{prompt-tag}.

Examples:

```
> (define tag
  (impersonate-prompt-tag
   (make-continuation-prompt-tag)
   (lambda (n) (* n 2))
   (lambda (n) (+ n 1))))

> (call-with-continuation-prompt
   (lambda ()
     (abort-current-continuation tag 5))
   tag
   (lambda (n) n))

12
```

```
(impersonate-continuation-mark-key  key
  get-proc
  set-proc
  prop
  prop-val ...
  ...
)
```

\texttt{→ (and/c continuation-mark? impersonator?)
key : continuation-mark-key?
get-proc : procedure?
set-proc : procedure?
prop : impersonator-property?
prop-val : any/c}

Returns an impersonator of \texttt{key}, which redirects \texttt{with-continuation-mark} and continuation mark accessors such as \texttt{continuation-mark-set->list}.
The get-proc must accept the value attached to a continuation mark and it must produce a replacement value, which will be returned by the continuation mark accessor.

The set-proc must accept a value passed to with-continuation-mark; it must produce a replacement value, which is attached to the continuation frame.

Pairs of prop and prop-val (the number of arguments to impersonate-continuation-mark-key must be odd) add impersonator properties or override impersonator-property values of key.

Examples:

> (define mark-key
   (impersonate-continuation-mark-key
    (make-continuation-mark-key)
    (lambda (l) (map char-upcase l))
    (lambda (s) (string->list s))))

> (with-continuation-mark mark-key "quiche"
   (continuation-mark-set-first
    (current-continuation-marks)
    mark-key))
'(\Q \U \I \C \H \E)

prop:impersonator-of : struct-type-property?

A structure type property (see §5.3 “Structure Type Properties”) that supplies a procedure for extracting an impersonated value from a structure that represents an impersonator. The property is used for impersonator-of? as well as equal?.

The property value must be a procedure of one argument, which is a structure whose structure type has the property. The result can be #f to indicate the structure does not represent an impersonator, otherwise the result is a value for which the original structure is an impersonator (so the original structure is an impersonator-of? and equal? to the result value). The result value must have the same prop:impersonator-of and prop:equal+hash property values as the original structure, if any, and the property values must be inherited from the same structure type (which ensures some consistency between impersonator-of? and equal?).

Impersonator property predicates and accessors applied to a structure with the prop:impersonator-of property first check for the property on the immediate structure, and if it is not found, the value produced by the prop:impersonator-of procedure is checked (recursively).

Changed in version 6.1.1.8 of package base: Made impersonator property predicates and accessors sensitive to prop:impersonator-of.

prop:authentic : struct-type-property?
A structure type property that declares a structure type as *authentic*. The value associated with the property is ignored; the presence of the property itself makes the structure type authentic.

Instances of an authentic structure type cannot be impersonated via `impersonate-struct` or chaperoned via `chaperone-struct`. As a consequence, an instance of an authentic structure type can be given a contract (see `struct/c`) only if it is a flat contract.

Declaring a structure type as authentic can prevent unwanted structure impersonation, but exposed structure types normally should support impersonators or chaperones to facilitate contracts. Declaring a structure type as authentic can also slightly improve the performance of structure predicates, selectors, and mutators, which can be appropriate for data structures that are private and frequently used within a library.

Added in version 6.9.0.4 of package `base`.

### 14.5.2 Chaperone Constructors

```scheme
(chaperone-procedure proc
    wrapper-proc
    prop
    prop-val ...
    ...)
```

→ `(and/c procedure? chaperone?)

- `proc`: `procedure?`
- `wrapper-proc`: `(or/c procedure? #f)`
- `prop`: `impersonator-property?`
- `prop-val`: `any/c`

Like `impersonate-procedure`, but for each value supplied to `wrapper-proc`, the corresponding result must be the same or a chaperone of (in the sense of `chaperone-of?`) the supplied value. The additional result, if any, that precedes the chaperoned values must be a procedure that accepts as many results as produced by `proc`; it must return the same number of results, each of which is the same or a chaperone of the corresponding original result.

For applications that include keyword arguments, `wrapper-proc` must return an additional value before any other values but after the result-chaperoning procedure (if any). The additional value must be a list of chaperones of the keyword arguments that were supplied to the chaperone procedure (i.e., not counting optional arguments that were not supplied). The arguments must be ordered according to the sorted order of the supplied arguments’ keywords.
Like `chaperone-procedure`, but `wrapper-proc` receives an extra argument as with `impersonate-procedure*`.

Added in version 6.1.1.5 of package `base`.

Like `impersonate-struct`, but with the following refinements, where `self` refers to the value to which a `orig-proc` is originally applied:

- With a structure-field accessor as `orig-proc, redirect-proc` must accept two arguments, `self` and the value `field-v` that `orig-proc` produces for `v`; it must return a chaperone of `field-v`. The corresponding field may be immutable.
- With structure-field mutator as `orig-proc, redirect-proc` must accept two arguments, `self` and the value `field-v` supplied to the mutator; it must return a chaperone of `field-v` to be propagated to `orig-proc` and `v`. 

1067
A property accessor can be supplied as orig-proc, and the property need not have been created with 'can-impersonate. The corresponding redirect-proc uses the same protocol as for a structure-field accessor.

With struct-info as orig-proc, the corresponding redirect-proc must accept two values, which are the results of struct-info on v; it must return each values or a chaperone of each value. The redirect-proc is not called if struct-info would return #f as its first argument. An orig-proc can be struct-info only if struct-type or some other orig-proc is supplied.

Any accessor or mutator orig-proc that is an impersonator must be specifically a chaperone.

Supplying a property accessor for orig-proc enables prop arguments, the same as supplying an accessor, mutator, or structure type.

Changed in version 6.1.1.2 of package base: Changed first argument to an accessor or mutator redirect-proc from v to self.
Changed in version 6.1.1.8: Added optional struct-type argument.

Like impersonate-vector, but with support for immutable vectors. The ref-proc procedure must produce the same value or a chaperone of the original value, and set-proc must produce the value that is given or a chaperone of the value. The set-proc will not be used if vec is immutable.

```
(chaperone-vector vec
  ref-proc
  set-proc
  prop
  prop-val ...
  ...) → (and/c vector? chaperone?)
vec : vector?
ref-proc : (or/c (vector? exact-nonnegative-integer? any/c . -> . any/c) #f)
set-proc : (or/c (vector? exact-nonnegative-integer? any/c . -> . any/c) #f)
prop : impersonator-property?
prop-val : any/c

(chaperone-vector* vec
  ref-proc
  set-proc
  prop
  prop-val ...
  ...) → (and/c vector? chaperone?)
vec : (and/c vector? (not/c immutable?))
ref-proc : (or/c (vector? vector? exact-nonnegative-integer? any/c . -> . any/c) #f)
set-proc : (or/c (vector? vector? exact-nonnegative-integer? any/c . -> . any/c) #f)
prop : impersonator-property?
prop-val : any/c
```
Like `chaperone-vector`, but `ref-proc` and `set-proc` receive an extra argument as with `impersonate-vector*`.

Added in version 6.9.0.2 of package `base`.

```
(chaperone-box  box
  unbox-proc
  set-proc
  prop
  prop-val ...
  ...)  → (and/c box? chaperone?)

box  : box?
unbox-proc  : (box? any/c . -> . any/c)
set-proc  : (box? any/c . -> . any/c)
prop  : impersonator-property?
prop-val  : any/c
```

Like `impersonate-box`, but with support for immutable boxes. The `unbox-proc` procedure must produce the same value or a chaperone of the original value, and `set-proc` must produce the same value or a chaperone of the value that it is given. The `set-proc` will not be used if `box` is immutable.

```
(chaperone-hash  hash
  ref-proc
  set-proc
  remove-proc
  key-proc
  [clear-proc
   equal-key-proc]
  prop
  prop-val ...
  ...)  → (and/c hash? chaperone?)

hash  : hash?
ref-proc  : (hash? any/c . -> . (values
  any/c
  (hash? any/c any/c . -> . any/c)))
set-proc  : (hash? any/c any/c . -> . (values any/c any/c))
remove-proc  : (hash? any/c . -> . any/c)
key-proc  : (hash? any/c . -> . any/c)
clear-proc  : (or/c #f (hash? . -> . any)) = #f
equal-key-proc  : (or/c #f (hash? any/c . -> . any/c)) = #f
prop  : impersonator-property?
prop-val  : any/c
```

Like `impersonate-hash`, but with constraints on the given functions and support for immutable hashes. The `ref-proc` procedure must return a found value or a chaperone of the
value. The \texttt{set-proc} procedure must produce two values: the key that it is given or a chaperone of the key and the value that it is given or a chaperone of the value. The \texttt{remove-proc}, \texttt{key-proc}, and \texttt{equal-key-proc} procedures must produce the given key or a chaperone of the key.

Changed in version 6.3.0.11 of package \texttt{base}: Added the \texttt{equal-key-proc} argument.

\begin{verbatim}
(chaperone-struct-type struct-type
   struct-info-proc
   make-constructor-proc
   guard-proc
   prop
   prop-val ...
   ...
) → (and/c struct-type? chaperone?)
 struct-type : struct-type?
 struct-info-proc : procedure?
 make-constructor-proc : (procedure? . -> . procedure?)
 guard-proc : procedure?
 prop : impersonator-property?
 prop-val : any/c
\end{verbatim}

Returns a chaperoned value like \texttt{struct-type}, but with \texttt{struct-type-info} and \texttt{struct-type-make-constructor} operations on the chaperoned structure type redirected. In addition, when a new structure type is created as a subtype of the chaperoned structure type, \texttt{guard-proc} is interposed as an extra guard on creation of instances of the subtype.

The \texttt{struct-info-proc} must accept 8 arguments—the result of \texttt{struct-type-info} on \texttt{struct-type}. It must return 8 values, where each is the same or a chaperone of the corresponding argument. The 8 values are used as the results of \texttt{struct-type-info} for the chaperoned structure type.

The \texttt{make-constructor-proc} must accept a single procedure argument, which is a constructor produced by \texttt{struct-type-make-constructor} on \texttt{struct-type}. It must return the same or a chaperone of the procedure, which is used as the result of \texttt{struct-type-make-constructor} on the chaperoned structure type.

The \texttt{guard-proc} is like a \texttt{guard} argument to \texttt{make-struct-type}: it must accept one more argument than a constructor for \texttt{struct-type}, where the last argument is the name the name of the instantiated structure type. It must return the number of values needed by the constructor (i.e. one value for each argument but the last), and each returned value must be the same as or a chaperone of the corresponding argument. The \texttt{guard-proc} is added as a constructor guard when a subtype is created of the chaperoned structure type.

Pairs of \texttt{prop} and \texttt{prop-val} (the number of arguments to \texttt{chaperone-struct-type} must be even) add impersonator properties or override impersonator-property values of \texttt{struct-type}. 

1070
(chaperone-evt evt proc prop prop-val ... ...)
→ (and/c evt? chaperone?)
evtt : evt?
proc : (evt? . -> . (values evt? (any/c . -> . any/c)))
prop : impersonator-property?
prop-val : any/c

Returns a chaperoned value like \( evt \), but with \( proc \) as an event generator when the result is synchronized with functions like \( \text{sync} \).

The \( proc \) generator is called on synchronization, much like the procedure passed to \( \text{guard-evt} \), except that \( proc \) is given \( evt \). The \( proc \) must return two values: a synchronizable event that is a chaperone of \( evt \), and a procedure that is used to check the event’s result if it is chosen in a selection. The latter procedure accepts the result of \( evt \), and it must return a chaperone of that value.

Pairs of \( prop \) and \( prop-val \) (the number of arguments to \( \text{chaperone-evt} \) must be even) add impersonator properties or override impersonator-property values of \( evt \).

The result is \( \text{chaperone-of} \) the argument \( evt \). However, if \( evt \) is a thread, semaphore, input port, output port, or will executor, the result is not recognized as such. For example, \( \text{thread?} \) applied to the result of \( \text{chaperone-evt} \) will always produce \#f.

Like \( \text{impersonate-channel} \), but with restrictions on the \( \text{get-proc} \) and \( \text{put-proc} \) procedures.

The \( \text{get-proc} \) must return two values: a channel that is a chaperone of \( channel \), and a procedure that is used to check the channel’s contents. The latter procedure must return the original value or a chaperone of that value.

The \( \text{put-proc} \) must produce a replacement value that is either the original value communicated on the channel or a chaperone of that value.

Pairs of \( prop \) and \( prop-val \) (the number of arguments to \( \text{chaperone-channel} \) must be
odd) add impersonator properties or override impersonator-property values of channel.

((chaperone-prompt-tag prompt-tag
  handle-proc
  abort-proc
  [cc-guard-proc
callcc-chaperone-proc]
  prop
  prop-val ...
...)
→ (and/c continuation-prompt-tag? chaperone?)
prompt-tag : continuation-prompt-tag?
handle-proc : procedure?
abort-proc : procedure?
cc-guard-proc : procedure? = values
callcc-chaperone-proc : (procedure? . -> . procedure?) = (lambda (p) p)
prop : impersonator-property?
prop-val : any/c

Like impersonate-prompt-tag, but produces a chaperoned value. The handle-proc procedure must produce the same values or chaperones of the original values, abort-proc must produce the same values or chaperones of the values that it is given, and cc-guard-proc must produce the same values or chaperones of the original result values, and callcc-chaperone-proc must produce a procedure that is a chaperone or the same as the given procedure.

Examples:

> (define bad-chaperone
  (chaperone-prompt-tag
    (make-continuation-prompt-tag)
    (lambda (n) (* n 2))
    (lambda (n) (+ n 1))))

> (call-with-continuation-prompt
  (lambda ()
    (abort-current-continuation bad-chaperone 5))
  bad-chaperone
  (lambda (n) n))
abort-current-continuation: non-chaperone result; received a prompt-abort argument that is not a chaperone of the original prompt-abort argument
original: 5
received: 6

> (define good-chaperone
  (chaperone-prompt-tag
(make-continuation-prompt-tag)
(lambda (n) (if (even? n) n (error "not even")))
(lambda (n) (if (even? n) n (error "not even"))))

> (call-with-continuation-prompt
  (lambda ()
    (abort-current-continuation good-chaperone 2))
  good-chaperone
  (lambda (n) n))

2

(chaperone-continuation-mark-key key
  get-proc
  set-proc
  prop
  prop-val ...
  ...
)

→ (and/c continuation-mark-key? chaperone?)
key : continuation-mark-key?
get-proc : procedure?
set-proc : procedure?
prop : impersonator-property?
prop-val : any/c

Like impersonate-continuation-mark-key, but produces a chaperoned value. The
get-proc procedure must produce the same value or a chaperone of the original value,
and set-proc must produce the same value or a chaperone of the value that it is given.

Examples:

> (define bad-chaperone
  (chaperone-continuation-mark-key
   (make-continuation-mark-key)
   (lambda (l) (map char-upcase l))
   string->list))

> (with-continuation-mark bad-chaperone "timballo"
  (continuation-mark-set-first
    (current-continuation-marks)
    bad-chaperone))

with-continuation-mark: non-chaperone result; received a
value that is not a chaperone of the original value
original: "timballo"
received: '(#z t z i z m z b z a z l z l z o)

> (define (checker s)
  (if (> (string-length s) 5)
    s
    (error "expected string of length at least 5")))
> (define good-chaperone
  (chaperone-continuation-mark-key
   (make-continuation-mark-key)
   checker
   checker))

> (with-continuation-mark good-chaperone "zabaione"
  (continuation-mark-set-first
   (current-continuation-marks)
   good-chaperone))
"zabaione"

14.5.3 Impersonator Properties

(make-impersonator-property name)
  ➔ impersonator-property?
    (-> any/c boolean?)
    (->* (impersonator?) (any/c) any)
name : symbol?

Creates a new impersonator property and returns three values:

• an impersonator property descriptor, for use with impersonate-procedure, chaperone-procedure, and other impersonator constructors;

• an impersonator property predicate procedure, which takes an arbitrary value and returns #t if the value is an impersonator with a value for the property, #f otherwise;

• an impersonator property accessor procedure, which returns the value associated with an impersonator for the property; if a value given to the accessor is not an impersonator or does not have a value for the property (i.e. if the corresponding impersonator property predicate returns #f), then a second optional argument to the selector determines its response: the exn:fail:contract exception is raised if a second argument is not provided, the second argument is tail-called with zero arguments if it is a procedure, and the second argument is returned otherwise.

(impersonator-property? v) ➔ boolean?
  v : any/c

Returns #t if v is a impersonator property descriptor value, #f otherwise.

(impersonator-property-accessor-procedure? v) ➔ boolean?
  v : any/c
Returns \texttt{#t} if \( v \) is an accessor procedure produced by \texttt{make-impersonator-property}, \texttt{#f} otherwise.

\texttt{impersonator-prop:application-mark : impersonator-property?}

An impersonator property that is recognized by \texttt{impersonate-procedure} and \texttt{chaperone-procedure}.

### 14.6 Security Guards

\begin{verbatim}
(security-guard? \( v \)) → boolean?
\( v \) : any/c
\end{verbatim}

Returns \texttt{#t} if \( v \) is a security guard value as created by \texttt{make-security-guard}, \texttt{#f} otherwise.

A security guard provides a set of access-checking procedures to be called when a thread initiates access of a file, directory, or network connection through a primitive procedure. For example, when a thread calls \texttt{open-input-file}, the thread’s current security guard is consulted to check whether the thread is allowed read access to the file. If access is granted, the thread receives a port that it may use indefinitely, regardless of changes to the security guard (although the port’s custodian could shut down the port; see §14.7 “Custodians”).

A thread’s current security guard is determined by the \texttt{current-security-guard} parameter. Every security guard has a parent, and a parent’s access procedures are called whenever a child’s access procedures are called. Thus, a thread cannot increase its own access arbitrarily by installing a new guard. The initial security guard enforces no access restrictions other than those enforced by the host platform.

\begin{verbatim}
(make-security-guard parent
   file-guard
   network-guard
   [link-guard]) → security-guard?

parent : security-guard?
file-guard : (symbol?
   (or/c path? #f)
   (listof symbol?)
   . -> . any)
network-guard : (symbol?
   (or/c (and/c string? immutable?) #f)
   (or/c (integer-in 1 65535) #f)
   (or/c 'server 'client)
   . -> . any)
link-guard : (or/c (symbol? path? path? . -> . any) #f) = #f
\end{verbatim}
Creates a new security guard as child of `parent`.

The `file-guard` procedure must accept three arguments:

- a symbol for the primitive procedure that triggered the access check, which is useful for raising an exception to deny access.
- a path (see §15.1 "Paths") or `#f` for pathless queries, such as `(current-directory)`, `(filesystem-root-list)`, and `(find-system-path symbol)`. A path provided to `file-guard` is not expanded or otherwise normalized before checking access; it may be a relative path, for example.
- a list containing one or more of the following symbols:
  - `'read` — read a file or directory
  - `'write` — modify or create a file or directory
  - `'execute` — execute a file
  - `'delete` — delete a file or directory
  - `'exists` — determine whether a file or directory exists, or that a path string is well-formed

The `'exists` symbol is never combined with other symbols in the last argument to `file-guard`, but any other combination is possible. When the second argument to `file-guard` is `#f`, the last argument always contains only `'exists`.

The `network-guard` procedure must accept four arguments:

- a symbol for the primitive operation that triggered the access check, which is useful for raising an exception to deny access.
- an immutable string representing the target hostname for a client connection or the accepting hostname for a listening server; `#f` for a listening server or UDP socket that accepts connections at all of the host’s address; or `#f` an unbound UDP socket.
- an exact integer between 1 and 65535 (inclusive) representing the port number, or `#f` for an unbound UDP socket. In the case of a client connection, the port number is the target port on the server. For a listening server, the port number is the local port number.
- a symbol, either `'client` or `'server`, indicating whether the check is for the creation of a client connection or a listening server. The opening of an unbound UDP socket is identified as a `'client` connection; explicitly binding the socket is identified as a `'server` action.

The `link-guard` argument can be `#f` or a procedure of three arguments:
• a symbol for the primitive procedure that triggered the access check, which is useful for raising an exception to deny access.

• a complete path (see §15.1 “Paths”) representing the file to create as link.

• a path representing the content of the link, which may be relative the second-argument path; this path is not expanded or otherwise normalized before checking access.

If \texttt{link-guard} is \#f, then a default procedure is used that always raises \texttt{exn:fail}.

The return value of \texttt{file-guard}, \texttt{network-guard}, or \texttt{link-guard} is ignored. To deny access, the procedure must raise an exception or otherwise escape from the context of the primitive call. If the procedure returns, the parent’s corresponding procedure is called on the same inputs, and so on up the chain of security guards.

The \texttt{file-guard}, \texttt{network-guard}, and \texttt{link-guard} procedures are invoked in the thread that called the access-checked primitive. Breaks may or may not be enabled (see §10.6 “Breaks”). Full continuation jumps are blocked going into or out of the \texttt{file-guard} or \texttt{network-guard} call (see §1.1.11 “Prompts, Delimited Continuations, and Barriers”).

\begin{verbatim}
  (current-security-guard) → security-guard?
  (current-security-guard guard) → void?
  guard : security-guard?
\end{verbatim}

A parameter that determines the current security guard that controls access to the filesystem and network.

### 14.7 Custodians

See §1.1.15 “Custodians” for basic information on the Racket custodian model.

\begin{verbatim}
  (custodian? v) → boolean?
  v : any/c
\end{verbatim}

Returns \#t if \(v\) is a custodian value, \#f otherwise.

\begin{verbatim}
  (make-custodian [cust]) → custodian?
  cust : (and/c custodian? (not/c custodian-shut-down?))
  = (current-custodian)
\end{verbatim}

Creates a new custodian that is subordinate to \texttt{cust}. When \texttt{cust} is directed (via \texttt{custodian-shutdown-all}) to shut down all of its managed values, the new subordinate custodian is automatically directed to shut down its managed values as well.
(custodian-shutdown-all cust) → void?
cust : custodian?

Closes all file-stream ports, TCP ports, TCP listeners, and UDP sockets that are managed by `cust` (and its subordinates), and empties all custodian boxes associated with `cust` (and its subordinates). It also removes `cust` (and its subordinates) as managers of all threads; when a thread has no managers, it is killed (or suspended; see thread/suspend-to-kill) if the current thread is to be killed, all other shut-down actions take place before killing the thread.

If `cust` is already shut down, then `custodian-shutdown-all` has no effect. When a custodian is shut down and it has subordinate custodians, the subordinates are not only shut down, they no longer count as subordinates.

(custodian-shut-down? cust) → boolean?
cust : custodian?

Returns #t if `cust` has been shut down with `custodian-shutdown-all` or if it was a subordinate of a custodian that is shut down, #f otherwise.

Added in version 6.11.0.5 of package base.

(custodian-memory-accounting-available?) → boolean?

Memory accounting is normally available, but not in the CGC implementation.

(custodian-managed-list cust super) → list?
cust : custodian?
super : custodian?

Returns a list of immediately managed objects (not including custodian boxes) and subordinate custodians for `cust`, where `cust` is itself subordinate to `super` (directly or indirectly). If `cust` is not strictly subordinate to `super`, the exn:fail:contract exception is raised.

If `cust` has been shut down, the result is '(). If `cust` was a subordinate of a custodian that was shut down, then it cannot be a subordinate of `super`.

(custodian-memory-accounting-available?) → boolean?

Returns #t if Racket is compiled with support for per-custodian memory accounting, #f otherwise.
(custodian-require-memory limit-cust need- amt stop-cust) → void?

  limit-cust : custodian?
  need- amt : exact-nonnegative-integer?
  stop-cust : custodian?

Registers a required-memory check if Racket is compiled with support for per-custodian memory accounting, otherwise the exn:fail:unsupported exception is raised.

If a check is registered, and if Racket later reaches a state after garbage collection (see §1.1.6 “Garbage Collection”) where allocating need- amt bytes charged to limit-cust would fail or trigger some shutdown, then stop- cust is shut down.

The stop- cust must be a subordinate custodian of limit-cust.

(custodian-limit-memory limit-cust limit- amt [stop-cust]) → void?

  limit-cust : custodian?
  limit- amt : exact-nonnegative-integer?
  stop-cust : custodian? = limit-cust

Registers a limited-memory check if Racket is compiled with support for per-custodian memory accounting, otherwise the exn:fail:unsupported exception is raised.

If a check is registered, and if Racket later reaches a state after garbage collection (see §1.1.6 “Garbage Collection”) where limit-cust owns more than limit- amt bytes, then stop- cust is shut down.

For reliable shutdown, limit- amt for custodian-limit-memory must be much lower than the total amount of memory available (minus the size of memory that is potentially used and not charged to limit-cust). Moreover, if individual allocations that are initially charged to limit- cust can be arbitrarily large, then stop- cust must be the same as limit- cust, so that excessively large immediate allocations can be rejected with an exn:fail:out-of-memory exception.

(make-custodian-box cust v) → custodian-box?

  cust : custodian?
  v : any/c

Returns a custodian box that contains v as long as cust has not been shut down. If cust is already shut down, the custodian box’s value is immediately removed.

A custodian box is a synchronizable event (see §11.2.1 “Events”). The custodian box becomes ready when its custodian is shut down; the synchronization result of a custodian box

A custodian’s limit is checked only after a garbage collection, except that it may also be checked during certain large allocations that are individually larger than the custodian’s limit. A single garbage collection may shut down multiple custodians, even if shutting down only one of the custodians would have reduced memory use for other custodians.
is the custodian box itself.

\[
(custodian-box?\ v) \rightarrow boolean?
\ v : any/c
\]

Returns \#t if \(v\) is a custodian box produced by `make-custodian-box`, \#f otherwise.

\[
(custodian-box-value\ cb) \rightarrow any
\ cb : custodian-box?
\]

Returns the value in the given custodian box, or \#f if the value has been removed.

### 14.8 Thread Groups

A thread group is a collection of threads and other thread groups that have equal claim to the CPU. By nesting thread groups and by creating certain threads within certain groups, a programmer can control the amount of CPU allocated to a set of threads. Every thread belongs to a thread group, which is determined by the `current-thread-group` parameter when the thread is created. Thread groups and custodians (see §14.7 "Custodians") are independent.

The root thread group receives all of the CPU that the operating system gives Racket. Every thread or nested group in a particular thread group receives equal allocation of the CPU (a portion of the group’s access), although a thread may relinquish part of its allocation by sleeping or synchronizing with other processes.

\[
(make-thread-group\ [\ group]) \rightarrow thread-group?
\ group : thread-group? = (current-thread-group)
\]

Creates a new thread group that belongs to \(group\).

\[
(thread-group?\ v) \rightarrow boolean?
\ v : any/c
\]

Returns \#t if \(v\) is a thread group value, \#f otherwise.

\[
(current-thread-group) \rightarrow thread-group?
\ (current-thread-group\ group) \rightarrow void?
\ group : thread-group?
\]

A parameter that determines the thread group for newly created threads.
14.9 Structure Inspectors

An *inspector* provides access to structure fields and structure type information without the normal field accessors and mutators. (Inspectors are also used to control access to module bindings; see §14.10 “Code Inspectors.”) Inspectors are primarily intended for use by debuggers.

When a structure type is created, an inspector can be supplied. The given inspector is not the one that will control the new structure type; instead, the given inspector’s parent will control the type. By using the parent of the given inspector, the structure type remains opaque to “peer” code that cannot access the parent inspector.

The `current-inspector` parameter determines a default inspector argument for new structure types. An alternate inspector can be provided though the `#:inspector` option of the `struct` form (see §5.1 “Defining Structure Types: struct”), or through an optional `inspector` argument to `make-struct-type`.

```scheme
(inspector? v) → boolean?
  v : any/c
```

Returns `#t` if `v` is an inspector, `#f` otherwise.

```scheme
(make-inspector [inspector]) → inspector?
  inspector : inspector? = (current-inspector)
```

Returns a new inspector that is a subinspector of `inspector`. Any structure type controlled by the new inspector is also controlled by its ancestor inspectors, but no other inspectors.

```scheme
(make-sibling-inspector [inspector]) → inspector?
  inspector : inspector? = (current-inspector)
```

Returns a new inspector that is a subinspector of the same inspector as `inspector`. That is, `inspector` and the result inspector control mutually disjoint sets of structure types.

```scheme
(inspector-superior? inspector maybe-subinspector) → boolean?
  inspector : inspector?
  maybe-subinspector : inspector?
```

Returns `#t` if `inspector` is an ancestor of `maybe-subinspector` (and not equal to `maybe-subinspector`), `#f` otherwise.

Added in version 6.5.0.6 of package base.
A parameter that determines the default inspector for newly created structure types.

```lisp
(struct-info v) → (or/c struct-type? #f) boolean?
v : any/c
```

Returns two values:

- `struct-type`: a structure type descriptor or `#f`; the result is a structure type descriptor of the most specific type for which `v` is an instance, and for which the current inspector has control, or the result is `#f` if the current inspector does not control any structure type for which the `struct` is an instance.

- `skipped?`: `#f` if the first result corresponds to the most specific structure type of `v`, `#t` otherwise.

```lisp
(struct-type-info struct-type)
→ symbol?
  exact-nonnegative-integer?
  exact-nonnegative-integer?
  struct-accessor-procedure?
  struct-mutator-procedure?
  (listof exact-nonnegative-integer?)
  (or/c struct-type? #f)
  boolean?
struct-type : struct-type?
```

Returns eight values that provide information about the structure type descriptor `struct-type`, assuming that the type is controlled by the current inspector:

- `name`: the structure type’s name as a symbol;
- `init-field-cnt`: the number of fields defined by the structure type provided to the constructor procedure (not counting fields created by its ancestor types);
- `auto-field-cnt`: the number of fields defined by the structure type without a counterpart in the constructor procedure (not counting fields created by its ancestor types);
- `accessor-proc`: an accessor procedure for the structure type, like the one returned by `make-struct-type`;
- `mutator-proc`: a mutator procedure for the structure type, like the one returned by `make-struct-type`;
- `immutable-k-list`: an immutable list of exact non-negative integers that correspond to immutable fields for the structure type;
• **super-type**: a structure type descriptor for the most specific ancestor of the type that is controlled by the current inspector, or `#f` if no ancestor is controlled by the current inspector;

• **skipped?**: `#f` if the seventh result is the most specific ancestor type or if the type has no supertype, `#t` otherwise.

If the type for `struct-type` is not controlled by the current inspector, the `exn:fail:contract` exception is raised.

```
(struct-type-sealed? struct-type) -> boolean?
struct-type : struct-type?
```

Reports whether `struct-type` has the `prop:sealed` structure type property.

Added in version 8.0.0.7 of package base.

```
(struct-type-authentic? struct-type) -> boolean?
struct-type : struct-type?
```

Reports whether `struct-type` has the `prop:authentic` structure type property.

Added in version 8.0.0.7 of package base.

```
(struct-type-make-constructor struct-type
                                 [constructor-name])
  -> struct-constructor-procedure?
  struct-type : struct-type?
  constructor-name : (or/c symbol? #f) = #f
```

Returns a constructor procedure to create instances of the type for `struct-type`. If `constructor-name` is not `#f`, it is used as the name of the generated constructor procedure. If the type for `struct-type` is not controlled by the current inspector, the `exn:fail:contract` exception is raised.

```
(struct-type-make-predicate struct-type) -> any
struct-type : any/c
```

Returns a predicate procedure to recognize instances of the type for `struct-type`. If the type for `struct-type` is not controlled by the current inspector, the `exn:fail:contract` exception is raised.

```
(object-name v) -> any
v : any/c
```
Returns a value for the name of \( v \) if \( v \) has a name, \#f otherwise. The argument \( v \) can be any value, but only (some) procedures, structures, structure types, structure type properties, regexp values, ports, loggers, and prompt tags have names. See also §1.2.6 “Inferred Value Names”.

If a structure’s type implements the prop:object-name property, and the value of the prop:object-name property is an integer, then the corresponding field of the structure is the name of the structure. Otherwise, the property value must be a procedure, which is called with the structure as argument, and the result is the name of the structure. If a structure is a procedure as implemented by one of its fields (i.e., the prop:procedure property value for the structure’s type is an integer), then its name is the implementing procedure’s name. Otherwise, its name matches the name of the structure type that it instantiates.

The name (if any) of a procedure is a symbol, unless the procedure is also a structure whose type has the prop:object-name property, in which case prop:object-name takes precedence. The procedure-rename function creates a procedure with a specific name.

The name of a regexp value is a string or byte string. Passing the string or byte string to regexp, byte-regexp, pregexp, or byte-pregexp (depending on the kind of regexp whose name was extracted) produces a value that matches the same inputs.

The name of a port can be any value, but many tools use a path or string name as the port’s for (to report source locations, for example).

The name of a logger is either a symbol or \#f.

The name of a prompt tag is either the optional symbol given to make-continuation-prompt-tag or \#f.

Changed in version 7.9.0.13 of package base: Recognize the name of continuation prompt tags.

**prop:object-name : struct-type-property?**

A structure type property that allows structure types to customize the result of object-name applied to their instances. The property value can be any of the following:

- A procedure proc of one argument: In this case, procedure proc receives the structure as an argument, and the result of proc is the object-name of the structure.
- An exact, non-negative integer between 0 (inclusive) and the number of non-automatic fields in the structure type (exclusive, not counting supertype fields): The integer identifies a field in the structure, and the field must be designated as immutable. The value of the field is used as the object-name of the structure.

Added in version 6.2 of package base.
In the same way that inspectors control access to structure fields (see §14.9 “Structure Inspectors”), inspectors also control access to module bindings. Inspectors used this way are code inspectors. The default code inspector for module bindings is determined by the current-code-inspector parameter, instead of the current-inspector parameter.

When a module declaration is evaluated, the value of the current-code-inspector parameter is associated with the module declaration. When the module is invoked via require or dynamic-require, a sub-inspector of the module’s declaration-time inspector is created, and this sub-inspector is associated with the module invocation. Any inspector that controls the sub-inspector (including the declaration-time inspector and its superior) controls the module invocation. In particular, if the value of current-code-inspector never changes, then no control is lost for any module invocation, since the module’s invocation is associated with a sub-inspector of current-code-inspector.

When an inspector that controls a module invocation is installed current-code-inspector, it enables the following module->namespace on the module, and it enables access to the module’s protected exports (i.e., those identifiers exported from the module with protect-out) via dynamic-require.

When a module form is expanded or a namespace is created, the value of current-code-inspector is associated with the module or namespace’s top-level lexical information. Syntax objects with that lexical information gain access to the protected and unexported bindings of any module that the inspector controls. In the case of a module, the inspector sticks with such syntax objects even if the syntax object is used in the expansion of code in a less powerful context; furthermore, if the syntax object is an identifier that is compiled as a variable reference, the inspector sticks with the variable reference even if it appears in a module form that is evaluated (i.e., declared) with a weaker inspector. When a syntax object or variable reference is within compiled code that is printed (see §1.4.16 “Printing Compiled Code”), the associated inspector is not preserved.

When compiled code in printed form is read back in, no inspectors are associated with the code. When the code is evaluated, the instantiated syntax-object literals and module-variable references acquire value of current-code-inspector as their inspector.

When a module instantiation is attached to multiple namespaces, each with its own module registry, the inspector for the module invocation can be registry-specific. The invocation inspector in a particular module registry can be changed via namespace-unprotect-module (but changing the inspector requires control over the old one).

```
(current-code-inspector) -> inspector?
(current-code-inspector insp) -> void?
    insp : inspector?
```

A parameter that determines an inspector to control access to module bindings and redefini-
If the code inspector is changed from its original value, then bytecode loaded by the default compiled-load handler is marked as non-runnable.

## 14.11 Plumbers

A plumber supports flush callbacks, which are normally triggered just before a Racket process or place exits. For example, a flush callback might flush an output port’s buffer.

There is no guarantee that a flush callback will be called before a process terminates—either because the plumber is not the original plumber that is flushed by the default exit handler, or because the process is terminated forcibly (e.g., through a custodian shutdown).

```racket
(plumber? v) → boolean?

v : any/c
```

Returns #t if v is a plumber value, #f otherwise.

Added in version 6.0.1.8 of package base.

```racket
(make-plumber) → plumber?
```

Creates a new plumber.

Plumbers have no hierarchy (unlike custodians or inspectors), but a flush callback can be registered in one plumber to call `plumber-flush-all` with another plumber.

Added in version 6.0.1.8 of package base.

```racket
(current-plumber) → plumber?

(current-plumber plumber) → void?

plumber : plumber?
```

A parameter that determines a current plumber for flush callbacks. For example, creating an output file stream port registers a flush callback with the current plumber to flush the port as long as the port is opened.

Added in version 6.0.1.8 of package base.

```racket
(plumber-flush-all plumber) → void?

plumber : plumber?
```

Calls all flush callbacks that are registered with plumber.
The flush callbacks to call are collected from plumber before the first one is called. If a flush callback registers a new flush callback, the new one is not called. If a flush callback raises an exception or otherwise escapes, then the remaining flush callbacks are not called.

Added in version 6.0.1.8 of package base.

```scheme
(plumber-flush-handle? v) → boolean?
  v : any/c
```

Returns #t if v is a flush handle represents the registration of a flush callback, #f otherwise.

Added in version 6.0.1.8 of package base.

```scheme
(plumber-add-flush! plumber proc [weak?]) → plumber-flush-handle?
  plumber : plumber?
  proc : (plumber-flush-handle? . -> . any)
  weak? : any/c = #f
```

Registers proc as a flush callback with plumber, so that proc is called when plumber-flush-all is applied to plumber.

The result flush handle represents the registration of the callback, and it can be used with plumber-flush-handle-remove! to unregister the callback.

The given proc is reachable from the flush handle, but if weak? is true, then plumber retains only a weak reference to the result flush handle (and thus proc).

When proc is called as a flush callback, it is passed the same value that is returned by plumber-add-flush! so that proc can conveniently unregister itself. The call of proc is within a continuation barrier.

Added in version 6.0.1.8 of package base.

```scheme
(plumber-flush-handle-remove! handle) → void?
  handle : plumber-flush-handle?
```

Unregisters the flush callback that was registered by the plumber-add-flush! call that produced handle.

If the registration represented by handle has been removed already, then plumber-flush-handle-remove! has no effect.

Added in version 6.0.1.8 of package base.

## 14.12 Sandboxed Evaluation

```scheme
(require racket/sandbox) package: sandbox-lib
```

1087
The bindings documented in this section are provided by the racket/sandbox library, not racket/base or racket.

The racket/sandbox module provides utilities for creating “sandboxed” evaluators, which are configured in a particular way and can have restricted resources (memory and time), filesystem and network access, and much more. Sandboxed evaluators can be configured through numerous parameters — and the defaults are set for the common use case where sandboxes are very limited.

```scheme
(make-evaluator
  language
  input-program ...
  #:requires requires
  #:allow-for-require allow-for-require
  #:allow-for-load allow-for-load
  #:allow-read allow-read)
→ (any/c . -> . any)
language : (or/c module-path? 
  (list/c 'special symbol?)
  (cons/c 'begin list?))
input-program : any/c
requires : (listof (or/c module-path? path-string?)
  (cons/c 'for-syntax (listof module-path?)))
  = null
allow-for-require : (listof (or/c module-path? path?)) = null
allow-for-load : (listof path-string?) = null
allow-read : (listof (or/c module-path? path-string?)) = null
(make-module-evaluator
  module-decl
  [#:language lang
   #:allow-for-require allow-for-require
   #:allow-for-load allow-for-load
   #:allow-read allow-read])
→ (any/c . -> . any)
module-decl : (or/c syntax? pair? path? input-port? string? bytes?)
lang : (or/c #f module-path?) = #f
allow-for-require : (listof (or/c module-path? path?)) = null
allow-for-load : (listof path-string?) = null
allow-read : (listof (or/c module-path? path-string?)) = null
```

The make-evaluator function creates an evaluator with a language and requires specification, and starts evaluating the given input programs. The make-module-evaluator function creates an evaluator that works in the context of a given module. The result in either case is a function for further evaluation.

The returned evaluator operates in an isolated and limited environment. In particular, filesystem access is restricted, which may interfere with using modules from the filesystem that are not in a collection. See below for information on the allow-for-require, allow-for-
load, and allow-read arguments. When language is a module path or when requires is provided, the indicated modules are implicitly included in the allow-for-require list. (For backward compatibility, non-module-path? path strings are allowed in requires; they are implicitly converted to paths before addition to allow-for-require.)

Each input-program or module-decl argument provides a program in one of the following forms:

- an input port used to read the program;
- a string or a byte string holding the complete input;
- a path that names a file holding the input; or
- an S-expression or a syntax object, which is evaluated as with eval (see also get-uncovered-expressions).

In the first three cases above, the program is read using sandbox-reader, with line-counting enabled for sensible error messages, and with 'program as the source (used for testing coverage). In the last case, the input is expected to be the complete program, and is converted to a syntax object (using 'program as the source), unless it already is a syntax object.

The returned evaluator function accepts additional expressions (each time it is called) in essentially the same form: a string or byte string holding a sequence of expressions, a path for a file holding expressions, an S-expression, or a syntax object. If the evaluator receives an eof value, it is terminated and raises errors thereafter. See also kill-evaluator, which terminates the evaluator without raising an exception.

For make-evaluator, multiple input-programs are effectively concatenated to form a single program. The way that the input-programs are evaluated depends on the language argument:

- The language argument can be a module path (i.e., a datum that matches the grammar for module-path of require).
  In this case, the input-programs are automatically wrapped in a module, and the resulting evaluator works within the resulting module’s namespace.

- The language argument can be a list starting with 'special, which indicates a built-in language with special input configuration. The possible values are '(special r5rs) or a value indicating a teaching language: '(special beginner), '(special beginner-abbr), '(special intermediate), '(special intermediate-lambda), or '(special advanced).
  In this case, the input-programs are automatically wrapped in a module, and the resulting evaluator works within the resulting module’s namespace. In addition, certain
parameters (such as such as read-accept-infix-dot) are set to customize reading programs from strings and ports.

This option is provided mainly for older test systems. Using make-module-evaluator with input starting with #lang is generally better.

- Finally, language can be a list whose first element is 'begin.

In this case, a new namespace is created using sandbox-namespace-specs, which by default creates a new namespace using sandbox-make-namespace (which, in turn, uses make-base-namespace or make-gui-namespace depending on sandbox-gui-available and gui-available?).

In the new namespace, language is evaluated as an expression to further initialize the namespace.

The requires list adds additional imports to the module or namespace for the input-programs, even in the case that require is not made available through the language.

The following examples illustrate the difference between an evaluator that puts the program in a module and one that merely initializes a top-level namespace:

```racket
> (define base-module-eval
  ; a module cannot have free variables...
  (make-evaluator 'racket/base '(define (f) later)))

program:1:0: later: unbound identifier
in: later

> (define base-module-eval
  (make-evaluator 'racket/base '(define (f) later)
                 '(define later 5)))

> (base-module-eval '(f))
5

> (define base-top-eval
  ; non-module code can have free variables:
  (make-evaluator '(begin) '(define (f) later)))

> (base-top-eval '(+ 1 2))
3

> (base-top-eval '(define later 5))
> (base-top-eval '(f))
5
```

The make-module-evaluator function is essentially a restriction of make-evaluator, where the program must be a module, and all imports are part of the program. In some cases it is useful to restrict the program to be a module using a specific module in its language position — use the optional lang argument to specify such a restriction (the default, #f, means no restriction is enforced). When the program is specified as a path, then the path is implicitly added to the allow-for-load list.
The `make-module-evaluator` function can be convenient for testing module files: pass in a path value for the file name, and you get back an evaluator in the module’s context which you can use with your favorite test facility.

In all cases, the evaluator operates in an isolated and limited environment:

- It uses a new custodian and namespace. When `gui-available?` and `sandbox-gui-available` produce true, it is also runs in its own eventspace.
- The evaluator works under the `sandbox-security-guard`, which restricts file system and network access.
- The evaluator is contained in a memory-restricted environment, and each evaluation is wrapped in a `call-with-limits` (when memory accounting is available); see also `sandbox-memory-limit`, `sandbox-eval-limits` and `set-eval-limits`.

Note that these limits apply to the creation of the sandbox environment too — so, for example, if the memory that is required to create the sandbox is higher than the limit, then `make-evaluator` will fail with a memory limit exception.

The `allow-for-require` and `allow-for-load` arguments adjust filesystem permissions to extend the set of files that are usable by the evaluator. Modules that are in a collection are automatically accessible, but the `allow-for-require` argument lists additional modules that can be `require`d along with their imports (transitively) through a filesystem path. The `allow-for-load` argument similarly lists files that can be `load`ed. (The precise permissions needed for `require` versus `load` can differ.) The `allow-read` argument is for backward compatibility, only; each `module-path?` element of `allow-read` is effectively moved to `allow-for-require`, while other elements are moved to `allow-for-load`.

The sandboxed environment is well isolated, and the evaluator function essentially sends it an expression and waits for a result. This form of communication makes it impossible to have nested (or concurrent) calls to a single evaluator. Usually this is not a problem, but in some cases you can get the evaluator function available inside the sandboxed code, for example:

```scheme
> (let ([e (make-evaluator 'racket/base)])
  (e `(e 1)))
evaluator: nested evaluator call with: 1
```

An error will be signaled in such cases.

```
(define base-module-eval2
  ; equivalent to base-module-eval:
  (make-module-evaluator '(module m racket/base
                            (define (f) later)
                            (define later 5)))
```

1091
If the value of `sandbox-propagate-exceptions` is true (the default) when the sandbox is created, then exceptions (both syntax and run-time) are propagated as usual to the caller of the evaluation function (i.e., catch them with `with-handlers`). If the value of `sandbox-propagate-exceptions` is `#f` when the sandbox is created, then uncaught exceptions in a sandbox evaluation cause the error to be printed to the sandbox’s error port, and the caller of the evaluation receives `#<void>`.

Finally, the fact that a sandboxed evaluator accept syntax objects makes it usable as the value for `current-eval`, which means that you can easily start a sandboxed read-eval-print-loop:

```racket
(define e (make-evaluator 'racket/base))
(parameterize ([current-eval e])
  (read-eval-print-loop))
```

Note that in this code only the REPL interactions will be printed to the current output ports; using I/O operations inside the REPL will still use the usual sandbox parameters (defaulting to no I/O). In addition, the code works only from an existing toplevel REPL — specifically, `read-eval-print-loop` reads a syntax value and gives it the lexical context of the current namespace. Here is a variation that also allows I/O over the current input and output ports, and works when used from a module (by using a new namespace):

```racket
(parameterize ([sandbox-input current-input-port]
               [sandbox-output current-output-port]
               [sandbox-error-output current-error-port]
               [current-namespace (make-empty-namespace)])
  (parameterize ([current-eval (make-evaluator 'racket/base)])
    (read-eval-print-loop)))
```

```racket
(exn:fail:sandbox-terminated? v) → boolean?
  v : any/c
(exn:fail:sandbox-terminated-reason exn) → symbol?
  exn : exn:fail:sandbox-terminated?
```

A predicate and accessor for exceptions that are raised when a sandbox is terminated. Once a sandbox raises such an exception, it will continue to raise it on further evaluation attempts.

### 14.12.1 Security Considerations

Although the sandbox is designed to provide a safe environment for executing Racket programs with restricted access to system resources, executing untrusted programs in a sandbox still carries some risk. Because a malicious program can exercise arbitrary functionality from the Racket runtime and installed collections, an attacker who identifies a vulnerability in Racket or an installed collection may be able to escape the sandbox.
To mitigate this risk, programs that use the sandbox should employ additional precautions when possible. Suggested measures include:

- Supplying a custom module language to `make-evaluator` or `make-module-evaluator` that gives untrusted code access to only the language constructs it absolutely requires.
- If untrusted code needs access to installed collections, installing only the collections required by your program.
- Using operating-system-level security features to provide defense-in-depth in case the process running the sandbox is compromised.
- Making sure your Racket installation and installed packages are up-to-date with the latest release.

### 14.12.2 Customizing Evaluators

The sandboxed evaluators that `make-evaluator` creates can be customized via many parameters. Most of the configuration parameters affect newly created evaluators; changing them has no effect on already-running evaluators.

The default configuration options are set for a very restricted sandboxed environment — one that is safe to make publicly available. Further customizations might be needed in case more privileges are needed, or if you want tighter restrictions. Another useful approach for customizing an evaluator is to begin with a relatively unrestricted configuration and add the desired restrictions. This approach is made possible by the `call-with-trusted-sandbox-configuration` function.

The sandbox environment uses two notions of restricting the time that evaluations takes: shallow time and deep time. **Shallow time** refers to the immediate execution of an expression. For example, a shallow time limit of five seconds would restrict `(sleep 6)` and other computations that take longer than five seconds. **Deep time** refers to the total execution of the expression and all threads and sub-processes that the expression creates. For example, a deep time limit of five seconds would restrict `(thread (λ () (sleep 6)))`, which shallow time would not, as well as all expressions that shallow time would restrict. By default, most sandboxes only restrict shallow time to facilitate expressions that use threads.

```
(call-with-trusted-sandbox-configuration thunk) → any

thunk : (-> any)
```

Invokes the `thunk` in a context where sandbox configuration parameters are set for minimal restrictions. More specifically, there are no memory or time limits, and the existing existing inspectors, security guard, exit handler, logger, plumber, and environment variable set are used. (Note that the I/O ports settings are not included.)
(sandbox-init-hook) → (→ any)
(sandbox-init-hook thunk) → void?

thunk : (→ any)

A parameter that determines a thunk to be called for initializing a new evaluator. The hook is called just before the program is evaluated in a newly-created evaluator context. It can be used to setup environment parameters related to reading, writing, evaluation, and so on. Certain languages ('(special r5rs) and the teaching languages) have initializations specific to the language; the hook is used after that initialization, so it can override settings.

(sandbox-reader) → (any/c . → . any)
(sandbox-reader proc) → void?

proc : (any/c . → . any)

A parameter that specifies a function that reads all expressions from (current-input-port). The function is used to read program source for an evaluator when a string, byte string, or port is supplied. The reader function receives a value to be used as input source (i.e., the first argument to read-syntax), and it should return a list of syntax objects. The default reader calls read-syntax, accumulating results in a list until it receives eof.

Note that the reader function is usually called as is, but when it is used to read the program input for make-module-evaluator, read-accept-lang and read-accept-reader are set to #t.

(sandbox-input) → (or/c #f
  string? bytes?
  input-port?
  'pipe
  (→ input-port?))

(sandbox-input in) → void?
in : (or/c #f
  string? bytes?
  input-port?
  'pipe
  (→ input-port?))

A parameter that determines the initial current-input-port setting for a newly created evaluator. It defaults to #f, which creates an empty port. The following other values are allowed:

- a string or byte string, which is converted to a port using open-input-string or open-input-bytes;
- an input port;
• the symbol "pipe", which triggers the creation of a pipe, where put-input can return the output end of the pipe or write directly to it;

• a thunk, which is called to obtain a port (e.g., using current-input-port means that the evaluator input is the same as the calling context’s input).

```
(sandbox-output) → (or/c #f
  output-port?
  'pipe
  'bytes
  'string
  (-> output-port?))
```

```
(sandbox-output in) → void?
in : (or/c #f
  output-port?
  'pipe
  'bytes
  'string
  (-> output-port?))
```

A parameter that determines the initial current-output-port setting for a newly created evaluator. It defaults to #f, which creates a port that discards all data. The following other values are allowed:

• an output port, which is used as-is;

• the symbol "bytes", which causes get-output to return the complete output as a byte string as long as the evaluator has not yet terminated (so that the size of the bytes can be charged to the evaluator);

• the symbol "string", which is similar to "bytes", but makes get-output produce a string;

• the symbol "pipe", which triggers the creation of a pipe, where get-output returns the input end of the pipe;

• a thunk, which is called to obtain a port (e.g., using current-output-port means that the evaluator output is not diverted).

```
(sandbox-error-output) → (or/c #f
  output-port?
  'pipe
  'bytes
  'string
  (-> output-port?))
```

```
(sandbox-error-output in) → void?
```

1095
Like `sandbox-output`, but for the initial `current-error-port` value. An evaluator’s error output is set after its output, so using `current-output-port` (the parameter itself, not its value) for this parameter value means that the error port is the same as the evaluator’s initial output port.

The default is `(lambda () (dup-output-port (current-error-port)))`, which means that the error output of the generated evaluator goes to the calling context’s error port.

A parameter that controls whether syntactic coverage information is collected by sandbox evaluators. Use `get-uncovered-expressions` to retrieve coverage information.

The default value is `#f`.

When both this boolean parameter and `(break-enabled)` are true, breaking while an evaluator is running propagates the break signal to the sandboxed context. This makes the sandboxed evaluator break, typically, but beware that sandboxed evaluation can capture and avoid the breaks (so if safe execution of code is your goal, make sure you use it with a time limit). Also, beware that a break may be received after the evaluator’s result, in which case the evaluation result is lost. Finally, beware that a break may be propagated after an evaluator has produced a result, so that the break is visible on the next interaction with the evaluator (or the break is lost if the evaluator is not used further). The default is `#t`.

A parameter that controls how uncaught exceptions during a sandbox evaluation are treated. When the parameter value is `#t`, then the exception is propagated to the caller of sandbox. When the parameter value is `#f`, the exception message is printed to the sandbox’s error port, and the caller of the sandbox receives `#<void>` for the evaluation. The default is `#t`. 1096
A parameter that holds a list of values that specify how to create a namespace for evaluation in make-evaluator or make-module-evaluator. The first item in the list is a thunk that creates the namespace, and the rest are module paths for modules to be attached to the created namespace using namespace-attach-module.

The default is (list sandbox-make-namespace).

The module paths are needed for sharing module instantiations between the sandbox and the caller. For example, sandbox code that returns posn values (from the lang/posn module) will not be recognized as such by your own code by default, since the sandbox will have its own instance of lang/posn and thus its own struct type for posns. To be able to use such values, include 'lang/posn in the list of module paths.

When testing code that uses a teaching language, the following piece of code can be helpful:

```scheme
(sandbox-namespace-specs
 (let ([specs (sandbox-namespace-specs)])
  `(,(car specs)
    ,(cdr specs)
    lang/posn
    ,(if (gui-available?) 'mrlib/cache-image-snip) '(()))))
```

Calls make-gui-namespace when (sandbox-gui-available) produces true, make-base-namespace otherwise.

Determines whether the racket/gui module can be used when a sandbox evaluator is created. If gui-available? produces #f during the creation of a sandbox evaluator, this parameter is forced to #f during initialization of the sandbox. The default value of the parameter is #t.

Various aspects of the library change when the GUI library is available, such as using a new eventspace for each evaluator.
A parameter that determines a list of collection directories to prefix `current-library-collection-paths` in an evaluator. This parameter is useful for cases when you want to test code using an alternate, test-friendly version of a collection, for example, testing code that uses a GUI (like the `htdp/world` teachpack) can be done using a fake library that provides the same interface but no actual interaction. The default is `null`.

A parameter that determines the initial `current-security-guard` for sandboxed evaluations. It can be either a security guard, or a function to construct one. The default is a function that restricts the access of the current security guard by forbidding all filesystem I/O except for specifications in `sandbox-path-permissions`, and it uses `sandbox-network-guard` for network connections.

A parameter that configures the behavior of the default sandbox security guard by listing paths and access modes that are allowed for them. The contents of this parameter is a list of specifications, each is an access mode and a byte-regexp for paths that are granted this access. The access mode symbol is one of: `'execute`, `'write`, `'delete`, `'read-bytecode`, `'read`, or `'exists`. These symbols are in decreasing order: each implies access for the following modes too (e.g., `'read` allows reading or checking for existence).

The path regexp is used to identify paths that are granted access. It can also be given as a path (or a string or a byte string), which is (made into a complete path, cleansed, simplified, and then) converted to a regexp that allows the path and sub-directories; e.g., `"/foo/bar"` applies to `"/foo/bar/baz"`.

An additional mode symbol, `'read-bytecode`, is not part of the linear order of these modes. Specifying this mode is similar to specifying `'read`, but it is not implied by any other mode.
(For example, even if you specify 'write for a certain path, you need to also specify 'read-bytecode to grant this permission.) The sandbox usually works in the context of a lower code inspector (see sandbox-make-code-inspector) which prevents loading of untrusted bytecode files — the sandbox is set-up to allow loading bytecode from files that are specified with 'read-bytecode. This specification is given by default to the Racket collection hierarchy (including user-specific libraries) and to libraries that are explicitly specified in an #:allow-read argument. (Note that this applies for loading bytecode files only, under a lower code inspector it is still impossible to use protected module bindings (see §14.10 "Code Inspectors").)

The default value is null, but when an evaluator is created, it is augmented by 'read-bytecode permissions that make it possible to use collection libraries (including sandbox-override-collection-paths). See make-evaluator for more information.

(sandbox-network-guard) → (symbol? (or/c (and/c string? immutable?) #f) (or/c integer-in 1 65535) #f) (or/c 'server 'client) . -> . any)

(sandbox-network-guard proc) → void?
proc : (symbol? (or/c (and/c string? immutable?) #f) (or/c integer-in 1 65535) #f) (or/c 'server 'client) . -> . any)

A parameter that specifies a procedure to be used (as is) by the default sandbox-security-guard. The default forbids all network connection.

(sandbox-exit-handler) → (any/c . -> . any)
(sandbox-exit-handler handler) → void?
handler : (any/c . -> . any)

A parameter that determines the initial (exit-handler) for sandboxed evaluations. The default kills the evaluator with an appropriate error message (see exn:fail:sandbox-terminated-reason).

(sandbox-memory-limit) → (or/c (>=/c 0) #f)
(sandbox-memory-limit limit) → void?
limit : (or/c (>=/c 0) #f)

A parameter that determines the total memory limit on the sandbox in megabytes (it can hold a rational or a floating point number). When this limit is exceeded, the sandbox is terminated. This value is used when the sandbox is created and the limit cannot be changed.
afterwards. It defaults to 30mb. See sandbox-eval-limits for per-evaluation limits and a description of how the two limits work together.

Note that (when memory accounting is enabled) memory is attributed to the highest custodian that refers to it. This means that if you inspect a value that sandboxed evaluation returns outside of the sandbox, your own custodian will be charged for it. To ensure that it is charged back to the sandbox, you should remove references to such values when the code is done inspecting it.

This policy has an impact on how the sandbox memory limit interacts with the per-expression limit specified by sandbox-eval-limits: values that are reachable from the sandbox, as well as from the interaction will count against the sandbox limit. For example, in the last interaction of this code,

```scheme
(define e (make-evaluator 'racket/base))
(e '(define a 1))
(e '(for ([i (in-range 20)]) (set! a (cons (make-bytes 500000) a))))
```

the memory blocks are allocated within the interaction limit, but since they’re chained to the defined variable, they’re also reachable from the sandbox — so they will count against the sandbox memory limit but not against the interaction limit (more precisely, no more than one block counts against the interaction limit).

A parameter that determines the default limits on each use of a make-evaluator function, including the initial evaluation of the input program. Its value should be a list of two numbers; where the first is a shallow time value in seconds, and the second is a memory limit in megabytes (note that they don’t have to be integers). Either one can be #f for disabling the corresponding limit; alternately, the parameter can be set to #f to disable all per-evaluation limits (useful in case more limit kinds are available in future versions). The default is (list 30 20).

Note that these limits apply to the creation of the sandbox environment too — even (make-evaluator 'racket/base) can fail if the limits are strict enough. For example,

```scheme
(parameterize ([sandbox-eval-limits '(0.25 5)])
    (make-evaluator 'racket/base '(sleep 2)))
```
will throw an error instead of creating an evaluator. Therefore, to avoid surprises you need to catch errors that happen when the sandbox is created.

When limits are set, call-with-limits (see below) is wrapped around each use of the evaluator, so consuming too much time or memory results in an exception. Change the limits of a running evaluator using set-eval-limits. The memory limit that is specified by this parameter applies to each individual evaluation, but not to the whole sandbox — that limit is specified via sandbox-memory-limit. When the global limit is exceeded, the sandbox is terminated, but when the per-evaluation limit is exceeded, an exception recognizable by exn:fail:resource? is raised. For example, say that you evaluate an expression like

```scheme
(for ([i (in-range 1000)])
  (set! a (cons (make-bytes 1000000) a))
  (collect-garbage))
```

then, assuming sufficiently small limits,

- if a global limit is set but no per-evaluation limit, the sandbox will eventually be terminated and no further evaluations possible;
- if there is a per-evaluation limit, but no global limit, the evaluation will abort with an error and it can be used again — specifically, a will still hold a number of blocks, and you can evaluate the same expression again which will add more blocks to it;
- if both limits are set, with the global one larger than the per-evaluation limit, then the evaluation will abort and you will be able to repeat it, but doing so several times will eventually terminate the sandbox (this will be indicated by the error message, and by the evaluator-alive? predicate).

```scheme
(sandbox-eval-handlers)
   → (list/c (or/c #f (-> any) . -> . any))
       (or/c #f (-> any) . -> . any))

(sandbox-eval-handlers handlers) → void?
   handlers : (list/c (or/c #f (-> any) . -> . any))
            (or/c #f (-> any) . -> . any))
```

A parameter that determines two (optional) handlers that wrap sandboxed evaluations. The first one is used when evaluating the initial program when the sandbox is being set-up, and the second is used for each interaction. Each of these handlers should expect a thunk as an argument, and they should execute these thunks — possibly imposing further restrictions. The default values are #f and call-with-custodian-shutdown, meaning no additional restrictions on initial sandbox code (e.g., it can start background threads), and a custodian-shutdown around each interaction that follows. Another useful function for this is call-with-killing-threads which kills all threads, but leaves other resources intact.
(sandbox-run-submodules) → (list/c symbol?)
(sandbox-run-submodules submod-syms) → void?
submod-syms : (list/c symbol?)

A parameter that determines submodules to run when a sandbox is created by `make-module-evaluator`. The parameter’s default value is the empty list.

(sandbox-make-inspector) → (-> inspector?)
(sandbox-make-inspector make) → void?
make : (-> inspector?)

A parameter that determines the (nullary) procedure that is used to create the inspector for sandboxed evaluation. The procedure is called when initializing an evaluator. The default parameter value is `(lambda () (make-inspector (current-inspector)))`.

(sandbox-make-code-inspector) → (-> inspector?)
(sandbox-make-code-inspector make) → void?
make : (-> inspector?)

A parameter that determines the (nullary) procedure that is used to create the code inspector for sandboxed evaluation. The procedure is called when initializing an evaluator. The default parameter value is `(lambda () (make-inspector (current-code-inspector)))`.

The `current-load/use-compiled` handler is setup to allow loading of bytecode files under the original code inspector when `sandbox-path-permissions` allows it through a `read-bytecode` mode symbol, which makes loading libraries possible.

(sandbox-make-logger) → (-> logger?)
(sandbox-make-logger make) → void?
make : (-> logger?)

A parameter that determines the procedure used to create the logger for sandboxed evaluation. The procedure is called when initializing an evaluator, and the default parameter value is `current-logger`. This means that it is not creating a new logger (this might change in the future).

(sandbox-make-plumber) → (or/c (-> plumber?) 'propagate)
(sandbox-make-plumber make) → void?
make : (or/c (-> plumber?) 'propagate)

A parameter that determines the procedure used to create the plumber for sandboxed evaluation. The procedure is called when initializing an evaluator.

If the value is `propagate` (the default), then a new plumber is created, and a flush callback is added to the current plumber to propagate the request to the new plumber within the created sandbox (if the sandbox has not already terminated).
Added in version 6.0.1.8 of package sandbox-lib.

```
(sandbox-make-environment-variables)
→ (→ environment-variables?)
(sandbox-make-environment-variables make) → void?
make : (→ environment-variables?)
```

A parameter that determines the procedure used to create the environment variable set for sandboxed evaluation. The procedure is called when initializing an evaluator, and the default parameter value constructs a new environment variable set using `(environment-variables-copy (current-environment-variables))`.

14.12.3 Interacting with Evaluators

The following functions are used to interact with a sandboxed evaluator in addition to using it to evaluate code.

```
(evaluator-alive? evaluator) → boolean?
evaluator : (any/c . → . any)
```

Determines whether the evaluator is still alive.

```
(kill-evaluator evaluator) → void?
evaluator : (any/c . → . any)
```

Releases the resources that are held by `evaluator` by shutting down the evaluator’s custodian. Attempting to use an evaluator after killing raises an exception, and attempts to kill a dead evaluator are ignored.

Killing an evaluator is similar to sending an `eof` value to the evaluator, except that an `eof` value will raise an error immediately.

```
(break-evaluator evaluator) → void?
evaluator : (any/c . → . any)
```

Sends a break to the running evaluator. The effect of this is as if Ctrl-C was typed when the evaluator is currently executing, which propagates the break to the evaluator’s context.

```
(get-user-custodian evaluator) → void?
evaluator : (any/c . → . any)
```

Retrieves the `evaluator`’s toplevel custodian. This returns a value that is different from `(evaluator '(current-custodian))` or `call-in-sandbox-context evaluator current-custodian` — each sandbox interaction is wrapped in its own custodian, which is what these would return.
(One use for this custodian is with current-memory-use, where the per-interaction sub-custodians will not be charged with the memory for the whole sandbox.)

\[
(set-eval-limits \text{ evaluator } \text{ secs } \text{ mb}) \rightarrow \text{ void?}
\]

\[
\begin{align*}
\text{evaluator} & : (\text{any/c } . \rightarrow . \text{ any}) \\
\text{secs} & : (\text{or/c exact-nonnegative-integer? } \#f) \\
\text{mb} & : (\text{or/c exact-nonnegative-integer? } \#f)
\end{align*}
\]

Changes the per-expression limits that \text{evaluator} uses to \text{secs} seconds of shallow time and \text{mb} megabytes (either one can be \#f, indicating no limit).

This procedure should be used to modify an existing evaluator limits, because changing the \text{sandbox-eval-limits} parameter does not affect existing evaluators. See also \text{call-with-limits}.

\[
(set-eval-handler \text{ evaluator } \text{ handler}) \rightarrow \text{ void?}
\]

\[
\begin{align*}
\text{evaluator} & : (\text{any/c } . \rightarrow . \text{ any}) \\
\text{handler} & : (\text{or/c } \#f \ (\rightarrow \text{ any}) . \rightarrow . \text{ any})
\end{align*}
\]

Changes the per-expression handler that the \text{evaluator} uses around each interaction. A \#f value means no handler is used.

This procedure should be used to modify an existing evaluator handler, because changing the \text{sandbox-eval-handlers} parameter does not affect existing evaluators. See also \text{call-with-custodian-shutdown} and \text{call-with-killing-threads} for two useful handlers that are provided.

\[
\begin{align*}
\text{call-with-custodian-shutdown} \text{ thunk} \rightarrow \text{ any} \\
\text{thunk} & : (\rightarrow \text{ any})
\end{align*}
\]

\[
\begin{align*}
\text{call-with-killing-threads} \text{ thunk} \rightarrow \text{ any} \\
\text{thunk} & : (\rightarrow \text{ any})
\end{align*}
\]

These functions are useful for use as an evaluation handler. \text{call-with-custodian-shutdown} will execute the \text{thunk} in a fresh custodian, then shutdown that custodian, making sure that \text{thunk} could not have left behind any resources. \text{call-with-killing-threads} is similar, except that it kills threads that were left, but leaves other resources as is.

\[
\begin{align*}
\text{put-input} \text{ evaluator} \rightarrow \text{ output-port?} \\
\text{evaluator} & : (\text{any/c } . \rightarrow . \text{ any})
\end{align*}
\]

\[
\begin{align*}
\text{put-input} \text{ evaluator } \text{i/o} \rightarrow \text{ void?} \\
\text{evaluator} & : (\text{any/c } . \rightarrow . \text{ any}) \\
\text{i/o} & : (\text{or/c bytes? string? } \text{eof-object?})
\end{align*}
\]

If \text{(sandbox-input)} is 'pipe when an evaluator is created, then this procedure can be used to retrieve the output port end of the pipe (when used with no arguments), or to add a string or a byte string into the pipe. It can also be used with \text{eof}, which closes the pipe.
(get-output evaluator) → (or/c #f input-port? bytes? string?)
  evaluator : (any/c . -> . any)
(get-error-output evaluator)
  → (or/c #f input-port? bytes? string?)
  evaluator : (any/c . -> . any)

Returns the output or error-output of the evaluator, in a way that depends on the setting of (sandbox-output) or (sandbox-error-output) when the evaluator was created:

• if it was 'pipe, then get-output returns the input port end of the created pipe;
• if it was 'bytes or 'string, then the result is the accumulated output, and the output port is reset so each call returns a different piece of the evaluator’s output (note that results are available only until the evaluator has terminated, and any allocations of the output are subject to the sandbox memory limit);
• otherwise, it returns #f.

(get-uncovered-expressions evaluator [prog? src]) → (listof syntax?)
  evaluator : (any/c . -> . any)
  prog? : any/c = #t
  src : any/c = default-src

Retrieves uncovered expression from an evaluator, as longs as the sandbox-coverage-enabled parameter had a true value when the evaluator was created. Otherwise, an exception is raised to indicate that no coverage information is available.

The prog? argument specifies whether to obtain expressions that were uncovered after only the original input program was evaluated (#t) or after all later uses of the evaluator (#f). Using #t retrieves a list that is saved after the input program is evaluated, and before the evaluator is used, so the result is always the same.

A #t value of prog? is useful for testing student programs to find out whether a submission has sufficient test coverage built in. A #f value is useful for writing test suites for a program to ensure that your tests cover the whole code.

The second optional argument, src, specifies that the result should be filtered to hold only syntax objects whose source matches src. The default is the source that was used in the program code, if there was one. Note that 'program is used as the source value if the input program was given as S-expressions or as a string (and in these cases it will be the default for filtering). If given #f, the result is the unfiltered list of expressions.

The resulting list of syntax objects has at most one expression for each position and span. Thus, the contents may be unreliable, but the position information is reliable (i.e., it always
indicates source code that would be painted red in DrRacket when coverage information is used).

Note that if the input program is a sequence of syntax values, either make sure that they have \texttt{program} as the source field, or use the \texttt{src} argument. Using a sequence of S-expressions (not syntax objects) for an input program leads to unreliable coverage results, since each expression may be assigned a single source location.

\[
\text{(call-in-sandbox-context \textit{evaluator} \textit{thunk} \{unrestricted\}) \rightarrow \text{any}}
\]

\textit{evaluator}: (any/c . -\rightarrow . any)
\textit{thunk}: (\rightarrow \text{any})
\text{unrestricted?}: \text{boolean?} = \#f

Calls the given \textit{thunk} in the context of a sandboxed evaluator. The call is performed under the resource limits and evaluation handler that are used for evaluating expressions, unless \text{unrestricted?} is specified as true.

This process is usually similar to \text{(evaluator (list thunk))}, except that it does not rely on the common meaning of a sexpr-based syntax with list expressions as function application (which is not true in all languages). Note that this is more useful for meta-level operations such as namespace manipulation, it is not intended to be used as a safe-evaluation replacement (i.e., using the sandbox evaluator as usual).

In addition, you can avoid some of the sandboxed restrictions by using your own permissions, for example,

\[
\begin{align*}
\text{(let ([guard \textit{current-security-guard})]}
\text{(call-in-sandbox-context ev lambda () (parameterize ([\textit{current-security-guard guard}} \; \text{can access anything you want here}
\text{(delete-file "/some/file"))})
\end{align*}
\]

\section*{14.12.4 Miscellaneous}

\textit{gui?}: \text{boolean?}

For backward compatibility, only: the result of \texttt{gui-available?} at the time that \texttt{racket/sandbox} was instantiated.

The value of \textit{gui?} is no longer used by \texttt{racket/sandbox} itself. Instead, \texttt{gui-available?} and \texttt{sandbox-gui-available} are checked at the time that a sandbox evaluator is created.
(call-with-limits secs mb thunk) → any
  secs : (or/c exact-nonnegative-integer? #f)
  mb : (or/c exact-nonnegative-integer? #f)
  thunk : (-> any)

Executes the given thunk with memory and time restrictions: if execution consumes more than mb megabytes or more than secs shallow time seconds, then the computation is aborted and an exception recognizable by exn:fail:resource? is raised. Otherwise, the result of the thunk is returned as usual (a value, multiple values, or an exception). Each of the two limits can be #f to indicate the absence of a limit. See also custodian-limit-memory for information on memory limits.

Sandboxed evaluators use call-with-limits, according to the sandbox-eval-limits setting and uses of set-eval-limits: each expression evaluation is protected from time-outs and memory problems. Use call-with-limits directly only to limit a whole testing session, instead of each expression.

(with-limits sec-expr mb-expr body ...)

A macro version of call-with-limits.

(call-with-deep-time-limit secs thunk) → any
  secs : exact-nonnegative-integer?
  thunk : (-> any)

Executes the given thunk with deep time restrictions, and returns the values produced by thunk.

The given thunk is run in a new thread. If it errors or if the thread terminates returning a value, then (values) is returned.

Changed in version 1.1 of package sandbox-lib: Changed to return thunk’s result if it completes normally.

(with-deep-time-limit secs-expr body ...)

A macro version of call-with-deep-time-limit.

(exn:fail:resource? v) → boolean?
  v : any/c
(exn:fail:resource-resource exn)
→ (or/c 'time 'memory 'deep-time)
  exn : exn:fail:resource?

A predicate and accessor for exceptions that are raised by call-with-limits. The resource field holds a symbol, representing the resource that was expended. ’time is used for shallow time and ’deep-time is used for deep time.
14.13 The racket/repl Library

```scheme
(require racket/repl)  package: base
```

The `racket/repl` provides the same `read-eval-print-loop` binding as `racket/base`, but with even fewer internal dependencies than `racket/base`. It is loaded in some situations on startup, as described in §18.1.1 “Initialization”.

14.14 Linklets and the Core Compiler

```scheme
(require racket/linklet)  package: base
```

A linklet is a primitive element of compilation, bytecode marshaling, and evaluation. Racket’s implementations of modules, macros, and top-level evaluation are all built on linklets. Racket programmers generally do not encounter linklets directly, but the `racket/linklet` library provides access to linklet facilities.

A single Racket module (or collection of top-level forms) is typically implemented by multiple linklets. For example, each phase of evaluation that exists in a module is implemented in a separate linklet. A linklet is also used for metadata such as the module path indexes for a module’s `require`. These linklets, plus some other metadata, are combined to form a linklet bundle. Information in a linklet bundle is keyed by either a symbol or a fixnum. A linklet bundle containing linklets can be marshaled to and from a byte stream by `write` and (with `read-accept-compiled` is enabled) `read`.

When a Racket module has submodules, the linklet bundles for the module and the submodules are grouped together in a linklet directory. A linklet directory can have nested linklet directories. Information in a linklet directory is keyed by `#f` or a symbol, where `#f` must be mapped to a linklet bundle (if anything) and each symbol must be mapped to a linklet directory. A linklet directory can be equivalently viewed as a mapping from a lists of symbols to a linklet bundle. Like linklet bundles, a linklet directory can be marshaled to and from a byte stream by `write` and `read`; the marshaled form allows individual linklet bundles to be loaded independently.

A linklet consists of a set of variable definitions and expressions, an exported subset of the defined variable names, a set of variables to export from the linklet despite having no corresponding definition, and a set of imports that provide other variables for the linklet to use. To run a linklet, it is instantiated as as linklet instance (or just instance, for short). When a linklet is instantiated, it receives other linklet instances for its imports, and it extracts a specified set of variables that are exported from each of the given instances. The newly created linklet instance provides its exported variables for use by other linklets or for direct access via `instance-variable-value`. A linklet instance can be synthesized directly with `make-instance`.

A linklet is created by compiling an enriched S-expression representation of its source. Since
linklets exist below the layer of macros and syntax objects, linklet compilation does not use syntax objects. Instead, linklet compilation uses correlated objects, which are like syntax objects without lexical-context information and without the constraint that content is coerced to correlated objects. Using an S-expression or correlated object, the grammar of a linklet as recognized by `compile-linklet` is

```
(linklet [[imported-id/renamed ...] ...]
  [exported-id/renamed ...]
  defn-or-expr ...)

imported-id/renamed = imported-id
| (external-imported-id internal-imported-id)

exported-id/renamed = exported-id
| (internal-exported-id external-exported-id)
```

Each import set [imported-id/renamed ...] refers to a single imported instance, and each imported-id/renamed corresponds to a variable from that instance. If separate external-imported-id and internal-imported-id are specified, then external-imported-id is the name of the variable as exported by the instance, and internal-imported-id is the name used to refer to the variable in the defn-or-exprs. For exports, separate internal-exported-id and external-exported-id names correspond to the variable name as exported as referenced in the defn-or-exprs, respectively.

The grammar of an defn-or-expr is similar to the expander's grammar of fully expanded expressions (see §1.2.3.1 "Fully Expanded Programs") with some exceptions: quote-syntax and #%top are not allowed; #%plain-lambda is spelled lambda; #%plain-app is omitted (i.e., application is implicit); lambda, case-lambda, let-values, and letrec-values can have only a single body expression; and numbers, booleans, strings, and byte strings are self-quoting. Primitives are accessed directly by name, and shadowing is not allowed within a linklet form for primitive names (see linklet-body-reserved-symbol?), imported variables, defined variables, or local variables.

When an exported-id/renamed has no corresponding definition among the defn-or-exprs, then the variable is effectively defined as uninitialized; referencing the variable will trigger exn:fail:contract:variable, the same as referencing a variable before it is defined. When a target instance is provided to instantiate-linklet, any existing variable with the same name will be left as-is, instead of set to undefined. This treatment of uninitialized variables provides core support for top-level evaluation where variables may be referenced and then defined in a separate element of compilation.

Added in version 6.90.0.1 of package base.

```
(linklet? v) → boolean?
  v : any/c
```
Returns #t if v is a linklet, #f otherwise.

\[
(\text{compile-linklet} \quad \text{form} \\
\quad \text{name}\ \\
\quad \text{import-keys}\ \\
\quad \text{get-import}\ \\
\quad \text{options}) \rightarrow \text{linklet?}
\]

\[
\text{form} : (\text{or/c correlated? any/c}) \\
\text{name} : \text{any/c} = \#f \\
\text{import-keys} : \#f = \#f \\
\text{get-import} : \#f = \#f \\
\text{options} : (\text{listof (or/c 'serializable 'unsafe 'static 'quick 'use-prompt 'uninterned-literal)})
\]

\[
= '(\text{serializable})
\]

\[
(\text{compile-linklet} \quad \text{form} \\
\quad \text{name} \\
\quad \text{import-keys} \quad \text{get-import} \\
\quad \text{options}) \rightarrow \text{linklet? vector?}
\]

\[
\text{form} : (\text{or/c correlated? any/c}) \\
\text{name} : \text{any/c} \\
\text{import-keys} : \text{vector?} \\
\text{get-import} : (\text{or/c #f (any/c . -> . (values (or/c linklet? instance? #f) (or/c vector? #f)))))
\]

\[
= \#f \\
\text{options} : (\text{listof (or/c 'serializable 'unsafe 'static 'quick 'use-prompt 'uninterned-literal)})
\]

\[
= '(\text{serializable})
\]

Takes an S-expression or correlated object for a linklet form and produces a linklet. As long as ‘serializable included in options, the resulting linklet can be marshaled to and from a byte stream when it is part of a linklet bundle (possibly in a linklet directory).

The optional name is associated to the linklet for debugging purposes and as the default name of the linklet’s instance.

The optional import-keys and get-import arguments support cross-linklet optimization. If import-keys is a vector, it must have as many elements as sets of imports in form. If the compiler becomes interested in optimizing a reference to an imported variable, it passes back to get-import (if non-#f) the element of import-keys that corresponds to the variable’s import set. The get-import function can then return a linklet or instance that represents an instance to be provided to the compiled linklet when it is eventually instantiated; ensuring consistency between reported linklet or instance and the eventual instance is up to the caller of compile-linklet. If get-import returns #f as its first value, the compiler will be prevented from making any assumptions about the imported instance. The second result from get-import is an optional vector of keys to provide transitive information on a returned
linklet’s imports (and is not allowed for a returned instance); the returned vector must have the same number of elements as the linklet has imports. When vector elements are eq? and non-#f, the compiler can assume that they correspond to the same run-time instance. A #f value for get-import is equivalent to a function that always returns two #f results.

When import-keys is not #f, then the compiler is allowed to grow or shrink the set of imported instances for the linklet. The result vector specifies the keys of the imports for the returned linklet. Any key that is #f or a linklet instance must be preserved intact, however.

If 'unsafe is included in options, then the linklet is compiled in unsafe mode: uses of safe operations within the linklet can be converted to unsafe operations on the assumption that the relevant contracts are satisfied. For example, car is converted to unsafe-car. Some substituted unsafe operations may not have directly accessible names, such as the unsafe variant of in-list that can be substituted in unsafe mode. An unsafe operation is substituted only if its (unchecked) contract is subsumed by the safe operation’s contract. The fact that the linklet is compiled in unsafe mode can be exposed through variable-reference-from-unsafe? using a variable reference produced by a #%variable-reference form within the module body.

If 'static is included in options, then the linklet must be instantiated only once; if the linklet is serialized, then any individual instance read from the serialized form must be instantiated at most once. Compilation with 'static is intended to improve the performance of references within the linklet to defined and imported variables.

If 'quick is included in options, then linklet compilation may trade run-time performance for compile-time performance—that is, spend less time compiling the linklet, but the resulting linklet may run more slowly.

If 'use-prompt is included in options, then instantiating resulting linklet always wraps a prompt around each definition and immediate expression in the linklet. Otherwise, supplying #t as the use-prompt? argument to instantiate-linklet may only wrap a prompt around the entire instantiation.

If 'uninterned-literal is included in options, then literals in form will not necessarily be interned via datum-intern-literal when compiling or loading the linklet. Disabling the use of datum-intern-literal can be especially useful of the linklet includes a large string or byte string constant that is not meant to be shared.

The symbols in options must be distinct, otherwise exn:fail:contract exception is raised.

Changed in version 7.1.0.8 of package base: Added the 'use-prompt option.
Changed in version 7.1.0.10: Added the 'uninterned-literal option.
Changed in version 7.5.0.14: Added the 'quick option.
Like `compile-linklet`, but takes an already-compiled linklet and potentially optimizes it further.

Changed in version 7.1.0.6 of package `base`: Added the `options` argument.

Changed in version 7.1.0.8: Added the `use-prompt` option.

Changed in version 7.1.0.10: Added the `uninterned-literal` option.

Changed in version 7.5.0.14: Added the `quick` option.

Returns a variant of a `linklet` that is prepared for JIT compilation such that every later use of the result linklet with `instantiate-linklet` shares the JIT-generated code. However, the result of `eval-linklet` cannot be marshaled to a byte stream as part of a linklet bundle, and it cannot be used with `recompile-linklet`.
```
(instantiate-linklet linklet
    import-instances
    [target-instance? boolean]
    [use-prompt? boolean]) → instance?

linklet : linklet?
import-instances : (listof instance?)
target-instance? : #f = #f
use-prompt? : any/c = #t

(instantiate-linklet linklet
    import-instances
    target-instance
    [use-prompt?]) → any

linklet : linklet?
import-instances : (listof instance?)
target-instance : instance?
use-prompt? : any/c = #t
```

Instantiates linklet by running its definitions and expressions, using the given import-instances for its imports. The number of instances in import-instances must match the number of import sets in linklet.

If target-instance is #f or not provided, the result is a fresh instance for the linklet. If target-instance is an instance, then the instance is used and modified for the linklet definitions and expressions, and the result is the value of the last expression in the linklet.

The linklet’s exported variables are accessible in the result instance or in target-instance using the linklet’s external name for each export. If target-instance is provided as non-#f, its existing variables remain intact if they are not modified by a linklet definition.

If use-prompt? is true, then a prompt is wrapped around the linklet instantiation in same ways as an expression in a module body. If the linklet contains multiple definitions or immediate expressions, then a prompt may or may not be wrapped around each definition or expression; supply 'use-prompt to compile-linklet to ensure that a prompt is used around each definition and expression.

```
(linklet-import-variables linklet) → (listof (listof symbol?))
linklet : linklet?
```

Returns a description of a linklet’s imports. Each element of the result list corresponds to an import set as satisfied by a single instance on instantiation, and each member of the set is a variable name that is used from the corresponding imported instance.

```
(linklet-export-variables linklet) → (listof symbol?)
linklet : linklet?
```

Returns a description of a linklet’s exports. Each element of the list corresponds to a variable that is made available by the linklet in its instance.
(linklet-directory? v) → boolean?

v : any/c

Returns #t if v is a linklet directory, #f otherwise.

(hash->linklet-directory content) → linklet-directory?

content : (and/c hash? hash-eq? immutable? (not/c impersonator?))

Constructs a linklet directory given mappings in the form of a hash table. Each key of content must be either a symbol or #f, each symbol must be mapped to a linklet directory, and #f must be mapped to a linklet bundle or not mapped.

(linklet-directory->hash linklet-directory)

→ (and/c hash? hash-eq? immutable? (not/c impersonator?))

linklet-directory : linklet-directory?

Extracts the content of a linklet directory into a hash table.

(linklet-bundle? v) → boolean?

v : any/c

Returns #t if v is a linklet bundle, #f otherwise.

(hash->linklet-bundle content) → linklet-bundle?

content : (and/c hash? hash-eq? immutable? (not/c impersonator?))

Constructs a linklet bundle given mappings in the form of a hash table. Each key of content must be either a symbol or a fixnum. Values in the hash table are unconstrained, but the intent is that they are all linklets or values that can be recovered from write output by read.

(linklet-bundle->hash linklet-bundle)

→ (and/c hash? hash-eq? immutable? (not/c impersonator?))

linklet-bundle : linklet-bundle?

Extracts the content of a linklet bundle into a hash table.

(linklet-body-reserved-symbol? sym) → boolean?

sym : symbol?

Return #t if sym is a primitive name or other identifier that is not allowed as a binding within a linklet, #f otherwise.

Added in version 8.2.0.1 of package base.
(instance? v) → boolean?
  v : any/c

Returns #t if v is a linklet instance, #f otherwise.

(make-instance name
  [data
    mode]
  variable-name
  variable-value ...
  ...) → instance?
  name : any/c
  data : any/c = #f
  mode : (or/c #f 'constant 'consistent) = #f
  variable-name : symbol?
  variable-value : any/c

Constructs a linklet instance directly. Besides associating an arbitrary name and data value to the instance, the instance is populated with variables as specified by variable-name and variable-value.

The optional data and mode arguments must be provided if any variable-name and variable-value arguments are provided. The mode argument is used as in instance-set-variable-value! for every variable-name.

(instance-name instance) → any/c
  instance : instance?

Returns the value associated to instance as its name—either the first value provided to make-instance or the name of a linklet that was instantiated to create the instance.

(instance-data instance) → any/c
  instance : instance?

Returns the value associated to instance as its data—either the second value provided to make-instance or the default #f.

(instance-variable-names instance) → (list symbol?)
  instance : instance?

Returns a list of all names for all variables accessible from instance.

(instance-variable-value instance name
  [fail-k]) → any

1115
instance : instance?
name : symbol?
fail-k : any/c = (lambda () (error ....))

Returns the value of the variable exported as name from instance. If no such variable is exported, then fail-k is used in the same way as by hash-ref.

(instance-set-variable-value! instance name v [mode]) → void?

Sets or creates the variable exported as name in instance so that its value is v, as long as the variable does not exist already as constant. If a variable for name exists as constant, the exn:fail:contract exception is raised.

If mode is 'constant or 'consistent, then the variable is created or changed to be constant. Furthermore, when the instance is reported for a linklet’s import though a get-import callback to compile-linklet, the compiler can assume that the variable will be constant in all future instances that are used to satisfy a linklet’s imports.

If mode is 'consistent, when the instance is reported though a callback to compile-linklet, the compiler can further assume that the variable’s value will be the same for future instances. For compilation purposes, “the same” can mean that a procedure value will have the same arity and implementation details, a structure type value will have the same configuration, a marshalable constant will be equal? to the current value, and so on.

(instance-unset-variable! instance name) → void?

Changes instance so that it does not export a variable as name, as long as name does not exist as a constant variable. If a variable for name exists as constant, the exn:fail:contract exception is raised.

(instance-describe-variable! instance name desc-v) → void?

1116
Registers information about `name` in `instance` that may be useful for compiling linklets where the instance is return via the `get-import` callback to `compile-linklet`. The `desc-\v` description can be any value; the recognized descriptions depend on virtual machine, but may include the following:

- `'(procedure ,arity-mask)` — the value is always a procedure that is not impersonated and not a structure, and its arity in the style of `procedure-arity-mask` is `arity-mask`.
- `'(procedure/succeeds ,arity-mask)` — like `'(procedure ,arity-mask)`, but for a procedure that never raises an exception of otherwise captures or escapes the calling context.
- `'(procedure/pure ,arity-mask)` — like `'(procedure/succeeds ,arity-mask)`, but with no observable side effects, so a call to the procedure can be reordered.

Added in version 7.1.0.8 of package base.

```
(variable-reference->instance varref [ref-site?])
  \rightarrow (if ref-site? (or/c instance? #f symbol?) instance?)
  varref    : variable-reference?
  ref-site? : any/c = #f
```

Extracts the instance where the variable of `varref` is defined if `ref-site?` is `#f`, and returns the instance where `varref` itself resides if `ref-site?` is true. This notion of variable reference is the same as at the module level and can reflect the linklet instance that implements a particular phase of a module instance.

When `ref-site?` is `#f`, the result is `#f` when `varref` is from `(#%variable-reference)` with no identifier. The result is a symbol if `varref` refers to a primitive.

```
(correlated? v) \rightarrow boolean?
  v    : any/c
(correlated-source crlt) \rightarrow any
  crlt : correlated?
(correlated-line crlt) \rightarrow (or/c exact-positive-integer? #f)
  crlt : correlated?
(correlated-column crlt) \rightarrow (or/c exact-nonnegative-integer? #f)
  crlt : correlated?
(correlated-position crlt) \rightarrow (or/c exact-positive-integer? #f)
  crlt : correlated?
(correlated-span crlt) \rightarrow (or/c exact-nonnegative-integer? #f)
  crlt : correlated?
(correlated-e crlt) \rightarrow any
  crlt : correlated?
```
(correlated->datum crlt)  ➞  any
  crlt : (or/c correlated? any/c)
(datum->correlated v [srcloc prop])  ➞  correlated?
  v : any/c
  srcloc : (or/c correlated? #f)
    (list/c any/c
      (or/c exact-positive-integer? #f)
      (or/c exact-nonnegative-integer? #f)
      (or/c exact-positive-integer? #f)
      (or/c exact-nonnegative-integer? #f))
    (vector/c any/c
      (or/c exact-positive-integer? #f)
      (or/c exact-nonnegative-integer? #f)
      (or/c exact-positive-integer? #f)
      (or/c exact-nonnegative-integer? #f)))
  = #f
  prop : (or/c correlated? #f) = #f
(correlated-property crlt key val)  ➞  correlated?
  crlt : correlated?
  key : any/c
  val : any/c
(correlated-property crlt key)  ➞  any/c
  crlt : correlated?
  key : any/c
(correlated-property-symbol-keys crlt)  ➞  list?
  crlt : correlated?


Unlike datum->syntax, datum->correlated does not recur through the given S-expression and convert pieces to correlated objects. Instead, a correlated object is simply wrapped around the immediate value. In contrast, correlated->datum recurs through its argument (which is not necessarily a correlated object) to discover any correlated objects and convert them to plain S-expressions.

Changed in version 7.6.0.6 of package base: Added the prop argument to datum->correlated.
15 Operating System

15.1 Paths

When a Racket procedure takes a filesystem path as an argument, the path can be provided either as a string or as an instance of the `path` datatype. If a string is provided, it is converted to a path using `string->path`. Beware that some paths may not be representable as strings; see §15.1.3.1 “Unix Path Representation” and §15.1.4.1 “Windows Path Representation” for more information. A Racket procedure that generates a filesystem path always generates a path value.

By default, paths are created and manipulated for the current platform, but procedures that merely manipulate paths (without using the filesystem) can manipulate paths using conventions for other supported platforms. The `bytes->path` procedure accepts an optional argument that indicates the platform for the path, either `unix` or `windows`. For other functions, such as `build-path` or `simplify-path`, the behavior is sensitive to the kind of path that is supplied. Unless otherwise specified, a procedure that requires a path accepts only paths for the current platform.

Two path values are `equal?` when they are use the same convention type and when their byte-string representations are `equal?`. A path string (or byte string) cannot be empty, and it cannot contain a nul character or byte. When an empty string or a string containing nul is provided as a path to any procedure except `absolute-path?`, `relative-path?`, or `complete-path?`, the `exn:fail:contract` exception is raised.

Most Racket primitives that accept paths first `cleanse` the path before using it. Procedures that build paths or merely check the form of a path do not cleanse paths, with the exceptions of `cleanse-path`, `expand-user-path`, and `simplify-path`. For more information about path cleansing and other platform-specific details, see §15.1.3 “Unix and Mac OS Paths” and §15.1.4 “Windows Paths”.

15.1.1 Manipulating Paths

\[ \text{(path? } v \text{)} \rightarrow \text{boolean?} \]
\[ v : \text{any/c} \]

Returns `#t` if `v` is a path value for the current platform (not a string, and not a path for a different platform), `#f` otherwise.

\[ \text{(path-string? } v \text{)} \rightarrow \text{boolean?} \]
\[ v : \text{any/c} \]

Returns `#t` if `v` is either a path or string: either a path for the current platform or a non-empty string without nul characters. Returns `#f` otherwise.
(path-for-some-system? v) → boolean?
  v : any/c

Returns #t if v is a path value for some platform (not a string), #f otherwise.

(string->path str) → path?
  str : string?

-produces a path whose byte-string encoding is (string->bytes/locale str (char->integer #\?)) on Unix and Mac OS or (string->bytes/utf-8 str) on Windows.

-Beware that the current locale might not encode every string, in which case string->path can produce the same path for different strs. See also string->path-element, which should be used instead of string->path when a string represents a single path element.

-For information on how strings and byte strings encode paths, see §15.1.3.1 "Unix Path Representation" and §15.1.4.1 "Windows Path Representation"

See also string->some-system-path, and see §15.1.3.1 "Unix Path Representation" and §15.1.4.1 "Windows Path Representation" for information on how strings encode paths.

-Changed in version 6.1.1.1 of package base: Changed Windows conversion to always use UTF-8.

(bytes->path bstr [type]) → path?
  bstr : bytes?
  type : (or/c 'unix 'windows) = (system-path-convention-type)

-produces a path (for some platform) whose byte-string encoding is bstr, where bstr must not contain a nul byte. The optional type specifies the convention to use for the path.

-For converting relative path elements from literals, use instead bytes->path-element, which applies a suitable encoding for individual elements.

-For information on how byte strings encode paths, see §15.1.3.1 "Unix Path Representation" and §15.1.4.1 "Windows Path Representation"

(path->string path) → string?
  path : path?

-produces a string that represents path by decoding path’s byte-string encoding using the current locale on Unix and Mac OS and by using UTF-8 on Windows. In the former case, ? is used in the result string where encoding fails, and if the encoding result is the empty string, then the result is "?".

-The resulting string is suitable for displaying to a user, string-ordering comparisons, etc., but it is not suitable for re-creating a path (possibly modified) via string->path, since decoding and re-encoding the path’s byte string may lose information.
Furthermore, for display and sorting based on individual path elements (such as pathless file names), use `path-element->string`, instead, to avoid special encodings use to represent some relative paths. See §15.1.4 “Windows Paths” for specific information about the conversion of Windows paths.

See also `some-system-path->string`.

Changed in version 6.1.1.1 of package `base`: Changed Windows conversion to always use UTF-8.

```
(path->bytes path) → bytes?
  path : path-for-some-system?
```

Produces `path`’s byte-string representation. No information is lost in this translation, so that `(bytes->path (path->bytes path) (path-convention-type path))` always produces a path that is `equal?` to `path`. The `path` argument can be a path for any platform.

Conversion to and from byte values is useful for marshaling and unmarshaling paths, but manipulating the byte form of a path is generally a mistake. In particular, the byte string may start with a `\?\REL` encoding for Windows paths. Instead of `path->bytes`, use `split-path` and `path-element->bytes` to manipulate individual path elements.

For information on how byte strings encode paths, see §15.1.3.1 “Unix Path Representation” and §15.1.4.1 “Windows Path Representation”.

```
(string->path-element str) → path?
  str : string?
```

Like `string->path`, except that `str` corresponds to a single relative element in a path, and it is encoded as necessary to convert it to a path. See §15.1.3 “Unix and Mac OS Paths” and §15.1.4 “Windows Paths” for more information on the conversion of paths.

If `str` does not correspond to any path element (e.g., it is an absolute path, or it can be split), or if it corresponds to an up-directory or same-directory indicator on Unix and Mac OS, then `exn:fail:contract` exception is raised.

Like `path->string`, information can be lost from `str` in the locale-specific conversion to a path.

```
(bytes->path-element bstr [type]) → path-for-some-system?
  bstr : bytes?
  type : (or/c 'unix 'windows) = (system-path-convention-type)
```

Like `bytes->path`, except that `bstr` corresponds to a single relative element in a path. In terms of conversions and restrictions on `bstr`, `bytes->path-element` is like `string->path-element`. 
The `bytes->path-element` procedure is generally the best choice for reconstructing a path based on another path (where the other path is deconstructed with `split-path` and `path-element->bytes`) when ASCII-level manipulation of path elements is necessary.

```scheme
(path-element->string path) → string?
  path : path-element?
```

Like `path->string`, except that trailing path separators are removed (as by `split-path`). On Windows, any `\??\REL` encoding prefix is also removed; see §15.1.4 "Windows Paths" for more information.

The `path` argument must be such that `split-path` applied to `path` would return `relative` as its first result and a path as its second result, otherwise the `exn:fail:contract` exception is raised.

The `path-element->string` procedure is generally the best choice for presenting a pathless file or directory name to a user.

```scheme
(path-element->bytes path) → bytes?
  path : path-element?
```

Like `path->bytes`, except that any encoding prefix is removed, etc., as for `path-element->string`.

For any reasonable locale, consecutive ASCII characters in the printed form of `path` are mapped to consecutive byte values that match each character’s code-point value, and a leading or trailing ASCII character is mapped to a leading or trailing byte, respectively. The `path` argument can be a path for any platform.

The `path-element->bytes` procedure is generally the right choice (in combination with `split-path`) for extracting the content of a path to manipulate it at the ASCII level (then reassembling the result with `bytes->path-element` and `build-path`).

```scheme
(path<? a-path b-path ...) → boolean?
  a-path : path?
  b-path : path?
```

Returns `#t` if the arguments are sorted, where the comparison for each pair of paths is the same as using `path->bytes` and `bytes<?`.

Changed in version 7.0.0.13 of package `base`: Allow one argument, in addition to allowing two or more.

```scheme
(path-convention-type path) → (or/c 'unix 'windows)
  path : path-for-some-system?
```

Accepts a path value (not a string) and returns its convention type.
\( (\text{system-path-convention-type}) \rightarrow (\text{o}/\text{c} \ '\text{unix} \ '\text{windows}) \)

Returns the path convention type of the current platform: 'unix for Unix and Mac OS, 'windows for Windows.

\[
(\text{build-path} \ base \ sub \ \ldots) \rightarrow \text{path-for-some-system}
\]

- \( base : (\text{o}/\text{c} \ \text{path-string}? \ \text{path-for-some-system}? \ 'up \ 'same) \)
- \( sub : (\text{o}/\text{c} \ (\text{and}/\text{c} \ (\text{o}/\text{c} \ \text{path-string}? \ \text{path-for-some-system}?) \ (\text{not}/\text{c} \ \text{complete-path}?) \)) \)
  - \( (\text{o}/\text{c} \ 'up \ 'same) \)

Creates a path given a base path and any number of sub-path extensions. If \( base \) is an absolute path, the result is an absolute path, otherwise the result is a relative path.

The \( base \) and each \( sub \) must be either a relative path, the symbol 'up (indicating the relative parent directory), or the symbol 'same (indicating the relative current directory). For Windows paths, if \( base \) is a drive specification (with or without a trailing slash) the first \( sub \) can be an absolute (driveless) path. For all platforms, the last \( sub \) can be a filename.

The \( base \) and \( sub \) arguments can be paths for any platform. The platform for the resulting path is inferred from the \( base \) and \( sub \) arguments, where string arguments imply a path for the current platform. If different arguments are for different platforms, the \text{exn:fail:contract} exception is raised. If no argument implies a platform (i.e., all are 'up or 'same), the generated path is for the current platform.

Each \( sub \) and \( base \) can optionally end in a directory separator. If the last \( sub \) ends in a separator, it is included in the resulting path.

If \( base \) or \( sub \) is an illegal path string (because it is empty or contains a nul character), the \text{exn:fail:contract} exception is raised.

The \text{build-path} procedure builds a path \textit{without} checking the validity of the path or accessing the filesystem.

See §15.1.3 "Unix and Mac OS Paths" and §15.1.4 "Windows Paths" for more information on the construction of paths.

The following examples assume that the current directory is "/home/joeuser" for Unix examples and "C:\Joe's Files" for Windows examples.

\[
(\text{define} \ p1 \ (\text{build-path} \ (\text{current-directory}) \ "src" \ "racket"))
\]

; Unix: \( p1 \) is "/home/joeuser/src/racket"
; Windows: \( p1 \) is "C:\Joe's Files\src\racket"
\[
(\text{define} \ p2 \ (\text{build-path} \ 'up \ 'up \ "docs" \ "Racket"))
\]

; Unix: \( p2 \) is ".../../docs/Racket"
; Windows: \( p2 \) is ".\\\\\..\docs\Racket"

1123
(build-path p2 p1)
; Unix and Windows: raises exn:fail:contract; p1 is absolute
(build-path p1 p2)
; Unix: is "/home/joeuser/src/racket/../../docs/Racket"
; Windows: is "C:\Joe's Files\src\racket\..\..\..\docs\Racket"

(build-path/convention-type type
  base                   path-for-some-system?
  sub (...)
  type : (or/c 'unix 'windows)
  base : (or/c path-string? path-for-some-system? 'up 'same)
  sub : (or/c (and/c (or/c path-string? path-for-some-system?)
               (not/c complete-path?))
         (or/c 'up 'same))

Like build-path, except a path convention type is specified explicitly.

Note that, just as with build-path, any string arguments for either base or sub will be
implicitly converted into a path for the current platform before being combined with the
others. For this reason, you cannot use this function to build paths from strings for any
platform other than the current one; in such attempts, type does not match the inferred
convention type for the strings and an exn:fail:contract exception is raised. (To create
paths for foreign platforms, see bytes->path.)

The usefulness of build-path/convention-type over build-path is limited to cases
where the sub-paths contain 'same or 'up elements.

(absolute-path? path) → boolean?
  path : (or/c path? string? path-for-some-system?)

Returns #t if path is an absolute path, #f otherwise. The path argument can be a path
for any platform. If path is not a legal path string (e.g., it contains a nul character), #f is
returned. This procedure does not access the filesystem.

(relative-path? path) → boolean?
  path : (or/c path? string? path-for-some-system?)

Returns #t if path is a relative path, #f otherwise. The path argument can be a path for any
platform. If path is not a legal path string (e.g., it contains a nul character), #f is returned.
This procedure does not access the filesystem.

(complete-path? path) → boolean?
  path : (or/c path? string? path-for-some-system?)

Returns #t if path is a completely determined path (not relative to a directory or drive),
#f otherwise. The path argument can be a path for any platform. Note that for Windows
paths, an absolute path can omit the drive specification, in which case the path is neither relative nor complete. If path is not a legal path string (e.g., it contains a nul character), #f is returned.

This procedure does not access the filesystem.

\[
\text{(path->complete-path } \text{ path [base]) } \rightarrow \text{ path-for-some-system?}
\]

\[
\text{path : (or/c path-string? path-for-some-system?)}
\]

\[
\text{base : (or/c path-string? path-for-some-system?)}
\]

\[
= \text{(current-directory)}
\]

Returns path as a complete path. If path is already a complete path, it is returned as the result. Otherwise, path is resolved with respect to the complete path base. If base is not a complete path, the exn:fail:contract exception is raised.

The path and base arguments can be paths for any platform; if they are for different platforms, the exn:fail:contract exception is raised.

This procedure does not access the filesystem.

\[
\text{(path->directory-path } \text{ path)} \rightarrow \text{ path-for-some-system?}
\]

\[
\text{path : (or/c path-string? path-for-some-system?)}
\]

Returns path if path syntactically refers to a directory and ends in a separator, otherwise it returns an extended version of path that specifies a directory and ends with a separator. For example, on Unix and Mac OS, the path "x/y/" syntactically refers to a directory and ends in a separator, but "x/y" would be extended to "x/y/", and "x/.." would be extended to "x/../". The path argument can be a path for any platform, and the result will be for the same platform.

This procedure does not access the filesystem.

\[
\text{(resolve-path } \text{ path)} \rightarrow \text{ path?}
\]

\[
\text{path : path-string?}
\]

Cleanses path and returns a path that references the same file or directory as path. If path is a soft link to another path, then the referenced path is returned (this may be a relative path with respect to the directory owning path), otherwise path is returned (after cleansing).

On Windows, the path for a link should be simplified syntactically, so that an up-directory indicator removes a preceding path element independent of whether the preceding element itself refers to a link. For relative-paths links, the path should be parsed specially; see §15.1.4 “Windows Paths” for more information.

Changed in version 6.0.1.12 of package base: Added support for links on Windows.

\[
\text{(cleanse-path } \text{ path)} \rightarrow \text{ path-for-some-system?}
\]

\[
\text{path : (or/c path-string? path-for-some-system?)}
\]
Cleanses path (as described at the beginning of this chapter) without consulting the file system.

Example:

```scheme
> (let ([p (string->some-system-path "tiny//dancer" 'unix)])
   (cleanse-path p))
#<path:tiny/dancer>
```

Example:

```scheme
(expand-user-path path) → path?
path : path-string?
```

Cleanses path. In addition, on Unix and Mac OS, a leading ~ is treated as user’s home directory and expanded; the username follows the ~ (before a / or the end of the path), where ~ by itself indicates the home directory of the current user.

```scheme
(simplify-path path [use-filesystem?]?) → path-for-some-system?
path : (or/c path-string? path-for-some-system?)
use-filesystem? : boolean? = #t
```

Eliminates redundant path separators (except for a single trailing separator), up-directory ..., and same-directory . indicators in path, and changes / separators to \ separators in Windows paths, such that the result accesses the same file or directory (if it exists) as path.

In general, the pathname is normalized as much as possible—without consulting the file system if use-filesystem? is #f, and (on Windows) without changing the case of letters within the path. If path syntactically refers to a directory, the result ends with a directory separator.

When path is simplified other than just converting slashes to backslashes and use-filesystem? is true (the default), a complete path is returned. If path is relative, it is resolved with respect to the current directory. On Unix and Mac OS, up-directory indicators are removed taking into account soft links (so that the resulting path refers to the same directory as before); on Windows, up-directory indicators are removed by deleting a preceding path element.

When use-filesystem? is #f, up-directory indicators are removed by deleting a preceding path element, and the result can be a relative path with up-directory indicators remaining at the beginning of the path; up-directory indicators are dropped when they refer to the parent of a root directory. Similarly, the result can be the same as (build-path 'same) (but with a trailing separator) if eliminating up-directory indicators leaves only same-directory indicators.

The path argument can be a path for any platform when use-filesystem? is #f, and the resulting path is for the same platform.
The filesystem might be accessed when `use-filesystem?` is true, but the source or simplified path might be a non-existent path. If `path` cannot be simplified due to a cycle of links, the `exn:fail:filesystem` exception is raised (but a successfully simplified path may still involve a cycle of links if the cycle did not inhibit the simplification).

See §15.1.3 “Unix and Mac OS Paths” and §15.1.4 “Windows Paths” for more information on simplifying paths.

Example:

```scheme
> (let ([p (string->some-system-path "tiny//in/my/head/../../../dancer" 'unix)])
  (simplify-path p #f))
#<path:tiny/dancer>

(normal-case-path path) → path-for-some-system?
  path : (or/c path-string? path-for-some-system?)
```

Returns `path` with “normalized” case characters. For Unix and Mac OS paths, this procedure always returns the input path, because filesystems for these platforms can be case-sensitive. For Windows paths, if `path` does not start with `\?\`, the resulting string uses only lowercase letters, based on the current locale. In addition, for Windows paths when the path does not start with `\?\`, all `/s` are converted to `\s`, and trailing spaces and `.s` are removed.

The `path` argument can be a path for any platform, but beware that local-sensitive decoding and conversion of the path may be different on the current platform than for the path’s platform.

This procedure does not access the filesystem.

```scheme
(split-path path) → (or/c path-for-some-system? 'relative #f)
  (or/c path-for-some-system? 'up 'same)
  boolean?
  path : (or/c path-string? path-for-some-system?)
```

Deconstructs `path` into a smaller path and an immediate directory or file name. Three values are returned:

- `base` is either
  - a path,
  - `'relative` if `path` is an immediate relative directory or filename, or
  - `#f` if `path` is a root directory.
- `name` is either
- a directory-name path,
- a filename,
- `'up` if the last part of `path` specifies the parent directory of the preceding path (e.g., `..` on Unix), or
- `'same` if the last part of `path` specifies the same directory as the preceding path (e.g., `.` on Unix).

- `must-be-dir?` is `#t` if `path` explicitly specifies a directory (e.g., with a trailing separator), `#f` otherwise. Note that `must-be-dir?` does not specify whether `name` is actually a directory or not, but whether `path` syntactically specifies a directory.

Compared to `path`, redundant separators (if any) are removed in the result `base` and `name`. If `base` is `#f`, then `name` cannot be `'up` or `'same`. The `path` argument can be a path for any platform, and resulting paths for the same platform.

This procedure does not access the filesystem.

See §15.1.3 “Unix and Mac OS Paths” and §15.1.4 “Windows Paths” for more information on splitting paths.

```scheme
(explode-path path) → (listof (or/c path-for-some-system? 'up 'same))
path : (or/c path-string? path-for-some-system?)
```

Returns the list of path elements that constitute `path`. If `path` is simplified in the sense of `simple-form-path`, then the result is always a list of paths, and the first element of the list is a root.

The `explode-path` function computes its result in time proportional to the length of `path` (unlike a loop in that uses `split-path`, which must allocate intermediate paths).

```scheme
(path-replace-extension path ext) → path-for-some-system?
path : (or/c path-string? path-for-some-system?)
```

Returns a path that is the same as `path`, except that the extension for the last element of the path (including the extension separator) is changed to `ext`. If the last element of `path` has no extension, then `ext` is added to the path.

An extension is defined as a `. ` that is not at the start of the path element followed by any number of non-`.` characters/bytes at the end of the path element, as long as the path element is not a directory indicator like `".."`.

The `path` argument can be a path for any platform, and the result is for the same platform. If `path` represents a root, the `exn:fail:contract` exception is raised. The given `ext` typically starts with `.`, but it is not required to start with an extension separator.
Examples:

```racket
> (path-replace-extension "x/y.ss" #".rkt")
<path:x/y.rkt>
> (path-replace-extension "x/y.ss" "")
<path:x/y>
> (path-replace-extension "x/y" #".rkt")
<path:x/y.rkt>
> (path-replace-extension "x/y.tar.gz" #".rkt")
<path:x/y.tar.rkt>
> (path-replace-extension "x/.racketrc" #".rkt")
<path:x/.racketrc.rkt>
```

Added in version 6.5.0.3 of package base.

```racket
(path-add-extension path ext [sep]) ➔ path-for-some-system?
path : (or/c path-string? path-for-some-system?)
ext : (or/c string? bytes?)
sep : (or/c string? bytes?) = #"_"
```

Similar to `path-replace-extension`, but any existing extension on `path` is preserved by replacing the . before the extension with `sep`, and then the `ext` is added to the end.

Examples:

```racket
> (path-add-extension "x/y.ss" #".rkt")
<path:x/y.ss.rkt>
> (path-add-extension "x/y" #".rkt")
<path:x/y.rkt>
> (path-add-extension "x/y.tar.gz" #".rkt")
<path:x/y.tar.gz.rkt>
> (path-add-extension "x/y.tar.gz" #".rkt" #")
<path:x/y.tar.gz.rkt>
> (path-add-extension "x/.racketrc" #".rkt")
<path:x/.racketrc.rkt>
```

Added in version 6.5.0.3 of package base.

Changed in version 6.8.0.2: Added the `sep` optional argument.

```racket
(path-replace-suffix path ext) ➔ path-for-some-system?
path : (or/c path-string? path-for-some-system?)
ext : (or/c string? bytes?)
```

**NOTE:** This function is deprecated; use `path-replace-extension`, instead.
Like `path-replace-extension`, but treats a leading `. in a path element as an extension separator.

```scheme
(path-add-suffix path ext) → path-for-some-system?
  path : (or/c path-string? path-for-some-system?)
  ext : (or/c string? bytes?)
```

**NOTE:** This function is deprecated; use `path-add-extension`, instead.

Like `path-add-extension`, but treats a leading `. in a path element as an extension separator.

```scheme
(reroot-path path root-path) → path-for-some-system?
  path : (or/c path-string? path-for-some-system?)
  root-path : (or/c path-string? path-for-some-system?)
```

Produces a path that extends `root-path` based on the complete form of `path`.

If `path` is not already complete, is it completed via `path->complete-path`, in which case `path` must be a path for the current platform. The `path` argument is also cleansed and case-normalized via `normal-case-path`. The path is then appended to `root-path`; in the case of Windows paths, a root letter drive becomes a letter path element, while a root UNC path is prefixed with "UNC" as a path element and the machine and volume names become path elements.

Examples:

```scheme
> (reroot-path (bytes->path #"/home/caprica/baltar" 'unix)
  (bytes->path #"/earth" 'unix))
#<path:/earth/home/caprica/baltar>
> (reroot-path (bytes->path #"c:\\usr\\adama" 'windows)
  (bytes->path #"\\\\earth\\africa\\" 'windows))
#<windows-path:\\\earth\\africa\\c\\usr\\adama>
> (reroot-path (bytes->path #"\\\\galactica\\cac\\adama" 'windows)
  (bytes->path #"s:\\\earth\\africa\\" 'windows))
#<windows-path:s:\\\earth\\africa\\UNC\\galactica\\cac\\adama>
```

### 15.1.2 More Path Utilities

```scheme
(require racket/path)
```

The bindings documented in this section are provided by the `racket/path` and `racket` libraries, but not `racket/base`.

1130
(file-name-from-path path) → (or/c path-for-some-system? #f)
path : (or/c path-string? path-for-some-system?)

Returns the last element of path. If path is syntactically a directory path (see split-path), then the result is #f.

(path-get-extension path) → (or/c bytes? #f)
path : (or/c path-string? path-for-some-system?)

Returns a byte string that is the extension part of the filename in path, including the . separator. If the path has no extension, #f is returned.

See path-replace-extension for the definition of a filename extension.

Examples:

> (path-get-extension "x/y.rkt")
".rkt"
> (path-get-extension "x/y")
#f
> (path-get-extension "x/y.tar.gz")
".gz"
> (path-get-extension "x/.racketrc")
#f

Added in version 6.5.0.3 of package base.

(path-has-extension? path ext) → (or/c bytes? #f)
path : (or/c path-string? path-for-some-system?)
ext : (or/c bytes? string?)

Determines whether the last element of path ends with ext but is not exactly the same as ext.

If ext is a byte string with the shape of an extension (i.e., starting with .), this check is equivalent to checking whether (path-get-extension path) produces ext.

Examples:

> (path-has-extension? "x/y.rkt" 
".rkt")
#t
> (path-has-extension? "x/y.ss" 
".rkt")
#f
> (path-has-extension? "x/y" 
".rkt")
#f
> (path-has-extension? "x/.racketrc" ".racketrc")
#f
> (path-has-extension? "x/compiled/y_rkt.zo" ".rkt.zo")
#t

Added in version 6.5.0.3 of package base.

(filename-extension path) → (or/c bytes? #f)
prefile : (or/c path-string? path-for-some-system?)

**NOTE:** This function is deprecated; use `path-get-extension`, instead.

Returns a byte string that is the extension part of the filename in `path` without the . separator. If `path` is syntactically a directory (see `split-path`) or if the path has no extension, #f is returned.

(find-relative-path base path [ #:more-than-root? more-than-root? #:more-than-same? more-than-same? #:normalize-case? normalize-case?])
→ (or/c path-for-some-system? path-string?)
based : (or/c path-string? path-for-some-system?)
prefile : (or/c path-string? path-for-some-system?)
more-than-root? : any/c = #f
more-than-same? : any/c = #t
normalize-case? : any/c = #t

Finds a relative pathname with respect to `base` that names the same file or directory as `prefile`. Both `base` and `prefile` must be simplified in the sense of `simple-form-path`. If `prefile` shares no subpath in common with `base`, `prefile` is returned.

If `more-than-root?` is true, if `base` and `prefile` share only a Unix root in common, and if neither `base` nor `prefile` is just a root path, then `prefile` is returned.

If `prefile` is the same as `base`, then (build-path 'same) is returned only if `more-than-same?` is true. Otherwise, `prefile` is returned when `prefile` is the same as `base`.

If `normalize-case?` is true (the default), then pairs of path elements to be compared are first converted via `normal-case-path`, which means that path elements are compared case-insentively on Windows. If `normalize-case?` is #f, then path elements and the path roots match only if they have the same case.

The result is normally a path in the sense of `path?`. The result is a string only if `prefile` is provided a string and also returned as the result.
Changed in version 6.8.0.3 of package base: Made path elements case-normalized for comparison by default, and added the #:normalize-case? argument.
Changed in version 6.90.0.21: Added the #:more-than-same? argument.

```
(normalize-path path [wrt]) → path?
path   : path-string?
(wrt)  : (and/c path-string? complete-path?)
        = (current-directory)
```

Returns a complete version of path by making the path complete, expanding the complete path, and resolving all soft links (which requires consulting the filesystem). If path is relative, then wrt is used as the base path.

Letter case is not normalized by normalize-path. For this and other reasons, such as whether the path is syntactically a directory, the result of normalize-path is not suitable for comparisons that determine whether two paths refer to the same file or directory (i.e., the comparison may produce false negatives).

An error is signaled by normalize-path if the input path contains an embedded path for a non-existent directory, or if an infinite cycle of soft links is detected.

Example:
```
> (equal? (current-directory) (normalize-path "."))
#t
```

```
(path-element? path) → boolean?
path   : any/c
```

Returns #t if path is a path element: a path value for some platform (see path-for-some-system?) such that split-path applied to path would return 'relative as its first result and a path as its second result. Otherwise, the result is #f.

```
(path-only path) → (or/c #f path-for-some-system?)
path   : (or/c path-string? path-for-some-system?)
```

Returns path without its final path element in the case that path is not syntactically a directory; if path has only a single, non-directory path element, #f is returned. If path is syntactically a directory, then path is returned unchanged (but as a path, if it was a string).

Examples:
```
> (path-only (build-path "a" "b"))
<path:a/>
> (path-only (build-path "a"))
#f
```
(simple-form-path path) → path?
path : path-string?

Returns (simplify-path (path->complete-path path)), which ensures that the result is a complete path containing no up- or same-directory indicators.

(some-system-path->string path) → string?
path : path-for-some-system?

Converts path to a string using a UTF-8 encoding of the path’s bytes.

Use this function when working with paths for a different system (whose encoding of path-names might be unrelated to the current locale’s encoding) and when starting and ending with strings.

(string->some-system-path str kind) → path-for-some-system?
str : string?
kind : (or/c 'unix 'windows)

Converts str to a kind path using a UTF-8 encoding of the path’s bytes.

Use this function when working with paths for a different system (whose encoding of path-names might be unrelated to the current locale’s encoding) and when starting and ending with strings.

(shrink-path-wrt pth other-pths) → (or/c #f path?)
pth : path?
other-pths : (listof path?)

Returns a suffix of pth that shares nothing in common with the suffixes of other-pths, or pth, if not possible (e.g. when other-pths is empty or contains only paths with the same elements as pth).

Examples:

> (shrink-path-wrt (build-path "racket" "list.rkt")
  (list (build-path "racket" "list.rkt")
        (build-path "racket" "base.rkt")))
#<path:list.rkt>
15.1.3 Unix and Mac OS Paths

In a path on Unix and Mac OS, a `/` separates elements of the path, `.` as a path element always means the directory indicated by preceding path, and `..` as a path element always means the parent of the directory indicated by the preceding path. A leading `~` in a path is not treated specially, but expand-user-path can be used to convert a leading `~` element to a user-specific directory. No other character or byte has a special meaning within a path. Multiple adjacent `/` are equivalent to a single `/` (i.e., they act as a single path separator).

A path root is always `/`. A path starting with `/` is an absolute, complete path, and a path starting with any other character is a relative path.

Any pathname that ends with a `/` syntactically refers to a directory, as does any path whose last element is `.` or `..`.

A Unix and Mac OS path is cleansed by replacing multiple adjacent `/s with a single `/`.

For `(bytes->path-element bstr)`, `bstr` must not contain any `/`, otherwise the `exn:fail:contract` exception is raised. The result of `(path-element->bytes path)` or `(path-element->string path)` is always the same as the result of `(path->bytes path)` and `(path->string path)`. Since that is not the case for other platforms, however, `path-element->bytes` and `path-element->string` should be used when converting individual path elements.

On Mac OS, Finder aliases are zero-length files.

Unix Path Representation

A path on Unix and Mac OS is natively a byte string. For presentation to users and for other string-based operations, a path is converted to/from a string using the current locale’s encoding with `?` (encoding) or `\uFFFD` (decoding) in place of errors. Beware that the encoding may not accommodate all possible paths as distinct strings.

15.1.4 Windows Paths

In general, a Windows pathname consists of an optional drive specifier and a drive-specific path. A Windows path can be `absolute` but still relative to the current drive; such paths start with a `/` or `\` separator and are not UNC paths or paths that start with `\?\`.

1135
A path that starts with a drive specification is complete. Roughly, a drive specification is either a Latin letter followed by a colon, a UNC path of the form \x\machine\volume\, or a \x?\ form followed by something other than REL\element\, or RED\element\). (Variants of \x?\ paths are described further below.)

Racket fails to implement the usual Windows path syntax in one way. Outside of Racket, a pathname "C:rant.txt" can be a drive-specific relative path. That is, it names a file "rant.txt" on drive "C:\", but the complete path to the file is determined by the current working directory for drive "C:\". Racket does not support drive-specific working directories (only a working directory across all drives, as reflected by the current-directory parameter). Consequently, Racket implicitly converts a path like "C:rant.txt" into "C:\rant.txt".

• Racket-specific: Whenever a path starts with a drive specifier \x letter\, that is not followed by a / or \, a \ is inserted as the path is cleansed.

Otherwise, Racket follows standard Windows path conventions, but also adds \x?\REL and \x?\RED conventions to deal with paths inexpressible in the standard convention, plus conventions to deal with excessive \s in \x?\ paths.

In the following, \letter\ stands for a Latin letter (case does not matter), \machine\ stands for any sequence of characters that does not include \ or / and is not ?, \volume\ stands for any sequence of characters that does not include \ or /, and \element\ stands for any sequence of characters that does not include \.

• Trailing spaces and \ in a path element are ignored when the element is the last one in the path, unless the path starts with \x?\ or the element consists of only spaces and \s.

• The following special "files", which access devices, exist in all directories, case-insensitively, and with all possible endings after a period or colon, except in path-names that start with \x?\: "NUL", "CON", "PRN", "AUX", "COM1", "COM2", "COM3", "COM4", "COM5", "COM6", "COM7", "COM8", "COM9", "LPT1", "LPT2", "LPT3", "LPT4", "LPT5", "LPT6", "LPT7", "LPT8", "LPT9".

• Except for \x?\ paths, /s are equivalent to \s. Except for \x?\ paths and the start of UNC paths, multiple adjacent /s and \s count as a single \. In a path that starts \x?\ paths, elements can be separated by either a single or double \.

• A directory can be accessed with or without a trailing separator. In the case of a non-\x?\ path, the trailing separator can be any number of /s and \s; in the case of a \x?\ path, a trailing separator must be a single \, except that two \s can follow \x?\/\letter\.

• Except for \x?\ paths, a single \ as a path element means "the current directory," and a .. as a path element means "the parent directory." Up-directory path elements (i.e., ..) immediately after a drive are ignored.
• A pathname that starts `\{machine\}\{volume\}` (where a `/` can replace any `\`) is a UNC path, and the starting `\{machine\}\{volume\}` counts as the drive specifier.

• Normally, a path element cannot contain a character in the range `\#x 0` to `\#x 1F` nor any of the following characters: 
  `<`  `>`  `:`  `"`  `/`  `\`  `

  Except for `\`, path elements containing these characters can be accessed using a `\\?\` path (assuming that the underlying filesystem allows the characters).

• In a pathname that starts `\\\{letter\}:\`, the `\\\{letter\}:\` prefix counts as the path’s drive, as long as the path does not both contain non-drive elements and end with two consecutive `\`, and as long as the path contains no sequence of three or more `\`. Two `\` can appear in place of the `\` before `{letter}`. A `/` cannot be used in place of a `\` (but `/` can be used in element names, though the result typically does not name an actual directory or file).

• In a pathname that starts `\\?\{UNC\}\{machine\}\{volume\}`, the `\\?\{UNC\}\{machine\}\{volume\}` prefix counts as the path’s drive, as long as the path does not end with two consecutive `\`, and as long as the path contains no sequence of three or more `\`. Two `\` can appear in place of the `\` before `\{UNC\}`, the `\` after `\{UNC\}`, and/or the `\` after `{machine}`. The letters in the `\{UNC\}` part can be uppercase or lowercase, and `/` cannot be used in place of `\`s (but `/` can be used in element names).

• **Racket-specific**: A pathname that starts `\\?\{REL\}\{element\}` or `\\?\{REL\}\{element\}` is a relative path, as long as the path does not end with two consecutive `\`, and as long as the path contains no sequence of three or more `\`. This Racket-specific path form supports relative paths with elements that are not normally expressible in Windows paths (e.g., a final element that ends in a space). The `\{REL\}` part must be exactly the three uppercase letters, and `/`s cannot be used in place of `\`. If the path starts `\\?\{REL}\..` then as long as the path continues with repetitions of `\..`, each element counts as an up-directory element; a single `\` must be used to separate the up-directory elements. As soon as a second `\` is used to separate the elements, or as soon as a non-`..` element is encountered, the remaining elements are all literals (never up-directory elements). When a `\\?\{REL\}` path value is converted to a string (or when the path value is written or displayed), the string does not contain the starting `\\?\{REL\}` or the immediately following `\`; converting a path value to a byte string preserves the `\\?\{REL\}` prefix.

• **Racket-specific**: A pathname that starts `\\?\{RED\}\{element\}` or `\\?\{RED\}\{element\}` is a drive-relative path, as long as the path does not end with two consecutive `\`, and as long as the path contains no sequence of three or more `\`. This Racket-specific path form supports drive-relative paths (i.e., absolute given a drive) with elements that are not normally expressible in Windows paths. The `\{RED\}` part must be exactly the three uppercase letters, and `/`s cannot be used in place of `\`. Unlike `\\?\{REL\}` paths, a `..` element is always a literal path element. When a `\\?\{RED\}` path value is converted to a string (or when the path value is written or displayed), the string does not contain the
starting `\?\RED` and it contains a single starting `\`; converting a path value to a byte string preserves the `\?\RED` prefix.

Three additional Racket-specific rules provide meanings to character sequences that are otherwise ill-formed as Windows paths:

- **Racket-specific:** In a pathname of the form `\?\RED` where `<any>` is any non-empty sequence of characters other than `<letter>`; or `<letter>\?\RED`, the entire path counts as the path’s (non-existent) drive.
- **Racket-specific:** In a pathname of the form `\?\RED` where `<any>` is any non-empty sequence of characters and `<elements>` is any sequence that does not start with a `\`, does not end with two `\`, and does not contain a sequence of three `\`, then `\?\RED` counts as the path’s (non-existent) drive.
- **Racket-specific:** In a pathname that starts `\?\RED` and does not match any of the patterns from the preceding bullets, `\?\RED` counts as the path’s (non-existent) drive.

Outside of Racket, except for `\?\RED` paths, pathnames are typically limited to 259 characters when used as a file path and 247 characters when used as a directory path. Racket internally converts pathnames longer than 247 characters to `\?\RED` form to avoid the limits; in that case, the path is first simplified syntactically (in the sense of `simplify-path`). The operating system cannot access files through `\?\RED` paths that are longer than 32,000 characters or so.

Where the above descriptions says “character,” substitute “byte” for interpreting byte strings as paths. The encoding of Windows paths into bytes preserves ASCII characters, and all special characters mentioned above are ASCII, so all of the rules are the same.

Beware that the `\` path separator is an escape character in Racket strings. Thus, the path `\?\RED` as a string must be written "\\?\\REL\\.\\.\\.\\.\\.\\.\\.\\.\\.\\.\\.\\.\\.\\.\\.\\.\\.\\.\\.\\.\\.\\.\\.\\.\\.\\.\\.\\.\\.\\.\\.\\.\\.\\.\\.\\.\\.\\.\\.\\.\\.\\.\\.\\.\\.\\.\\.\\.\\.\\.\\.\\.\\.\\.\\.\\.\\.\\.\\.\\.\\.\\.\\.\\.\\.\\.\\.\\.\\.\\.\\.\\.\\.\\.\\.\\.\\.\\.\\.\\.\\.\\.\\.\\.\\.\\.\\.\\.\\.\\.\\.\\.\\.\\.\\.\\.\\.\\.\\.\\.\\.\\.\\.\\.\\.\\.\\.\\.\\.\\.\\.\\.\\.\\.\\.\\.\\.\\.\\.\\.\\.\\.\\.\\.\\.\\.\\.\\.\\.\\.\\.\\.\\.\\.\\.\\.\\.\\.\\.\\.\\.\\.\\.\\.\\.\\.\\.\\.\\.\\.\\.\\.\\.\\.\\.\\.\\.\\.\\.\\.\\.\\.\\.\\.\\.\\.\\.\\.\\.\\.\\.\\.\\.\\.\\.\\.\\.\\.\\.\\.\\.\\.\\.\\.\\.\\.\\.\\.\\.\\.\\.\\.\\.\\.\\.\\.\\.\\.\\.\\.\\.\\.\\.\\.\\.\\.\\.\\.\\.\\.\\.\\.\\.\\.\\.\\.\\.\\.\\.\\.\\.\\.\\.\\.\\.\\.\\.\\.\\.\\.\\.\\.\\.\\.\\.\\.\\.\\.\\.\\.\\.\\.\\.\\.\\.\\.\\.\\.\\.\\.\\.\\.\\.\\.\\.\\.\\.\\.\\.\\.\\.\\.\\.\\.\\.\\.\\.\\.\\.\\.\\.\\.\\.\\.\\.\\.\\.\\.\\.\\.\\.\\.\\.\\.\\.\\.\\.\\.\\.\\.\\.\\.\\.\\.\\.\\.\\.\\.\\.\\.\\.\\.\\.\\.\\.\\.\\.\\.\\.\\.\\.\\.\\.\\.\\.\\.\\.\\.\\.\\.\\.\\.\\.\\.\\.\\.\\.\\.\\.\\.\\.\\.\\.\\.\\.\\.\\.\\.\\.\\.\\.\\.\\.\\.\\.\\.\\.\\.\\.\\.\\.\\.\\.\\.\\.\\.\\.\\.\\.\\.\\.\\.\\.\\.\\.\\.\\.\\.\\.\\.\\.\\.\\.\\.\\.\\.\\.\\.\\.\\.\\.\\.\\.\\.\\.\\.\\.\\.\\.\\.\\.\\.\\.\\.\\.\\.\\.\\.\\.\\.\\.\\.\\.\\.\\.\\.\\.\\.\\.\\.\\.\\.\\.\\.\\.\\.\\.\\.\\.\\.\\.\\.\\.\\.\\.\\.\\.\\.\\.\\.\\.\\.\\.\\.\\.\\.\\.\\.\\.\\.\\.\\.\\.\\.\\.\\.\\.\\.\\.\\.\\.\\.\\.\\.\\.\\.\\.\\.\\.\\.\n
A path that ends with a directory separator syntactically refers to a directory. In addition, a path syntactically refers to a directory if its last element is a same-directory or up-directory indicator (not quoted by a `\?\RED` form), or if it refers to a root.

Even on variants of Windows that support symbolic links, up-directory `..` indicators in a path are resolved syntactically, not sensitive to links. For example, if a path ends with `d\..\f` and `d` refers to a symbolic link that references a directory with a different parent than `d`, the path nevertheless refers to `f` in the same directory as `d`. A relative-path link is parsed as if prefixed with `\?\REL` paths, except that `..` and `.` elements are allowed throughout the path, and any number of redundant `\` separators are allowed.

Windows paths are cleansed as follows: In paths that start `\?\RED`, redundant `\`s are removed, an extra `\` is added in a `\?\REL` if an extra one is not already present to separate up-directory indicators from literal path elements, and an extra `\` is similarly added after `\?\RED` if an extra one is not already present. For other paths, multiple `/s and `\`s are converted to single
/s or \ (except at the beginning of a shared folder name), and a \ is inserted after the colon in a drive specification if it is missing.

For (bytes->path-element bstr), /s, colons, trailing dots, trailing whitespace, and special device names (e.g., “aux”) in bstr are encoded as a literal part of the path element by using a \?\REL prefix. The bstr argument must not contain a \, otherwise the exn:fail:contract exception is raised.

For (path-element->bytes path) or (path-element->string path), if the byte-string form of path starts with a \?\REL, the prefix is not included in the result.

For (build-path base-path sub-path ...) , trailing spaces and periods are removed from the last element of base-path and all but the last sub-path (unless the element consists of only spaces and periods), except for those that start with \?\. If base-path starts \?\, then after each non-\?\REL and non-\?\RED sub-path is added, all /s in the addition are converted to \s, multiple consecutive \s are converted to a single \, added . elements are removed, and added .. elements are removed along with the preceding element; these conversions are not performed on the original base-path part of the result or on any \?\REL or \?\RED or sub-path. If a \?\REL or \?\RED sub-path is added to a non-\?\ base-path, the base-path (with any additions up to the \?\REL or \?\RED sub-path) is simplified and converted to a \?\ path. In other cases, a \ may be added or removed before combining paths to avoid changing the root meaning of the path (e.g., combining //x and y produces /x/y, because //x/y would be a UNC path instead of a drive-relative path).

For (simplify-path path use-filesystem?) , path is expanded, and if path does not start with \?\, trailing spaces and periods are removed, a / is inserted after the colon in a drive specification if it is missing, and a \ is inserted after \?\ as a root if there are elements and no extra \ already. Otherwise, if no indicators or redundant separators are in path, then path is returned.

For (split-path path) producing base, name, and must-be-dir?, splitting a path that does not start with \?\ can produce parts that start with \?\. For example, splitting C:/x /aux/ twice produces \?\REL\x and \?\REL\aux: the \?\ is needed in these cases to preserve a trailing space after x and to avoid referring to the AUX device instead of an "aux" file.

**Windows Path Representation**

A path on Windows is natively a sequence of UTF-16 code units, where the sequence can include unpaired surrogates. This sequence is encoded as a byte string through an extension of UTF-8, where unpaired surrogates in the UTF-16 code-unit sequence are converted as if they were non-surrogate values. The extended encodings are implemented on Windows as the "platform-UTF-16" and "platform-UTF-8" encodings for bytes-open-converter.

Racket’s internal representation of a Windows path is a byte string, so that path->bytes and bytes->path are always inverses. When converting a path to a native UTF-16 code-
unit sequence, \texttt{\backslash tab} is used in place of platform-UTF-8 decoding errors (on the grounds that tab is normally disallowed as a character in a Windows path, unlike \texttt{\backslash uFFFD}).

A Windows path is converted to a string by treating the platform-UTF-8 encoding as a UTF-8 encoding with \texttt{\backslash uFFFD} in place of decoding errors. Similarly, a string is converted to a path by UTF-8 encoding (in which case no errors are possible).

15.2 Filesystem

15.2.1 Locating Paths

\begin{verbatim}
(find-system-path kind) \rightarrow path?
  kind : symbol?
\end{verbatim}

Returns a machine-specific path for a standard type of path specified by \texttt{kind}, which must be one of the following:

- \texttt{'home-dir} — the current user’s home directory.
  
  On all platforms, if the PLTUSERHOME environment variable is defined as a complete path, then the path is used as the user’s home directory.

  On Unix and Mac OS, when PLTUSERHOME does not apply, the user’s home directory is determined by expanding the path "~", which is expanded by first checking for a HOME environment variable. If none is defined, the USER and LOGNAME environment variables are consulted (in that order) to find a user name, and then system files are consulted to locate the user’s home directory.

  On Windows, when PLTUSERHOME does not apply, the user’s home directory is the user-specific profile directory as determined by the Windows registry. If the registry cannot provide a directory for some reason, the value of the USERPROFILE environment variable is used instead, as long as it refers to a directory that exists. If USERPROFILE also fails, the directory is the one specified by the HOMEDRIVE and HOMEPATH environment variables. If those environment variables are not defined, or if the indicated directory still does not exist, the directory containing the current executable is used as the home directory.

- \texttt{'pref-dir} — the standard directory for storing the current user’s preferences. The preferences directory might not exist.

  On Unix, the preferences directory is normally the \texttt{racket} subdirectory of the path specified by \texttt{XDG\_CONFIG\_HOME}, or \texttt{.config/racket} in the user’s home directory if \texttt{XDG\_CONFIG\_HOME} is not set to an absolute path or if PLTUSERHOME is set. Either way, if that directory does not exist but a \texttt{.racket} directory exists in the user’s home directory, then that directory is the preference directory, instead.
On Windows, the preferences directory is "Racket" in the user’s home directory if determined by PLTUSERHOME, otherwise in the user’s application-data folder as specified by the Windows registry; the application-data folder is usually "Application Data" in the user’s profile directory.

On Mac OS, the preferences directory is "Library/Preferences" in the user’s home directory.

- **'pref-file** — a file that contains a symbol-keyed association list of preference values. The file’s directory path always matches the result returned for 'pref-dir. The file name is "racket-preferences.rktd" on Unix and Windows, and it is "org.racket-lang.preferences.rktd" on Mac OS. The file’s directory might not exist. See also get-preference.

- **'temp-dir** — the standard directory for storing temporary files. On Unix and Mac OS, this is the directory specified by the TMPDIR environment variable, if it is defined, otherwise it is the first path that exists among "/var/tmp", "/usr/tmp", and "/tmp". On Windows, the result is the directory specified by the TMP or TEMP environment variable, if it is defined, otherwise it is the current directory.

- **'init-dir** — the directory containing the initialization file used by the Racket executable.

  On Unix, the initialization directory is the same as the result returned for 'pref-dir—unless that directory does not exist and a ".racketrc" file exists in the user’s home directory, in which case the home directory is the initialization directory.

  On Windows, the initialization directory is the same as the user’s home directory.

  On Mac OS, the initialization directory is "Library/Racket" in the user’s home directory—unless no "racketrc.rktl" exists there and a ".racketrc" file does exist in the home directory, in which case the home directory is the initialization directory.

- **'init-file** — the file loaded at start-up by the Racket executable. The directory part of the path is the same path as returned for 'init-dir.

  On Windows, the file part of the name is "racketrc.rktl".

  On Unix and Mac OS, the file part of the name is "racketrc.rktl"—unless the path returned for 'init-dir is the user’s home directory, in which case the file part of the name is ".racketrc".

- **'config-dir** — a directory for the installation’s configuration. This directory is specified by the PLTCONFIGDIR environment variable, and it can be overridden by the --config or -G command-line flag. If no environment variable or flag is specified, or if the value is not a legal path name, then this directory defaults to an "etc" directory relative to the current executable. If the result of (find-system-path 'config-dir) is a relative path, it is relative to the current executable. The directory might not exist.
• **host-config-dir** — like **config-dir**, but when cross-platform build mode has been selected (through the **-C** or **--cross** argument to **racket**; see §18.1.4 "Command Line"), the result refers to a directory for the current system’s installation, instead of for the target system.

• **addon-dir** — a directory for user-specific Racket configuration, packages, and extension. This directory is specified by the **PLTADDONDIR** environment variable, and it can be overridden by the **--addon** or **-A** command-line flag. If no environment variable or flag is specified, or if the value is not a legal path name, then this directory defaults to a platform-specific locations. The directory might not exist.

  On Unix, the default is normally the "racket" subdirectory of the path specified by **XDG_DATA_HOME**, or ".local/share/racket" in the user’s home directory if **XDG_CONFIG_HOME** is not set to an absolute path or if **PLTUSERHOME** is set. If that directory does not exist but a ".racket" directory exists in the user’s home directory, that the ".racket" directory path is the default, instead.

  On Windows, the default is the same as the **pref-dir** directory.

  On Mac OS, the default is "Library/Racket" within the user’s home directory.

• **cache-dir** — a directory for storing user-specific caches. The directory might not exist.

  On Unix, the cache directory is normally the "racket" subdirectory of the path specified by **XDG_CACHE_HOME**, or ".cache/racket" in the user’s home directory if **XDG_CONFIG_HOME** is not set to an absolute path or if **PLTUSERHOME** is set. If that directory does not exist but a ".racket" directory exists in the home directory, then the ".racket" directory is the cache directory, instead.

  On Windows, the cache directory is the same as the result returned for **addon-dir**.

  On Mac OS, the cache directory is "Library/Caches/Racket" within the user’s home directory.

• **doc-dir** — the standard directory for storing the current user’s documents. On Unix, it’s the user’s home directory. On Windows, it is the user’s home directory if determined by **PLTUSERHOME**, otherwise it is the user’s documents folder as specified by the Windows registry; the documents folder is usually "My Documents" in the user’s home directory. On Mac OS, it’s the "Documents" directory in the user’s home directory.

• **desk-dir** — the directory for the current user’s desktop. On Unix, it’s the user’s home directory. On Windows, it is the user’s home directory if determined by **PLTUSERHOME**, otherwise it is the user’s desktop folder as specified by the Windows registry; the desktop folder is usually "Desktop" in the user’s home directory. On Mac OS, it is "Desktop" in the user’s home directory.

• **sys-dir** — the directory containing the operating system for Windows. On Unix and Mac OS, the result is "/".
• 'exec-file — the path of the Racket executable as provided by the operating system for the current invocation. For some operating systems, the path can be relative.

• 'run-file — the path of the current executable; this may be different from result for 'exec-file because an alternate path was provided through a --name or -N command-line flag to the Racket (or GRacket) executable, or because an embedding executable installed an alternate path. In particular a “launcher” script created by make-racket-launcher sets this path to the script’s path.

• 'collects-dir — a path to the main collection of libraries (see §18.2 “Libraries and Collections”). If this path is relative, then it is relative to the executable as reported by (find-system-path 'exec-file)—though the latter could be a soft-link or relative to the user’s executable search path, so that the two results should be combined with find-executable-path. The 'collects-dir path is normally embedded in the Racket executable, but it can be overridden by the --collects or -X command-line flag.

• 'host-collects-dir — like 'collects-dir, but when cross-platform build mode has been selected (through the -C or --cross argument to racket; see §18.1.4 “Command Line”), the result refers to a directory for the current system’s installation, instead of for the target system. In cross-platform build mode, collection files are normally read from the target system’s installation, but some tasks require current-system directories (such as the one that holds foreign libraries) that are configured relative to the main library-collection path.

• 'orig-dir — the current directory at start-up, which can be useful in converting a relative-path result from (find-system-path 'exec-file) or (find-system-path 'run-file) to a complete path.

Changed in version 6.0.0.3 of package base: Added PLTUSERHOME.
Changed in version 6.9.0.1: Added 'host-config-dir and 'host-collects-dir.
Changed in version 7.8.0.9: Added 'cache-dir, and changed to use XDG directories as preferred on Unix with the previous paths as a fallback, and with similar adjustments for Mac OS.

(path-list-string->path-list str
default-path-list)
→ (listof (or/c path? 'same))
str : (or/c string? bytes?)
default-path-list : (listof (or/c path? 'same))

Parses a string or byte string containing a list of paths, and returns a list of paths. On Unix and Mac OS, paths in a path-list string are separated by a ; on Windows, paths are separated by a ; and all #s in the string are discarded. Whenever the path list contains an empty path, the list default-path-list is spliced into the returned list of paths. Parts of str that do not form a valid path are not included in the returned list. The given str must not contain a null character or null byte.

Changed in version 8.0.0.10 of package base: Changed to allow 'same in default-path-list.
Finds a path for the executable `program`, returning `#f` if the path cannot be found.

If `related` is not `#f`, then it must be a relative path string, and the path found for `program` must be such that the file or directory `related` exists in the same directory as the executable. The result is then the full path for the found `related`, instead of the path for the executable.

This procedure is used by the Racket executable to find the standard library collection directory (see §18.2 “Libraries and Collections”). In this case, `program` is the name used to start Racket and `related` is "collects". The `related` argument is used because, on Unix and Mac OS, `program` may involve a sequence of soft links; in this case, `related` determines which link in the chain is relevant.

If `related` is not `#f`, then when `find-executable-path` does not find a `program` that is a link to another file path, the search can continue with the destination of the link. Further links are inspected until `related` is found or the end of the chain of links is reached. If `deepest?` is `#f` (the default), then the result corresponds to the first path in a chain of links for which `related` is found (and further links are not actually explored); otherwise, the result corresponds to the last link in the chain for which `related` is found.

If `program` is a pathless name, `find-executable-path` gets the value of the PATH environment variable; if this environment variable is defined, `find-executable-path` tries each path in `PATH` as a prefix for `program` using the search algorithm described above for path-containing `programs`. If the PATH environment variable is not defined, `program` is prefixed with the current directory and used in the search algorithm above. (On Windows, the current directory is always implicitly the first item in `PATH`, so `find-executable-path` checks the current directory first on Windows.)

### 15.2.2 Files

\[
(file-exists? path) \rightarrow boolean?
\]

\[
path : path-string?
\]

Returns `#t` if a file (not a directory) `path` exists, `#f` otherwise.

On Windows, `file-exists?` reports `#t` for all variations of the special filenames (e.g., "LPT1", "x:/baddir/LPT1").
(link-exists? path) → boolean?
  path : path-string?

Returns #t if a link path exists, #f otherwise.

The predicates file-exists? or directory-exists? work on the final destination of a link or series of links, while link-exists? only follows links to resolve the base part of path (i.e., everything except the last name in the path).

This procedure never raises the exn:fail:filesystem exception.

On Windows, link-exists? reports #t for both symbolic links and junctions.

Changed in version 6.0.1.12 of package base: Added support for links on Windows.

(file-or-directory-type path [must-exist?])
  → (or/c 'file 'directory 'link 'directory-link #f)
  path : path-string?
  must-exist?: any/c = #f

Reports whether path refers to a file, directory, link, or directory link (in the case of Windows; see also make-file-or-directory-link), assuming that path can be accessed.

If path cannot be accessed, the result is #f if must-exist? is #f, otherwise the exn:fail:filesystem exception is raised.

Added in version 7.8.0.5 of package base.

(delete-file path) → void?
  path : path-string?

Deletes the file with path path if it exists, otherwise the exn:fail:filesystem exception is raised. If path is a link, the link is deleted rather than the destination of the link.

On Windows, if an initial attempt to delete the file fails with a permission error and the value of current-force-delete-permissions is true, then delete-file attempts to change the file’s permissions (to allow writes) and then delete the file; the permission change followed by deletion is a non-atomic sequence, with no attempt to revert a permission change if the deletion fails.

On Windows, delete-file can delete a symbolic link, but not a junction. Use delete-directory to delete a junction.

On Windows, beware that if a file is deleted while it remains in use by some process (e.g., a background search indexer), then the file’s content will eventually go away, but the file’s
name remains occupied until the file is no longer used. As long as the name remains occupied, attempts to open, delete, or replace the file will trigger a permission error (as opposed to a file-exists error). A common technique to avoid this pitfall is to move the file to a generated temporary name before deleting it. See also delete-directory/files.

Changed in version 6.1.1.7 of package base: Changed Windows behavior to use current-force-delete-permissions.

```scheme
(rename-file-or-directory old new [exists-ok?]) → void?
old : path-string?
new : path-string?
exists-ok? : any/c = #f
```

Renames the file or directory with path old—if it exists—to the path new. If the file or directory is not renamed successfully, the exn:fail:filesystem exception is raised.

This procedure can be used to move a file/directory to a different directory (on the same filesystem) as well as rename a file/directory within a directory. Unless exists-ok? is provided as a true value, new cannot refer to an existing file or directory, but the check is not atomic with the rename operation on Unix and Mac OS. Even if exists-ok? is true, new cannot refer to an existing file when old is a directory, and vice versa.

If new exists and is replaced, the replacement is atomic on Unix and Mac OS, but it is not guaranteed to be atomic on Windows. Furthermore, if new exists and is opened by any process for reading or writing, then attempting to replace it will typically fail on Windows. See also call-with-atomic-output-file.

If old is a link, the link is renamed rather than the destination of the link, and it counts as a file for replacing any existing new.

On Windows, beware that a directory cannot be renamed if any file within the directory is open. That constraint is particularly problematic if a search indexer is running in the background (as in the default Windows configuration). A possible workaround is to combine copy-directory/files and delete-directory/files, since the latter can deal with open files, although that sequence is obviously not atomic and temporarily duplicates files.

```scheme
(file-or-directory-modify-seconds path [secs-n]) → exact-integer?
path : path-string?
secs-n : #f = #f
```

```scheme
(file-or-directory-modify-seconds path secs-n) → void?
path : path-string?
secs-n : exact-integer?
```
(file-or-directory-modify-seconds path
  [secs-n
    fail-thunk]) → any

path : path-string?
secs-n : (or/c exact-integer? #f) = #f
fail-thunk : (→ any)
  = (lambda () (raise (make-exn:fail:filesystem ....)))

Returns the file or directory’s last modification date in seconds since midnight UTC, January 1, 1970 (see also §15.6 “Time” when secs-n is not provided or is #f.

For FAT filesystems on Windows, directories do not have modification dates. Therefore, the creation date is returned for a directory, but the modification date is returned for a file.

If secs-n is provided and not #f, the access and modification times of path are set to the given time.

On error (e.g., if no such file exists), then fail-thunk is called (through a tail call) to produce the result of the file-or-directory-modify-secs call. If fail-thunk is not provided, an error raises exn:fail:filesystem.

(file-or-directory-permissions path [mode])
→ (listof (or/c 'read 'write 'execute))

path : path-string?
mode : #f = #f

(file-or-directory-permissions path mode) → (integer-in 0 65535)

path : path-string?
mode : 'bits

(file-or-directory-permissions path mode) → void

path : path-string?
mode : (integer-in 0 65535)

When given one argument or #f as the second argument, returns a list containing 'read, 'write, and/or 'execute to indicate permission the given file or directory path by the current user and group. On Unix and Mac OS, permissions are checked for the current effective user instead of the real user.

If 'bits is supplied as the second argument, the result is a platform-specific integer encoding of the file or directory properties (mostly permissions), and the result is independent of the current user and group. The lowest nine bits of the encoding are somewhat portable, reflecting permissions for the file or directory’s owner, members of the file or directory’s group, or other users:

• #o400 : owner has read permission
• #o200 : owner has write permission
• #0100: owner has execute permission
• #0040: group has read permission
• #0020: group has write permission
• #010: group has execute permission
• #004: others have read permission
• #002: others have write permission
• #001: others have execute permission

See also user-read-bit, etc. On Windows, permissions from all three (owner, group, and others) are always the same, and read and execute permission are always available. On Unix and Mac OS, higher bits have a platform-specific meaning.

If an integer is supplied as the second argument, it is used as an encoding of properties (mostly permissions) to install for the file.

In all modes, the exn:fail:filesystem exception is raised on error (e.g., if no such file exists).

(file-or-directory-identity path [as-link?])
→ exact-positive-integer?
path : path-string?
as-link? : any/c = #f

Returns a number that represents the identity of path in terms of the device and file or directory that it accesses. This function can be used to check whether two paths correspond to the same filesystem entity under the assumption that the path’s entity selection does not change.

If as-link? is a true value, then if path refers to a filesystem link, the identity of the link is returned instead of the identity of the referenced file or directory (if any).

(file-size path) → exact-nonnegative-integer?
path : path-string?

Returns the (logical) size of the specified file in bytes. On Mac OS, this size excludes the resource-fork size. On error (e.g., if no such file exists), the exn:fail:filesystem exception is raised.

copy-file src dest [exists-ok?] → void?
src : path-string?
dest : path-string?
exists-ok? : any/c = #f

1148
Creates the file dest as a copy of src, if dest does not already exist. If dest already exists and exists-ok? is #f, the copy fails with exn:fail:filesystem:exists? exception is raised; otherwise, if dest exists, its content is replaced with the content of src. File permissions are transferred from src to dest; on Windows, the modification time of src is also transferred to dest. If src refers to a link, the target of the link is copied, rather than the link itself; if dest refers to a link and exists-ok? is true, the target of the link is updated.

(make-file-or-directory-link to path) → void?
  to : path-string?
  path : path-string?

Creates a link path to to. The creation will fail if path already exists. The to need not refer to an existing file or directory, and to is not expanded before writing the link. If the link is not created successfully, the exn:fail:filesystem exception is raised.

On Windows XP and earlier, the exn:fail:unsupported exception is raised. On later versions of Windows, the creation of links tends to be disallowed by security policies. Windows distinguishes between file and directory links, and a directory link is created only if to parses syntactically as a directory (see path->directory-path). Furthermore, a relative-path link is parsed specially by the operating system; see §15.1.4 “Windows Paths” for more information. When make-file-or-directory-link succeeds, it creates a symbolic link as opposed to a junction or hard link. Beware that directory links must be deleted using delete-directory instead of delete-file.

Changed in version 6.0.1.12 of package base: Added support for links on Windows.

(current-force-delete-permissions) → boolean?
  (current-force-delete-permissions any/c) → void?
  any/c : boolean?

A parameter that determines on Windows whether delete-file and delete-directory attempt to change a file or directory’s permissions to delete it. The default value is #t.

15.2.3 Directories

See also: rename-file-or-directory, file-or-directory-modify-seconds, file-or-directory-permissions.

(current-directory) → (and/c path? complete-path?)
  (current-directory path) → void?
  path : path-string?

A parameter that determines the current directory for resolving relative paths.
When the parameter procedure is called to set the current directory, the path argument is cleansed using `cleanse-path`, simplified using `simplify-path`, and then converted to a directory path with `path->directory-path`; cleansing and simplification raise an exception if the path is ill-formed. Thus, the current value of `current-directory` is always a cleansed, simplified, complete, directory path.

The path is not checked for existence when the parameter is set.

On Unix and Mac OS, the initial value of the parameter for a Racket process is taken from the `PWD` environment variable—if the value of the environment variable identifies the same directory as the operating system’s report of the current directory.

```
(current-directory-for-user)  →  (and/c path? complete-path?)
(current-directory-for-user path)  →  void?
path : path-string?
```

Like `current-directory`, but use only by `srcloc->string` for reporting paths relative to a directory.

Normally, `current-directory-for-user` should stay at its initial value, reflecting the directory where a user started a process. A tool such as DrRacket, however, implicitly lets a user select a directory (for the file being edited), in which case updating `current-directory-for-user` makes sense.

```
(current-drive)  →  path?
```

Returns the current drive name Windows. For other platforms, the `exn:fail:unsupported` exception is raised. The current drive is always the drive of the current directory.

```
(directory-exists? path)  →  boolean?
path : path-string?
```

Returns `#t` if `path` refers to a directory, `#f` otherwise.

```
(make-directory path)  →  void?
path : path-string?
```

Creates a new directory with the path `path`. If the directory is not created successfully, the `exn:fail:filesystem` exception is raised.

```
(delete-directory path)  →  void?
path : path-string?
```

Deletes an existing directory with the path `path`. If the directory is not deleted successfully, the `exn:fail:filesystem` exception is raised.

1150
On Windows, if an initial attempt to delete the directory fails with a permission error and
the value of `current-force-delete-permissions` is true, then `delete-file` attempts
to change the directory’s permissions (to allow writes) and then delete the directory; the
permission change followed by deletion is a non-atomic sequence, with no attempt to revert
a permission change if the deletion fails.

Changed in version 6.1.1.7 of package `base`: Changed Windows behavior to use
`current-force-delete-permissions`.

```racket
(directory-list [path #:build? build?]) -> (listof path?)
path : path-string? = (current-directory)
build? : any/c = #f
```

Returns a list of all files and directories in the directory specified by `path`. If `build?` is
`#f`, the resulting paths are all path elements; otherwise, the individual results are combined
with `path` using `build-path`. On Windows, an element of the result list may start with
`\\?\REL\`.

The resulting paths are always sorted using `path<?`.

```racket
(filesystem-root-list) -> (listof path?)
```

Returns a list of all current root directories. Obtaining this list can be particularly slow on
Windows.

### 15.2.4 Detecting Filesystem Changes

Many operating systems provide notifications for filesystem changes, and those notifications
are reflected in Racket by filesystem change events.

```racket
(filesystem-change-evt? v) -> boolean?
v : any/c
```

Returns `#t` if `v` is a filesystem change event, `#f` otherwise.

```racket
(filesystem-change-evt path [failure-thunk])
-> (or/c filesystem-change-evt? any)
path : path-string?
failure-thunk : (or/c (-> any) #f) = #f
```

Creates a `filesystem change event`, which is a synchronizable event that becomes ready for
synchronization after a change to `path`:

- If `path` refers to a file, the event becomes ready for synchronization when the file’s
  content or attributes change, or when the file is deleted.
• If `path` refers to a directory, the event becomes ready for synchronization if a file or subdirectory is added, renamed, or removed within the directory.

The event also becomes ready for synchronization if it is passed to `filesystem-change-evt-cancel`.

Finally, depending on the precision of information available from the operating system, the event may become ready for synchronization under other circumstances. For example, on Windows, an event for a file becomes ready when any file changes within in the same directory as the file.

After a filesystem change event becomes ready for synchronization, it stays ready for synchronization. The event’s synchronization result is the event itself.

If the current platform does not support filesystem-change notifications, then the `exn:fail:unsupported` exception is raised if `failure-thunk` is not provided as a procedure, or `failure-thunk` is called in tail position if provided. Similarly, if there is any operating-system error when creating the event (such as a non-existent file), then the `exn:fail:filesystem` exception is raised or `failure-thunk` is called.

Creation of a filesystem change event allocates resources at the operating-system level. The resources are released at latest when the event is synchronized and ready for synchronization, when the event is canceled with `filesystem-change-evt-cancel`, or when the garbage collector determine that the filesystem change event is unreachable. See also `system-type` in `fs-change` mode.

A filesystem change event is placed under the management of the current custodian when it is created. If the custodian is shut down, `filesystem-change-evt-cancel` is applied to the event.

Changed in version 7.3.0.8 of package `base`: Allow `#f` for `failure-thunk`.

```
(filesystem-change-evt-cancel evt) → void?

evt : filesystem-change-evt?
```

Causes `evt` to become immediately ready for synchronization, whether it was ready or not before, and releases the resources (at the operating-system level) for tracking filesystem changes.

### 15.2.5 Declaring Paths Needed at Run Time

```
(require racket/runtime-path)  package:base
```

The bindings documented in this section are provided by the `racket/runtime-path` library, not `racket/base` or `racket`. 

1152
The **racket/runtime-path** library provides forms for accessing files and directories at run time using a path that are usually relative to an enclosing source file. Unlike using **collection-path**, **define-runtime-path** exposes each run-time path to tools like the executable and distribution creators, so that files and directories needed at run time are carried along in a distribution.

In addition to the bindings described below, **racket/runtime-path** provides `%datum` in phase level 1, since string constants are often used as compile-time expressions with **define-runtime-path**.

```racket
(define-runtime-path id maybe-runtime?-id expr)
```

| maybe-runtime? = | #:runtime?-id runtime?-id |

Uses **expr** as both a compile-time (i.e., phase 1) expression and a run-time (i.e., phase 0) expression. In either context, **expr** should produce a path, a string that represents a path, a list of the form `(list 'lib str ...+)`, or a list of the form `(list 'so str)` or `(list 'so str vers)`. If runtime?-id is provided, then it is bound in the context of **expr** to `#f` for the compile-time instance of **expr** and `#t` for the run-time instance of **expr**.

For run time, **id** is bound to a path that is based on the result of **expr**. The path is normally computed by taking a relative path result from **expr** and adding it to a path for the enclosing file (which is computed as described below). However, tools like the executable creator can also arrange (by colluding with **racket/runtime-path**) to have a different base path substituted in a generated executable. If **expr** produces an absolute path, it is normally returned directly, but again may be replaced by an executable creator. In all cases, the executable creator preserves the relative locations of all paths within a given package (treating paths outside of any package as being together). When **expr** produces a relative or absolute path, then the path bound to **id** is always an absolute path.

If **expr** produces a list of the form `(list 'lib str ...+)`, the value bound to **id** is an absolute path. The path refers to a collection-based file similar to using the value as a module path.

If **expr** produces a list of the form `(list 'so str)` or `(list 'so str vers)`, the value bound to **id** can be either **str** or an absolute path; it is an absolute path when searching in the Racket-specific shared-object library directories (as determined by **get-lib-search-dirs**) locates the path. In this way, shared-object libraries that are installed specifically for Racket get carried along in distributions. The search tries each directory in order; within a directory, the search tries using **str** directly, then it tries adding each version specified by **vers**—which defaults to `'#f')—along with a platform-specific shared-library extension—as produced by `(system-type 'so-suffix)`. A **vers** can be a string, or it can be a list of strings and `#f`.

If **expr** produces a list of the form `(list 'share str)`, the value bound to **id** can be either **str** or an absolute path; it is an absolute path when searching in the directories reported
by `find-user-share-dir` and `find-share-dir` (in that order) locates the path. In this way, files that are installed in Racket’s "share" directory get carried along in distributions.

If `expr` produces a list of the form `(list 'module module-path var-ref)` or `(list 'so str (list str-or-false ...))`, the value bound to `id` is a module path index, where `module-path` is treated as relative (if it is relative) to the module that is the home of the variable reference `var-ref`, where `var-ref` can be `#f` if `module-path` is absolute. In an executable, the corresponding module is carried along, including all of its dependencies.

For compile-time, the `expr` result is used by an executable creator—but not the result when the containing module is compiled. Instead, `expr` is preserved in the module as a compile-time expression (in the sense of `begin-for-syntax`). Later, at the time that an executable is created, the compile-time portion of the module is executed (again), and the result of `expr` is the file to be included with the executable. The reason for the extra compile-time execution is that the result of `expr` might be platform-dependent, so the result should not be stored in the (platform-independent) bytecode form of the module; the platform at executable-creation time, however, is the same as at run time for the executable. Note that `expr` is still evaluated at run-time; consequently, avoid procedures like `collection-path`, which depends on the source installation, and instead use relative paths and forms like `(list 'lib str ...+)`.

If a path is needed only on some platforms and not on others, use `define-runtime-path-list` with an `expr` that produces an empty list on platforms where the path is not needed.

Beware that `define-runtime-path` in a phase level other than 0 does not cooperate properly with an executable creator. To work around that limitation, put `define-runtime-path` in a separate module—perhaps a submodule created by `module`—then export the definition, and then the module containing the definition can be required into any phase level. Using `define-runtime-path` in a phase level other than 0 logs a warning at expansion time.

The enclosing path for a `define-runtime-path` is determined as follows from the `define-runtime-path` syntactic form:

- If the form has a source module according to `syntax-source-module`, then the source location is determined by preserving the original expression as a syntax object, extracting its source module path at run time (again using `syntax-source-module`), and then resolving the resulting module path index. Note that `syntax-source-module` is based on a syntax object’s lexical information, not its source location.
- If the expression has no source module, the `syntax-source` location associated with the form is used, if is a string or path.
- If no source module is available, and `syntax-source` produces no path, then `current-load-relative-directory` is used if it is not `#f`. Finally, `current-directory` is used if all else fails.

In the latter two cases, the path is normally preserved in (platform-specific) byte form, but if the enclosing path corresponds to a result of `collection-file-path`, then the path is...
record as relative to the corresponding module path.

Changed in version 6.0.1.6 of package base: Preserve relative paths only within a package.

Changed in version 7.5.0.7: Added support for '*share in expr.

Examples:

```scheme
; Access a file "data.txt" at run-time that is originally located in the same directory as the module source file:
(define-runtime-path data-file "data.txt")
(define (read-data)
  (with-input-from-file data-file
    (lambda ()
      (read-bytes (file-size data-file))))

; Load a platform-specific shared object (using ffi-lib) that is located in a platform-specific sub-directory of the module's source directory:
(define-runtime-path libfit-path
  (build-path "compiled" "native" (system-library-subpath #f)
    (path-replace-suffix "libfit"
      (system-type 'so-suffix))))
(define libfit (ffi-lib libfit-path))

; Load a platform-specific shared object that might be installed as part of the operating system, or might be installed specifically for Racket:
(define-runtime-path libssl-so
  (case (system-type)
    [(windows) '(so "ssleay32")]
    [else '(so "libssl")])
(define libssl (ffi-lib libssl-so))
```

Changed in version 6.4 of package base: Added #:runtime?-id.

```scheme
(define-runtime-paths (id ...?) maybe-runtime?-id expr)
```

Like define-runtime-path, but declares and binds multiple paths at once. The expr should produce as many values as ids.

```scheme
(define-runtime-path-list id maybe-runtime?-id expr)
```

Like define-runtime-path, but expr should produce a list of paths.

```scheme
(define-runtime-module-path-index id maybe-runtime?-id module-path-expr)
```
Similar to `define-runtime-path`, but `id` is bound to a module path index that encapsulates the result of `module-path-exp` relative to the enclosing module.

Use `define-runtime-module-path-index` to bind a module path that is passed to a reflective function like `dynamic-require` while also creating a module dependency for building and distributing executables.

```scheme
(runtime-require module-path)
```

Similar to `define-runtime-module-path-index`, but creates the distribution dependency without binding a module path index. When `runtime-require` is used multiple times within a module with the same `module-path`, all but the first use expands to an empty `begin`.

```scheme
(define-runtime-module-path id module-path)
```

Similar to `define-runtime-path`, but `id` is bound to a resolved module path. The resolved module path for `id` corresponds to `module-path` (with the same syntax as a module path for `require`), which can be relative to the enclosing module.

The `define-runtime-module-path-index` form is usually preferred, because it creates a weaker link to the referenced module. Unlike `define-runtime-module-path-index`, the `define-runtime-module-path` form creates a `for-label` dependency from an enclosing module to `module-path`. Since the dependency is merely `for-label`, `module-path` is not instantiated or visited when the enclosing module is instantiated or visited (unless such a dependency is created by other `require`s), but the code for the referenced module is loaded when the enclosing module is loaded.

```scheme
(runtime-paths module-path)
```

This form is mainly for use by tools such as executable builders. It expands to a quoted list containing the run-time paths declared by `module-path`, returning the compile-time results of the declaration `expr`s, except that paths are converted to byte strings. The enclosing module must require (directly or indirectly) the module specified by `module-path`, which is an unquoted module path. The resulting list does not include module paths bound through `define-runtime-module-path`.

### 15.2.6 More File and Directory Utilities

```scheme
(require racket/file)  package: base
```

The bindings documented in this section are provided by the `racket/file` and `racket` libraries, but not `racket/base`.  

1156
(file->string path [#:mode mode-flag]) → string?
  path : path-string?
  mode-flag : (or/c 'binary 'text) = 'binary

Reads all characters from path and returns them as a string. The mode-flag argument is the same as for open-input-file.

(file->bytes path [#:mode mode-flag]) → bytes?
  path : path-string?
  mode-flag : (or/c 'binary 'text) = 'binary

Reads all characters from path and returns them as a byte string. The mode-flag argument is the same as for open-input-file.

(file->value path [#:mode mode-flag]) → any
  path : path-string?
  mode-flag : (or/c 'binary 'text) = 'binary

Reads a single S-expression from path using read. The mode-flag argument is the same as for open-input-file.

(file->list path [proc #:mode mode-flag]) → (listof any/c)
  path : path-string?
  proc : (input-port? . -> . any/c) = read
  mode-flag : (or/c 'binary 'text) = 'binary

Repeatedly calls proc to consume the contents of path, until eof is produced. The mode-flag argument is the same as for open-input-file.

(file->lines path [#:mode mode-flag #:line-mode line-mode]) → (listof string?)
  path : path-string?
  mode-flag : (or/c 'binary 'text) = 'binary
  line-mode : (or/c 'linefeed 'return 'return-linefeed 'any 'any-one) = 'any

Read all characters from path, breaking them into lines. The line-mode argument is the same as the second argument to read-line, but the default is 'any instead of 'linefeed. The mode-flag argument is the same as for open-input-file.

(file->bytes-lines path [#:mode mode-flag #:line-mode line-mode]) → (listof bytes?)

1157
Like `file->lines`, but reading bytes and collecting them into lines like `read-bytes-line`.

```scheme
(display-to-file  v
  path
  [#:mode mode-flag
   #:exists exists-flag]) → void?

v : any/c
path : path-string?
mode-flag : (or/c 'binary 'text) = 'binary
exists-flag : (or/c 'error 'append 'update
  'replace 'truncate 'truncate/replace)
  = 'error
```

Uses `display` to print `v` to `path`. The `mode-flag` and `exists-flag` arguments are the same as for `open-output-file`.

```scheme
(write-to-file  v
  path
  [#:mode mode-flag
   #:exists exists-flag]) → void?

v : any/c
path : path-string?
mode-flag : (or/c 'binary 'text) = 'binary
exists-flag : (or/c 'error 'append 'update
  'replace 'truncate 'truncate/replace)
  = 'error
```

Like `display-to-file`, but using `write` instead of `display`.

```scheme
(display-lines-to-file  lst
  path
  [#:separator separator
   #:mode mode-flag
   #:exists exists-flag]) → void?

lst : list?
path : path-string?
separator : any/c = "\n"
mode-flag : (or/c 'binary 'text) = 'binary
exists-flag : (or/c 'error 'append 'update
  'replace 'truncate 'truncate/replace)
  = 'error
```
Displays each element of `lst` to `path`, adding `separator` after each element. The `mode-flag` and `exists-flag` arguments are the same as for `open-output-file`.

```scheme
(copy-directory/files
  src
dest
#:keep-modify-seconds? keep-modify-seconds?
#:preserve-links? preserve-links?)
→ void?
src : path-string?
dest : path-string?
keep-modify-seconds? : #f
preserve-links? : #f
```

Copies the file or directory `src` to `dest`, raising `exn:fail:filesystem` if the file or directory cannot be copied, possibly because `dest` exists already. If `src` is a directory, the copy applies recursively to the directory’s content. If a source is a link and `preserve-links?` is `#f`, the target of the link is copied rather than the link itself; if `preserve-links?` is `#t`, the link is copied.

If `keep-modify-seconds?` is `#f`, then file copies keep only the properties kept by `copy-file`. If `keep-modify-seconds?` is `true`, then each file copy also keeps the modification date of the original.

Changed in version 6.3 of package `base`: Added the `#:preserve-links?` argument.

```scheme
(delete-directory/files path
 #:must-exist? must-exist?) → void?
path : path-string?
must-exist? : #t
```

Deletes the file or directory specified by `path`, raising `exn:fail:filesystem` if the file or directory cannot be deleted. If `path` is a directory, then `delete-directory/files` is first applied to each file and directory in `path` before the directory is deleted.

If `must-exist?` is `true`, then `exn:fail:filesystem` is raised if `path` does not exist. If `must-exist?` is `false`, then `delete-directory/files` succeeds if `path` does not exist (but a failure is possible if `path` initially exists and is removed by another thread or process before `delete-directory/files` deletes it).

On Windows, `delete-directory/files` attempts to move a file into the temporary-file directory before deleting it, which avoids problems caused by deleting a file that is currently open (e.g., by a search indexer running as a background process). If the move attempt fails (e.g., because the temporary directory is on a different drive than the file), then the file is deleted directly with `delete-file`.

Changed in version 7.0 of package `base`: Added Windows-specific file deletion.
(find-files
  predicate
  [start-path]
  #:skip-filtered-directory? skip-filtered-directory?
  #:follow-links? follow-links?)
→ (listof path?)
  predicate : (path? . -> . any/c)
  start-path : (or/c path-string? #f) = #f
  skip-filtered-directory? : #f
  follow-links? : #f

Traverses the filesystem starting at start-path and creates a list of all files and directories for which predicate returns true. If start-path is #f, then the traversal starts from (current-directory). In the resulting list, each directory precedes its content.

The predicate procedure is called with a single argument for each file or directory. If start-path is #f, the argument is a pathname string that is relative to the current directory. Otherwise, it is a path building on start-path. Consequently, supplying (current-directory) for start-path is different from supplying #f, because predicate receives complete paths in the former case and relative paths in the latter. Another difference is that predicate is not called for the current directory when start-path is #f.

If skip-filtered-directory? is true, then when predicate returns #f for a directory, the directory’s content is not traversed.

If follow-links? is true, the find-files traversal follows links, and links are not included in the result. If follow-links? is #f, then links are not followed, and links are included in the result.

If start-path does not refer to an existing file or directory, then predicate will be called exactly once with start-path as the argument.

The find-files procedure raises an exception if it encounters a directory for which directory-list fails.

Changed in version 6.3.0.11 of package base: Added the #:skip-filtered-directory? argument.

(pathlist-closure path-list
  [#:path-filter path-filter
   #:follow-links? follow-links?])
→ (listof path?)
  path-list : (listof path-string?)
  path-filter : (or/c #f (path? . -> . any/c)) = #f
  follow-links? : any/c = #f

Given a list of paths, either absolute or relative to the current directory, returns a list such
that

• if a nested path is given, all of its ancestors are also included in the result (but the same ancestor is not added twice);

• if a path refers to directory, all of its descendants are also included in the result, except as omitted by path-filter;

• ancestor directories appear before their descendants in the result list, as long as they are not misordered in the given path-list.

If path-filter is a procedure, then it is applied to each descendant of a directory. If path-filter returns #f, then the descendant (and any of its descendants, in the case of a subdirectory) are omitted from the result.

If follow-links? is true, then the traversal of directories and files follows links, and the link paths are not included in the result. If follow-links? is #f, then the result list includes paths to link and the links are not followed.

Changed in version 6.3.0.11 of package base: Added the #:path-filter argument.

(fold-files proc
  init-val
  [start-path
   follow-links?]) -> any
proc : (or/c (path? (or/c 'file 'dir 'link) any/c
  . -> . any/c)
   (path? (or/c 'file 'dir 'link) any/c
  . -> . (values any/c any/c)))
init-val : any/c
start-path : (or/c path-string? #f) = #f
follow-links? : any/c = #t

Traverses the filesystem starting at start-path, calling proc on each discovered file, directory, and link. If start-path is #f, then the traversal starts from (current-directory).

The proc procedure is called with three arguments for each file, directory, or link:

• If start-path is #f, the first argument is a pathname string that is relative to the current directory. Otherwise, the first argument is a pathname that starts with start-path. Consequently, supplying (current-directory) for start-path is different from supplying #f, because proc receives complete paths in the former case and relative paths in the latter. Another difference is that proc is not called for the current directory when start-path is #f.
• The second argument is a symbol, either 'file, 'dir, or 'link. The second argument can be 'link when follow-links? is #f, in which case the filesystem traversal does not follow links. If follow-links? is #t, then proc will only get a 'link as a second argument when it encounters a dangling symbolic link (one that does not resolve to an existing file or directory).

• The third argument is the accumulated result. For the first call to proc, the third argument is init-val. For the second call to proc (if any), the third argument is the result from the first call, and so on. The result of the last call to proc is the result of fold-files.

The proc argument is used in an analogous way to the procedure argument of foldl, where its result is used as the new accumulated result. There is an exception for the case of a directory (when the second argument is 'dir): in this case the procedure may return two values, the second indicating whether the recursive scan should include the given directory or not. If it returns a single value, the directory is scanned. In the cases of files or links (when the second argument is 'file or 'link), a second value is permitted but ignored.

If the start-path is provided but no such path exists, or if paths disappear during the scan, then an exception is raised.

\[
\text{(make-directory* path)} \rightarrow \text{void?}
\]
\[
\text{path} : \text{path-string?}
\]

Creates directory specified by path, creating intermediate directories as necessary, and never failing if path exists already.

If path is a relative path and the current directory does not exist, then make-directory* will not create the current directory, because it considers only explicit elements of path.

\[
\text{(make-parent-directory* path)} \rightarrow \text{void?}
\]
\[
\text{path} : \text{path-string?}
\]

Creates the parent directory of the path specified by path, creating intermediate directories as necessary, and never failing if an ancestor of path exists already.

If path is a filesystem root or a relative path with a single path element, then no directory is created. Like make-directory*, if path is a relative path and the current directory does not exist, then make-parent-directory* will not create it.

Added in version 6.1.1.3 of package base.
Creates a new temporary file and returns a pathname string for the file. Instead of merely generating a fresh file name, the file is actually created; this prevents other threads or processes from picking the same temporary name.

The `template` argument must be a format string suitable for use with `format` and one additional string argument (where the string contains only digits). If the resulting string is a relative path, it is combined with the result of `(find-system-path 'temp-dir)`, unless `directory` is provided and non-#f, in which case the file name generated from `template` is combined with `directory` to obtain a full path.

The `template` argument’s default is only the string "rkttmp-a" when there is no source location information for the callsite of `make-temporary-file` (or if `make-temporary-file` is used in a higher-order position). If there is such information, then the template string is based on the source location.

If `copy-from-filename` is provided as path, the temporary file is created as a copy of the named file (using `copy-file`). If `copy-from-filename` is #f, the temporary file is created as empty. If `copy-from-filename` is 'directory, then the temporary “file” is created as a directory.

When a temporary file is created, it is not opened for reading or writing when the pathname is returned. The client program calling `make-temporary-file` is expected to open the file with the desired access and flags (probably using the 'truncate flag; see `open-output-file`) and to delete it when it is no longer needed.

```scheme
(call-with-atomic-output-file
 file
 proc
 [:security-guard security-guard
 #:rename-fail-handler rename-fail-handler])
 -> any
 file : path-string?
 proc : ([port output-port?] [tmp-path path?] . -> . any)
 security-guard : (or/c #f security-guard?) = #f
 rename-fail-handler : (or/c #f (exn:fail:filesystem? path> . -> . any))
 = #f
```

Opens a temporary file for writing in the same directory as `file`, calls `proc` to write to the temporary file, and then atomically (except on Windows) moves the temporary file in place of `file`. The move simply uses `rename-file-or-directory` on Unix and Mac OS, and it uses `rename-file-or-directory` on Windows if `rename-fail-handler` is provided; otherwise, on Windows, the moves uses an extra rename step (see below) on Windows to avoid problems due to concurrent readers of `file`.

The `proc` function is called with an output port for the temporary file, plus the path of the temporary file. The result of `proc` is the result of `call-with-atomic-output-file`.

1163
The `call-with-atomic-output-file` function arranges to delete temporary files on exceptions.

Windows prevents programs from deleting or replacing files that are open, but it allows renaming of open files. Therefore, on Windows, `call-with-atomic-output-file` by default creates a second temporary file `extra-temp-file`, renames `file` to `extra-temp-file`, renames the temporary file written by `proc` to `file`, and finally deletes `extra-temp-file`. Since that process is not atomic, however, `rename-file-or-directory` is used if `rename-fail-handler` is provided, where `rename-file-or-directory` has some chance of being atomic, since that the source and destination of the moves will be in the same directory; any filesystem exception while attempting to rename the file is sent to `rename-fail-handler`, which can re-raise the exception or simply return to try again, perhaps after a delay. In addition to a filesystem exception, the `rename-fail-handler` procedure also receives the temporary file path to be moved to `path`. The `rename-fail-handler` argument is used only on Windows.

Changed in version 7.1.0.6 of package `base`: Added the `#:rename-fail-handler` argument.

```scheme
(get-preference name
    [failure-thunk
        flush-mode
        filename
        #:use-lock? use-lock?
        #:timeout-lock-there timeout-lock-there
        #:lock-there lock-there]) → any
name : symbol?
failure-thunk : (-> any) = (lambda () #f)
flush-mode : any/c = 'timestamp
filename : (or/c string-path? #f) = #f
use-lock? : any/c = #t
timeout-lock-there : (or/c (path? . -> . any) #f) = #f
lock-there : (or/c (path? . -> . any) #f)
    = (make-handle-get-preference-locked
        0.01 name failure-thunk flush-mode filename
        #:lock-there timeout-lock-there)
```

Extracts a preference value from the file designated by `(find-system-path 'pref-file)`, or by `filename` if it is provided and is not `#f`. In the former case, if the preference file doesn’t exist, `get-preferences` attempts to read an old preferences file, and then a "racket-prefs.rktd" file in the configuration directory (as reported by `find-config-dir"), instead. If none of those files exists, the preference set is empty.

The preference file should contain a list of symbol–value lists written with the default parameter settings. Keys starting with `racket:`, `mzscheme:`, `mred:`, and `plt: in any letter case are reserved for use by Racket implementors. If the preference file does not contain a list of symbol–value lists, an error is logged via `log-error` and `failure-thunk` is called.
The result of `get-preference` is the value associated with `name` if it exists in the association list, or the result of calling `failure-thunk` otherwise.

Preference settings are cached (weakly) across calls to `get-preference`, using `(path->complete-path filename)` as a cache key. If `flush-mode` is provided as `#f`, the cache is used instead of re-consulting the preferences file. If `flush-mode` is provided as `'times-tamp` (the default), then the cache is used only if the file has a timestamp that is the same as the last time the file was read. Otherwise, the file is re-consulted.

On platforms for which `preferences-lock-file-mode` returns `'file-lock` and when `use-lock?` is true, preference-file reading is guarded by a lock; multiple readers can share the lock, but writers take the lock exclusively. If the preferences file cannot be read because the lock is unavailable, `lock-there` is called on the path of the lock file; if `lock-there` is `#f`, an exception is raised. The default `lock-there` handler retries about 5 times (with increasing delays between each attempt) before trying `timeout-lock-there`, and the default `timeout-lock-there` triggers an exception.

See also `put-preferences`. For a more elaborate preference system, see `preferences:get`.

**Old preferences files:** When a `filename` is not provided and the file indicated by `(find-system-path 'pref-file)` does not exist, the following paths are checked for compatibility with old versions of Racket:

- **Windows:** 
  ```scheme
  (build-path (find-system-path 'pref-dir) 'up "PLT Scheme" plt-prefs.ss"
  )
  ```
- **Mac OS:** 
  ```scheme
  (build-path (find-system-path 'pref-dir) "org.plt-scheme.prefs.ss"
  )
  ```
- **Unix:** 
  ```scheme
  (expand-user-path "~/.plt-scheme/plt-prefs.ss"
  )
  ```

```
(put-preferences
  names vals
  [locked-proc filename]) → void?

names : (listof symbol?)
vals : list?
locked-proc : (or/c #f (path? . -> . any)) = #f
filename : (or/c #f path-string?) = #f
```

Installs a set of preference values and writes all current values to the preference file designated by `(find-system-path 'pref-file)`, or `filename` if it is supplied and not `#f`.

The `names` argument supplies the preference names, and `vals` must have the same length as `names`. Each element of `vals` must be an instance of a built-in data type whose `write`
output is readable (i.e., the print-unreadable parameter is set to #f while writing preferences).

Current preference values are read from the preference file before updating, and a write lock is held starting before the file read, and lasting until after the preferences file is updated. The lock is implemented by the existence of a file in the same directory as the preference file; see preferences-lock-file-mode for more information. If the directory of the preferences file does not already exist, it is created.

If the write lock is already held, then locked-proc is called with a single argument: the path of the lock file. The default locked-proc (used when the locked-proc argument is #f) reports an error; an alternative thunk might wait a while and try again, or give the user the choice to delete the lock file (in case a previous update attempt encountered disaster and locks are implemented by the presence of the lock file).

If filename is #f or not supplied, and the preference file does not already exist, then values read from the "defaults" collection (if any) are written for preferences that are not mentioned in names.

(preferences-lock-file-mode) → (or/c 'exists 'file-lock)

Reports the way that the lock file is used to implement preference-file locking on the current platform.

The 'exists mode is currently used on all platforms except Windows. In 'exists mode, the existence of the lock file indicates that a write lock is held, and readers need no lock (because the preferences file is atomically updated via rename-file-or-directory).

The 'file-lock mode is currently used on Windows. In 'file-lock mode, shared and exclusive locks (in the sense of port-try-file-lock?) on the lock file reflect reader and writer locks on the preference-file content. (The preference file itself is not locked, because a lock would interfere with replacing the file via rename-file-or-directory.)

(make-handle-get-preference-locked delay name
  [failure-thunk
   flush-mode
   filename
   #:lock-there lock-there
   #:max-delay max-delay])

→ (path-string? . -> . any)
  delay : real?
  name : symbol?
  failure-thunk : (-> any) = (lambda () #f)
  flush-mode : any/c = 'timestamp
  filename : (or/c path-string? #f) = #f
  lock-there : (or/c (path? . -> . any) #f) = #f
max-delay : real? = 0.2

Creates a procedure suitable for use as the #:lock-there argument to get-preference, where the name, failure-thunk, flush-mode, and filename are all passed on to get-preference by the result procedure to retry the preferences lookup.

Before calling get-preference, the result procedure uses (sleep delay) to pause. Then, if (* 2 delay) is less than max-delay, the result procedure calls make-handle-get-preference-locked to generate a new retry procedure to pass to get-preference, but with a delay of (* 2 delay). If (* 2 delay) is not less than max-delay, then get-preference is called with the given lock-there, instead.

(call-with-file-lock/timeout filename kind thunk failure-thunk [#:lock-file lock-file #:delay delay #:max-delay max-delay]) → any

filename : (or/c path-string? #f)
kind : (or/c 'shared 'exclusive)
thunk : (-> any)
failure-thunk : (-> any)
lock-file : (or/c #f path-string?) = #f
delay : (and/c real? (not/c negative?)) = 0.01
max-delay : (and/c real? (not/c negative?)) = 0.2

Obtains a lock for the filename lock-file and then calls thunk. The filename argument specifies a file path prefix that is used only to generate the lock filename when lock-file is #f. Specifically, when lock-file is #f, then call-with-file-lock/timeout uses make-lock-file-name to build the lock filename. If the lock file does not yet exist, it is created; beware that the lock file is not deleted by call-with-file-lock/timeout.

When thunk returns, call-with-file-lock/timeout releases the lock, returning the result of thunk. The call-with-file-lock/timeout function will retry after delay seconds and continue retrying with exponential backoff until delay reaches max-delay. If call-with-file-lock/timeout fails to obtain the lock, failure-thunk is called in tail position. The kind argument specifies whether the lock is 'shared or 'exclusive in the sense of port-try-file-lock?.

Examples:

> (call-with-file-lock/timeout filename 'exclusive
  (lambda () (printf "File is locked\n"))
  (lambda () (printf "Failed to obtain lock for file\n")))
File is locked
(call-with-file-lock/timeout #f 'exclusive
  (lambda ()
    (call-with-file-lock/timeout filename 'shared
      (lambda () (printf "Shouldn't get here\n"))
      (lambda () (printf "Failed to obtain lock for file\n")))
    (lambda () (printf "Shouldn't get here either\n"))
    #:lock-file (make-lock-file-name filename))
Failed to obtain lock for file

(make-lock-file-name path) → path?
  path : (or path-string? path-for-some-system?)
(make-lock-file-name dir name) → path?
  dir : (or path-string? path-for-some-system?)
  name : path-element?

Creates a lock filename by prepending ".LOCK" on Windows (i.e., when cross-system-type reports 'windows) or ".LOCK" on other platforms to the file portion of the path.

Example:

> (make-lock-file-name "~/home/george/project/important-file")
<path:/home/george/project/.LOCKimportant-file>

user-read-bit : #o400
user-write-bit : #o200
user-execute-bit : #o100
group-read-bit : #o040
group-write-bit : #o020
group-execute-bit : #o010
other-read-bit : #o004
other-write-bit : #o002
other-execute-bit : #o001

Constants that are useful with file-or-directory-permissions and bitwise operations such as bitwise-ior, and bitwise-and.

15.3 Networking

15.3.1 TCP

(require racket/tcp)  package:base

The bindings documented in this section are provided by the racket/tcp and racket libraries, but not racket/base.
For information about TCP in general, see *TCP/IP Illustrated, Volume 1* by W. Richard Stevens.

```
(tcp-listen  port-no
     [max-allow-wait
      reuse?
      hostname])  -> tcp-listener?

port-no : listen-port-number?
max-allow-wait : exact-nonnegative-integer? = 4
reuse? : any/c = #f
hostname : (or/c string? #f) = #f
```

Creates a “listening” server on the local machine at the port number specified by `port-no`. If `port-no` is 0 the socket binds to an ephemeral port, which can be determined by calling `tcp-addresses`. The `max-allow-wait` argument determines the maximum number of client connections that can be waiting for acceptance. (When `max-allow-wait` clients are waiting acceptance, no new client connections can be made.)

If the `reuse?` argument is true, then `tcp-listen` will create a listener even if the port is involved in a TIME_WAIT state. Such a use of `reuse?` defeats certain guarantees of the TCP protocol; see Stevens’s book for details. Furthermore, on many modern platforms, a true value for `reuse?` overrides TIME_WAIT only if the listener was previously created with a true value for `reuse?`.

If `hostname` is `#f` (the default), then the listener accepts connections to all of the listening machine’s addresses. Otherwise, the listener accepts connections only at the interface(s) associated with the given hostname. For example, providing "127.0.0.1" as `hostname` creates a listener that accepts only connections to "127.0.0.1" (the loopback interface) from the local machine.

(Racket implements a listener with multiple sockets, if necessary, to accommodate multiple addresses with different protocol families. On Linux, if `hostname` maps to both IPv4 and IPv6 addresses, then the behavior depends on whether IPv6 is supported and IPv6 sockets can be configured to listen to only IPv6 connections: if IPv6 is not supported or IPv6 sockets are not configurable, then the IPv6 addresses are ignored; otherwise, each IPv6 listener accepts only IPv6 connections.)

The return value of `tcp-listen` is a TCP listener. This value can be used in future calls to `tcp-accept`, `tcp-accept-ready?`, and `tcp-close`. Each new TCP listener value is placed into the management of the current custodian (see §14.7 "Custodians").

If the server cannot be started by `tcp-listen`, the `exn:fail:network` exception is raised.

A TCP listener can be used as a synchronizable event (see §11.2.1 "Events"). A TCP listener is ready for synchronization when `tcp-accept` would not block; the synchronization result of a TCP listener is the TCP listener itself.
Attempts to connect as a client to a listening server. The `hostname` argument is the server host’s Internet address name, and `port-no` is the port number where the server is listening.

(If `hostname` is associated with multiple addresses, they are tried one at a time until a connection succeeds. The name "localhost" generally specifies the local machine.)

The optional `local-hostname` and `local-port-no` specify the client’s address and port. If both are #f (the default), the client’s address and port are selected automatically. If `local-hostname` is not #f, then `local-port-no` must be non-#f. If `local-port-no` is non-#f and `local-hostname` is #f, then the given port is used but the address is selected automatically.

Two values are returned by `tcp-connect`: an input port and an output port. Data can be received from the server through the input port and sent to the server through the output port. If the server is a Racket program, it can obtain ports to communicate to the client with `tcp-accept`. These ports are placed into the management of the current custodian (see §14.7 "Custodians").

Initially, the returned input port is block-buffered, and the returned output port is block-buffered. Change the buffer mode using `file-stream-buffer-mode`.

Both of the returned ports must be closed to terminate the TCP connection. When both ports are still open, closing the output port with `close-output-port` sends a TCP close to the server (which is seen as an end-of-file if the server reads the connection through a port). In contrast, `tcp-abandon-port` (see below) closes the output port, but does not send a TCP close until the input port is also closed.

Note that the TCP protocol does not support a state where one end is willing to send but not read, nor does it include an automatic message when one end of a connection is fully closed. Instead, the other end of a connection discovers that one end is fully closed only as a response to sending data; in particular, some number of writes on the still-open end may appear to succeed, though writes will eventually produce an error.

If a connection cannot be established by `tcp-connect`, the `exn:fail:network` exception is raised.
Like `tcp-connect`, but breaking is enabled (see §10.6 “Breaks”) while trying to connect. If breaking is disabled when `tcp-connect/enable-break` is called, then either ports are returned or the `exn:break` exception is raised, but not both.

```
(tcp-connect/enable-break hostname port-no [local-hostname] local-port-no)
```

→ input-port? output-port?

 Like `tcp-connect`, but breaking is enabled (see §10.6 “Breaks”) while trying to accept a connection. If breaking is disabled when `tcp-accept/enable-break` is called, then either ports are returned or the `exn:break` exception is raised, but not both.

```
(tcp-accept/enable-break listener) → input-port? output-port?
```

→ input-port? output-port?

Like `tcp-accept`, but breaking is enabled (see §10.6 “Breaks”) while trying to accept a connection. If breaking is disabled when `tcp-accept/enable-break` is called, then either ports are returned or the `exn:break` exception is raised, but not both.

```
(tcp-accept-ready? listener) → boolean?
```

→ boolean?

Tests whether an unaccepted client has connected to the server associated with `listener`. If a client is waiting, the return value is `#t`, otherwise it is `#f`. A client is accepted with the `tcp-accept` procedure, which returns ports for communicating with the client and removes the client from the list of unaccepted clients.
If the listener has been closed, the `exn:fail:network` exception is raised.

```scheme
(tcp-close listener) → void?
listener : tcp-listener?
```

Shuts down the server associated with `listener`. All unaccepted clients receive an end-of-file from the server; connections to accepted clients are unaffected.

If the listener has already been closed, the `exn:fail:network` exception is raised.

The listener’s port number may not become immediately available for new listeners (with the default `reuse?` argument of `tcp-listen`). For further information, see Stevens’s explanation of the `TIME_WAIT` TCP state.

```scheme
(tcp-listener? v) → boolean?
v : any/c
```

Returns `#t` if `v` is a TCP listener created by `tcp-listen`, `#f` otherwise.

```scheme
(tcp-accept-evt listener) → evt?
listener : tcp-listener?
```

Returns a synchronizable event (see §11.2.1 “Events”) that is ready for synchronization when `tcp-accept` on `listener` would not block. The synchronization result is a list of two items, which correspond to the two results of `tcp-accept`. (If the event is not chosen in a `sync`, no connections are accepted.) The ports are placed into the management of the custodian that is the current custodian (see §14.7 “Custodians”) at the time that `tcp-accept-evt` is called.

```scheme
(tcp-abandon-port tcp-port) → void?
tcp-port : tcp-port?
```

Like `close-output-port` or `close-input-port` (depending on whether `tcp-port` is an input or output port), but if `tcp-port` is an output port and its associated input port is not yet closed, then the other end of the TCP connection does not receive a TCP close message until the input port is also closed.

The TCP protocol does not include a “no longer reading” state on connections, so `tcp-abandon-port` is equivalent to `close-input-port` on input TCP ports.

```scheme
(tcp-addresses tcp-port [port-numbers?])
→ (or/c (values string? string?)
   (values string? port-number?
    string? listen-port-number?))
tcp-port : (or/c tcp-port? tcp-listener? udp?)
port-numbers? : any/c = #f
```
Returns two strings when `port-numbers?` is `#f` (the default). The first string is the Internet address for the local machine as viewed by the given TCP port’s connection, for the TCP listener, or the UDP socket. (When a machine serves multiple addresses, as it usually does if you count the loopback device, the result is connection-specific or listener-specific.) If a listener or UDP socket is given and it has no specific host, the first string result is "0.0.0.0". The second string is the Internet address for the other end of the connection, or always "0.0.0.0" for a listener or unconnected UDP socket.

If `port-numbers?` is true, then four results are returned: a string for the local machine’s address, an exact integer between 1 and 65535 for the local machine’s port number, a string for the remote machine’s address, and an exact integer between 1 and 65535 for the remote machine’s port number or 0 for a listener.

If the given port, listener, or socket has been closed, the `exn:fail:network` exception is raised.

```
(tcp-port? v) → boolean?
  v : any/c
```

Returns `#t` if `v` is a TCP port—which is a port returned by `tcp-accept`, `tcp-connect`, `tcp-accept/enable-break`, or `tcp-connect/enable-break`—`#f` otherwise.

```
port-number? : contract?
```

Equivalent to `(integer-in 1 65535)`.

Added in version 6.3 of package `base`.

```
listen-port-number? : contract?
```

Equivalent to `(integer-in 0 65535)`.

Added in version 6.3 of package `base`.

### 15.3.2 UDP

```
(require racket/udp)  package:base
```

The bindings documented in this section are provided by the `racket/udp` and `racket` libraries, but not `racket/base`.

For information about UDP in general, see *TCP/IP Illustrated, Volume 1* by W. Richard Stevens.

```
(udp-open-socket [family-hostname
  family-port-no]) → udp?
```
family-hostname : (or/c string? #f) = #f
family-port-no : (or/c port-number? #f) = #f

Creates and returns a *UDP socket* to send and receive datagrams (broadcasting is allowed). Initially, the socket is not bound or connected to any address or port.

If *family-hostname* or *family-port-no* is not #f, then the socket’s protocol family is determined from these arguments. The socket is not bound to the hostname or port number. For example, the arguments might be the hostname and port to which messages will be sent through the socket, which ensures that the socket’s protocol family is consistent with the destination. Alternately, the arguments might be the same as for a future call to `udp-bind!`, which ensures that the socket’s protocol family is consistent with the binding. If neither *family-hostname* nor *family-port-no* is non-#f, then the socket’s protocol family is IPv4.

```lisp
(udp-bind! udp-socket
    hostname-string
    port-no
    [reuse?]) → void?
```

Binds an unbound *udp-socket* to the local port number *port-no*. If *port-no* is 0 the *udp-socket* is bound to an ephemeral port, which can be determined by calling `udp-addresses`.

If *hostname-string* is #f, then the socket accepts connections to all of the listening machine’s IP addresses at *port-no*. Otherwise, the socket accepts connections only at the IP address associated with the given name. For example, providing "127.0.0.1" as *hostname-string* typically creates a listener that accepts only connections to "127.0.0.1" from the local machine.

A socket cannot receive datagrams until it is bound to a local address and port. If a socket is not bound before it is used with a sending procedure `udp-send`, `udp-send-to`, etc., the sending procedure binds the socket to a random local port. Similarly, if an event from `udp-send-evt` or `udp-send-to-evt` is chosen for a synchronization (see §11.2.1 “Events”), the socket is bound; if the event is not chosen, the socket may or may not become bound.

The binding of a bound socket cannot be changed, with one exception: on some systems, if the socket is bound automatically when sending, if the socket is disconnected via `udp-connect!`, and if the socket is later used again in a send, then the later send may change the socket’s automatic binding.

If *udp-socket* is already bound or closed, the `exn:fail:network` exception is raised.
If the `reuse?` argument is true, then `udp-bind!` will set the SO_REUSEADDR socket option before binding, permitting the sharing of access to a UDP port between many processes on a single machine when using UDP multicast.

```scheme
(udp-connect! udp-socket
  hostname-string
  port-no) → void?

udp-socket : udp?
hostname-string : (or/c string? #f)
port-no : (or/c port-number? #f)
```

Connects the socket to the indicated remote address and port if `hostname-string` is a string and `port-no` is an exact integer.

If `hostname-string` is `#f`, then `port-no` also must be `#f`, and the port is disconnected (if connected). If one of `hostname-string` or `port-no` is `#f` and the other is not, the `exn:fail:contract` exception is raised.

A connected socket can be used with `udp-send` (not `udp-send-to`), and it accepts datagrams only from the connected address and port. A socket need not be connected to receive datagrams. A socket can be connected, re-connected, and disconnected any number of times.

If `udp-socket` is closed, the `exn:fail:network` exception is raised.

```scheme
(udp-send-to udp-socket
  hostname
  port-no
  bstr
  [start-pos end-pos]) → void?

udp-socket : udp?
hostname : string?
port-no : port-number?
bstr : bytes?
start-pos : exact-nonnegative-integer? = 0
end-pos : exact-nonnegative-integer? = (bytes-length bstr)
```

Sends `(subbytes bytes start-pos end-pos)` as a datagram from the unconnected `udp-socket` to the socket at the remote machine `hostname-address` on the port `port-no`. The `udp-socket` need not be bound or connected; if it is not bound, `udp-send-to` binds it to a random local port. If the socket’s outgoing datagram queue is too full to support the send, `udp-send-to` blocks until the datagram can be queued.

If `start-pos` is greater than the length of `bstr`, or if `end-pos` is less than `start-pos` or greater than the length of `bstr`, the `exn:fail:contract` exception is raised.

If `udp-socket` is closed or connected, the `exn:fail:network` exception is raised.
Like `udp-send-to`, except that `udp-socket` must be connected, and the datagram goes to the connection target. If `udp-socket` is closed or unconnected, the `exn:fail:network` exception is raised.

Like `udp-send-to`, but never blocks; if the socket’s outgoing queue is too full to support the send, `#f` is returned, otherwise the datagram is queued and the result is `#t`.

Like `udp-send`, except that (like `udp-send-to`) it never blocks and returns `#f` or `#t`. 
Like `udp-send-to`, but breaking is enabled (see §10.6 “Breaks”) while trying to send the datagram. If breaking is disabled when `udp-send-to/enable-break` is called, then either the datagram is sent or the `exn:break` exception is raised, but not both.

```
(udp-send/enable-break  udp-socket
  bstr
  [start-pos
   end-pos])  → void?
```

Like `udp-send`, except that breaks are enabled like `udp-send-to/enable-break`.

```
(udp-receive!  udp-socket
  bstr
  [start-pos
   end-pos])  → exact-nonnegative-integer?
```

Accepts up to `end-pos-start-pos` bytes of `udp-socket`’s next incoming datagram into `bstr`, writing the datagram bytes starting at position `start-pos` within `bstr`. The `udp-socket` must be bound to a local address and port (but need not be connected). If no incoming datagram is immediately available, `udp-receive!` blocks until one is available.

Three values are returned: the number of received bytes (between 0 and `end-pos-start-pos`, a hostname string indicating the source address of the datagram, and an integer indicating the source port of the datagram. If the received datagram is longer than `end-pos-start-pos` bytes, the remainder is discarded.

If `start-pos` is greater than the length of `bstr`, or if `end-pos` is less than `start-pos` or greater than the length of `bstr`, the `exn:fail:contract` exception is raised.

```
(udp-receive!*  udp-socket
  bstr
  [start-pos
   end-pos])
  → (or/c exact-nonnegative-integer? #f)
    (or/c string? #f)
    (or/c port-number? #f)
```
Like `udp-receive!`, except that it never blocks. If no datagram is available, the three result values are all `#f`.

```scheme
(udp-receive!/enable-break udp-socket bstr [start-pos end-pos])
```

Like `udp-receive!`, but breaking is enabled (see §10.6 “Breaks”) while trying to receive the datagram. If breaking is disabled when `udp-receive!/enable-break` is called, then either a datagram is received or the `exn:break` exception is raised, but not both.

```scheme
(udp-set-receive-buffer-size! udp-socket size) → void?
```

Set the receive buffer size (SO_RCVBUF) for `udp-socket`. Using a larger buffer can minimize packet loss that can occur due to slow polling of a connection, including during a major garbage collection.

If `size` is greater than the maximum allowed by the system, the `exn:fail:network` exception is raised.

Added in version 7.1.0.11 of package `base`.

```scheme
(udp-close udp-socket) → void?
```

Closes `udp-socket`, discarding unreceived datagrams. If the socket is already closed, the `exn:fail:network` exception is raised.

```scheme
(udp? v) → boolean?
```

v : any/c
Returns #t if v is a socket created by udp-open-socket, #f otherwise.

\[(udp-bound? udp-socket) \rightarrow boolean?\]
\[udp-socket : udp?\]

Returns #t if udp-socket is bound to a local address and port, #f otherwise.

\[(udp-connected? udp-socket) \rightarrow boolean?\]
\[udp-socket : udp?\]

Returns #t if udp-socket is connected to a remote address and port, #f otherwise.

\[(udp-send-ready-evt udp-socket) \rightarrow evt?\]
\[udp-socket : udp?\]

Returns a synchronizable event (see §11.2.1 “Events”) that is in a blocking state when udp-send-to on udp-socket would block. The synchronization result is the event itself.

\[(udp-receive-ready-evt udp-socket) \rightarrow evt?\]
\[udp-socket : udp?\]

Returns a synchronizable event (see §11.2.1 “Events”) that is in a blocking state when udp-receive! on udp-socket would block. The synchronization result is the event itself.

\[(udp-send-to-evt udp-socket
  hostname
  port-no
  bstr
  [start-pos
    end-pos]) \rightarrow evt?\]
\[udp-socket : udp?\]
\[hostname : string?\]
\[port-no : port-number?\]
\[bstr : bytes?\]
\[start-pos : exact-nonnegative-integer? = 0\]
\[end-pos : exact-nonnegative-integer? = (bytes-length bstr)\]

Returns a synchronizable event. The event is in a blocking state when udp-send-to on udp-socket would block. Otherwise, if the event is chosen in a synchronization, data is sent as for (udp-send-to udp-socket hostname-address port-no bstr start-pos end-pos), and the synchronization result is #<void>. (No bytes are sent if the event is not chosen.)

\[(udp-send-evt udp-socket
  bstr
  [start-pos
    end-pos]) \rightarrow evt?\]
**udp-socket**: `udp?`

**bstr**: `bytes?`

**start-pos**: `exact-nonnegative-integer? = 0`

**end-pos**: `exact-nonnegative-integer? = (bytes-length bstr)`

Returns a synchronizable event. The event is ready for synchronization when `udp-send` on `udp-socket` would not block. Otherwise, if the event is chosen in a synchronization, data is sent as for `(udp-send-to udp-socket bstr start-pos end-pos)`, and the synchronization result is `#<void>`. (No bytes are sent if the event is not chosen.) If `udp-socket` is closed or unconnected, the `exn:fail:network` exception is raised during a synchronization attempt.

```
(udp-receive!-evt udp-socket
  bstr
  [start-pos
   end-pos]) → evt?
```

**udp-socket**: `udp?`

**bstr**: `(and/c bytes? (not immutable?))`

**start-pos**: `exact-nonnegative-integer? = 0`

**end-pos**: `exact-nonnegative-integer? = (bytes-length bstr)`

Returns a synchronizable event. The event is ready for synchronization when `udp-receive` on `udp-socket` would not block. Otherwise, if the event is chosen in a synchronization, data is received into `bstr` as for `(udp-receive! udp-socket bytes start-pos end-pos)`, and the synchronization result is a list of three values, corresponding to the three results from `udp-receive!`. (No bytes are received and the `bstr` content is not modified if the event is not chosen.)

```
(udp-addresses udp-port [port-numbers?])
→ (or/c (values string? string?)
         (values string? listen-port-number?
           string? listen-port-number?))
```

**udp-port**: `udp?`

**port-numbers?**: `any/c = #f`

Returns two strings when `port-numbers?` is `#f` (the default). The first string is the Internet address for the local machine a viewed by the given UDP socket’s connection. (For most machines, the answer corresponds to the current machine’s only Internet address, but when a machine serves multiple addresses, the result is connection-specific.) The second string is the Internet address for the other end of the connection.

If `port-numbers?` is true, then four results are returned: a string for the local machine’s address, an exact integer between 1 and 65535 for the local machine’s port number or 0 if the socket is unbound, a string for the remote machine’s address, and an exact integer between 1 and 65535 for the remote machine’s port number or 0 if the socket is unconnected.
If the given port has been closed, the `exn:fail:network` exception is raised.

```scheme
(udp-set-ttl! udp-socket ttl) → void?
  udp-socket : udp?
  ttl : byte?
(udp-ttl udp-socket) → byte?
  udp-socket : udp?
```

Sets or retrieves the current time-to-live setting of `udp-socket`.

Added in version 7.5.0.5 of package base.

```scheme
(udp-multicast-join-group! udp-socket multicast-addr? hostname) → void?
  udp-socket : udp?
  multicast-addr : string?
  hostname : (or/c string? #f)
(udp-multicast-leave-group! udp-socket multicast-addr? hostname) → void?
  udp-socket : udp?
  multicast-addr : string?
  hostname : (or/c string? #f)
```

Adds or removes `udp-socket` to a named multicast group.

The `multicast-addr` argument must be a valid IPv4 multicast IP address; for example, "224.0.0.251" is the appropriate address for the mDNS protocol. The `hostname` argument selects the interface that the socket uses to receive (not send) multicast datagrams; if `hostname` is `#f` or "0.0.0.0", the kernel selects an interface automatically.

Leaving a group requires the same `multicast-addr` and `hostname` arguments that were used to join the group.

```scheme
(udp-multicast-interface udp-socket) → string?
  udp-socket : udp?
(udp-multicast-set-interface! udp-socket hostname) → void?
  udp-socket : udp?
  hostname : (or/c string? #f)
```

Retrieves or sets the interface that `udp-socket` uses to send (not receive) multicast datagrams. If the result or `hostname` is either `#f` or "0.0.0.0", the kernel automatically selects an interface when a multicast datagram is sent.
Sets or checks whether `udp-socket` receives its own multicast datagrams: a `#t` result or a true value for `loopback?` indicates that self-receipt is enabled, and `#f` indicates that self-receipt is disabled.

Sets or retrieves the current time-to-live setting of `udp-socket`.

The time-to-live setting should almost always be 1, and it is important that this number is as low as possible. In fact, these functions seldom should be used at all. See the documentation for your platform’s IP stack.

### 15.4 Processes

```
(subprocess stdout
  stdin
  stderr [group]
  command
  arg ...) → subprocess?
```

Time-to-live settings correspond to the IP_MULTICAST_TTL setting of the socket.

```
(udp-multicast-set-loopback! udp-socket loopback?) → void?
udp-socket : udp?
loopback? : any/c
(udp-multicast-loopback? udp-socket) → boolean?
udp-socket : udp?
```

Loopback settings correspond to the IP_MULTICAST_LOOP setting of the socket.

```
(udp-multicast-set-ttl! udp-socket ttl) → void?
udp-socket : udp?
ttl : byte?
(udp-multicast-ttl udp-socket) → byte?
udp-socket : udp?
```

1182
(subprocess stdout
  stdin
  stderr
  [group]
  command
  exact
  arg)

→ subprocess?
  (or/c (and/c input-port? file-stream-port?) #f)
  (or/c (and/c output-port? file-stream-port?) #f)
  (or/c (and/c input-port? file-stream-port?) #f)
stdout : (or/c (and/c output-port? file-stream-port?) #f)
stdin : (or/c (and/c input-port? file-stream-port?) #f)
stderr : (or/c (and/c output-port? file-stream-port?) #f)
group : (or/c #f 'new subprocess)
  = (and (subprocess-group-enabled) 'new)
command : path-string?
exact : 'exact
arg : string?

Creates a new process in the underlying operating system to execute command asynchronously, providing the new process with environment variables current-environment-variables. See also system and process from racket/system.

The command argument is a path to a program executable, and the args are command-line arguments for the program. See find-executable-path for locating an executable based on the PATH environment variable. On Unix and Mac OS, command-line arguments are passed as byte strings, and string args are converted using the current locale’s encoding (see §13.1.1 “Encodings and Locales”). On Windows, command-line arguments are passed as strings, and byte strings are converted using UTF-8.

On Windows, the first arg can be replaced with 'exact, which triggers a Windows-specific behavior: the sole arg is used exactly as the command-line for the subprocess. Otherwise, on Windows, a command-line string is constructed from command and arg so that a typical Windows console application can parse it back to an array of arguments. If 'exact is provided on a non-Windows platform, the exn:fail:contract exception is raised.

When provided as a port, stdout is used for the launched process’s standard output, stdin is used for the process’s standard input, and stderr is used for the process’s standard error. All provided ports must be file-stream ports. Any of the ports can be #f, in which case a system pipe is created and returned by subprocess. The stdout argument can be 'stdout, in which case the same file-stream port or system pipe that is supplied as standard output is also used for standard error. For each port or 'stdout that is provided, no pipe is created and the corresponding returned value is #f. If stdout or stderr is a port for which port-waiting-peer? returns true, then subprocess waits for the port to become ready for writing before proceeding with the subprocess creation.

On Unix and Mac OS, subprocess creation is separate from starting the program indicated by command. In particular, if command refers to a non-existent or non-executable file, an error will be reported (via standard error and a non-0 exit code) in the subprocess, not in the creating process.

For information on the Windows command-line conventions, search for “command line parsing” at http://msdn.microsoft.com/.
If `group` is `'new`, then the new process is created as a new OS-level process group. In that case, `subprocess-kill` attempts to terminate all processes within the group, which may include additional processes created by the subprocess. See `subprocess-kill` for details. If `group` is a subprocess, then that subprocess must have been created with `'new`, and the new subprocess will be added to the group; adding to the group will succeed only on Unix and Mac OS, and only in the same cases that `subprocess-kill` would have an effect (i.e., the subprocess is not known to have terminated), otherwise it will fail silently.

The `subprocess` procedure returns four values:

- a `subprocess` value representing the created process;
- an input port piped from the process’s standard output, or `#f` if `stdout-output-port` was a port;
- an output port piped to the process’s standard input, or `#f` if `stdin-input-port` was a port;
- an input port piped from the process’s standard error, or `#f` if `stderr-output-port` was a port or `'stdout`.

**Important:** All ports returned from `subprocess` must be explicitly closed, usually with `close-input-port` or `close-output-port`.

The returned ports are file-stream ports (see §13.1.5 “File Ports”), and they are placed into the management of the current custodian (see §14.7 “Custodians”). The `exn:fail` exception is raised when a low-level error prevents the spawning of a process or the creation of operating system pipes for process communication.

The `current-subprocess-custodian-mode` parameter determines whether the subprocess itself is registered with the current custodian so that a custodian shutdown calls `subprocess-kill` for the subprocess.

A subprocess can be used as a synchronizable event (see §11.2.1 “Events”). A subprocess value is ready for synchronization when `subprocess-wait` would not block; the synchronization result of a subprocess value is the subprocess value itself.

Example:

```scheme
(define-values (sp out in err)
  (subprocess #f #f #f "'/bin/ls" "-l")
(printf "stdout: \n" (port->string out))
(printf "stderr: \n" (port->string err))
(close-input-port out)
(close-output-port in)
(close-input-port err)
(subprocess-wait sp)
```

Beware that creating a group may interfere with the job control in an interactive shell, since job control is based on process groups.

A file-stream port for communicating with a subprocess is normally a pipe with a limited capacity. Beware of creating deadlock by serializing a write to a subprocess followed by a read, while the subprocess does the same, so that both processes end up blocking on a write because the other end must first read to make room in the pipe. Beware also of waiting for a subprocess to finish without reading its output, because the subprocess may be blocked attempting to write output into a full pipe.

1184
Changed in version 6.11.0.1 of package base: Added the group argument.
Changed in version 7.4.0.5: Added waiting for a fifo without a reader as stdout and/or stderr.

```
(subprocess-wait subproc) → void?
  subproc : subprocess?
```

Blocks until the process represented by subproc terminates. The subproc value also can be used with sync and sync/timeout.

```
(subprocess-status subproc) → (or/c 'running exact-nonnegative-integer?)
  subproc : subprocess?
```

Returns 'running if the process represented by subproc is still running, or its exit code otherwise. The exit code is an exact integer, and 0 typically indicates success. If the process terminated due to a fault or signal, the exit code is non-zero.

```
(subprocess-kill subproc force?) → void?
  subproc : subprocess?
  force? : any/c
```

Terminates the subprocess represented by subproc. The precise action depends on whether force? is true, whether the process was created in its own group by setting the subprocess-group-enabled parameter to a true value, and the current platform:

- force? is true, not a group, all platforms: Terminates the process if the process still running.
- force? is false, not a group, on Unix or Mac OS: Sends the process an interrupt signal instead of a kill signal.
- force? is false, not a group, on Windows: No action is taken.
- force? is true, a group, on Unix or Mac OS: Terminates all processes in the group, but only if subprocess-status has never produced a non-'running result for the subprocess and only if functions like subprocess-wait and sync have not detected the subprocess’s completion. Otherwise, no action is taken (because the immediate process is known to have terminated while the continued existence of the group is unknown).
- force? is true, a group, on Windows: Terminates the process if the process still running.
- force? is false, a group, on Unix or Mac OS: The same as when force? is #t, but when the group is sent a signal, it is an interrupt signal instead of a kill signal.
• force? is false, a group, on Windows: All processes in the group receive a CTRL-BREAK signal (independent of whether the immediate subprocess has terminated).

If an error occurs during termination, the exn:fail exception is raised.

```
(subprocess-pid subproc) -> exact-nonnegative-integer?
subproc : subprocess?
```

Returns the operating system’s numerical ID (if any) for the process represented by sub-
proc. The result is valid only as long as the process is running.

```
(subprocess? v) -> boolean?
 v : any/c
```

Returns #t if v is a subprocess value, #f otherwise.

```
(current-subprocess-custodian-mode)
 -> (or/c #f 'kill 'interrupt)
(current-subprocess-custodian-mode mode) -> void?
 mode : (or/c #f 'kill 'interrupt)
```

A parameter that determines whether a subprocess (as created by subprocess or wrappers like
process) is registered with the current custodian. If the parameter value is #f, then the
subprocess is not registered with the custodian—although any created ports are registered.
If the parameter value is 'kill or 'interrupt, then the subprocess is shut down through
subprocess-kill, where 'kill supplies a #t value for the force? argument and 'in-
terrupt supplies a #f value. The shutdown may occur either before or after ports created
for the subprocess are closed.

Custodian-triggered shutdown is limited by details of process handling in the host system.
For example, process and system may create an intermediate shell process to run a pro-
gram, in which case custodian-based termination shuts down the shell process and proba-
bly not the process started by the shell. See also subprocess-kill. Process groups (see
subprocess-group-enabled) can address some limitations, but not all of them.

```
(subprocess-group-enabled) -> boolean?
(subprocess-group-enabled on?) -> void?
 on? : any/c
```

A parameter that determines whether a subprocess is created as a new process group by
default. See subprocess and subprocess-kill for more information.

```
(shell-execute verb
target
 parameters
dir
 show-mode) -> #f
```

1186
verb : (or/c string? #f)
target : string?
parameters : string?
dir : path-string?
show-mode : symbol?

Performs the action specified by verb on target in Windows. For platforms other than Windows, the exn:fail:unsupported exception is raised.

For example,

(shell-execute #f "http://racket-lang.org" ""
 (current-directory) 'sw_shownormal)

Opens the Racket home page in a browser window.

The verb can be #f, in which case the operating system will use a default verb. Common verbs include "open", "edit", "find", "explore", and "print".

The target is the target for the action, usually a filename path. The file could be executable, or it could be a file with a recognized extension that can be handled by an installed application.

The parameters argument is passed on to the system to perform the action. For example, in the case of opening an executable, the parameters is used as the command line (after the executable name).

The dir is used as the current directory when performing the action.

The show-mode sets the display mode for a Window affected by the action. It must be one of the following symbols; the description of each symbol’s meaning is taken from the Windows API documentation.

- 'sw_hide or 'SW_HIDE — Hides the window and activates another window.
- 'sw_maximize or 'SW_MAXIMIZE — Maximizes the window.
- 'sw_minimize or 'SW_MINIMIZE — Minimizes the window and activates the next top-level window in the z-order.
- 'sw_restore or 'SW_RESTORE — Activates and displays the window. If the window is minimized or maximized, Windows restores it to its original size and position.
- 'sw_show or 'SW_SHOW — Activates the window and displays it in its current size and position.
- 'sw_showdefault or 'SW_SHOWDEFAULT — Uses a default.
• 'sw_showmaximized or 'SW_SHOWMAXIMIZED — Activates the window and displays it as a maximized window.

• 'sw_showminimized or 'SW_SHOWMINIMIZED — Activates the window and displays it as a minimized window.

• 'sw_showminnoactive or 'SW_SHOWMINNOACTIVE — Displays the window as a minimized window. The active window remains active.

• 'sw_showna or 'SW_SHOWNA — Displays the window in its current state. The active window remains active.

• 'sw_shownoactivate or 'SW_SHOWNOACTIVATE — Displays a window in its most recent size and position. The active window remains active.

• 'sw_shownormal or 'SW_SHOWNORMAL — Activates and displays a window. If the window is minimized or maximized, Windows restores it to its original size and position.

If the action fails, the exn:fail exception is raised. If the action succeeds, the result is #f.

In future versions of Racket, the result may be a subprocess value if the operating system did returns a process handle (but if a subprocess value is returned, its process ID will be 0 instead of the real process ID).

### 15.4.1 Simple Subprocesses

```racket
(require racket/system)  package: base
```

The bindings documented in this section are provided by the racket/system and racket libraries, but not racket/base.

```racket
(system command [#:set-pwd? set-pwd?]) → boolean?
command : (or/c string-no-nuls? bytes-no-nuls?)
set-pwd? : any/c = (member (system-type) '(unix macosx))
```

Executes a Unix, Mac OS, or Windows shell command synchronously (i.e., the call to system does not return until the subprocess has ended). The command argument is a string or byte string containing no nul characters. If the command succeeds, the return value is #t, #f otherwise.

If set-pwd? is true, then the PWD environment variable is set to the value of (current-directory) when starting the shell process.

See also current-subprocess-custodian-mode and subprocess-group-enabled, which affect the subprocess used to implement system.
The resulting process writes to (current-output-port), reads from (current-input-port), and logs errors to (current-error-port). To gather the process’s non-error output to a string, for example, use with-output-to-string, which sets current-output-port while calling the given function:

(with-output-to-string (lambda () (system "date")))

Like system, except that command is a filename that is executed directly (instead of through a shell command; see find-executable-path for locating an executable based on the PATH environment variable), and the args are the arguments. The executed file is passed the specified string arguments (which must contain no nul characters).

On Windows, the first argument after command can be 'exact, and the final arg is a complete command line. See subprocess for details.

Like system, except that the result is the exit code returned by the subprocess. A 0 result normally indicates success.
command : path-string?
exact : 'exact
arg : string?
set-pwd? : any/c = (member (system-type) '(unix macosx))

Like system*, but returns the exit code like system/exit-code.

(process command [#:set-pwd? set-pwd?])
  → (list input-port?
       output-port?
       exact-nonnegative-integer?
       input-port?
       ((or/c 'status 'wait 'interrupt 'kill) . -> . any))
command : (or/c string-no-nuls? bytes-no-nuls?)
set-pwd? : any/c = (member (system-type) '(unix macosx))

Executes a shell command asynchronously (using $@ on Unix and Mac OS, cmd on Windows). The result is a list of five values:

• an input port piped from the subprocess’s standard output,
• an output port piped to the subprocess’s standard input,
• the system process id of the subprocess,
• an input port piped from the subprocess’s standard error, and
• a procedure of one argument, either 'status, 'wait, 'interrupt, 'exit-code or 'kill:
  – 'status returns the status of the subprocess as one of 'running, 'done-ok, or 'done-error.
  – 'exit-code returns the integer exit code of the subprocess or #f if it is still running.
  – 'wait blocks execution in the current thread until the subprocess has completed.
  – 'interrupt sends the subprocess an interrupt signal on Unix and Mac OS, and takes no action on Windows. The result is #<void>.
  – 'kill terminates the subprocess and returns #<void>. Note that the immediate process created by process is a shell process that may run another program; terminating the shell process may not terminate processes that the shell starts, particularly on Windows.

Important: All three ports returned from process must be explicitly closed with close-input-port or close-output-port.

See also subprocess for notes about error handling and the limited buffer capacity of subprocess pipes.

On Unix and Mac OS, if command runs a single program, then $@ typically runs the program in such a way that it replaces $@ in the same process. For reliable and precise control over process creation, however, use process*.
If `set-pwd?` is true, then PWD is set in the same way as `system`.

See also `current-subprocess-custodian-mode` and `subprocess-group-enabled`, which affect the subprocess used to implement `process`. In particular, the `interrupt` and `kill` process-control messages are implemented via `subprocess-kill`, so they can affect a process group instead of a single process.

```
(process* command
  arg ...
  [#:set-pwd? set-pwd?]) → list?
command : path-string?
arg   : (or/c path? string-no-nuls? bytes-no-nuls?)
set-pwd? : any/c = (member (system-type) '(unix macosx))

(process* command
  exact
  arg
  [#:set-pwd? set-pwd?]) → list?
command : path-string?
exact : 'exact
arg    : string?
set-pwd? : any/c = (member (system-type) '(unix macosx))
```

Like `process`, except that `command` is a filename that is executed directly like `system*`, and the `args` are the arguments. On Windows, as for `system*`, the first `arg` can be replaced with `exact`.

```
(process/ports out
  in
  error-out
  command
  [#:set-pwd? set-pwd?]) → list?
out   : (or/c #f output-port?)
in    : (or/c #f input-port?)
error-out : (or/c #f output-port? 'stdout)
command : (or/c path? string-no-nuls? bytes-no-nuls?)
set-pwd? : any/c = (member (system-type) '(unix macosx))
```

Like `process`, except that `out` is used for the process’s standard output, `in` is used for the process’s standard input, and `error-out` is used for the process’s standard error. Any of the ports can be `#f`, in which case a system pipe is created and returned, as in `process`. If `error-out` is `stdout`, then standard error is redirected to standard output. For each port or `stdout` that is provided, no pipe is created, and the corresponding value in the returned list is `#f`. 

1191
Like `process*`, but with the port handling of `process/ports`.

The contracts of `system` and related functions may signal a contract error with references to the following functions.

```scheme
(string-no-nuls? x) → boolean?
```

Ensures that `x` is a string and does not contain "$\0000$".

```scheme
(bytes-no-nuls? x) → boolean?
```

Ensures that `x` is a byte-string and does not contain "#$0".
15.5 Logging

A logger accepts events that contain information to be logged for interested parties. A log receiver represents an interested party that receives logged events asynchronously. Each event has a topic and level of detail, and a log receiver subscribes to logging events at a certain level of detail (and lower) for a specific topic or for all topics. The levels, in increasing order of detail, are 'none, 'fatal, 'error, 'warning, 'info, and 'debug.

To help organize logged events, a logger can have a default topic and/or a parent logger. Every event reported to a logger is propagated to its parent (if any), and the event message is prefixed with the logger’s topic (if any) if the message doesn’t already have a topic. Furthermore, events that are propagated from a logger to its parent can be filtered by level and topic.

On start-up, Racket creates an initial logger that is used to record events from the core runtime system. For example, a 'debug event is reported for each garbage collection (see §1.1.6 "Garbage Collection"). For this initial logger, two log receivers are also created: one that writes events to the process’s original error output port, and one that writes events to the system log. The level of written events in each case is system-specific, and the default can be changed through command-line flags (see §18.1.4 "Command Line") or through environment variables:

• If the PLTSTDERR environment variable is defined and is not overridden by a command-line flag, it determines the level of the log receiver that propagates events to the original error port.

The environment variable’s value can be a ⟨level⟩: none, fatal, error, warning, info, or debug (from low detail to high detail); all events at the corresponding level of detail or lower are printed. After an initial ⟨level⟩, the value can contain whitespace-separated specifications of the form ⟨level⟩@⟨topic⟩, which prints events whose topics match ⟨topic⟩ only at the given ⟨level⟩ or higher (where a ⟨topic⟩ contains any character other than whitespace or @). Leading and trailing whitespace is ignored. For example, the value "error debug@GC" prints all events at the 'error level and higher, but prints events for the topic 'GC at the 'debug level and higher (which includes all levels).

The default is "error".

• If the PLTSTDOUT environment variable is defined and is not overridden by a command-line flag, it determines the level of the log receiver that propagates events to the original output port. The possible values are the same as for PLTSTDERR.

The default is "none".

• If the PLTSYSLOG environment variable is defined and is not overridden by a command-line flag, it determines the level of the log receiver that propagates events to the system log. The possible values are the same as for PLTSTDERR.

The default is "none" for Unix or "error" for Windows and Mac OS.
The `current-logger` parameter determines the current logger that is used by forms such as `log-warning`. On start-up, the initial value of this parameter is the initial logger. The run-time system sometimes uses the current logger to report events. For example, the byte-code compiler sometimes reports `warning` events when it detects an expression that would produce a run-time error if evaluated.

Changed in version 6.6.0.2 of package base: Prior to version 6.6.0.2, parsing of PLTSTDERR and PLTSYSLOG was very strict. Leading and trailing whitespace was forbidden, and anything other than exactly one space character separating two specifications was rejected.

Changed in version 6.90.0.17: Added PLTSTDOUT.

### 15.5.1 Creating Loggers

```scheme
 getLogger? v → boolean?
 v : any/c
```

Returns `#t` if `v` is a logger, `#f` otherwise.

```scheme
 (make-logger [topic
              parent
              propagate-level
              propagate-topic ...]
     ...) → logger?

 topic : (or/c symbol? #f) = #f
 parent : (or/c logger? #f) = #f
 propagate-level : log-level/c = 'debug
 propagate-topic : (or/c #f symbol?) = #f
```

Creates a new logger with an optional topic and parent.

The optional `propagate-level` and `propagate-topic` arguments constrain the events that are propagated from the new logger to `parent` (when `parent` is not `#f`) in the same way that events are described for a log receiver in `make-log-receiver`. By default, all events are propagated to `parent`.

Changed in version 6.1.1.3 of package base: Removed an optional argument to specify a notification callback, and added `propagate-level` and `propagate-topic` constraints for events to propagate.

```scheme
 (logger-name logger) → (or/c symbol? #f)

 logger : logger?
```

Reports `logger`’s default topic, if any.

```scheme
 (current-logger) → logger?
 (current-logger logger) → void?

 logger : logger?
```
A parameter that determines the current logger.

```
(define-logger id maybe-parent)
```

```
maybe-parent =
   | #:parent parent-expr
parent-expr : (or/c logger? #f)
```

Defines log-id-fatal, log-id-error, log-id-warning, log-id-info, and log-id-debug as forms like log-fatal, log-error, log-warning, log-info, and log-debug. The define-logger form also defines id-logger, which is a logger with default topic 'id that is a child of the result of parent-expr (if parent-expr does not produce #f), or of (current-logger) if parent-expr not provided; the log-id-fatal, etc. forms use this new logger. The new logger is created when define-logger is evaluated.

Changed in version 7.1.0.9 of package base: Added the #:parent option.

### 15.5.2 Logging Events

```
(log-message logger level [topic] message [data prefix-message?]) → void?
```

```
logger : logger?
level : log-level/c
[topic] message : string?
data : any/c = #f
prefix-message? : any/c = #t
```

Reports an event to logger, which in turn distributes the information to any log receivers attached to logger or its ancestors that are interested in events at level or higher.

Log receivers can filter events based on topic. In addition, if topic and prefix-message? are not #f, then message is prefixed with the topic followed by ": " before it is sent to receivers.

Changed in version 6.0.1.10 of package base: Added the prefix-message? argument.
Changed in version 7.2.0.7: Made the data argument optional.

```
(log-level? logger level [topic]) → boolean?
```

```
logger : logger?
```

1195
(level : log-level/c
topic : (or/c symbol? #f) = #f)

Reports whether any log receiver attached to logger or one of its ancestors is interested in level events (or potentially lower) for topic. If topic is #f, the result indicates whether a log receiver is interested in events at level for any topic.

Use this function to avoid work generating an event for log-message if no receiver is interested in the information; this shortcut is built into log-fatal, log-error, log-warning, log-info, log-debug, and forms bound by define-logger, however, so it should not be used with those forms.

The result of this function can change if a garbage collection determines that a log receiver is no longer accessible (and therefore that any event information it receives will never become accessible).

Changed in version 6.1.1.3 of package base: Added the topic argument.

((log-max-level logger [topic]) Þ (or/c log-level/c #f)
  logger : logger?
topic : (or/c symbol? #f) = #f)

Similar to log-level?, but reports the maximum-detail level of logging for which log-level? on logger and topic returns #t. The result is #f if log-level? with logger and topic currently returns #f for all levels.

Changed in version 6.1.1.3 of package base: Added the topic argument.

((log-all-levels logger) Þ (list/c (or/c #f log-level/c)
  (or/c #f symbol?)
  ...
  ...
)
  logger : logger?)

Summarizes the possible results of log-max-level on all possible interned symbols. The result list contains a sequence of symbols and #f, where the first, third, etc., list element corresponds to a level, and the second, fourth, etc., list element indicates a corresponding topic. The level is the result that log-max-level would produce for the topic, where the level for the #f topic (which is always present in the result list) indicates the result for any interned-symbol topic that does not appear in the list.

The result is suitable as a sequence of arguments to make-log-receiver (after a logger argument) to create a new receiver for events that currently have receivers in logger.

Added in version 6.1.1.4 of package base.

((log-level-evt logger) Þ evt?
  logger : logger?)
Creates a synchronizable event that is ready for synchronization when the result of \texttt{log-level?}, \texttt{log-max-level}, or \texttt{log-all-levels} can be different than before \texttt{log-level-evt} was called. The event’s synchronization result is the event itself.

The condition reported by the event is a conservative approximation: the event can become ready for synchronization even if the results of \texttt{log-level?}, \texttt{log-max-level}, and \texttt{log-all-levels} are unchanged. Nevertheless, the expectation is that events produced by \texttt{log-level-evt} become ready infrequently, because they are triggered by the creation of a log receiver.

Added in version 6.1.1.4 of package \texttt{base}.

\begin{verbatim}
(log-fatal string-expr)
(log-fatal format-string-expr v ...)
(log-error string-expr)
(log-error format-string-expr v ...)
(log-warning string-expr)
(log-warning format-string-expr v ...)
(log-info string-expr)
(log-info format-string-expr v ...)
(log-debug string-expr)
(log-debug format-string-expr v ...)
\end{verbatim}

Log an event with the current logger, evaluating \texttt{string-expr} or \texttt{(format format-string-expr v ...)} only if the logger has receivers that are interested in the event. In addition, the current continuation’s continuation marks are sent to the logger with the message string.

These form are convenient for using the current logger, but libraries should generally use a logger for a specific topic—typically through similar convenience forms generated by \texttt{define-logger}.

For each \texttt{log-level},

\begin{verbatim}
(log-level string-expr)
\end{verbatim}

is equivalent to

\begin{verbatim}
(let ([l (current-logger)])
  (when (log-level? l 'level)
    (log-message l 'level string-expr
      (current-continuation-marks)))))
\end{verbatim}

while

\begin{verbatim}
(log-level format-string-expr v ...)
\end{verbatim}
is equivalent to

```
(log-level (format format-string-expr v ...))
```

### 15.5.3 Receiving Logged Events

```
(log-receiver? v) → boolean?

v : any/c
```

Returns #t if `v` is a log receiver, #f otherwise.

```
(make-log-receiver logger level [topic ...] ...) → log-receiver?

logger : logger?
level : log-level/c
topic : (or/c #f symbol?) = #f
```

Creates a log receiver to receive events of detail `level` and lower as reported to `logger` and its descendants, as long as either `topic` is #f or the event’s topic matches `topic`.

A log receiver is a synchronizable event. It becomes ready for synchronization when a logging event is received, so use `sync` to receive a logged event. The log receiver’s synchronization result is an immutable vector containing four values: the level of the event as a symbol, an immutable string for the event message, an arbitrary value that was supplied as the last argument to `log-message` when the event was logged, and a symbol or #f for the event topic.

Multiple pairs of `level` and `topic` can be provided to indicate different specific `levels` for different `topics` (where `topic` defaults to #f only for the last given `level`). A `level` for a #f `topic` applies only to events whose topic does not match any other provided `topic`. If the same `topic` is provided multiple times, the `level` provided with the last instance in the argument list takes precedence.

### 15.5.4 Additional Logging Functions

```
(require racket/logging)  package: base
```

The bindings documented in this section are provided by the `racket/logging` library, not `racket/base` or `racket`.

```
(log-level/c v) → boolean?

v : any/c
```

Returns #t if `v` is a valid logging level (\'none, \'fatal, \'error, \'warning, \'info, or \'debug), #f otherwise.
Added in version 6.3 of package base.

```scheme
(with-intercepted-logging  
  interceptor
  proc
  [#:logger logger]
  level
  [topic ...]
  ...)
```

`interceptor` : `(-> (vector/c
  log-level/c
  string?
  any/c
  (or/c symbol? #f))
  any)

`proc` : `(-> any)

`logger` : `logger? = #f

`level` : `log-level/c

`topic` : `(or/c #f symbol?) = #f

Runs `proc`, calling `interceptor` on any log event that the execution of `proc` emits to `current-logger` at the specified levels and topics. If `#:logger` is specified, intercepts events sent to that logger, otherwise uses a new child logger of the current logger. Returns whatever `proc` returns.

Example:

```scheme
> (let ([warning-counter 0])
  (with-intercepted-logging
    (lambda (l)
      (when (eq? (vector-ref l 0) 'warning)
        (set! warning-counter (add1 warning-counter)))
    (lambda ()
      (log-warning "Warning!")
      (log-warning "Warning again!")
      (+ 2 2)
      'warning)
    warning-counter)

2
```

Added in version 6.3 of package base.

Changed in version 6.7.0.1: Added `#:logger` argument.
(with-logging-to-port  
  port  
  proc  
  [#:logger logger]  
  level  
  [topic ...]  
  ...)  
  → any

  port : output-port?
  proc : (→ any)
  logger : logger? = #f
  level : log-level/c
  topic : (or/c #f symbol?) = #f

Runs proc, outputting any logging that the execution of proc emits to current-logger at the specified levels and topics. If #:logger is specified, intercepts events sent to that logger, otherwise uses a new child logger of the current logger. Returns whatever proc returns.

Example:

> (let ([my-log (open-output-string)])
   (with-logging-to-port my-log
     (lambda ()
       (log-warning "Warning World!")
       (+ 2 2))
     'warning)
   (get-output-string my-log))
"Warning World!
"

Added in version 6.3 of package base.
Changed in version 6.7.0.1: Added #:logger argument.

### 15.6 Time

| (current-seconds) → exact-integer? |

Returns the current time in seconds since midnight UTC, January 1, 1970.

| (current-inexact-milliseconds) → real? |

Returns the current time in milliseconds since midnight UTC, January 1, 1970. The result may contain fractions of a millisecond.

Example:

> (current-inexact-milliseconds)
1289513737015.418
In this example, 1289513737015 is in milliseconds and 418 is in microseconds.

\[(\text{current-inexact-monotonic-milliseconds}) \rightarrow \text{real?}\]

Returns the number of milliseconds since an unspecified starting time. Unlike \text{current-inexact-milliseconds}, which is sensitive to the system clock and may therefore retreat or advance more quickly than real time if the system clock is adjusted, results from \text{current-inexact-monotonic-milliseconds} will always advance with real time within a Racket process, but results across processes are not comparable.

Example:

```
> (current-inexact-monotonic-milliseconds)
12772.418
```

Added in version 8.1.0.4 of package base.

\[(\text{seconds->date} \text{ secs-n} \text{ [local-time?]}) \rightarrow \text{date?}\]

\text{secs-n} : \text{real?}

\text{local-time?} : \text{any/c} = \#t

Takes \text{secs-n}, a platform-specific time in seconds returned by \text{current-seconds}, \text{file-or-directory-modify-seconds}, or 1/1000th of \text{current-inexact-milliseconds}, and returns an instance of the \text{date?} structure type. Note that \text{secs-n} can include fractions of a second. If \text{secs-n} is too small or large, the \text{exn:fail} exception is raised.

The resulting \text{date?} reflects the time according to the local time zone if \text{local-time?} is \#t, otherwise it reflects a date in UTC.

\[(\struct \text{date} \text{ (second minute hour day month year week-day year-day dst? time-zone-offset)} \#:extra-constructor-name \text{make-date} \#:transparent)\]

\text{second} : (\text{integer-in} 0 60)

\text{minute} : (\text{integer-in} 0 59)

\text{hour} : (\text{integer-in} 0 23)

\text{day} : (\text{integer-in} 1 31)

\text{month} : (\text{integer-in} 1 12)
Represent a date. The second field reaches 60 only for leap seconds. The week-day field is 0 for Sunday, 1 for Monday, etc. The year-day field is 0 for January 1, 1 for January 2, etc.; the year-day field reaches 365 only in leap years.

The dst? field is #t if the date reflects a daylight-saving adjustment. The time-zone-offset field reports the number of seconds east of UTC (GMT) for the current time zone (e.g., Pacific Standard Time is -28800), including any daylight-saving adjustment (e.g., Pacific Daylight Time is -25200). When a date record is generated by seconds->date with #f as the second argument, then the dst? and time-zone-offset fields are #f and 0, respectively.

The date constructor accepts any value for dst? and converts any non-#f value to #t.

The value produced for the time-zone-offset field tends to be sensitive to the value of the TZ environment variable, especially on Unix platforms; consult the system documentation (usually under tzset) for details.

See also the racket/date library.

```
(struct date* date (nanosecond time-zone-name)
 #:extra-constructor-name make-date*)
```

Extends date with nanoseconds and a time zone name, such as "MDT", "Mountain Daylight Time", or "UTC".

When a date* record is generated by seconds->date with #f as the second argument, then the time-zone-name field is "UTC".

The date* constructor accepts a mutable string for time-zone-name and converts it to an immutable one.

```
(current-milliseconds) → exact-integer?
```

Like current-inexact-milliseconds, but coerced to a fixnum (possibly negative). Since the result is a fixnum, the value increases only over a limited (though reasonably long) time on a 32-bit platform.

```
(current-process-milliseconds [scope]) → exact-integer?
scope : (or/c #f thread? 'subprocesses) = #f
```
Returns an amount of processor time in fixnum milliseconds that has been consumed by on the underlying operating system, including both user and system time.

- If `scope` is `#f`, the reported time is for all Racket threads and places.
- If `scope` is a thread, the result is specific to the time while the thread ran, but it may include time for other places. The more a thread synchronizes with other threads, the less precisely per-thread processor time is recorded.
- If `scope` is `subprocesses`, the result is the sum of process times for known-completed subprocesses (see §15.4 “Processes”)—and known-completed children of the subprocesses, etc., on Unix and Mac OS—across all places.

The precision of the result is platform-specific, and since the result is a fixnum, the value increases only over a limited (though reasonably long) time on a 32-bit platform.

Changed in version 6.1.1.4 of package `base`: Added `subprocesses` mode.

```
(current-gc-milliseconds) → exact-integer?
```

Returns the amount of processor time in fixnum milliseconds that has been consumed by Racket’s garbage collection so far. This time is a portion of the time reported by `(current-process-milliseconds)`, and is similarly limited.

```
(time-apply proc lst) → list?
  exact-integer?
  exact-integer?
  exact-integer?
proc : procedure?
lst : list?
```

Collects timing information for a procedure application.

Four values are returned: a list containing the result(s) of applying `proc` to the arguments in `lst`, the number of milliseconds of CPU time required to obtain this result, the number of “real” milliseconds required for the result, and the number of milliseconds of CPU time (included in the first result) spent on garbage collection.

The reliability of the timing numbers depends on the platform. If multiple Racket threads are running, then the reported time may include work performed by other threads.

```
(time body ...+)
```

Reports `time-apply`-style timing information for the evaluation of `expr` directly to the current output port. The result is the result of the last `body`. 
15.6.1 Date Utilities

```
(require racket/date)  ; package: base
```

The bindings documented in this section are provided by the `racket/date` library, not `racket/base` or `racket`.

```
(current-date) → date?
```

An abbreviation for `(seconds->date (* 0.001 (current-inexact-milliseconds)))`.

```
(date->string date [time?]) → string?
  date : date?
  time? : any/c = #f
```

Converts a date to a string. The returned string contains the time of day only if `time?`. See also `date-display-format`.

```
(date-display-format) → (or/c 'american 'chinese 'german 'indian 'irish 'iso-8601 'rfc2822 'julian)
```

```
(date-display-format format) → void?
  format : (or/c 'american 'chinese 'german 'indian 'irish 'iso-8601 'rfc2822 'julian)
```

Parameter that determines the date string format. The initial format is `american`.

```
(date->seconds date [local-time?]) → exact-integer?
  date : date?
  local-time? : any/c = #t
```

Finds the representation of a date in platform-specific seconds. If the platform cannot represent the specified date, `exn:fail` exception is raised.
The \textit{week-day}, \textit{year-day} fields of \textit{date} are ignored. The \textit{dst?} and \textit{time-zone-offset} fields of \textit{date} are also ignored; the date is assumed to be in local time by default or in UTC if \textit{local-time?} is \#f.

\begin{verbatim}
(date*->seconds date [local-time?]) → real?
  date : date?
  local-time? : any/c = #t
\end{verbatim}

Like \textit{date->seconds}, but returns an exact number that can include a fraction of a second based on \textit{(date*-nanosecond date)} if \textit{date} is a \textit{date*} instance.

\begin{verbatim}
(find-seconds second minute hour day month year [local-time?] → exact-integer?
  second : (integer-in 0 61)
  minute : (integer-in 0 59)
  hour : (integer-in 0 23)
  day : (integer-in 1 31)
  month : (integer-in 1 12)
  year : exact-nonnegative-integer?
  local-time? : any/c = #t
\end{verbatim}

Finds the representation of a date in platform-specific seconds. The arguments correspond to the fields of the \textit{date} structure—in local time by default or UTC if \textit{local-time?} is \#f. If the platform cannot represent the specified date, an error is signaled, otherwise an integer is returned.

\begin{verbatim}
(date->julian/scaliger date) → exact-integer?
  date : date?
\end{verbatim}

Converts a date structure (up to 2099 BCE Gregorian) into a Julian date number. The returned value is not a strict Julian number, but rather Scaliger's version, which is off by one for easier calculations.

\begin{verbatim}
(julian/scaliger->string date-number) → string?
  date-number : exact-integer?
\end{verbatim}

Converts a Julian number (Scaliger's off-by-one version) into a string.
The same as `date->julian/scaliger` and `julian/scaliger->string`, but misspelled.

## 15.7 Environment Variables

An environment variable set encapsulates a partial mapping from byte strings to byte strings. A Racket process’s initial environment variable set is connected to the operating system’s environment variables: accesses or changes to the set read or change operating-system environment variables for the Racket process.

Since Windows environment variables are case-insensitive, environment variable set’s key byte strings on Windows are case-folded. More precisely, key byte strings are coerced to a UTF-8 encoding of characters that are converted to lowercase via `string-locale-downcase`.

The current environment variable set, which is determined by the `current-environment-variables` parameter, is propagated to a subprocess when the subprocess is created.

```
(environment-variables? v) → boolean?
  v : any/c
```

Returns `#t` if `v` is an environment variable set, `#f` otherwise.

```
(current-environment-variables) → environment-variables?
(current-environment-variables env) → void?
  env : environment-variables?
```

A parameter that determines the environment variable set that is propagated to a subprocess and that is used as the default set for `getenv` and `putenv`.

```
(bytes-environment-variable-name? v) → boolean?
  v : any/c
```

Returns `#t` if `v` is a byte string and if it is valid for an environment variable name. An environment variable name must contain no bytes with the value 0 or 61, where 61 is `(char->integer #\=)`. On Windows, an environment variable name also must have a non-zero length.

```
(make-environment-variables name val ... ...) → environment-variables?
  name : bytes-environment-variable-name?
  val : bytes-no-nuls?
```

Creates a fresh environment variable set that is initialized with the given `name` to `val` mappings.
(environment-variables-ref env name)
→ (or/c #f (and/c bytes-no-nuls? immutable?))
env : environment-variables?
name : bytes-environment-variable-name?

Returns the mapping for name in env, returning #f if name has no mapping.

Normally, name should be a byte-string encoding of a string using the default encoding of the current locale. On Windows, name is coerced to a UTF-8 encoding and case-normalized.

(environment-variables-set! env name maybe-bstr [fail])
→ any
env : environment-variables?
name : bytes-environment-variable-name?
maybe-bstr : (or/c bytes-no-nuls? #f)
fail : (-> any) = (lambda ()
  (raise (make-exn:fail ....)))

Changes the mapping for name in env to maybe-bstr. If maybe-bstr is #f and env is the initial environment variable set of the Racket process, then the operating system environment-variable mapping for name is removed.

Normally, name and maybe-bstr should be a byte-string encoding of a string using the default encoding of the current locale. On Windows, name is coerced to a UTF-8 encoding and case-normalized, and maybe-bstr is coerced to a UTF-8 encoding if env is the initial environment variable set of the Racket process.

On success, the result of environment-variables-set! is #<void>. If env is the initial environment variable set of the Racket process, then attempting to adjust the operating system environment-variable mapping might fail for some reason, in which case fail is called in tail position with respect to the environment-variables-set!. The default fail raises an exception.

(environment-variables-names env)
→ (listof (and/c bytes-environment-variable-name? immutable?))
env : environment-variables?

Returns a list of byte strings that corresponds to names mapped by env.

(environment-variables-copy env) → environment-variables?
env : environment-variables?

Returns an environment variable set that is initialized with the same mappings as env.
(getenv name) → (or/c string-no-nuls? #f)
name : string-environment-variable-name?
(putenv name value) → boolean?
name : string-environment-variable-name?
value : string-no-nuls?

Convenience wrappers for environment-variables-ref and environment-
variables-set! that convert between strings and byte strings using the current locale’s
default encoding (using #\? as the replacement character for encoding errors) and always
using the current environment variable set from current-environment-variables. The
putenv function returns #t for success and #f for failure.

(string-environment-variable-name? v) → boolean?
v : any/c

Returns #t if v is a string and if its encoding using the current locale’s encoding is valid for
an environment variable name according to bytes-environment-variable-name?.

15.8 Environment and Runtime Information

(system-type [mode])
→ (or/c symbol? string? bytes? exact-positive-integer? vector? #f)
mode : (or/c 'os 'os* 'arch 'word 'vm 'gc 'link 'machine 'target-machine
 'so-suffix 'so-mode 'fs-change 'cross)
    = 'os

Returns information about the operating system, build mode, or machine for a running
Racket. (Installation tools should use cross-system-type, instead, to support cross-installation.)

In 'os mode, the possible symbol results are:

• 'unix
• 'windows
• 'macos

In 'os* mode, the result is similar to 'os mode, but refined to a specific operating system,
such as 'linux or 'freebsd, instead of a generic 'unix classification.

In 'arch mode, the result is a symbol representing an architecture. Possible results include
'x86_64, 'i386, 'aarch64, 'arm (32-bit), and 'ppc (32-bit).
In 'word mode, the result is either 32 or 64 to indicate whether Racket is running as a 32-bit program or 64-bit program.

In 'vm mode, the possible symbol results are (see also §1.5 "Implementations"):

- 'racket
- 'chez-scheme

In 'gc mode, the possible symbol results are (see also §1.5 "Implementations"):

- 'cgc — when (system-type 'vm) is 'racket
- '3m — when (system-type 'vm) is 'racket
- 'cs — when (system-type 'vm) is 'chez-scheme

In 'link mode, the possible symbol results are:

- 'static (Unix)
- 'shared (Unix)
- 'dll (Windows)
- 'framework (Mac OS)

Future ports of Racket may expand the list of 'os, 'os*, 'arch, 'vm, 'gc, and 'link results.

In 'machine mode, then the result is a string, which contains further details about the current machine in a platform-specific format.

In 'target-machine mode, the result is a symbol for the running Racket’s native bytecode format, or it is #f if there is no native format other than the machine-independent format. If the result is a symbol, then compile-target-machine? returns #t when applied to the symbol; see also current-compile-target-machine.

In 'so-suffix mode, then the result is a byte string that represents the file extension used for shared objects on the current platform. The byte string starts with a period, so it is suitable as a second argument to path-replace-suffix.

In 'so-mode mode, then the result is 'local if foreign libraries should be opened in “local” mode by default (as on most platforms) or 'global if foreign libraries should be opened in “global” mode.
In 'fs-change' mode, the result is an immutable vector of four elements. Each element is either #f or a symbol, where a symbol indicates the presence of a property and #f indicates the absence of a property. The possible symbols, in order, are:

- 'supported — filesystem-change-evt can produce a filesystem change event to monitor filesystem changes; if this symbol is not first in the vector, all other vector elements are #f
- 'scalable — resources consumed by a filesystem change event are effectively limited only by available memory, as opposed to file-descriptor limits; this property is #f on Mac OS and BSD variants of Unix
- 'low-latency — creation and checking of a filesystem change event is practically instantaneous; this property is #f on Linux
- 'file-level — a filesystem change event can track changes at the level of a file, as opposed to the file's directory; this property is #f on Windows

In 'cross' mode, the result reports whether cross-platform build mode has been selected (through the -C or --cross argument to racket; see §18.1.4 "Command Line"). The possible symbols are:

- 'infer — infer cross-platform mode based on whether (system-type) and (cross-system-type) report the same symbol
- 'force — use cross-platform mode, even if the current and target system types are the same, because the current and target executables can be different

Changed in version 6.8.0.2 of package base: Added 'vm mode.
Changed in version 6.9.0.1: Added 'cross mode.
Changed in version 7.1.0.6: Added 'target-machine mode.
Changed in version 7.9.0.6: Added 'os* and 'arch modes.

(system-language+country) → string?

Returns a string to identify the current user’s language and country.

On Unix and Mac OS, the string is five characters: two lowercase ASCII letters for the language, an underscore, and two uppercase ASCII letters for the country. On Windows, the string can be arbitrarily long, but the language and country are in English (all ASCII letters or spaces) separated by an underscore.

On Unix, the result is determined by checking the LC_ALL, LC_TYPE, and LANG environment variables, in that order (and the result is used if the environment variable's value starts with two lowercase ASCII letters, an underscore, and two uppercase ASCII letters, followed by either nothing or a period). On Windows and Mac OS, the result is determined by system calls.
(system-library-subpath [mode]) → path?

mode : (or/c 'cgc '3m 'cs #f) = (system-type 'gc)

Returns a relative directory path. This string can be used to build paths to system-specific files. For example, when Racket is running on Solaris on a Sparc architecture, the subpath starts "sparc-solaris", while the subpath for Windows on an i386 architecture starts "win32\i386".

The optional mode argument specifies the relevant garbage-collection variant, which one of the possible results of (system-type 'gc): 'cgc, '3m, or 'cs. It can also be #f, in which case the result is independent of the garbage-collection variant.

Installation tools should use cross-system-library-subpath, instead, to support cross-installation.

Changed in version 7.0 of package base: Added 'cs mode.

(version) → (and/c string? immutable?)

Returns an immutable string indicating the currently executing version of Racket.

(banner) → (and/c string? immutable?)

Returns an immutable string for Racket’s start-up banner text (or the banner text for an embedding program, such as GRacket). The banner string ends with a newline.

(current-command-line-arguments) → (vectorof (and/c string? immutable?))

(current-command-line-arguments argv) → void?

argv : (vectorof string?)

A parameter that is initialized with command-line arguments when Racket starts (not including any command-line arguments that were treated as flags for the system).

On Unix and Mac OS, command-line arguments are provided to the Racket process as byte strings. The arguments are converted to strings using bytes->string/locale and \uFFFD as the encoding-error character.

(current-thread-initial-stack-size) → exact-positive-integer?

(current-thread-initial-stack-size size) → void?

size : exact-positive-integer?

A parameter that provides a hint about how much space to reserve for a newly created thread’s local variables. The actual space used by a computation is affected by JIT compilation, but it is otherwise platform-independent.
(vector-set-performance-stats! results [thd]) \rightarrow void?
results : (and/c vector? (not/c immutable?))

thd : (or/c thread? #f) = #f

Sets elements in results to report current performance statistics. If thd is not #f, a particular set of thread-specific statistics are reported, otherwise a different set of global (within the current place) statistics are reported.

For global statistics, up to 12 elements are set in the vector, starting from the beginning. If results has \( n \) elements where \( n < 12 \), then the \( n \) elements are set to the first \( n \) performance-statistics values. The reported statistics values are as follows, in the order that they are set within results:

- 0: The same value as returned by current-process-milliseconds.
- 1: The same value as returned by current-milliseconds.
- 2: The same value as returned by current-gc-milliseconds.
- 3: The number of garbage collections performed since start-up within the current place.
- 4: The number of thread context switches performed since start-up.
- 5: The number of internal stack overflows handled since start-up.
- 6: The number of threads currently scheduled for execution (i.e., threads that are running, not suspended, and not unscheduled due to a synchronization).
- 7: The number of syntax objects read from compiled code since start-up.
- 8: The number of hash-table searches performed. When this counter reaches the maximum value of a fixnum, it overflows to the most negative fixnum.
- 9: The number of additional hash slots searched to complete hash searches (using double hashing). When this counter reaches the maximum value of a fixnum, it overflows to the most negative fixnum.
- 10: The number of bytes allocated for machine code that is not reported by current-memory-use.
- 11: The peak number of allocated bytes just before a garbage collection.

For thread-specific statistics, up to 4 elements are set in the vector:

- 0: #t if the thread is running, #f otherwise (same result as thread-running?).
• 1: #t if the thread has terminated, #f otherwise (same result as thread-dead?).
• 2: #t if the thread is currently blocked on a synchronizable event (or sleeping for some number of milliseconds), #f otherwise.
• 3: The number of bytes currently in use for the thread’s continuation.

Changed in version 6.1.1.8 of package base: Added vector position 11 for global statistics.

15.9 Command-Line Parsing

(require racket/cmdline) package:base

The bindings documented in this section are provided by the racket/cmdline and racket libraries, but not racket/base.

(command-line optional-name-expr optional-argv-expr
  flag-clause ...
  finish-clause)
optional-name-expr =
  | #:program name-expr

optional-argv-expr =
  | #:argv argv-expr

flag-clause = #:multi flag-spec ...
  | #:once-each flag-spec ...
  | #:once-any flag-spec ...
  | #:final flag-spec ...
  | #:usage-help string ...
  | #:help-labels string ...
  | #:ps string ...

flag-spec = (flags id ... help-spec body ...+)
  | (flags => handler-expr help-expr)

flags = flag-string
  | (flag-string ...+)

help-spec = string
  | (string-expr ...+)

finish-clause =
  | #:args arg-formals body ...
  | #:handlers handlers-exprs

arg-formals = rest-id
  | (arg ...)
  | (arg ...+ . rest-id)

arg = id
  | [id default-expr]

handlers-exprs = finish-expr arg-strings-expr
  | finish-expr arg-strings-expr help-expr
  | finish-expr arg-strings-expr help-expr help-expr
  | unknown-expr

Parses a command line according to the specification in the flag-clauses.

The name-expr, if provided, should produce a path or string to be used as the program name for reporting errors when the command-line is ill-formed. It defaults to (find-system-path 'run-file). When a path is provided, only the last element of the path is used to report an error.
The `argv-expr`, if provided, must evaluate to a list or a vector of strings. It defaults to `(current-command-line-arguments)`.

The command-line is disassembled into flags, each possibly with flag-specific arguments, followed by (non-flag) arguments. Command-line strings starting with `-` or `*` are parsed as flags, but arguments to flags are never parsed as flags, and integers and decimal numbers that start with `-` or `*` are not treated as flags. Non-flag arguments in the command-line must appear after all flags and the flags’ arguments. No command-line string past the first non-flag argument is parsed as a flag. The built-in `--` flag signals the end of command-line flags; any command-line string past the `--` flag is parsed as a non-flag argument.

A `#:multi`, `#:once-each`, `#:once-any`, or `#:final` clause introduces a set of command-line flag specifications. The clause tag indicates how many times the flag can appear on the command line:

- `#:multi` — Each flag specified in the set can be represented any number of times on the command line; i.e., the flags in the set are independent and each flag can be used multiple times.

- `#:once-each` — Each flag specified in the set can be represented once on the command line; i.e., the flags in the set are independent, but each flag should be specified at most once. If a flag specification is represented in the command line more than once, the `exn:fail` exception is raised.

- `#:once-any` — Only one flag specified in the set can be represented on the command line; i.e., the flags in the set are mutually exclusive. If the set is represented in the command line more than once, the `exn:fail` exception is raised.

- `#:final` — Like `#:multi`, except that no argument after the flag is treated as a flag. Note that multiple `#:final` flags can be specified if they have short names; for example, if `-a` is a `#:final` flag, then `-aa` combines two instances of `-a` in a single command-line argument.

A normal flag specification has four parts:

- `flags` — a flag string, or a set of flag strings. If a set of flags is provided, all of the flags are equivalent. Each flag string must be of the form `"-x"` or `"+x"` for some character `x`, or `"--x"` or `"++x"` for some sequence of characters `x`. An `x` cannot contain only digits or digits plus a single decimal point, since simple (signed) numbers are not treated as flags. In addition, the flags `"--"`, `"-h"`, and `"--help"` are predefined and cannot be changed.

- `ids` — identifier that are bound to the flag’s arguments. The number of identifiers determines how many arguments can be provided on the command line with the flag, and the names of these identifiers will appear in the help message describing the flag. The `ids` are bound to string values in the `bodys` for handling the flag.
• **help-spec** — a string or sequence of strings that describes the flag. This string is used in the help message generated by the handler for the built-in `-h` (or `--help`) flag. A single literal string can be provided, or any number of expressions that produce strings; in the latter case, strings after the first one are displayed on subsequent lines.

• **bodys** — expressions that are evaluated when one of the flags appears on the command line. The flags are parsed left-to-right, and each sequence of bodys is evaluated as the corresponding flag is encountered. When the bodys are evaluated, the preceding ids are bound to the arguments provided for the flag on the command line.

A flag specification using `=>` escapes to a more general method of specifying the handler and help strings. In this case, the handler procedure and help string list returned by `handler-expr` and `help-expr` are used as in the table argument of `parse-command-line`.

A `#:usage-help` clause inserts text lines immediately after the usage line. Each string in the clause provides a separate line of text.

A `#:help-labels` clause inserts text lines into the help table of command-line flags. Each string in the clause provides a separate line of text.

A `#:ps` clause inserts text lines at the end of the help output. Each string in the clause provides a separate line of text.

After the flag clauses, a final clause handles command-line arguments that are not parsed as flags:

• Supplying no finish clause is the same as supplying `#:args () (void)`.

• For an `#:args` finish clause, identifiers in `arg-formals` are bound to the leftover command-line strings in the same way that identifiers are bound for a `lambda` expression. Thus, specifying a single `id` (without parentheses) collects all of the leftover arguments into a list. The effective arity of the `arg-formals` specification determines the number of extra command-line arguments that the user can provide, and the names of the identifiers in `arg-formals` are used in the help string. When the command-line is parsed, if the number of provided arguments cannot be matched to identifiers in `arg-formals`, the `exn:fail` exception is raised. Otherwise, `args` clause’s bodys are evaluated to handle the leftover arguments, and the result of the last body is the result of the `command-line` expression.

• A `#:handlers` finish clause escapes to a more general method of handling the leftover arguments. In this case, the values of the expressions are used like the last two to four arguments `parse-command-line`.

Example:

```scheme
(define verbose-mode (make-parameter #f))
```
(define profiling-on (make-parameter #f))
(define optimize-level (make-parameter 0))
(define link-flags (make-parameter null))

(define file-to-compile
  (command-line
   #:program "compiler"
   #:once-each
   ["-v" "--verbose"] "Compile with verbose messages"
   (verbose-mode #t)
   ["-p" "--profile"] "Compile with profiling"
   (profiling-on #t)
   #:once-any
   ["-o" "--optimize-1"] "Compile with optimization level 1"
   (optimize-level 1))
   ["--optimize-2" (; show help on separate lines
    "Compile with optimization level 2,"
    "which includes all of level 1")
    (optimize-level 2)]
   #:multi
   ["-l" "--link-flags"] lf ; flag takes one argument
   "Add a flag <lf> for the linker"
   (link-flags (cons lf (link-flags)))
   #:args (filename) ; expect one command-line argument: <filename>
   ; return the argument as a filename to compile
   filename))

(parse-command-line name
  argv
  table
  finish-proc
  arg-help-strs
  [help-proc
    unknown-proc]) -> any
name : (or/c string? path?)
argv : (or/c (listof string?) (vectorof string?))
table : (listof (cons/c symbol? list?))
finish-proc : ((list?) () #:rest list? . ->* . any)
arg-help-strs : (listof string?)
help-proc : (string? . -> . any) = (lambda (str) ....)
unknown-proc : (string? . -> . any) = (lambda (str) ...)

Parses a command-line using the specification in table. For an overview of command-line parsing, see the command-line form, which provides a more convenient notation for most purposes.
The *table* argument to this procedural form encodes the information in *command-line*’s clauses, except for the *args* clause. Instead, arguments are handled by the *finish-proc* procedure, and help information about non-flag arguments is provided in *arg-help-strs*. In addition, the *finish-proc* procedure receives information accumulated while parsing flags. The *help-proc* and *unknown-proc* arguments allow customization that is not possible with *command-line*.

When there are no more flags, *finish-proc* is called with a list of information accumulated for command-line flags (see below) and the remaining non-flag arguments from the command-line. The arity of *finish-proc* determines the number of non-flag arguments accepted and required from the command-line. For example, if *finish-proc* accepts either two or three arguments, then either one or two non-flag arguments must be provided on the command-line. The *finish-proc* procedure can have any arity (see *procedure-arity*) except 0 or a list of 0s (i.e., the procedure must at least accept one or more arguments).

The *arg-help-strs* argument is a list of strings identifying the expected (non-flag) command-line arguments, one for each argument. If an arbitrary number of arguments are allowed, the last string in *arg-help-strs* represents all of them.

The *help-proc* procedure is called with a help string if the -h or --help flag is included on the command line. If an unknown flag is encountered, the *unknown-proc* procedure is called just like a flag-handling procedure (as described below); it must at least accept one argument (the unknown flag), but it may also accept more arguments. The default *help-proc* displays the string and exits and the default *unknown-proc* raises the *exn:fail* exception.

A *table* is a list of flag specification sets. Each set is represented as a pair of two items: a mode symbol and a list of either help strings or flag specifications. A mode symbol is one of *once-each*, *once-any*, *multi*, *final*, *help-labels*, *usage-help*, or *ps* with the same meanings as the corresponding clause tags in *command-line*. For the *help-labels*, *usage-help* or *ps* mode, a list of help strings is provided. For the other modes, a list of flag specifications is provided, where each specification maps a number of flags to a single handler procedure. A specification is a list of three items:

- A list of strings for the flags defined by the spec. See *command-line* for information about the format of flag strings.
- A procedure to handle the flag and its arguments when one of the flags is found on the command line. The arity of this handler procedure determines the number of arguments consumed by the flag: the handler procedure is called with a flag string plus the next few arguments from the command line to match the arity of the handler procedure. The handler procedure must accept at least one argument to receive the flag. If the handler accepts arbitrarily many arguments, all of the remaining arguments are passed to the handler. A handler procedure’s arity must either be a number or an *arity-at-least* value.

The return value from the handler is added to a list that is eventually passed to *finish-proc*. If the handler returns *<void>* , no value is added onto this list. For
all non-\texttt{void} values returned by handlers, the order of the values in the list is the same as the order of the arguments on the command-line.

• A non-empty list for constructing help information for the spec. The first element of the list describes the flag: it can be a string or a non-empty list of strings, and in the latter case, each string is shown on its own line. Additional elements of the main list must be strings to name the expected arguments for the flag. The number of extra help strings provided for a spec must match the number of arguments accepted by the spec’s handler procedure.

The following example is the same as the core example for command-line, translated to the procedural form:

\begin{verbatim}
(parse-command-line "compile" (current-command-line-arguments)
  `(once-each
    `(["-v" "--verbose"]
      , (lambda (flag) (verbose-mode #t))
      ("Compile with verbose messages")
    )
    `(["-p" "--profile"]
      , (lambda (flag) (profiling-on #t))
      ("Compile with profiling")
    )
  )
  (once-any
    `(["-o" "--optimize-1"]
      , (lambda (flag) (optimize-level 1))
      ("Compile with optimization level 1")
    )
    `(["--optimize-2"]
      , (lambda (flag) (optimize-level 2))
      ("Compile with optimization level 2,"
        "which implies all optimizations of level 1")
    )
  )
  (multi
    `(["-l" "--link-flags"]
      , (lambda (flag lf) (link-flags (cons lf (link-flags))))
      ("Add a flag <lf> for the linker" "lf")
    )
    (lambda (flag-accum file) file)
      ("filename")
  )
)
\end{verbatim}

15.10 Additional Operating System Functions

\begin{verbatim}
(require racket/os) package: base
\end{verbatim}

The \texttt{racket/os} library additional functions for querying the operating system.

Added in version 6.3 of package base.

\begin{verbatim}
| (gethostname) → string?
\end{verbatim}
Returns a string for the current machine’s hostname (including its domain).

```
(getpid) ➞ exact-integer?
```

Returns an integer identifying the current process within the operating system.
16 Memory Management

16.1 Weak Boxes

A weak box is similar to a normal box (see §4.13 “Boxes”), but when the garbage collector (see §1.1.6 “Garbage Collection”) can prove that the content value of a weak box is only reachable via weak references, the content of the weak box is replaced with #f. A weak reference is a reference through a weak box, through a key reference in a weak hash table (see §4.14 “Hash Tables”), through a value in an ephemeron where the value can be replaced by #f (see §16.2 “Ephemerons”), or through a custodian (see §14.7 “Custodians”).

\[(\text{make-weak-box } v) \to \text{weak-box?}\]
\[v : \text{any/c}\]

Returns a new weak box that initially contains \(v\).

\[(\text{weak-box-value weak-box } [\text{gcd}-v]) \to \text{any/c}\]
\[\text{weak-box} : \text{weak-box?}\]
\[\text{gcd}-v : \text{any/c} = \#f\]

Returns the value contained in \text{weak-box}. If the garbage collector has proven that the previous content value of \text{weak-box} was reachable only through a weak reference, then \text{gcd}-v (which defaults to \#f) is returned.

\[(\text{weak-box? } v) \to \text{boolean?}\]
\[v : \text{any/c}\]

Returns \#t if \(v\) is a weak box, \#f otherwise.

16.2 Ephemerons

An ephemeron [Hayes97] is a generalization of a weak box (see §16.1 “Weak Boxes”). Instead of just containing one value, an ephemeron holds two values: one that is considered the value of the ephemeron and another that is the ephemeron’s key. Like the value in a weak box, the value in an ephemeron may be replaced by #f, but when the key is no longer reachable (except possibly via weak references) instead of when the value is no longer reachable.

As long as an ephemeron’s value is retained, the reference is considered a non-weak reference. References to the key via the value are treated specially, however, in that the reference does not necessarily count toward the key’s reachability. A weak box can be seen as a specialization of an ephemeron where the key and value are the same.
One particularly common use of ephemerons is to combine them with a weak hash table (see §4.14 “Hash Tables”) to produce a mapping where the memory manager can reclaim key–value pairs even when the value refers to the key; see `make-ephemeron-hash`. A related use is to retain a reference to a value as long as any value for which it is an impersonator is reachable; see `impersonator-ephemeron`.

More precisely,

- the value in an ephemeron is replaced by `#f` when the automatic memory manager can prove that either the ephemeron or the key is reachable only through weak references (see §16.1 “Weak Boxes”); and
- nothing reachable from the value in an ephemeron counts toward the reachability of an ephemeron key (whether for the same ephemeron or another), unless the same value is reachable through a non-weak reference, or unless the value’s ephemeron key is reachable through a non-weak reference (see §16.1 “Weak Boxes” for information on weak references).

```
(make-ephemeron key v) → ephemeron?
key : any/c
v : any/c
```

Returns a new ephemeron whose key is `key` and whose value is initially `v`.

```
(ephemeron-value ephemeron [gced-v retain-v]) → any/c
ephemeron : ephemeron?
gced-v : any/c = #f
retain-v : any/c = #f
```

Returns the value contained in `ephemeron`. If the garbage collector has proven that the key for `ephemeron` is only weakly reachable, then the result is `gced-v` (which defaults to `#f`).

The `retain-v` argument is retained as reachable until the ephemeron’s value is extracted. It is useful, for example, when `ephemeron` was obtained through a weak, `eq?`-based mapping from `key` and `ephemeron` was created with `key` as the key; in that case, supplying `key` as `retain-v` ensures that `ephemeron` retains its value long enough for it to be extracted, even if `key` is otherwise unreachable.

Changed in version 7.1.0.10 of package `base`: Added the `retain-v` argument.

```
(ephemeron? v) → boolean?
v : any/c
```

Returns `#t` if `v` is an ephemeron, `#f` otherwise.
16.3 Wills and Executors

A will executor manages a collection of values and associated will procedures (a.k.a. finalizers). The will procedure for each value is ready to be executed when the value has been proven (by the garbage collector) to be unreachable, except through weak references (see §16.1 “Weak Boxes”) or as the registrant for other will executors. A will is useful for triggering clean-up actions on data associated with an unreachable value, such as closing a port embedded in an object when the object is no longer used.

Calling the will-execute or will-try-execute procedure executes a will that is ready in the specified will executor. A will executor is also a synchronizable event, so sync or sync/timeout can be used to detect when a will executor has ready wills. Wills are not executed automatically, because certain programs need control to avoid race conditions. However, a program can create a thread whose sole job is to execute wills for a particular executor.

If a value is registered with multiple wills (in one or multiple executors), the wills are readied in the reverse order of registration. Since readying a will procedure makes the value reachable again, the will must be executed and the value must be proven again unreachable through only weak references before another of the wills is readied or executed. However, wills for distinct unreachable values are readied at the same time, regardless of whether the values are reachable from each other.

A will executor’s registrant is held non-weakly until after the corresponding will procedure is executed. Thus, if the content value of a weak box (see §16.1 “Weak Boxes”) is registered with a will executor, the weak box’s content is not changed to #f until all wills have been executed for the value and the value has been proven again reachable through only weak references.

A will executor can be used as a synchronizable event (see §11.2.1 “Events”). A will executor is ready for synchronization when will-execute would not block; the synchronization result of a will executor is the will executor itself.

These examples show how to run cleanup actions when no synchronization is necessary. It simply runs the registered executors as they become ready in another thread.

Examples:

```scheme
> (define an-executor (make-will-executor))
> (void
   (thread
    (λ ()
     (let loop ()
      (will-execute an-executor)
      (loop))))))
> (define (executor-proc v) (printf "a-box is now garbage\n"))
> (define a-box-to-track (box #f))
```
> (will-register an-executor a-box-to-track executor-proc)
> (collect-garbage)
> (set! a-box-to-track #f)
> (collect-garbage)
> a-box is now garbage

\textbf{(make-will-executor)} \rightarrow \text{will-executor?}

Returns a new will executor with no managed values.

\textbf{(will-executor? v)} \rightarrow \text{boolean?}
\hspace{1cm} v : \text{any/c}

Returns \#t if \(v\) is a will executor, \#f otherwise.

\textbf{(will-register executor v proc)} \rightarrow \text{void?}
\hspace{1cm} executor : \text{will-executor?}
\hspace{1cm} v : \text{any/c}
\hspace{1cm} proc : (\text{any/c} \rightarrow \text{. any})

Registers the value \(v\) with the will procedure \(\text{proc}\) in the will executor \(\text{executor}\). When \(v\) is proven unreachable, then the procedure \(\text{proc}\) is ready to be called with \(v\) as its argument via \text{will-execute} or \text{will-try-execute}. The \(\text{proc}\) argument is strongly referenced until the will procedure is executed.

\textbf{(will-execute executor)} \rightarrow \text{any}
\hspace{1cm} executor : \text{will-executor?}

Invokes the will procedure for a single “unreachable” value registered with the executor \(\text{executor}\). The values returned by the will procedure are the result of the \text{will-execute} call. If no will is ready for immediate execution, \text{will-execute} blocks until one is ready.

\textbf{(will-try-execute executor [v])} \rightarrow \text{any}
\hspace{1cm} executor : \text{any/c}
\hspace{1cm} v : \text{any/c} = \#f

Like \text{will-execute} if a will is ready for immediate execution. Otherwise, \(v\) is returned.

Changed in version 6.90.0.4 of package \text{base}: Added the \(v\) argument.

### 16.4 Garbage Collection

Set the PLTDISABLEGC environment variable (to any value) before Racket starts to disable garbage collection. Set the PLT_INCREMENTAL_GC environment variable to a value
that starts with \texttt{1, y,} or \texttt{Y} to request incremental mode at all times, but calling \texttt{(collect-garbage 'incremental)} in a program with a periodic task is generally a better mechanism for requesting incremental mode. Set the PLT\_INCREMENTAL\_GC environment variable to a value that starts with \texttt{0, n,} or \texttt{N} to disable incremental-mode requests.

Each garbage collection logs a message (see §15.5 "Logging") at the \texttt{debug} level with topic \texttt{GC}. In the CS and 3m implementations of Racket, ‘major’ collections are also logged at the \texttt{debug} level with the topic \texttt{GC:major}. In the CS and 3m implementations of Racket, the data portion of the message is an instance of a \texttt{gc-info} prefab structure type with 10 fields as follows, but future versions of Racket may use a \texttt{gc-info} prefab structure with additional fields:

\begin{verbatim}
(struct gc-info (mode pre-amount pre-admin-amount code-amount
    post-amount post-admin-amount
    start-process-time end-process-time
    start-time end-time)
    #:prefab)
\end{verbatim}

- The \texttt{mode} field is a symbol \texttt{'major}, \texttt{'minor}, or \texttt{'incremental}; \texttt{'major} indicates a collection that inspects all memory, \texttt{'minor} indicates collection that mostly inspects just recent allocations, and \texttt{'incremental} indicates a minor collection that performs extra work toward the next major collection.

  Changed in version 6.3.0.7 of package base: Changed first field from a boolean \texttt{(#t for 'major, #f for 'minor)} to a mode symbol.

- The \texttt{pre-amount} field reports place-local memory use (i.e., not counting the memory use of child places) in bytes at the time that the garbage collection started. Additional bytes registered via \texttt{make-phantom-bytes} are included.

- The \texttt{pre-admin-amount} is a larger number that includes memory use for the garbage collector’s overhead, such as space on memory pages that are mapped but not currently used.

- The \texttt{code-amount} field reports additional memory use for generated native code (which is the same just before and after a garbage collection, since it is released via finalization).

- The \texttt{post-amount} and \texttt{post-admin-amount} fields correspond to \texttt{pre-amount} and \texttt{pre-admin-amount}, but after garbage collection. In typical configurations, the difference between \texttt{post-amount} and \texttt{pre-amount} contributes to \texttt{post-admin-amount}, since reclaimed pages tend to stay in reserve with the expectation that they’ll be needed again (but the pages are released if multiple collections pass without need for the pages).

- The \texttt{start-process-time} and \texttt{end-process-time} fields report processor time (in the sense of \texttt{current-process-milliseconds}) at the start and end of garbage collection. The difference between the times is the processor time consumed by collection.
• The start-time and end-time fields report real time (in the sense of current-inexact-milliseconds) at the start and end of garbage collection. The difference between the times is the real time consumed by garbage collection.

The format of the logged message’s text is subject to change. Currently, after a prefix that indicates the place and collection mode, the text has the format

\[
\langle \text{used} \rangle (\langle \text{admin} \rangle) [\langle \text{code} \rangle]; \ \text{free} \ \langle \text{reclaimed} \rangle (\langle \text{adjust} \rangle) \ \langle \text{elapsed} \rangle \ \text{@} \ \langle \text{timestamp} \rangle
\]

- \langle \text{used} \rangle: Collectable memory in use just prior to garbage collection
- \langle \text{admin} \rangle: Additional memory used as to manage collectable memory
- \langle \text{code} \rangle: Additional memory used for generated machine code
- \langle \text{reclaimed} \rangle: Collectable memory reclaimed by garbage collection
- \langle \text{adjust} \rangle: Negation of change to administrative memory minus \langle \text{reclaimed} \rangle
- \langle \text{elapsed} \rangle: Processor time used to perform garbage collection
- \langle \text{timestamp} \rangle: Processor time since startup of garbage collection’s start

Changed in version 6.3.0.7 of package base: Added PLT_INCREMENTAL_GC.

Changed in version 7.6.0.9: Added major-collection logging for the topic ‘GC:major.

\[
\text{(collect-garbage [request])} \rightarrow \text{void}?
\]

request : (or/c 'major 'minor 'incremental) = 'major

Requests an immediate garbage collection or requests a garbage-collection mode, depending on request:

• 'major — Forces a “major” collection, which inspects all memory. Some effectively unreachable data may remain uncollected, because the collector cannot prove that it is unreachable.

  This mode of \text{collect-garbage} procedure provides some control over the timing of collections, but garbage will obviously be collected even if this procedure is never called—unless garbage collection is disabled by setting PLTDISABLEGC.

• 'minor — Requests a “minor” collection, which mostly inspects only recent allocations. If minor collection is not supported (e.g., when \text{(system-type 'gc) returns 'cgc}) or if the next collection must be a major collection, no collection is performed. More generally, minor collections triggered by \text{(collect-garbage 'minor) do not cause major collections any sooner than they would occur otherwise.

• 'incremental — Does not request an immediate collection, but requests extra effort going forward to avoid major collections, even if it requires more work per minor collection to incrementally perform the work of a major collection. This incremental-mode request expires at the next major collection.
The intent of incremental mode is to significantly reduce pause times due to major collections, but incremental mode may imply longer minor-collection times and higher memory use. Currently, incremental mode is only meaningful for CS and 3m Racket implementations; it has no effect in other Racket implementations.

If the PLT_INCREMENTAL_GC environment variable’s value starts with 0, n, or N on start-up, then incremental-mode requests are ignored.

#### current-memory-use

```
(current-memory-use [mode]) → exact-nonnegative-integer?
```

Returns information about memory use:

- If `mode` is `#f` (the default), the result is an estimate of the number of bytes reachable from any custodian.
- If `mode` is `'cumulative`, returns an estimate of the total number of bytes allocated since start up, including bytes that have since been reclaimed by garbage collection.
- If `mode` is a custodian, returns an estimate of the number of bytes of memory occupied by reachable data from `mode`. This estimate is calculated by the last garbage collection, and can be 0 if none occurred (or if none occurred since the given custodian was created). The `current-memory-use` function does not perform a collection by itself; doing one before the call will generally decrease the result (or increase it from 0 if no collections happened yet).

When Racket is compiled without support for memory accounting, the estimate is the same as when `mode` is `#f` (i.e., all memory) for any individual custodian. See also `custodian-memory-accounting-available?`.

See also `vector-set-performance-stats!`.

#### dump-memory-stats

```
(dump-memory-stats v ...) → any
```

Dumps information about memory usage to the low-level error port or console.

Various combinations of `v` arguments can control the information in a dump. The information that is available depends on your Racket build; check the end of a dump from a particular build to see if it offers additional information; otherwise, all `v`s are ignored.
16.5 Phantom Byte Strings

A *phantom byte string* is a small Racket value that is treated by the Racket memory manager as having an arbitrary size, which is specified when the phantom byte string is created or when it is changed via `set-phantom-bytes!`.

A phantom byte string acts as a hint to Racket’s memory manager that memory is allocated within the process but through a separate allocator, such as through a foreign library that is accessed via `ffi/unsafe`. This hint is used to trigger garbage collections or to compute the result of `current-memory-use`.

```
(phantom-bytes? v) → boolean?
 v : any/c
```

Returns `#t` if `v` is a phantom byte string. `#f` otherwise.

```
(make-phantom-bytes k) → phantom-bytes?
 k : exact-nonnegative-integer?
```

Creates a phantom byte string that is treated by the Racket memory manager as being `k` bytes in size. For a large enough `k`, the `exn:fail:out-of-memory` exception is raised—either because the size is implausibly large, or because a memory limit has been installed with `custodian-limit-memory`.

```
(set-phantom-bytes! phantom-bstr k) → phantom-bytes?
 phantom-bstr : phantom-bytes?
 k : exact-nonnegative-integer?
```

Adjusts the size of a phantom byte string as it is treated by the Racket memory manager.

For example, if the memory that `phantom-bstr` represents is released through a foreign library, then `(set-phantom-bytes! phantom-bstr 0)` can reflect the change in memory use.

When `k` is larger than the current size of `phantom-bstr`, then this function can raise `exn:fail:out-of-memory`, like `make-phantom-bytes`. 
17 Unsafe Operations

(require racket/unsafe/ops)  package: base

All functions and forms provided by racket/base and racket check their arguments to ensure that the arguments conform to contracts and other constraints. For example, vector-ref checks its arguments to ensure that the first argument is a vector, that the second argument is an exact integer, and that the second argument is between 0 and one less than the vector’s length, inclusive.

Functions provided by racket/unsafe/ops are unsafe. They have certain constraints, but the constraints are not checked, which allows the system to generate and execute faster code. If arguments violate an unsafe function’s constraints, the function’s behavior and result is unpredictable, and the entire system can crash or become corrupted.

All of the exported bindings of racket/unsafe/ops are protected in the sense of protect-out, so access to unsafe operations can be prevented by adjusting the code inspector (see §14.10 “Code Inspectors”).

17.1 Unsafe Numeric Operations

(unsafe-fx+ a ...) → fixnum?
  a : fixnum?
(unsafe-fx- a b ...) → fixnum?
  a : fixnum?
  b : fixnum?
(unsafe-fx* a ...) → fixnum?
  a : fixnum?
(unsafe-fxquotient a b) → fixnum?
  a : fixnum?
  b : fixnum?
(unsafe-fxremainder a b) → fixnum?
  a : fixnum?
  b : fixnum?
(unsafe-fxmodulo a b) → fixnum?
  a : fixnum?
  b : fixnum?
(unsafe-fxabs a) → fixnum?
  a : fixnum?

For fixnums: Unchecked versions of fx+, fx-, fx*, fxquotient, fxremainder, fxmodulo, and fxabs.

Changed in version 7.0.0.13 of package base: Allow zero or more arguments for unsafe-fx+ and unsafe-fx*
and allow one or more arguments for `unsafe-fx-`.

```scheme
(unsafe-fxand a ...) → fixnum?
  a : fixnum?
(unsafe-fxior a ...) → fixnum?
  a : fixnum?
(unsafe-fxxor a ...) → fixnum?
  a : fixnum?
(unsafe-fxnot a) → fixnum?
  a : fixnum?
(unsafe-fxlshift a b) → fixnum?
  a : fixnum?
  b : fixnum?
(unsafe-fxrshift a b) → fixnum?
  a : fixnum?
  b : fixnum?
```

For fixnums: Unchecked versions of `fxand`, `fxior`, `fxxor`, `fxnot`, `fxlshift`, and `fxrshift`.

Changed in version 7.0.0.13 of package base: Allow zero or more arguments for `unsafe-fxand`, `unsafe-fxior`, and `unsafe-fxxor`.

```scheme
(unsafe-fx+/wraparound a b) → fixnum?
  a : fixnum?
  b : fixnum?
(unsafe-fx-/wraparound a b) → fixnum?
  a : fixnum?
  b : fixnum?
(unsafe-fx*/wraparound a b) → fixnum?
  a : fixnum?
  b : fixnum?
(unsafe-fxlshift/wraparound a b) → fixnum?
  a : fixnum?
  b : fixnum?
(unsafe-fxrshift/wraparound a b) → fixnum?
  a : fixnum?
  b : fixnum?
```

For fixnums: Unchecked versions of `fx+/wraparound`, `fx-/wraparound`, `fx*/wraparound`, and `fxlshift/wraparound`.

Added in version 7.9.0.6 of package base.

```scheme
(unsafe-fx= a b ...) → boolean?
  a : fixnum?
  b : fixnum?
(unsafe-fx< a b ...) → boolean?
  a : fixnum?
  b : fixnum?
```
(unsafe-fx> a b ...) → boolean?
   a : fixnum?
   b : fixnum?
(unsafe-fx<= a b ...) → boolean?
   a : fixnum?
   b : fixnum?
(unsafe-fx>= a b ...) → boolean?
   a : fixnum?
   b : fixnum?
(unsafe-fxmin a b ...) → fixnum?
   a : fixnum?
   b : fixnum?
(unsafe-fxmax a b ...) → fixnum?
   a : fixnum?
   b : fixnum?

For fixnums: Unchecked versions of fx=, fx<, fx>, fx<=, fx>=, fxmin, and fxmax.

Changed in version 7.0.0.13 of package base: Allow one or more argument, instead of allowing just two.

(unsafe-fl+ a ...) → flonum?
   a : flonum?
(unsafe-fl- a b ...) → flonum?
   a : flonum?
   b : flonum?
(unsafe-fl* a ...) → flonum?
   a : flonum?
(unsafe-fl/ a b ...) → flonum?
   a : flonum?
   b : flonum?
(unsafe-flabs a) → flonum?
   a : flonum?

For flonums: Unchecked versions of fl+, fl-, fl*, fl/, and flabs.

Changed in version 7.0.0.13 of package base: Allow zero or more arguments for unsafe-fl+ and unsafe-fl* and one or more arguments for unsafe-fl- and unsafe-fl/.

(unsafe-fl= a b ...) → boolean?
   a : flonum?
   b : flonum?
(unsafe-fl< a b ...) → boolean?
   a : flonum?
   b : flonum?
(unsafe-fl> a b ...) → boolean?
   a : flonum?
   b : flonum?
(unsafe-fl<= a b ...)  →  boolean?
    a : flonum?
    b : flonum?
(unsafe-fl>= a b ...)  →  boolean?
    a : flonum?
    b : flonum?
(unsafe-flmin a b ...)  →  flonum?
    a : flonum?
    b : flonum?
(unsafe-flmax a b ...)  →  flonum?
    a : flonum?
    b : flonum?

For flonums: Unchecked versions of \( fl=, fl<, fl>, fl<=, fl>=, flmin, \) and \( flmax \).

Changed in version 7.0.0.13 of package base: Allow one or more argument, instead of allowing just two.

(unsafe-flround a)  →  flonum?
    a : flonum?
(unsafe-flfloor a)  →  flonum?
    a : flonum?
(unsafe-flceiling a)  →  flonum?
    a : flonum?
(unsafe-fltruncate a)  →  flonum?
    a : flonum?

For flonums: Unchecked (potentially) versions of \( flround, flfloor, flceiling, \) and \( fltruncate \). Currently, these bindings are simply aliases for the corresponding safe bindings.

(unsafe-flsingle a)  →  flonum?
    a : flonum?

For flonums: Unchecked (potentially) version of \( flsingle \).

Added in version 7.8.0.7 of package base.

(unsafe-flasin a)  →  flonum?
    a : flonum?
(unsafe-flcos a)  →  flonum?
    a : flonum?
(unsafe-fltan a)  →  flonum?
    a : flonum?
(unsafe-flasin a)  →  flonum?
    a : flonum?
(unsafe-flacos a)  →  flonum?
    a : flonum?
(unsafe-flatan a) → flonum?
  a : flonum?
(unsafe-fllog a) → flonum?
  a : flonum?
(unsafe-flexp a) → flonum?
  a : flonum?
(unsafe-flsqrt a) → flonum?
  a : flonum?
(unsafe-flexpt a b) → flonum?
  a : flonum?
  b : flonum?

For flonums: Unchecked (potentially) versions of flsin, flcos, fltan, flasin, flacos, flatan, fllog, flexp, flsqrt, and flexpt. Currently, some of these bindings are simply aliases for the corresponding safe bindings.

(unsafe-make-flrectangular a b)
  → (and/c complex?
      (lambda (c) (flonum? (real-part c)))
      (lambda (c) (flonum? (imag-part c))))
  a : flonum?
  b : flonum?
(unsafe-flreal-part a) → flonum?
  a : (and/c complex?
      (lambda (c) (flonum? (real-part c)))
      (lambda (c) (flonum? (imag-part c))))
(unsafe-flimag-part a) → flonum?
  a : (and/c complex?
      (lambda (c) (flonum? (real-part c)))
      (lambda (c) (flonum? (imag-part c))))

For flonums: Unchecked versions of make-flrectangular, flreal-part, and flimag-part.

(unsafe-fx->fl a) → flonum?
  a : fixnum?
(unsafe-fl->fx a) → fixnum?
  a : flonum?

Unchecked versions of fx->fl and fl->fx.

Changed in version 7.7.0.8 of package base: Changed unsafe-fl->fx to truncate.

(unsafe-flrandom rand-gen) → (and flonum? (>/c 0) (</c 1))
  rand-gen : pseudo-random-generator?

Unchecked version of flrandom.
17.2 Unsafe Character Operations

(unsafe-char=? a b ...) → boolean?
  a : char?
  b : char?
(unsafe-char<? a b ...) → boolean?
  a : char?
  b : char?
(unsafe-char>? a b ...) → boolean?
  a : char?
  b : char?
(unsafe-char<=? a b ...) → boolean?
  a : char?
  b : char?
(unsafe-char>=? a b ...) → boolean?
  a : char?
  b : char?
(unsafe-char->integer a) → fixnum?
  a : char?


Added in version 7.0.0.14 of package base.

17.3 Unsafe Compound-Data Operations

(unsafe-car p) → any/c
  p : pair?
(unsafe-cdr p) → any/c
  p : pair?
(unsafe-mcar p) → any/c
  p : mpair?
(unsafe-mcdr p) → any/c
  p : mpair?
(unsafe-set-mcar! p v) → void?
  p : mpair?
  v : any/c
(unsafe-set-mcdr! p v) → void?
  p : mpair?
  v : any/c

Unsafe variants of car, cdr, mcar, mcdr, set-mcar!, and set-mcdr!.
(unsafe-cons-list v rest) → (and/c pair? list?)
  v : any/c
  rest : list?

Unsafe variant of cons that produces a pair that claims to be a list—without checking whether rest is a list.

(unsafe-list-ref lst pos) → any/c
  lst : pair?
  pos : (and/c exact-nonnegative-integer? fixnum?)
(unsafe-list-tail lst pos) → any/c
  lst : any/c
  pos : (and/c exact-nonnegative-integer? fixnum?)

Unsafe variants of list-ref and list-tail, where pos must be a fixnum, and lst must start with at least (add1 pos) (for unsafe-list-ref) or pos (for unsafe-list-tail) pairs.

(unsafe-set-immutable-car! p v) → void?
  p : pair?
  v : any/c
(unsafe-set-immutable-cdr! p v) → void?
  p : pair?
  v : any/c

As their oxymoronic names should suggest, there is no generally correct way to use these functions. They may be useful nevertheless, as a last resort, in settings where pairs are used in a constrained way and when making correct assumptions about Racket’s implementation (including limits on the compiler’s optimizations).

Some pitfalls of using unsafe-set-immutable-car! and unsafe-set-immutable-cdr!:

- Functions that consume a pair may take advantage of immutability, such as computing a list’s length once and expecting the list to retain that length, or checking a list against a contract and expecting the contract to hold thereafter.

- The result of list? for a pair may be cached internally, so that changing the cdr of a pair from a list to a non-list or vice versa may cause list? to produce the wrong value—for the mutated pair or for another pair that reaches the mutated pair.

- The compiler may reorder or even optimize away a call to car or cdr on the grounds that pairs are immutable, in which case a unsafe-set-immutable-car! or unsafe-set-immutable-cdr! may not have an effect on the use of car or cdr.
Added in version 7.9.0.18 of package base.

(unsafe-unbox b) → fixnum?
  b : box?
(unsafe-set-box! b k) → void?
  b : box?
  k : fixnum?
(unsafe-unbox* v) → any/c
  v : (and/c box? (not/c impersonator?))
(unsafe-set-box*! v val) → void?
  v : (and/c box? (not/c impersonator?))
  val : any/c

Unsafe versions of unbox and set-box!, where the box* variants can be faster but do not work on impersonators.

(unsafe-box*-cas! loc old new) → boolean?
  loc : box?
  old : any/c
  new : any/c

Unsafe version of box-cas!. Like unsafe-set-box*, it does not work on impersonators.

(unsafe-vector-length v) → fixnum?
  v : vector?
(unsafe-vector-ref v k) → any/c
  v : vector?
  k : fixnum?
(unsafe-vector-set! v k val) → void?
  v : vector?
  k : fixnum?
  val : any/c
(unsafe-vector*-length v) → fixnum?
  v : (and/c vector? (not/c impersonator?))
(unsafe-vector*-ref v k) → any/c
  v : (and/c vector? (not/c impersonator?))
  k : fixnum?
(unsafe-vector*-set! v k val) → void?
  v : (and/c vector? (not/c impersonator?))
  k : fixnum?
  val : any/c
(unsafe-vector*-cas! v k old-val new-val) → boolean?
  v : (and/c vector? (not/c impersonator?))
  k : fixnum?
  old-val : any/c
  new-val : any/c
Unsafe versions of `vector-length`, `vector-ref`, `vector-set!`, and `vector-cas!`, where the `vector*` variants can be faster but do not work on impersonators.

A vector’s size can never be larger than a fixnum, so even `vector-length` always returns a fixnum.

Changed in version 6.11.0.2 of package `base`: Added `unsafe-vector*-cas!`.

```
(unsafe-vector*->immutable-vector! v)
  → (and/c vector? immutable?)
  v : (and/c vector? (not/c impersonator?))
```

Similar to `vector->immutable-vector`, but potentially destroys `v` and reuses it space, so `v` must not be used after calling `unsafe-vector*->immutable-vector!`.

Added in version 7.7.0.6 of package `base`.

```
(unsafe-string-length str) → fixnum?
  str : string?
(unsafe-string-ref str k)
  → (and/c char? (lambda (ch) (<= 0 (char->integer ch) 255)))
  str : string?
  k : fixnum?
(unsafe-string-set! str k ch) → void?
  str : (and/c string? (not/c immutable?))
  k : fixnum?
  ch : char?
```

Unsafe versions of `string-length`, `string-ref`, and `string-set!`. The `unsafe-string-ref` procedure can be used only when the result will be a Latin-1 character. A string’s size can never be larger than a fixnum (so even `string-length` always returns a fixnum).

```
(unsafe-string->immutable-string! str)
  → (and/c string? immutable?)
  str : string?
```

Similar to `string->immutable-string`, but potentially destroys `str` and reuses it space, so `str` must not be used after calling `unsafe-string->immutable-string!`.

Added in version 7.7.0.6 of package `base`.

```
(unsafe-bytes-length bstr) → fixnum?
  bstr : bytes?
(unsafe-bytes-ref bstr k) → byte?
  bstr : bytes?
  k : fixnum?
```
(unsafe-bytes-set! bstr k b) → void?
  bstr : (and/c bytes? (not/c immutable?))
  k : fixnum?
  b : byte?

(unsafe-bytes-copy!
  dest
  dest-start
  src
  [src-start
    src-end]) → void?
  dest : (and/c bytes? (not/c immutable?))
  dest-start : fixnum?
  src : bytes?
  src-start : fixnum? = 0
  src-end : fixnum? = (bytes-length src)

Unsafe versions of bytes-length, bytes-ref, bytes-set!, and bytes-copy!. A byte’s size can never be larger than a fixnum (so even bytes-length always returns a fixnum).

Changed in version 7.5.0.15 of package base: Added unsafe-bytes-copy!.

(unsafe-bytes->immutable-bytes!
  bstr
) → (and/c bytes? immutable?)
  bstr : bytes?

Similar to bytes->immutable-bytes, but potentially destroys bstr and reuses it space, so bstr must not be used after calling unsafe-bytes->immutable-bytes!.

Added in version 7.7.0.6 of package base.

(unsafe-fxvector-length v) → fixnum?
  v : fxvector?
(unsafe-fxvector-ref v k) → fixnum?
  v : fxvector?
  k : fixnum?
(unsafe-fxvector-set! v k x) → void?
  v : fxvector?
  k : fixnum?
  x : fixnum?

Unsafe versions of fxvector-length, fxvector-ref, and fxvector-set!. A fxvector’s size can never be larger than a fixnum (so even fxvector-length always returns a fixnum).

(unsafe-flvector-length v) → fixnum?
  v : flvector?
(unsafe-flvector-ref v k) → flonum?
  v : flvector?
  k : fixnum?
(unsafe-flvector-set! v k x) → void?
  v : flvector?
  k : fixnum?
  x : flonum?

Unsafe versions of flvector-length, flvector-ref, and flvector-set!. A flvector’s size can never be larger than a fixnum (so even flvector-length always returns a fixnum).

(unsafe-f64vector-ref vec k) → flonum?
  vec : f64vector?
  k : fixnum?
(unsafe-f64vector-set! vec k n) → void?
  vec : f64vector?
  k : fixnum?
  n : flonum?

Unsafe versions of f64vector-ref and f64vector-set!.

(unsafe-s16vector-ref vec k) → (integer-in -32768 32767)
  vec : s16vector?
  k : fixnum?
(unsafe-s16vector-set! vec k n) → void?
  vec : s16vector?
  k : fixnum?
  n : (integer-in -32768 32767)

Unsafe versions of s16vector-ref and s16vector-set!.

(unsafe-u16vector-ref vec k) → (integer-in 0 65535)
  vec : u16vector?
  k : fixnum?
(unsafe-u16vector-set! vec k n) → void?
  vec : u16vector?
  k : fixnum?
  n : (integer-in 0 65535)

Unsafe versions of u16vector-ref and u16vector-set!.

(unsafe-struct-ref v k) → any/c
  v : any/c
  k : fixnum?
(unsafe-struct-set! v k val) → void?
  v : any/c
  k : fixnum?
  val : any/c
Unsafe field access and update for an instance of a structure type, where the `struct*` variants can be faster but do not work on impersonators. The index `k` must be between 0 (inclusive) and the number of fields in the structure (exclusive). In the case of `unsafe-struct*-set!`, `unsafe-struct*-set!`, and `unsafe-struct*-cas!`, the field must be mutable. The `unsafe-struct*-cas!` operation is analogous to `box-cas!` to perform an atomic compare-and-set.

Changed in version 6.11.0.2 of package `base`: Added `unsafe-struct*-cas!`.

```scheme
(unsafe-struct*-ref v k) -> any/c
  v : (not/c impersonator?)
  k : fixnum?

(unsafe-struct*-set! v k val) -> void?
  v : (not/c impersonator?)
  k : fixnum?
  val : any/c

(unsafe-struct*-cas! v k old-val new-val) -> boolean?
  v : (not/c impersonator?)
  k : fixnum?
  old-val : any/c
  new-val : any/c
```

```scheme
(unsafe-mutable-hash-iterate-first hash) -> (or/c #f any/c)
  hash : (and/c hash? (not/c immutable?) hash-strong?)

(unsafe-mutable-hash-iterate-next hash pos) -> (or/c #f any/c)
  hash : (and/c hash? (not/c immutable?) hash-strong?)
  pos : any/c

(unsafe-mutable-hash-iterate-key hash pos) -> any/c
  hash : (and/c hash? (not/c immutable?) hash-strong?)
  pos : any/c

(unsafe-mutable-hash-iterate-key hash pos bad-index-v) -> any/c
  hash : (and/c hash? (not/c immutable?) hash-strong?)
  pos : any/c
  bad-index-v : any/c

(unsafe-mutable-hash-iterate-value hash pos) -> any/c
  hash : (and/c hash? (not/c immutable?) hash-strong?)
  pos : any/c

(unsafe-mutable-hash-iterate-value hash pos bad-index-v) -> any/c
  hash : (and/c hash? (not/c immutable?) hash-strong?)
  pos : any/c
  bad-index-v : any/c
```
(unsafe-mutable-hash-iterate-key+value hash
  pos) → any/c any/c
hash : (and/c hash? (not/c immutable?) hash-strong?)
pos : any/c
(unsafe-mutable-hash-iterate-key+value hash
  pos
  bad-index-v)
→ any/c any/c
hash : (and/c hash? (not/c immutable?) hash-strong?)
pos : any/c
bad-index-v : any/c
(unsafe-mutable-hash-iterate-pair hash pos) → pair?
hash : (and/c hash? (not/c immutable?) hash-strong?)
pos : any/c
(unsafe-mutable-hash-iterate-pair hash
  pos
  bad-index-v) → pair?
hash : (and/c hash? (not/c immutable?) hash-strong?)
pos : any/c
bad-index-v : any/c
(unsafe-immutable-hash-iterate-first hash) → (or/c #f any/c)
hash : (and/c hash? immutable?)
(unsafe-immutable-hash-iterate-next hash
  pos) → (or/c #f any/c)
hash : (and/c hash? immutable?)
pos : any/c
(unsafe-immutable-hash-iterate-key hash
  pos) → any/c
hash : (and/c hash? immutable?)
pos : any/c
(unsafe-immutable-hash-iterate-key hash
  pos
  bad-index-v) → any/c
hash : (and/c hash? immutable?)
pos : any/c
bad-index-v : any/c
(unsafe-immutable-hash-iterate-value hash
  pos) → any/c
hash : (and/c hash? immutable?)
pos : any/c
(unsafe-immutable-hash-iterate-value hash
  pos
  bad-index-v) → any/c
hash : (and/c hash? immutable?)
pos : any/c
bad-index-v : any/c
(unsafe-weak-hash-iterate-key+value hash
  pos) → any/c any/c

  hash : (and/c hash? hash-weak?)
pos : any/c

(unsafe-weak-hash-iterate-key+value hash
  pos
  bad-index-v) → any/c any/c

  hash : (and/c hash? hash-weak?)
pos : any/c

  bad-index-v : any/c

(unsafe-weak-hash-iterate-pair hash pos) → pair?

  hash : (and/c hash? hash-weak?)
pos : any/c

  bad-index-v : any/c

(unsafe-weak-hash-iterate-pair hash
  pos
  bad-index-v) → pair?

  hash : (and/c hash? hash-weak?)
pos : any/c

  bad-index-v : any/c

(unsafe-ephemeron-hash-iterate-first hash) → (or/c #f any/c)

  hash : (and/c hash? hash-ephemeron?)

(unsafe-ephemeron-hash-iterate-next hash
  pos) → (or/c #f any/c)

  hash : (and/c hash? hash-ephemeron?)
pos : any/c

(unsafe-ephemeron-hash-iterate-key hash
  pos) → any/c

  hash : (and/c hash? hash-ephemeron?)
pos : any/c

(unsafe-ephemeron-hash-iterate-key hash
  pos
  bad-index-v) → any/c

  hash : (and/c hash? hash-ephemeron?)
pos : any/c

  bad-index-v : any/c

(unsafe-ephemeron-hash-iterate-value hash
  pos) → any/c

  hash : (and/c hash? hash-ephemeron?)
pos : any/c

(unsafe-ephemeron-hash-iterate-value hash
  pos
  bad-index-v) → any/c

  hash : (and/c hash? hash-ephemeron?)
pos : any/c

  bad-index-v : any/c

1243
Unsafe versions of `hash-iterate-key` and similar procedures. These operations support chaperones and impersonators.

Each unsafe `...-first` and `...-next` procedure may return, instead of a number index, an internal representation of a view into the hash structure, enabling faster iteration. The result of these `...-first` and `...-next` functions should be given as `pos` to the corresponding unsafe accessor functions.

If the `pos` provided to an accessor function for a mutable `hash` was formerly a valid hash index but is no longer a valid hash index for `hash`, and if `bad-index-v` is not provided, then the `exn:fail:contract` exception is raised. No behavior is specified for a `pos` that was never a valid hash index for `hash`. Note that `bad-index-v` argument is technically not useful for the `unsafe-immutable-hash-iterate-` functions, since an index cannot become invalid for an immutable `hash`.

Added in version 6.4.0.6 of package base.
Changed in version 7.0.0.10: Added the optional `bad-index-v` argument.
Changed in version 8.0.0.10: Added `ephemeron` variants.
Unsafe version of `srcloc`.

Added in version 7.2.0.10 of package base.

### 17.4 Unsafe Extflonum Operations

```
(unsafe-extfl+ a b) → extflonum?
  a : extflonum?
  b : extflonum?

(unsafe-extfl- a b) → extflonum?
  a : extflonum?
  b : extflonum?

(unsafe-extfl* a b) → extflonum?
  a : extflonum?
  b : extflonum?

(unsafe-extfl/ a b) → extflonum?
  a : extflonum?
  b : extflonum?

(unsafe-extflabs a) → extflonum?
  a : extflonum?
```

Unchecked versions of `extfl+, extfl-, extfl*, extfl/`, and `extflabs`.

```
(unsafe-extfl= a b) → boolean?
  a : extflonum?
  b : extflonum?

(unsafe-extfl< a b) → boolean?
  a : extflonum?
  b : extflonum?

(unsafe-extfl> a b) → boolean?
  a : extflonum?
  b : extflonum?

(unsafe-extfl<= a b) → boolean?
  a : extflonum?
  b : extflonum?

(unsafe-extfl>= a b) → boolean?
  a : extflonum?
  b : extflonum?
```
Unchecked versions of \texttt{extfl=}, \texttt{extfl<}, \texttt{extfl>}, \texttt{extfl<=}, \texttt{extfl>=}, \texttt{extflmin}, and \texttt{extflmax}.

\begin{verbatim}
(unsafe-extflmin a b) → extflonum?
  a : extflonum?
  b : extflonum?
(unsafe-extflmax a b) → extflonum?
  a : extflonum?
  b : extflonum?
\end{verbatim}

Unchecked (potentially) versions of \texttt{extflround}, \texttt{extflfloor}, \texttt{extflceiling}, and \texttt{extfltruncate}. Currently, these bindings are simply aliases for the corresponding safe bindings.

\begin{verbatim}
(unsafe-extflround a) → extflonum?
  a : extflonum?
(unsafe-extflfloor a) → extflonum?
  a : extflonum?
(unsafe-extflceiling a) → extflonum?
  a : extflonum?
(unsafe-extfltruncate a) → extflonum?
  a : extflonum?
\end{verbatim}

Unchecked (potentially) versions of \texttt{extflsin}, \texttt{extflcos}, \texttt{extfltan}, \texttt{extflasin}, \texttt{extflacos}, \texttt{extflatan}, \texttt{extfllog}, \texttt{extflexp}, \texttt{extflsqrt}, \texttt{extflexpt a b}

\begin{verbatim}
(unsafe-extflsin a) → extflonum?
  a : extflonum?
(unsafe-extflcos a) → extflonum?
  a : extflonum?
(unsafe-extfltan a) → extflonum?
  a : extflonum?
(unsafe-extflasin a) → extflonum?
  a : extflonum?
(unsafe-extflacos a) → extflonum?
  a : extflonum?
(unsafe-extflatan a) → extflonum?
  a : extflonum?
(unsafe-extfllog a) → extflonum?
  a : extflonum?
(unsafe-extflexp a) → extflonum?
  a : extflonum?
(unsafe-extflsqrt a) → extflonum?
  a : extflonum?
(unsafe-extflexpt a b) → extflonum?
  a : extflonum?
  b : extflonum?
\end{verbatim}

Unchecked (potentially) versions of \texttt{extflsin}, \texttt{extflcos}, \texttt{extfltan}, \texttt{extflasin}, \texttt{extflacos}, \texttt{extflatan}, \texttt{extfllog}, \texttt{extflexp}, \texttt{extflsqrt}, \texttt{extflexpt a b}.
extflacos, extflatan, extfllog, extflexp, extflsqrt, and extflexpt. Currently, some of these bindings are simply aliases for the corresponding safe bindings.

```scheme
(unsafe-fx->extfl a) → extflonum?
  a : fixnum?
(unsafe-extfl->fx a) → fixnum?
  a : extflonum?
```

Unchecked (potentially) versions of fx->extfl and extfl->fx.

Changed in version 7.7.0.8 of package base: Changed unsafe-fl->fx to truncate.

```scheme
(unsafe-extflvector-length v) → fixnum?
  v : extflvector?
(unsafe-extflvector-ref v k) → extflonum?
  v : extflvector?
  k : fixnum?
(unsafe-extflvector-set! v k x) → void?
  v : extflvector?
  k : fixnum?
  x : extflonum?
```

Unchecked versions of extflvector-length, extflvector-ref, and extflvector-set!. A extflvector’s size can never be larger than a fixnum (so even extflvector-length always returns a fixnum).

### 17.5 Unsafe Impersonators and Chaperones

```scheme
(unsafe-impersonate-procedure proc
  replacement-proc
  prop
  prop-val ...
  ...
) → (and/c procedure? impersonator?)
  proc : procedure?
  replacement-proc : procedure?
  prop : impersonator-property?
  prop-val : any
```

Like impersonate-procedure, but assumes that replacement-proc calls proc itself. When the result of unsafe-impersonate-procedure is applied to arguments, the arguments are passed on to replacement-proc directly, ignoring proc. At the same time, impersonator-of? reports #t when given the result of unsafe-impersonate-procedure and proc.
If \( \text{proc} \) is itself an impersonator that is derived from \texttt{impersonate-procedure\*} or \texttt{chaperone-procedure\*}, beware that \texttt{replacement-proc} will not be able to call it correctly. Specifically, the impersonator produced by \texttt{unsafe-impersonate-procedure} will not get passed to a wrapper procedure that was supplied to \texttt{impersonate-procedure\*} or \texttt{chaperone-procedure\*} to generate \( \text{proc} \).

Finally, unlike \texttt{impersonate-procedure}, \texttt{unsafe-impersonate-procedure} does not specially handle \texttt{impersonator-prop:application-mark} as a \texttt{prop}.

The unsafety of \texttt{unsafe-impersonate-procedure} is limited to the above differences from \texttt{impersonate-procedure}. The contracts on the arguments of \texttt{unsafe-impersonate-procedure} are checked when the arguments are supplied.

As an example, assuming that \( f \) accepts a single argument and is not derived from \texttt{impersonate-procedure\*} or \texttt{chaperone-procedure\*}, then

\[
(\lambda (f) \\
(\text{unsafe-impersonate-procedure} \\
f \\
(\lambda (x) \\
(if (\text{number?} x) \\
(\text{error} '\text{no-numbers!}) \\
(f x))))))
\]

is equivalent to

\[
(\lambda (f) \\
(\text{impersonate-procedure} \\
f \\
(\lambda (x) \\
(if (\text{number?} x) \\
(\text{error} '\text{no-numbers!}) \\
x))))
\]

Similarly, with the same assumptions about \( f \), the following two procedures \texttt{wrap-f1} and \texttt{wrap-f2} are almost equivalent; they differ only in the error message produced when their arguments are functions that return multiple values (and that they update different global variables). The version using \texttt{unsafe-impersonate-procedure} will signal an error in the \texttt{let} expression about multiple return values, whereas the one using \texttt{impersonate-procedure} signals an error from \texttt{impersonate-procedure} about multiple return values.

\[
(\text{define log1-args } '()) \\
(\text{define log1-results } '()) \\
(\text{define wrap-f1 } \\
(\lambda (f) \\
(\text{impersonate-procedure} \\
f \\
(\lambda (x) \\
(if (\text{number?} x) \\
(\text{error} '\text{no-numbers!}) \\
x))))
\]

1248
\[
\text{f} \\
(\lambda (\text{arg}) \\
  (\text{set! log1-args} (\text{cons arg log1-args})) \\
  (\text{values } (\lambda (\text{res}) \\
    (\text{set! log1-results} (\text{cons res log1-results})) \\
    \text{res}) \\
    \text{arg})))\))
\]

(\text{define log2-args '()} \\
(\text{define log2-results '()} \\
(\text{define wrap-f2 } \\
  (\lambda (f) \\
    (\text{unsafe-impersonate-procedure} f \\
    (\lambda (\text{arg}) \\
      (\text{set! log2-args} (\text{cons arg log2-args})) \\
      (\text{let } ([\text{res} (f \text{ arg})]) \\
        (\text{set! log2-results} (\text{cons res log2-results})) \\
        \text{res}))))))

Added in version 6.4.0.4 of package base.

\[
(\text{unsafe-chaperone-procedure} \text{ proc} \\
  \text{wrapper-proc} \text{ prop} \text{ prop-val} \ldots \ldots) \\
\rightarrow (\text{and/c procedure? chaperone?})
\]

\text{proc} : \text{procedure?} \\
\text{wrapper-proc} : \text{procedure?} \\
\text{prop} : \text{impersonator-property?} \\
\text{prop-val} : \text{any}

Like \text{unsafe-impersonate-procedure}, but creates a chaperone. Since \text{wrapper-proc} will be called in lieu of \text{proc}, \text{wrapper-proc} is assumed to return a chaperone of the value that \text{proc} would return.

Added in version 6.4.0.4 of package base.

\[
(\text{unsafe-impersonate-vector} \text{ vec} \\
  \text{replacement-vec} \text{ prop} \text{ prop-val} \ldots \ldots) \\
\rightarrow (\text{and/c vector? impersonator?})
\]

\text{vec} : \text{vector?}
replacement-vec : (and/c vector? (not/c impersonator?))
prop : impersonator-property?
prop-val : any/c

Like impersonate-vector, but instead of going through interposition procedures, all accesses to the impersonator are dispatched to replacement-vec.

The result of unsafe-impersonate-vector is an impersonator of vec.

Added in version 6.9.0.2 of package base.

(unsafe-chaperone-vector vec
  replacement-vec
  prop
  prop-val ...
  ...
) → (and/c vector? chaperone?)
vec : vector?
replacement-vec : (and/c vector? (not/c impersonator?))
prop : impersonator-property?
prop-val : any/c

Like unsafe-impersonate-vector, but the result of unsafe-chaperone-vector is a chaperone of vec.

Added in version 6.9.0.2 of package base.

17.6 Unsafe Assertions

(unsafe-assert-unreachable) → none/c

Like assert-unreachable, but the contract of unsafe-assert-unreachable is never satisfied, and the “unsafe” implication is that anything at all can happen if a call to unsafe-assert-unreachable is reached.

The compiler may take advantage of its liberty to pick convenient or efficient behavior in place of a call to unsafe-assert-unreachable. For example, the expression

(lambda (x)
  (if (pair? x)
      (car x)
      (unsafe-assert-unreachable))))

may be compiled to code equivalent to
(lambda (x) (unsafe-car x))

because choosing to make (unsafe-assert-unreachable) behave the same as (unsafe-car x) makes both branches of the if the same, and then pair? test can be eliminated.

Added in version 8.0.0.11 of package base.

17.7 Unsafe Undefined

(require racket/unsafe/undefined)  package: base

The bindings documented in this section are provided by the racket/unsafe/undefined library, not racket/base or racket.

The constant unsafe-undefined is used internally as a placeholder value. For example, it is used by letrec as a value for a variable that has not yet been assigned a value. Unlike the undefined value exported by racket/undefined, however, the unsafe-undefined value should not leak as the result of a safe expression, and it should not be passed as an optional argument to a procedure (because it may count as “no value provided”). Expression results that potentially produce unsafe-undefined can be guarded by check-not-unsafe-undefined, so that an exception can be raised instead of producing an undefined value.

The unsafe-undefined value is always eq? to itself.

Added in version 6.0.1.2 of package base.
 Changed in version 6.90.0.29: Procedures with optional arguments sometimes use the unsafe-undefined value internally to mean “no argument supplied.”

unsafe-undefined : any/c

The unsafe “undefined” constant.

See above for important constraints on the use of unsafe-undefined.

(check-not-unsafe-undefined v sym)
→ (and/c any/c (not/c (one-of/c unsafe-undefined)))
  v : any/c
  sym : symbol?

Checks whether v is unsafe-undefined, and raises exn:fail:contract:variable in that case with an error message along the lines of “sym: undefined; use before initialization.” If v is not unsafe-undefined, then v is returned.
The same as `check-not-unsafe-undefined`, except that the error message (if any) is along the lines of “`sym: undefined; assignment before initialization`.”

Chaperones `v` if it is a structure (as viewed through some inspector). Every access of a field in the structure is checked to prevent returning `unsafe-undefined`. Similarly, every assignment to a field in the structure is checked (unless the check disabled as described below) to prevent assignment of a field whose current value is `unsafe-undefined`.

When a field access would otherwise produce `unsafe-undefined` or when a field assignment would replace `unsafe-undefined`, the `exn:fail:contract` exception is raised.

The chaperone’s field-assignment check is disabled whenever `(continuation-mark-set-first #f prop:chaperone-unsafe-undefined)` returns `unsafe-undefined`. Thus, a field-initializing assignment—one that is intended to replace the `unsafe-undefined` value of a field—should be wrapped with `(with-continuation-mark prop:chaperone-unsafe-undefined unsafe-undefined ....)`.

A structure type property that causes a structure type’s constructor to produce a chaperone of an instance in the same way as `chaperone-struct-unsafe-undefined`.

The property value should be a list of symbols used as field names, but the list should be in reverse order of the structure’s fields. When a field access or assignment would produce or replace `unsafe-undefined`, the `exn:fail:contract:variable` exception is raised if a field name is provided by the structure property’s value, otherwise the `exn:fail:contract` exception is raised.
18 Running Racket

18.1 Running Racket or GRacket

The core Racket run-time system is available in two main variants:

- Racket, which provides the primitives libraries on which racket/base is implemented. On Unix and Mac OS, the executable is called racket. On Windows, the executable is called Racket.exe.

- GRacket, which is a GUI variant of racket to the degree that the system distinguishes them. On Unix, the executable is called gracket, and single-instance flags and X11-related flags are handled and communicated specially to the racket/gui/base library. On Windows, the executable is called GRacket.exe, and it is a GUI application (as opposed to a console application) that implements single-instance support. On Mac OS, the gracket script launches GRacket.app.

18.1.1 Initialization

On start-up, the top-level environment contains no bindings—not even #%app for function application. Primitive modules with names that start with #% are defined, but they are not meant for direct use, and the set of such modules can change. For example, the '#%kernel module is eventually used to bootstrap the implementation of racket/base.

The first action of Racket or GRacket is to initialize current-library-collection-paths to the result of (find-library-collection-paths pre-extras extras), where pre-extras is normally null and extras are extra directory paths provided in order in the command line with -S/--search. An executable created from the Racket or GRacket executable can embed paths used as pre-extras.

Racket and GRacket next require racket/init and racket/gui/init, respectively, but only if the command line does not specify a require flag (-t/--require, -l/--lib, or -u/--require-script) before any eval, load, or read-eval-print-loop flag (-e/--eval, -f/--load, -r/--script, -m/--main, or -i/--repl). The initialization library can be changed with the -I configuration option. The configure-runtime submodule of the initialization library or the 'configure-runtime property of the initialization library’s language is used before the library is instantiated; see §18.1.5 “Language Run-Time Configuration”.

After potentially loading the initialization module, expression evals, files loads, and module requires are executed in the order that they are provided on the command line. If any raises an uncaught exception, then the remaining evals, loads, and requires are skipped. If the first require precedes any eval or load so that the initialization library is skipped,
then the configure-runtime submodule of the required module or the 'configure-runtime property of the required module’s library language is used before the module is instantiated; see §18.1.5 “Language Run-Time Configuration”.

After running all command-line expressions, files, and modules, Racket or GRacket then starts a read-eval-print loop for interactive evaluation if no command line flags are provided other than configuration options. For Racket, the read-eval-print loop is run by calling read-eval-print-loop from racket/repl. For GRacket, the read-eval-print loop is run by calling graphical-read-eval-print-loop from racket/gui/base. If any command-line argument is provided that is not a configuration option, then the read-eval-print-loop is not started, unless the -i/--repl flag is provided on the command line to specifically re-enable it.

In addition, just before the read-eval-print loop is started, Racket runs racket/interactive and GRacket runs racket/gui/interactive, unless a different interactive file is specified in the the installation’s "config.rktd" file found in (find-config-dir), or the file "interactive.rkt" is found in (find-system-path 'addon-dir). If the -q/--no-init-file flag is specified on the command line, then no interactive file is run.

Finally, before Racket or GRacket exits, it calls the procedure that is the current value of executable-yield-handler in the main thread, unless the -V/--no-yield command-line flag is specified. Requiring racket/gui/base sets this parameter call (racket 'yield).

Changed in version 6.7 of package base: Run racket/interactive file rather than directly running (find-system-path 'init-file).
Changed in version 6.90.0.30: Run a read-eval-print loop by using racket/repl or racket/gui/base instead of racket/base or racket/gui/init.

18.1.2 Exit Status

The default exit status for a Racket or GRacket process is non-zero if an error occurs during a command-line eval (via -e, etc.), load (via -f, -r, etc.), or require (via -l, -t, etc.)—or, more generally, if the abort handler of the prompt surrounding those evaluations is called—but only when no read-eval-print loop is started. Otherwise, the default exit status is 0.

In all cases, a call to exit (when the default exit handler is in place) can end the process with a specific status value.

18.1.3 Init Libraries

(require racket/init) package: base

1254
The `racket/init` library is the default start-up library for Racket. It re-exports the `racket`, `racket/enter` and `racket/help` libraries, and it sets `current-print` to use `pretty-print`.

```scheme
(require racket/init)
```

The `racket/interactive` is the default start up library when the REPL begins. It is not run if the `-q/--no-init-file` is specified. The interactive file can be changed by modifying `interactive-file` in the "config.rkt" file found in `find-config-dir`. Alternative, if the file "interactive.rkt" exists in `find-system-path 'addon-dir` it is run rather than the installation wide interactive module.

The default interactive module starts `xrepl` and runs the `find-system-path 'init-file` file in the users home directory. A different interactive file can keep this behavior by requiring `racket/interactive`.

Added in version 6.7 of package `base`.

```scheme
(require racket/interactive)
```

The `racket/language-info` library provides a `get-info` function that takes any value and returns another function; the returned function takes a key value and a default value, and it returns `'(=(racket/runtime-config configure #f))` if the key is 'configure-runtime or the default value otherwise.

The vector `'#(racket/language-info get-info #f)` is suitable for attaching to a module as its language info to get the same language information as the `racket/base` language.

```scheme
(require racket/language-info)
```

The `racket/runtime-config` library provides a `configure` function that takes any value and sets `print-as-expression` to `#t`.

The vector `#(racket/runtime-config configure #f)` is suitable as a member of a list of runtime-configuration specification (as returned by a module’s language-information function for the key 'configure-runtime) to obtain the same runtime configuration as for the `racket/base` language.

```
(require racket/runtime-config)
```

### 18.1.4 Command Line

The Racket and GRacket executables recognize the following command-line flags:

- File and expression options:
  ```
  -e <expr> or --eval <expr> : evals <expr>. The results of the evaluation are printed via current-print.
  ```
- -f <file> or --load <file> : loads <file>; if <file> is "-", then expressions are read and evaluated from standard input.

- -t <file> or --require <file> : requires <file>, and then requires (submod (file "<file>") main) if available.

- -l <path> or --lib <path> : requires (lib "<path>"), and then requires (submod (lib "<path>") main) if available.

- -p <package> : requires (planet "<package>"), and then requires (submod (planet "<package>") main) if available.

- -r <file> or --script <file> : loads <file> as a script. This flag is like -t <file> plus -N <file> to set the program name and -- to cause all further command-line elements to be treated as non-flag arguments.

- -u <file> or --require-script <file> : requires <file> as a script; This flag is like -t <file> plus -N <file> to set the program name and -- to cause all further command-line elements to be treated as non-flag arguments.

- -k (n) (m) (p) : Loads code embedded in the executable from file position (n) to (m) and from (m) to (p). (On Mac OS, (n), (m), and (p) are relative to a __PLTScheme segment in the executable. On Windows, they are relative to a resource of type 257 and ID 1.) The first range is loaded in every new place, and any modules declared in that range are considered predefined in the sense of module-predefined?. This option is normally embedded in a stand-alone binary that also embeds Racket code.

- -Y (n) (m) (p) : Like -k (n) (m) (p), but reading from <file> (without any adjustment on Mac OS or Windows for a segment or resource offset).

- -m or --main : Evaluates a call to main as bound in the top-level environment. All of the command-line arguments that are not processed as options (i.e., the arguments put into current-command-line-arguments) are passed as arguments to main. The results of the call are printed via current-print.

The call to main is constructed as an expression (main arg-str ...) where the lexical context of the expression gives #%app and #%datum bindings as #%plain-app and #%datum, but the lexical context of main is the top-level environment.

• Interaction options:

- -i or --repl : Runs an interactive read-eval-print loop, using either read-eval-print-loop (Racket) or graphical-read-eval-print-loop (GRacket) after showing (banner) and loading (find-system-path 'init-file). In the case of Racket, (read-eval-print-loop) is followed by (newline). For GRacket, supply the -z/--text-repl configuration option to use read-eval-print-loop (and newline) instead of graphical-read-eval-print-loop.

- -n or --no-lib : Skips requiring the initialization library (i.e., racket/init or racket/gui/init, unless it is changed with the -I flag) when not otherwise disabled.
• Configuration options:

- `-c` or `--no-compiled`: Disables loading of compiled byte-code ".zo" files, by initializing `use-compiled-file-paths` to `null`. Use judiciously: this effectively ignores the content of all "compiled" subdirectories, so that any used modules are compiled on the fly—even racket/base and its dependencies—which leads to prohibitively expensive run times.

- `-q` or `--no-init-file`: Skips loading `(find-system-path 'init-file)` for `-i/-repl`.

- `-z` or `--text-repl`: GRacket only; changes `-i/-repl` to use `textual-read-eval-print-loop` instead of `graphical-read-eval-print-loop`.

- `-I path`: Sets `(lib "<path>") as the path to require to initialize the namespace, unless namespace initialization is disabled. Using this flag can effectively set the language for the read-eval-print loop and other top-level evaluation.

- `-X dir` or `--collects dir`: Sets `dir` as the path to the main collection of libraries by making `(find-system-path 'collects-dir)` produce `dir`. If `dir` is an empty string, then `(find-system-path 'collects-dir)` returns ".", but `current-library-collection-paths` is initialized to the empty list, and `use-collection-link-paths` is initialized to `#f`.

- `-S dir` or `--search dir`: Adds `dir` to the default library collection search path after the main collection directory. If the `-S/-dir` flag is supplied multiple times, the search order is as supplied.

- `-G dir` or `--config dir`: Sets the directory that is returned by `(find-system-path 'config-dir)`.

- `-A dir` or `--addon dir`: Sets the directory that is returned by `(find-system-path 'addon-dir)`.

- `-U` or `--no-user-path`: Omits user-specific paths in the search for collections, C libraries, etc. by initializing the `use-user-specific-search-paths` parameter to `#f`.

- `-R paths` or `--compiled paths`: Sets the initial value of the `current-compiled-file-roots` parameter, overriding any PLTCOMPILEDROOTS setting. The `paths` argument is parsed in the same way as PLTCOMPILEDROOTS (see `current-compiled-file-roots`).
- -C or --cross: Select cross-platform build mode, causing (system-type 'cross) to report 'force, and sets the current configuration of (find-system-path 'config-dir) and (find-system-path 'collects-dir) to be the results of (find-system-path 'host-config-dir) and (find-system-path 'host-collects-dir), respectively. If -C or --cross is provided multiple times, only the first instance has an effect.

- -N <file> or --name <file>: sets the name of the executable as reported by (find-system-path 'run-file) to <file>.

- -E <file> or --exe <file>: sets the name of the executable as reported by (find-system-path 'exec-file) to <file>.

- -J <name> or --wm-class <name>: GRacket, Unix only; sets the WM_CLASS program class to <name> (while the WM_CLASS program name is derived from the executable name or a -N/--name argument).

- -j or --no-jit: Disables the native-code just-in-time compiler by setting the eval-jit-enabled parameter to #f.

- -M or --compile-any: Enables machine-independent bytecode by setting the current-compile-target-machine parameter to #f.

- -d or --no-delay: Disables on-demand parsing of compiled code and syntax objects by setting the read-on-demand-source parameter to #f.

- -b or --binary: Requests binary mode, instead of text mode, for the process’s input, out, and error ports. This flag currently has no effect, because binary mode is always used.

- -W <levels> or --warn <levels>: Sets the logging level for writing events to the original error port. The possible <level> values are the same as for the PLTST-DERR environment variable. See $15.5 “Logging” for more information.

- -O <levels> or --stdout <levels>: Sets the logging level for writing events to the original output port. The possible <level> values are the same as for the PLTSTDOUT environment variable. See $15.5 “Logging” for more information.

- -L <levels> or --syslog <levels>: Sets the logging level for writing events to the system log. The possible <level> values are the same as for the PLTSYSLOG environment variable. See $15.5 “Logging” for more information.

- -Z: The argument following this flag is ignored. This flag can be handy in some impoverished scripting environments to replace or cancel another command-line argument.

- --: No argument following this flag is itself used as a flag.

- -h or --help: Shows information about the command-line flags and start-up process and exits, ignoring all other flags.

If at least one command-line argument is provided, and if the first one after any configuration option is not a flag, then a -u/--require-script flag is implicitly added before the first non-flag argument.
If no command-line arguments are supplied other than configuration options, then the
-i/--repl flag is effectively added.

For GRacket on Unix, the follow flags are recognized when they appear at the beginning
of the command line, and they count as configuration options (i.e., they do not disable the
read-eval-print loop or prevent the insertion of -u/--require-script):

- **-display**: Sets the X11 display to use.

- **-geometry arg, -bg arg, -background arg, -fg arg, -foreground arg, -fn arg, -font arg, -iconic, -name arg, -rv, -reverse, +rv, -selectionTimeout arg, -synchronous, -title arg, -xnllanguage arg, or -xrm arg**: Standard X11 arguments that are mostly ignored but accepted for
compatibility with other X11 programs. The -synchronous flag behaves in the usual way.

- **-singleInstance**: If an existing GRacket is already running on the same X11 dis-
play, if it was started on a machine with the same hostname, and if it was started
with the same name as reported by (find-system-path 'run-file)—possibly
set with the -N/--name command-line argument—then all non-option command-line
arguments are treated as filenames and sent to the existing GRacket instance via the
application file handler (see application-file-handler).

Similarly, on Mac OS, a leading switch starting with -psn_ is treated as a special configura-
tion option. It indicates that Finder started the application, so the current input, output, and
error output are redirected to a GUI window.

Multiple single-letter switches (the ones preceded by a single -) can be collapsed into a
single switch by concatenating the letters, as long as the first switch is not --. The arguments
for each switch are placed after the collapsed switches (in the order of the switches). For
example,

```
-ifve file expr
```

and

```
-i -f file -v -e expr
```

are equivalent. If a collapsed -- appears before other collapsed switches in the same col-
lapsed set, it is implicitly moved to the end of the collapsed set.

Extra arguments following the last option are available from the current-command-line-
arguments parameter.

Changed in version 6.90.0.17 of package base: Added -O/--stdout.
Changed in version 7.1.0.5: Added -M/--compile-any.
Changed in version 7.8.0.6: Added -Z.

1259
18.1.5 Language Run-Time Configuration

A module can have a `configure-runtime` submodule that is `dynamic-required` before the module itself when a module is the main module of a program. Normally, a `configure-runtime` submodule is added to a module by the module’s language (i.e., by the `#%module-begin` form among a module’s initial bindings).

Alternatively or in addition, an older protocol is in place. When a module is implemented using `#lang`, the language after `#lang` can specify configuration actions to perform when a module using the language is the main module of a program. The language specifies run-time configuration by

- attaching a `module-language` syntax property to the module as read from its source (see module and `module-compiled-language-info`);
- having the function indicated by the `module-language` syntax property recognize the `configure-runtime` key, for which it returns a list of vectors; each vector must have the form `(vector mp name val)` where `mp` is a module path, `name` is a symbol, and `val` is an arbitrary value; and
- having each function called as `((dynamic-require mp name) val)` configure the run-time environment, typically by setting parameters such as `current-print`.

A `configure-runtime` query returns a list of vectors, instead of directly configuring the environment, so that the indicated modules to be bundled with a program when creating a stand-alone executable; see §2 “raco exe: Creating Stand-Alone Executables” in raco: Racket Command-Line Tools.

For information on defining a new `#lang` language, see `syntax/module-reader`.

18.2 Libraries and Collections

A `library` is module declaration for use by multiple programs. Racket further groups libraries into collections. Typically, collections are added via packages (see Package Management in Racket); the package manager works outside of the Racket core, but it configures the core run-time system through collection links files.

Libraries in collections are referenced through lib paths (see `require`) or symbolic short-hands. For example, the following module uses the "getinfo.rkt" library module from the "setup" collection, and the "cards.rkt" library module from the "games" collection’s "cards" subcollection:
#lang racket
(require (lib "setup/getinfo.rkt")
  (lib "games/cards/cards.rkt")
  ....
This example is more compactly and more commonly written using symbolic shorthands:

#lang racket
(require setup/getinfo
games/cards/cards)
  ....

When an identifier id is used in a require form, it is converted to (lib rel-string) where rel-string is the string form of id.

A rel-string in (lib rel-string) consists of one or more path elements that name collections, and then a final path element that names a library file; the path elements are separated by /. If rel-string contains no /s, then /main.rkt is implicitly appended to the path. If rel-string contains / but does not end with a file suffix, then .rkt is implicitly appended to the path.

Libraries also can be distributed via PLaneT packages. Such libraries are referenced through a planet module path (see require) and are downloaded by Racket on demand, instead of referenced through collections.

The translation of a planet or lib path to a module declaration is determined by the module name resolver, as specified by the current-module-name-resolver parameter.

18.2.1 Collection Search Configuration

For the default module name resolver, the search path for collections is determined by the current-library-collection-links parameter and the current-library-collection-paths parameter:

- The most primitive collection-based modules are located in "collects" directory relative to the Racket executable. Libraries for a collection are grouped within a directory whose name matches the collection name. The path to the "collects" directory is normally included in current-library-collection-paths.

- Collection-based libraries also can be installed other directories, perhaps user-specific, that are structured like the "collects" directory. Those additional directories can be included in the current-library-collection-paths parameter either dynamically, through command-line arguments to racket, or by setting the PLTCOLLECTS environment variable; see find-library-collection-paths.
• Collection links files provide a mapping from top-level collection names to directories, plus additional "collects"-like directories (that have subdirectories with names that match collection names). Each collection links file to be searched is referenced by the current-library-collection-links parameter; the parameter references the file, and not the file’s content, so that changes to the file can be detected and affect later module resolution. See also find-library-collection-links.

• The current-library-collection-links parameter’s value can also include hash tables that provide the same content as collection links files: a mapping from collection names in symbol form to a list of paths for the collection, or from #f to a list of "collects"-like paths.

• Finally, the current-library-collection-links parameter’s value includes #f to indicate the point in the search process at which the module-name resolver should check current-library-collection-paths relative to the files and hash tables in current-library-collection-links.

To resolve a module reference rel-string, the default module name resolver searches collection links in current-library-collection-links from first to last to locate the first directory that contains rel-string, splicing a search through in current-library-collection-paths where in current-library-collection-links contains #f. The filesystem tree for each element in the link table and search path is effectively spliced together with the filesystem trees of other path elements that correspond to the same collection. Some Racket tools rely on unique resolution of module path names, so an installation and configuration should not allow multiple files to match the same collection and file combination.

The value of the current-library-collection-links parameter is initialized by the racket executable to the result of (find-library-collection-links), and the value of the current-library-collection-paths parameter is initialized to the result of (find-library-collection-paths).

### 18.2.2 Collection Links

Collection links files are used by collection-file-path, collection-path, and the default module name resolver to locate collections before trying the (current-library-collection-paths) search path. The collection links files to use are determined by the current-library-collection-links parameter, which is initialized to the result of find-library-collection-links.

A collection links file is read with default reader parameter settings to obtain a list. Every element of the list must be a link specification with one of the forms (list string path), (list string path regexp), (list 'root path), (list 'root path regexp), (list 'static-root path), (list 'static-root path regexp). A string names a top-level collection, in which case path is a path that can be used as
the collection’s path (directly, as opposed to a subdirectory of path named by string). A 'root entry, in contrast, acts like an path in (current-library-collection-paths). A 'static-root entry is like a 'root entry, but where the immediate content of the directory is assumed not to change unless the collection links file changes. If path is a relative path, it is relative to the directory containing the collection links file. If regexp is specified in a link, then the link is used only if (regexp-match? regexp (version)) produces a true result.

A single top-level collection can have multiple links in a collection links file, and any number of 'root entries can appear. The corresponding paths are effectively spliced together, since the paths are tried in order to locate a file or sub-collection.


18.2.3 Collection Paths and Parameters

(find-library-collection-paths [pre-extras post-extras]) → (listof path?)
pre-extras : (listof path-string?) = null
post-extras : (listof path-string?) = null

Produces a list of paths, which is normally used to initialize current-library-collection-paths, as follows:

- The path produced by (build-path (find-system-path 'addon-dir) (get-installation-name) "collects") is the first element of the default collection path list, unless the value of the use-user-specific-search-paths parameter is #f.
- Extra directories provided in pre-extras are included next to the default collection path list, converted to complete paths relative to the executable.
- If the directory specified by (find-system-path 'collects-dir) is absolute, or if it is relative (to the executable) and it exists, then it is added to the end of the default collection path list.
- Extra directories provided in post-extras are included last in the default collection path list, converted to complete paths relative to the executable.
- If the PLTCOLLECTS environment variable is defined, it is combined with the default list using path-list-string->path-list, as long as the value of use-user-specific-search-paths is #f, the default collection path list (as constructed by the first three bullets above) is used directly.
Note that on Unix and Mac OS, paths are separated by `,` and on Windows by `;`. Also, `path-list-string->path-list` splices the default paths at an empty path, for example, with many Unix shells you can set PLTCOLLECTS to "`pwd`", "`pwd:`", or "`pwd`" to specify search the current directory after, before, or instead of the default paths, respectively.

```scheme
(find-library-collection-links)
  → (listof (or/c #f (and/c path? complete-path?)))
```

Produces a list of paths and `#f`, which is normally used to initialize `current-library-collection-links`, as follows:

- The list starts with `#f`, which causes the default module name resolver, `collection-file-path`, and `collection-path` to try paths in `current-library-collection-paths` before collection links files.
- As long as the values of `use-user-specific-search-paths` and `use-collection-link-paths` are true, the second element in the result list is the path of the user–specific collection links file, which is `(build-path (find-system-path 'addon-dir) (get-installation-name) "links.rktd")`.
- As long as the value of `use-collection-link-paths` is true, the rest of the list contains the result of `get-links-search-files`. Typically, that function produces a list with a single path, `(build-path (find-config-dir) "links.rktd")`.

```scheme
(collection-file-path file
collection ...+
  [#:check-compiled? check-compiled?])
→ path?
  file : path-string?
  collection : path-string?
  check-compiled? : any/c = (regexp-match? #rx"\.rkt$" file)
```

```scheme
(collection-file-path file
collection ...+
  #:fail fail-proc
  [#:check-compiled? check-compiled?]) → any

  file : path-string?
  collection : path-string?
  fail-proc : (string? . -> . any)
  check-compiled? : any/c = (regexp-match? #rx"\.rkt$" file)
```

Returns the path to the file indicated by `file` in the collection specified by the `collections`, where the second `collection` (if any) names a sub-collection, and so on. The search uses the values of `current-library-collection-links` and `current-library-collection-paths`.

See also `collection-search` in `setup/collection-search`.
If file is not found, but file ends in ".rkt" and a file with the suffix ".ss" exists, then the directory of the ".ss" file is used. If file is not found and the ".rkt"/".ss" conversion does not apply, but a directory corresponding to the collections is found, then a path using the first such directory is returned.

If check-compiled? is true, then the search also depends on use-compiled-file-paths and current-compiled-file-roots; if file is not found, then a compiled form of file with the suffix ".zo" is checked in the same way as the default compiled-load handler. If a compiled file is found, the result from collection-file-path reports the location that file itself would occupy (if it existed) for the found compiled file.

Finally, if the collection is not found, and if fail-proc is provided, then fail-proc is applied to an error message (that does not start "collection-file-path:" or otherwise claim a source), and its result is the result of collection-file-path. If fail-proc is not provided and the collection is not found, then the exn:fail:filesystem exception is raised.

Examples:

```scheme
> (collection-file-path "main.rkt" "racket" "base")
#<path:path/to/collects/racket/base/main.rkt>
> (collection-file-path "sandwich.rkt" "bologna")
collection-file-path: collection not found
collection: "bologna"
in collection directories:
    /Users/robby/git/snapshot/racket/racket/collects/
    ... [206 additional linked and package directories]
```

Changed in version 6.0.1.12 of package base: Added the check-compiled? argument.

```scheme
(collection-path collection ...) -> path?
collection : path-string?
(collection-path collection ...+ #:fail fail-proc) -> any
collection : path-string?
fail-proc : (string? . -> . any)
```

**NOTE:** This function is deprecated; use collection-file-path, instead. Collection splicing implies that a given collection can have multiple paths, such as when multiple packages provide modules for a collection.

Like collection-file-path, but without a specified file name, so that a directory indicated by collections is returned.
When multiple directories correspond to the collection, the first one found in the search sequence (see §18.2.1 “Collection Search Configuration”) is returned.

```
(current-library-collection-paths)
  -> (listof (and/c path? complete-path?))
(current-library-collection-paths paths) -> void?
paths : (listof (and/c path-string? complete-path?))
```

Parameter that determines a list of complete directory paths for finding libraries (as referenced in require, for example) through the default module name resolver and for finding paths through collection-path and collection-file-path. See §18.2.1 “Collection Search Configuration” for more information.

```
(current-library-collection-links)
  -> (listof (or/c #f
               (and/c path? complete-path?)
               (hash/c (or/c (and/c symbol? module-path?) #f)
                       (listof (and/c path? complete-path?))))))
(current-library-collection-links paths) -> void?
paths : (listof (or/c #f
                 (and/c path-string? complete-path?)
                 (hash/c (or/c (and/c symbol? module-path?) #f)
                         (listof (and/c path-string? complete-path?))))))
```

Parameter that determines collection links files, additional paths, and the relative search order of current-library-collection-paths for finding libraries (as referenced in require, for example) through the default module name resolver and for finding paths through collection-path and collection-file-path. See §18.2.1 “Collection Search Configuration” for more information.

```
(use-user-specific-search-paths) -> boolean?
(use-user-specific-search-paths on?) -> void?
on? : any/c
```

Parameter that determines whether user-specific paths, which are in the directory produced by (find-system-path 'addon-dir), are included in search paths for collections and other files. For example, the initial value of find-library-collection-paths omits the user-specific collection directory when this parameter’s value is #f.

If -U or --no-user-path argument to racket, then use-user-specific-search-paths is initialized to #f.

```
(use-collection-link-paths) -> boolean?
(use-collection-link-paths on?) -> void?
on? : any/c
```
Parameter that determines whether collection links files are included in the result of find-library-collection-links.

If this parameter’s value is #f on start-up, then collection links files are effectively disabled permanently for the Racket process. In particular, if an empty string is provided as the -X or --collects argument to racket, then not only is current-library-collection-paths initialized to the empty list, but use-collection-link-paths is initialized to #f.

18.3 Interactive Help

(require racket/help)   package: base

The bindings documented in this section are provided by the racket/help and racket/init libraries, which means that they are available when the Racket executable is started with no command-line arguments. They are not provided by racket/base or racket.

help
  (help string ...)   (help id)
  (help id #:from module-path)
  (help #:search datum ...)

For general help, see the main documentation page.

The help form searches the documentation and opens a web browser (using the user’s selected browser) to display the results.

A simple help or (help) form opens the main documentation page.

The (help string ...) form—using literal strings, as opposed to expressions that produce strings—performs a string-matching search. For example,

(help "web browser" "firefox")

searches the documentation index for references that include the phrase “web browser” or “firefox.”

A (help id) form looks for documentation specific to the current binding of id. For example,

(require net/url)
  (help url->string)

opens a web browser to show the documentation for url->string from the net/url library.
For the purposes of help, a for-label require introduces a binding without actually executing the net/url library—for cases when you want to check documentation, but cannot or do not want to run the providing module.

```scheme
(require racket/gui) ; does not work in racket
(require (for-label racket/gui)) ; ok in racket
(help frame%)
```

If `id` has no for-label and normal binding, then help lists all libraries that are known to export a binding for `id`.

The `(help `id` #:from module-path)` variant is similar to `(help `id`)`, but using only the exports of `module-path`. (The `module-path` module is required for-label in a temporary namespace.)

```scheme
(help frame% #:from racket/gui) ; equivalent to the above
```

The `(help #:search datum ...)` form is similar to `(help string ...)`, where any non-string form of `datum` is converted to a string using `display`. No `datum` is evaluated as an expression.

For example,

```scheme
(help #:search "web browser" firefox)
```

also searches the documentation index for references that include the phrase “web browser” or “firefox.”

### 18.4 Interactive Module Loading

The `racket/rerequire` and `racket/enter` libraries provide support for loading, reloading, and using modules.

#### 18.4.1 Entering Modules

```scheme
(require racket/enter) ; package: base
```

The bindings documented in this section are provided by the `racket/enter` and `racket/init` libraries, which means that they are available when the Racket executable is started with no command-line arguments. They are not provided by `racket/base` or `racket`.
(enter! module-path)
(enter! #f)
(enter! module-path flag ...+)

flag = #:quiet
| #:verbose-reload
| #:verbose
| #:dont-re-require-enter

Intended for use in a REPL, such as when racket is started in interactive mode. When a module-path is provided (in the same sense as for require), the corresponding module is loaded or invoked via dynamic-rerequire, and the current namespace is changed to the body of the module via module->namespace. When #f is provided, then the current namespace is restored to the original one.

Additional flags can customize aspects of enter!:

- The #:verbose, #:verbose-reload, and #:quiet flags correspond to 'all, 'reload, and 'none verbosity for dynamic-rerequire. The default corresponds to #:verbose-reload.
- After switching namespaces to the designated module, enter! automatically requires racket/enter into the namespace, so that enter! can be used to switch namespaces again. In some cases, requiring racket/enter might not be desirable (e.g., in a tool that uses racket/enter); use the #:dont-re-require-enter flag to disable the require.

(dynamic-enter! mod
  [#:verbosity verbosity
    #:re-require-enter? re-require-enter?]) → void?

mod : (or/c module-path? #f)
verbosity : (or/c 'all 'reload 'none) = 'reload
re-require-enter? : any/c = #t

Procedure variant of enter!, where verbosity is passed along to dynamic-rerequire and re-require-enter? determines whether dynamic-enter! requires racket/enter in a newly entered namespace.

Added in version 6.0.0.1 of package base.

18.4.2 Loading and Reloading Modules

(require racket/rerequire) package: base
The bindings documented in this section are provided by the `racket/rerequire` library, not `racket/base` or `racket`.

```
(dynamic-rerequire module-path [#:verbosity verbosity]) → (listof path?)
```

*module-path*: `module-path?`
*verbosity*: `(or/c 'all 'reload 'none) = 'reload`

Like `(dynamic-require module-path 0)`, but with reloading support. The `dynamic-rerequire` function is intended for use in an interactive environment, especially via `enter!`.

If invoking `module-path` requires loading any files, then modification dates of the files are recorded. If the file is modified, then a later `dynamic-rerequire` re-loads the module from source; see also §1.9.4 “Module Redeclarations”. Similarly if a later `dynamic-rerequire` transitively requires a modified module, then the required module is re-loaded.

Re-loading support works only for modules that are first loaded (either directly or indirectly through transitive requires) via `dynamic-rerequire`. The returned list contains the absolute paths to the modules that were reloaded on this call to `dynamic-rerequire`. If the returned list is empty, no modules were changed or loaded.

When `enter!` loads or re-loads a module from a file, it can print a message to `(current-error-port)`, depending on `verbosity`: `'all prints a message for all loads and re-loads, `'reload` prints a message only for re-loaded modules, and `'none` disables printouts.

### 18.5 Debugging

Racket’s built-in debugging support is limited to context (i.e., “stack trace”) information that is printed with an exception. In some cases, for BC implementation of Racket, disabling the JIT compiler can affect context information. For the CS implementation of Racket, setting the `PLT_CS_DEBUG` environment variable causes compilation to record expression-level context information, instead of just function-level information.

The `errortrace` library supports more consistent (independent of the compiler) and precise context information. The `racket/trace` library provides simple tracing support. Finally, the DrRacket programming environment provides much more debugging support.

#### 18.5.1 Tracing

```
(require racket/trace) package: base
```

The bindings documented in this section are provided by the `racket/trace` library, not `racket/base` or `racket`. 1270
The **racket/trace** library mimics the tracing facility available in Chez Scheme.

```
(trace id ...)
```

Each `id` must be bound to a procedure in the environment of the `trace` expression. Each `id` is set!ed to a new procedure that traces procedure calls and returns by printing the arguments and results of the call via `current-trace-notify`. If multiple values are returned, each value is displayed starting on a separate line.

When traced procedures invoke each other, nested invocations are shown by printing a nesting prefix. If the nesting depth grows to ten and beyond, a number is printed to show the actual nesting depth.

The `trace` form can be used on an identifier that is already traced. In this case, assuming that the variable’s value has not been changed, `trace` has no effect. If the variable has been changed to a different procedure, then a new trace is installed.

Tracing respects tail calls to preserve loops, but its effect may be visible through continuation marks. When a call to a traced procedure occurs in tail position with respect to a previous traced call, then the tailness of the call is preserved (and the result of the call is not printed for the tail call, because the same result will be printed for an enclosing call). Otherwise, however, the body of a traced procedure is not evaluated in tail position with respect to a call to the procedure.

The result of a `trace` expression is `#<void>`.

Examples:

```
> (define (f x) (if (zero? x) 0 (add1 (f (sub1 x)))))
> (trace f)
> (f 10)
> (f 9)
> (f 8)
> (f 7)
> (f 5)
> (f 4)
> (f 3)
> (f 2)
> (f 1)
> [10] (f 0)
< < < <[10] 0
< < < < < 1
< < < < < 2
< < < < 3
< < < < 4
```
trace can also be used to debug syntax transformers. This is verbose to do directly with trace; refer to trace-definition-syntax for a simpler way to do this.

Examples:

```
> (require (for-syntax racket/trace))
> (begin-for-syntax
   (define let
     (syntax-rules ()
       [(_ ([x v]) e) ((lambda (x) e) v)]))
   (trace let))
> (define-syntax let
> (let ([x 120]) x)
> (_let #<syntax:eval:9:0 (let ((x 120)) x)>>
<#<syntax:eval:9:0 ((lambda (x) x) 120)>
120
```

When tracing syntax transformers, it may be helpful to modify current-trace-print-args and current-trace-print-results to make the trace output more readable; see current-trace-print-args for an extended example.

```
(trace-define id expr)
(trace-define (head args) body ...+)
```

The trace-define form is short-hand for first defining a function then tracing it. This form supports all define forms.

Examples:

```
> (trace-define (f x) (if (zero? x) 0 (add1 (f (sub1 x)))))
> (f 5)
> (f 5)
> (f 4)
> >(f 3)
> > (f 2)
> > > (f 1)
> > > > (f 0)
```
Examples:

```
> (trace-define ((+ n n) x) (+ n x))
> (map (+n 5) (list 1 3 4))
>(+n 5)
<#<procedure>
'(6 8 9)
```

```
(trace-define-syntax id expr)
(trace-define-syntax (head args) body ...+)
```

The trace-define-syntax form is short-hand for first defining a syntax transformer then tracing it. This form supports all define-syntax forms.

For example:

Examples:

```
> (trace-define-syntax fact
   (syntax-rules ()
      [(_ x) 120]))
> (fact 5)
>(fact #<syntax:eval:15:0 (fact 5)>)
<#<syntax:eval:15:0 120> 120
```

By default, trace prints out syntax objects when tracing a syntax transformer. This can result in too much output if you do not need to see, e.g., source information. To get more readable output by printing syntax objects as datums, we can modify the current-trace-print-args and current-trace-print-results. See current-trace-print-args for an example.

```
(trace-lambda [#:name id] args expr)
```

The trace-lambda form enables tracing an anonymous function. This form will attempt to infer a name using syntax-local-infer-name, or a name can be specified using the
optional #:name argument. A syntax error is raised if a name is not given and a name cannot be inferred.

Example:

```
> ((trace-lambda (x) 120) 5)
> (eval:16:0 5)
<120
120
```

```
(trace-let id ([arg expr] ...) body ...+)
```

The trace-let form enables tracing a named let.

Example:

```
> (trace-let f ([x 5])
   (if (zero? x)
       1
       (* x (f (sub1 x))))
> (f 5)
> (f 4)
> >(f 3)
> >>(f 2)
> >>>(f 1)
> >>>(f 0)
< < < 1
< < <1
< < 2
< <6
< 24
<120
120
```

```
(untrace id ...)
```

Undoes the effects of the trace form for each id, setting each id back to the untraced procedure, but only if the current value of id is a traced procedure. If the current value of a id is not a procedure installed by trace, then the variable is not changed.

The result of an untrace expression is #<void>.

```
(current-trace-notify) → (string? . -→ . any)
(current-trace-notify proc) → void?
proc : (string? . -→ . any)
```
A parameter that determines the way that trace output is displayed. The string given to \texttt{proc} is a trace; it does not end with a newline, but it may contain internal newlines. Each call or result is converted into a string using \texttt{pretty-print}. The parameter’s default value prints the given string followed by a newline to \texttt{(current-output-port)}.

\begin{verbatim}
(trace-call id proc #:<kw> kw-arg ...) \to\ any/c
  id : symbol?
  proc : procedure?
  kw-arg : any/c
\end{verbatim}

Calls \texttt{proc} with the arguments supplied in \texttt{args}, and possibly using keyword arguments. Also prints out the trace information during the call, as described above in the docs for \texttt{trace}, using \texttt{id} as the name of \texttt{proc}.

\begin{verbatim}
(current-trace-print-args) \to\ (-> symbol?
  list?
  (listof keyword?)
  list?
  number?
  void?)
\end{verbatim}

\begin{verbatim}
(current-trace-print-args trace-print-args) \to\ void?
  trace-print-args : (-> symbol?
    list?
    (listof keyword?)
    list?
    number?
    void?)
\end{verbatim}

The value of this parameter is invoked to print out the arguments of a traced call. It receives the name of the function, the function’s ordinary arguments, its keywords, the values of the keywords, and a number indicating the depth of the call.

Modifying this and \texttt{current-trace-print-results} is useful to get more readable or additional output when tracing syntax transformers. For example, we can use \texttt{debug-scopes} to add scopes information to the trace, (see \texttt{debug-scopes} for an example), or remove source location information to just display the shape of the syntax object.

In this example, we update the printers \texttt{current-trace-print-args} and \texttt{current-trace-print-results} by storing the current printers \texttt{(ctp\texttt{a} and ctpr)} to cast syntax objects to datum using \texttt{syntax->datum} and then pass the transformed arguments and results to the previous printer. When tracing, syntax arguments will be displayed without source location information, shortening the output.

Examples:

\begin{verbatim}
> (require (for-syntex racket/trace))
\end{verbatim}
We must take care when modifying these parameters, especially when the transformation makes assumptions about or changes the type of the argument/result of the traced identifier. This modification of \texttt{current-trace-print-args} and \texttt{current-trace-print-results} is an imperative update, and will affect all traced identifiers. This example assumes all arguments and results to all traced functions will be syntax objects, which is the case only if you are only tracing syntax transformers. If used as-is, the above code could result in type errors when tracing both functions and syntax transformers. It would be better to use \texttt{syntax->datum} only when the argument or result is actually a syntax object, for example, by defining \texttt{maybe-syntax->datum} as follows.

Examples:

\begin{verbatim}
> (require (for-syntax racket/trace))
> (begin-for-syntax
  (define (maybe-syntax->datum syn?)
    (if (syntax? syn?)
      (syntax->datum syn?)
      syn?))
  (current-trace-print-args
    (let ([ctpa (current-trace-print-args)])
      (lambda (s l kw l2 n)
        (ctpa s (map maybe-syntax->datum l) kw 12 n))))
  (current-trace-print-results
    (let ([ctpr (current-trace-print-results)])
      (lambda (s r n)
        (ctpr s (map maybe-syntax->datum r) n))))
> (trace-define-syntax fact
  (syntax-rules ()
    [(_ x) 120]))
> (fact 5)
>(fact '(fact 5))
<120
120
\end{verbatim}
> (trace-define (run-time-fact n) (apply * (build-list n add1)))
> (require (for-syntax syntax/parse))
> (trace-define-syntax (fact syn)
  (syntax-parse syn
   [(_ x:nat) (precompute-fact syn (syntax->datum #'
   x))]  
   [(_ x) #'(run-time-fact x)]))

> (fact 5)
> (fact '(fact 5))
> (precompute-fact '(fact 5) 5)
<120
120
> (fact (+ 2 3))
> (fact '(fact (+ 2 3)))
<'(run-time-fact (+ 2 3))
>(run-time-fact 5)
<120
120

(current-trace-print-results) → (-> symbol?
  list?
  number?
  any)
(current-trace-print-results trace-print-results) → void?
trace-print-results : (-> symbol?
  list?
  number?
  any)

The value of this parameter is invoked to print out the results of a traced call. It receives the
name of the function, the function’s results, and a number indicating the depth of the call.

(current-prefix-in) → string?
(current-prefix-in prefix) → void?
  prefix : string?
This string is used by the default value of current-trace-print-args indicating that the
current line is showing the a call to a traced function.

It defaults to ">".

(current-prefix-out) → string?
(current-prefix-out prefix) → void?
  prefix : string?
This string is used by the default value of current-trace-print-results indicating that
the current line is showing the result of a traced call.
18.6 Controlling and Inspecting Compilation

Racket programs and expressions are compiled automatically and on-the-fly. The `raco make` tool (see §1 “raco make: Compiling Source to Bytecode”) can compile a Racket module to a compiled ".zo" file, but that kind of ahead-to-time compilation simply allows a program to start more quickly, and it does not affect the performance of a Racket program.

18.6.1 Compilation Modes

All Racket variants suppose a machine-independent compilation mode, which generates compiled ".zo" files that work with all Racket variants on all platforms. To select machine-independent compilation mode, set the `current-compile-target-machine` parameter to `#f` or supplying the `--compile-any/-M` flag on startup. See `current-compile-target-machine` for more information.

Other compilation modes depend on the Racket implementation (see §1.5 “Implementations”).

BC Compilation Modes

The BC implementation of Racket supports two compilation modes: bytecode and machine-independent. The bytecode format is also machine-independent in the sense that it works the same on all operating systems for the BC implementation of Racket, but it does not work with the CS implementation of Racket.

Bytecode is further compiled to machine code at run time, unless the JIT compiler is disabled. See `eval-jit-enabled`.

CS Compilation Modes

The CS implementation of Racket supports several compilation modes: machine code, machine-independent, interpreted, and JIT. Machine code is the primary mode, and the machine-independent mode is the same as for BC. Interpreted mode uses an interpreter at the level of core linklet forms with no compilation. JIT mode triggers compilation of individual function forms on demand.

The default mode is a hybrid of machine-code and interpreter modes, where interpreter mode is used only for the outer contour of an especially large linklet, and machine-code mode is used for functions that are small enough within that outer contour. “Small enough” is determined by the `PLT_CS_COMPILE_LIMIT` environment variable, and the default value of 10000 means that most Racket modules have no interpreted component.
JIT compilation mode is used only if the `PLT_CS_JIT` environment variable is set on startup, otherwise pure interpreter mode is used only if `PLT_CS_INTERP` is set on startup, and the default hybrid machine code and interpreter mode is used if `PLT_CS_MACH` is set and `PLT_CS_JIT` is not set or if none of those environment variables is set. A module compiled in any mode can be loaded into the CS variant of Racket independent of the current compilation mode.

The `PLT_CS_DEBUG` environment variable, as described in §18.5 “Debugging”, affects only compilation in machine-code mode. Generated machine code is much larger when `PLT_CS_DEBUG` is enabled, but performance is not otherwise affected.

### 18.6.2 Inspecting Compiler Passes

When the `PLT_LINKLET_SHOW` environment variable is set on startup, the Racket process’s standard output shows intermediate compiled forms whenever a Racket form is compiled. For all Racket variants, the output shows one or more linklets that are generated from the original Racket form.

For the CS implementation of Racket, a “schemified” version of the linklet is also shown as the translation of the `linklet` form to a Chez Scheme procedure form. The following environment variables imply `PLT_LINKLET_SHOW` and show additional intermediate compiled forms or adjust the way forms are displayed:

- `PLT_LINKLET_SHOW_GENSYM` — prints full generated names, instead of abbreviations that may conflate different symbols
- `PLT_LINKLET_SHOW_PRE_JIT` — shows a schemified forms before a transformation to JIT mode, which applies only when `PLT_CS_JIT` is set
- `PLT_LINKLET_SHOW_LAMBDA` — shows individual schemified forms that are compiled within a larger form that has an interpreted outer contour
- `PLT_LINKLET_SHOW_POST_LAMBDA` — shows an outer form after inner individual forms are compiled
- `PLT_LINKLET_SHOW_POST_INTERP` — shows an outer form after its transformation to interpretable form
- `PLT_LINKLET_SHOW_JIT_DEMAND` — shows JIT compilation of form that were previously prepared by compilation with `PLT_CS_JIT` set
- `PLT_LINKLET_SHOW_PATHS` — show lifted path and serialization information alongside a schemified form
- `PLT_LINKLET_SHOWKNOWN` — show recorded known-binding information alongside a schemified form
• **PLT_LINKLET_SHOW_CP0** — show a schemified form after transformation by Chez Scheme’s front-end optimizer

• **PLT_LINKLET_SHOW_ASSEMBLY** — show the compiled form of a schemified linklet in Chez Scheme’s abstraction of machine instructions

When the PLT_LINKLET_TIMES environment variable is set on startup, then Racket prints cumulative timing information about compilation and evaluation times on exit. When the PLT_EXPANDER_TIMES environment variable is set, information about macro-expansion time is printed on exit.

### 18.7 Kernel Forms and Functions

```
#lang racket/kernel    package: base
```

The `racket/kernel` library is a cross-phase persistent module that provides a minimal set of syntactic forms and functions.

“Minimal” means that `racket/kernel` includes only forms that are built into the Racket compiler and only functions that are built into the run-time system. Currently, the set of bindings is not especially small, nor is it particularly well-defined, since the set of built-in functions can change frequently. Use `racket/kernel` with care, and beware that its use can create compatibility problems.

The `racket/kernel` module exports all of the bindings in the grammar of fully expanded programs (see §1.2.3.1 “Fully Expanded Programs”), but it provides `#%plain-lambda` as `lambda` and `λ`, `#%plain-app` as `#%app`, and `#%plain-module-begin` as `#%module-begin`. Aside from `#%datum` (which expands to `quote`), `racket/kernel` provides no other syntactic bindings.

The `racket/kernel` module also exports many of the function bindings from `racket/base`, and it exports a few other functions that are not exported by `racket/base` because `racket/base` exports improved variants. The exact set of function bindings exported by `racket/kernel` is unspecified and subject to change across versions.

```
(require racket/kernel/init)          package: base
```

The `racket/kernel/init` library re-provides all of `racket/kernel`. It also provides `#%top-interaction`, which makes `racket/kernel/init` useful with the `-I` command-line flag for `racket`.
Bibliography


1281


+nan.t, 231 ->.*, 640
+rv, 1259 ->*m, 574
++, 60 ->d, 644
., 60 ->dm, 575
-. 187 ->extfl, 233
-- , 1258 ->fl, 222
--addon, 1257 ->i, 641
--addon, 1257 ->m, 574
--back, 1257 -A, 1257
--binary, 1258 -A, 1257
--collects, 1257 -b, 1258
--compile-any, 1258 -background, 1259
--compiled, 1257 -bg, 1259
--config, 1257 -C, 1258
--cross, 1258 -c, 1258
--cross, 1258 -c, 1257
--eval, 1255 -d, 1258
--exe, 1258 -display, 1259
--help, 1258 -E, 1258
--lib, 1256 -e, 1255
--load, 1256 -f, 1256
--main, 1256 -fg, 1259
--name, 1258 -fn, 1259
--no-compiled, 1257 -font, 1259
--no-delay, 1258 -foreground, 1259
--no-init-file, 1257 -g, 1257
--no-jit, 1258 -geometry, 1259
--no-lib, 1256 -h, 1258
--no-user-path, 1257 -i, 1257
--no-yield, 1257 -i, 1256
--repl, 1256 -iconic, 1259
--require, 1256 -inf.0, 179
--require-script, 1256 -inf.f, 179
--script, 1256 -inf.t, 231
--search, 1257 -j, 1258
--stdout, 1258 -j, 1258
--syslog, 1258 -k, 1257
--text-repl, 1257 -k, 1256
--version, 1257 -L, 1258
--warn, 1258 -l, 1256
--wm-class, 1258 -M, 1258
->, 636 -m, 1256
<table>
<thead>
<tr>
<th>Term</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>contract?</td>
<td>688</td>
</tr>
<tr>
<td>contracted</td>
<td>596</td>
</tr>
<tr>
<td>Contracted Dictionaries</td>
<td>446</td>
</tr>
<tr>
<td>Contracts</td>
<td>609</td>
</tr>
<tr>
<td>Contrats</td>
<td>609</td>
</tr>
<tr>
<td>Contracts and Impersonators on Asynchronous Channels</td>
<td>789</td>
</tr>
<tr>
<td>Contracts as structs</td>
<td>679</td>
</tr>
<tr>
<td>control</td>
<td>757</td>
</tr>
<tr>
<td>Control Flow</td>
<td>722</td>
</tr>
<tr>
<td>control-at</td>
<td>758</td>
</tr>
<tr>
<td>control0</td>
<td>759</td>
</tr>
<tr>
<td>control0-at</td>
<td>759</td>
</tr>
<tr>
<td>Controlling and Inspecting Compilation</td>
<td>1278</td>
</tr>
<tr>
<td>convert-relative-module-path</td>
<td>865</td>
</tr>
<tr>
<td>convert-stream</td>
<td>951</td>
</tr>
<tr>
<td>Converting Values to Strings</td>
<td>252</td>
</tr>
<tr>
<td>copy-directory/files</td>
<td>1159</td>
</tr>
<tr>
<td>copy-file</td>
<td>1148</td>
</tr>
<tr>
<td>copy-port</td>
<td>952</td>
</tr>
<tr>
<td>Copying and Updating Structures</td>
<td>519</td>
</tr>
<tr>
<td>Copying Streams</td>
<td>951</td>
</tr>
<tr>
<td>correlated objects</td>
<td>1109</td>
</tr>
<tr>
<td>correlated-&gt;datum</td>
<td>1118</td>
</tr>
<tr>
<td>correlated-column</td>
<td>1117</td>
</tr>
<tr>
<td>correlated-e</td>
<td>1117</td>
</tr>
<tr>
<td>correlated-line</td>
<td>1117</td>
</tr>
<tr>
<td>correlated-position</td>
<td>1117</td>
</tr>
<tr>
<td>correlated-property</td>
<td>1118</td>
</tr>
<tr>
<td>correlated-property-symbol-keys</td>
<td>1118</td>
</tr>
<tr>
<td>correlated-source</td>
<td>1117</td>
</tr>
<tr>
<td>correlated-span</td>
<td>1117</td>
</tr>
<tr>
<td>correlated?</td>
<td>1117</td>
</tr>
<tr>
<td>cos</td>
<td>201</td>
</tr>
<tr>
<td>cosh</td>
<td>217</td>
</tr>
<tr>
<td>count</td>
<td>357</td>
</tr>
<tr>
<td>count property</td>
<td>699</td>
</tr>
<tr>
<td>Counting Positions, Lines, and Columns</td>
<td>900</td>
</tr>
<tr>
<td>Creating and Touching Futures</td>
<td>797</td>
</tr>
<tr>
<td>Creating and Using Asynchronous Channels</td>
<td>787</td>
</tr>
<tr>
<td>Creating Classes</td>
<td>533</td>
</tr>
<tr>
<td>Creating formatted identifiers</td>
<td>886</td>
</tr>
<tr>
<td>Creating Interfaces</td>
<td>532</td>
</tr>
<tr>
<td>Creating Loggers</td>
<td>1194</td>
</tr>
<tr>
<td>Creating Objects</td>
<td>556</td>
</tr>
<tr>
<td>Creating Ports</td>
<td>938</td>
</tr>
<tr>
<td>Creating Structure Types</td>
<td>505</td>
</tr>
<tr>
<td>Creating Threads</td>
<td>771</td>
</tr>
<tr>
<td>Creating Units</td>
<td>593</td>
</tr>
<tr>
<td>'cross,</td>
<td>1210</td>
</tr>
<tr>
<td>cross-phase persistent</td>
<td>24</td>
</tr>
<tr>
<td>Cross-Phase Persistent Module Declarations</td>
<td>49</td>
</tr>
<tr>
<td>Cross-Phase Persistent Modules</td>
<td>24</td>
</tr>
<tr>
<td>crypto-random-bytes</td>
<td>210</td>
</tr>
<tr>
<td>Cryptographic Hashing</td>
<td>1012</td>
</tr>
<tr>
<td>CS</td>
<td>76</td>
</tr>
<tr>
<td>'cs,</td>
<td>287</td>
</tr>
<tr>
<td>'cs,</td>
<td>1209</td>
</tr>
<tr>
<td>CS Compilation Modes</td>
<td>1278</td>
</tr>
<tr>
<td>cupto,</td>
<td>760</td>
</tr>
<tr>
<td>current custodian</td>
<td>29</td>
</tr>
<tr>
<td>current logger</td>
<td>1194</td>
</tr>
<tr>
<td>current namespace</td>
<td>47</td>
</tr>
<tr>
<td>current plumber</td>
<td>1086</td>
</tr>
<tr>
<td>current-blame-format</td>
<td>678</td>
</tr>
<tr>
<td>current-break-parameterization</td>
<td>768</td>
</tr>
<tr>
<td>current-code-inspector</td>
<td>1085</td>
</tr>
<tr>
<td>current-command-line-arguments</td>
<td>1211</td>
</tr>
<tr>
<td>current-command-line-arguments</td>
<td>1259</td>
</tr>
<tr>
<td>current-compile</td>
<td>1033</td>
</tr>
<tr>
<td>current-compile-target-machine</td>
<td>1034</td>
</tr>
<tr>
<td>current-compiled-file-roots</td>
<td>1030</td>
</tr>
<tr>
<td>current-continuation-marks</td>
<td>761</td>
</tr>
<tr>
<td>current-contract-region</td>
<td>662</td>
</tr>
<tr>
<td>current-custodian</td>
<td>1078</td>
</tr>
<tr>
<td>current-date</td>
<td>1204</td>
</tr>
</tbody>
</table>
Custom Hash Tables, 447
Custom Ports, 914
custom ports, 914
custom-print-quotable-accessor, 1000
custom-print-quotable?, 1000
custom-write-accessor, 999
custom-write?, 999
Customized Unreachable Reporting, 769
Customizing Evaluators, 1093
Data-structure Contracts, 610
Datatypes, 169
date, 1201
Date Utilities, 1204
date*, 1202
date*->seconds, 1205
date*->nanosecond, 1202
date*->time-zone-name, 1202
date*?, 1202
date->julian/scaliger, 1205
date->julian/scalinger, 1205
date->seconds, 1204
date->string, 1204
date-day, 1201
date-display-format, 1204
date-dst?, 1201
date-hour, 1201
date-minute, 1201
date-month, 1201
date-second, 1201
date-time-zone-offset, 1201
date-week-day, 1201
date-year, 1201
date-year-day, 1201
date?, 1201
datum, 51
datum->correlated, 1118
datum->syntax, 830
datum-intern-literal, 832
Debugging, 1270
declared, 20
Declaring Paths Needed at Run Time, 1152
Deep time, 1093
default method, 512
default-continuation-prompt-tag, 750
define, 134
define-compound-unit, 601
define-compound-unit/infer, 601
define-contract-struct, 651
define-custom-hash-types, 447
define-custom-set-types, 470
define-for-syntax, 137
define-generics, 512
define-inline, 165
define-local-member-name, 554
define-ipex, 1195
define-match-expander, 717
define-member-name, 555
define-module-boundary-contract, 663
define-module-boundary-contract, 663
define-namespace-anchor, 1014
define-opt/c, 692
define-provide-syntax, 139
define-rename-transformer-parameter, 871
define-require-syntax, 138
define-runtime-module-path, 1156
define-runtime-module-path-index, 1155
define-runtime-path, 1153
define-runtime-path-list, 1155
define-runtime-paths, 1155
define-sequence-syntax, 157
define-serializable-class, 581
define-serializable-class*, 580
define-serializable-struct, 1006
define-serializable-struct/versions, 1008
define-signature, 595
define-signature-form, 604
define-struct, 503
define-struct/contract, 600
define-struct/derived, 604
<table>
<thead>
<tr>
<th>Definition</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>define-syntax</td>
<td>136</td>
</tr>
<tr>
<td>define-syntax-parameter</td>
<td>870</td>
</tr>
<tr>
<td>define-syntax-rule</td>
<td>824</td>
</tr>
<tr>
<td>define-synaxes</td>
<td>136</td>
</tr>
<tr>
<td>define-unit</td>
<td>600</td>
</tr>
<tr>
<td>define-unit-binding</td>
<td>601</td>
</tr>
<tr>
<td>define-unit-from-context</td>
<td>602</td>
</tr>
<tr>
<td>define-unit/contract</td>
<td>605</td>
</tr>
<tr>
<td>define-unit/new-import-export</td>
<td>603</td>
</tr>
<tr>
<td>define-values</td>
<td>135</td>
</tr>
<tr>
<td>define-values-for-export</td>
<td>596</td>
</tr>
<tr>
<td>define-values-for-syntax</td>
<td>138</td>
</tr>
<tr>
<td>define-values/invoking-unit</td>
<td>598</td>
</tr>
<tr>
<td>define-values/invoking-unit/infer</td>
<td>602</td>
</tr>
<tr>
<td>define/augment</td>
<td>547</td>
</tr>
<tr>
<td>define/augment-final</td>
<td>548</td>
</tr>
<tr>
<td>define/augride</td>
<td>547</td>
</tr>
<tr>
<td>define/contract</td>
<td>658</td>
</tr>
<tr>
<td>define/final-prop</td>
<td>686</td>
</tr>
<tr>
<td>define/generic</td>
<td>514</td>
</tr>
<tr>
<td>define/match</td>
<td>714</td>
</tr>
<tr>
<td>define/overment</td>
<td>547</td>
</tr>
<tr>
<td>define/override</td>
<td>547</td>
</tr>
<tr>
<td>define/override-final</td>
<td>547</td>
</tr>
<tr>
<td>define/private</td>
<td>548</td>
</tr>
<tr>
<td>define/public</td>
<td>546</td>
</tr>
<tr>
<td>define/public-final</td>
<td>547</td>
</tr>
<tr>
<td>define/pubment</td>
<td>547</td>
</tr>
<tr>
<td>define/subexpression-pos-prop</td>
<td>687</td>
</tr>
<tr>
<td>define/with-syntax</td>
<td>888</td>
</tr>
<tr>
<td>Delayed Evaluation</td>
<td>746</td>
</tr>
<tr>
<td>'delete'</td>
<td>1076</td>
</tr>
<tr>
<td>delete-directory</td>
<td>1150</td>
</tr>
<tr>
<td>delete-directory/files</td>
<td>1159</td>
</tr>
<tr>
<td>delete-file</td>
<td>1145</td>
</tr>
<tr>
<td>delimited continuation</td>
<td>26</td>
</tr>
<tr>
<td>Delimiters and Dispatch</td>
<td>51</td>
</tr>
<tr>
<td>denominator</td>
<td>194</td>
</tr>
<tr>
<td>depth marker</td>
<td>814</td>
</tr>
<tr>
<td>derived class</td>
<td>531</td>
</tr>
<tr>
<td>Derived Dictionary Methods</td>
<td>438</td>
</tr>
<tr>
<td>Deriving New Iteration Forms</td>
<td>155</td>
</tr>
<tr>
<td>deserialize</td>
<td>1001</td>
</tr>
<tr>
<td>deserialize-module-guard</td>
<td>1005</td>
</tr>
<tr>
<td>'desk-dir'</td>
<td>1142</td>
</tr>
<tr>
<td>Detecting Filesystem Changes</td>
<td>1151</td>
</tr>
<tr>
<td>dict-&gt;list</td>
<td>445</td>
</tr>
<tr>
<td>dict-can-functional-set?</td>
<td>432</td>
</tr>
<tr>
<td>dict-can-remove-keys?</td>
<td>432</td>
</tr>
<tr>
<td>dict-clear</td>
<td>443</td>
</tr>
<tr>
<td>dict-clear!</td>
<td>444</td>
</tr>
<tr>
<td>dict-copy</td>
<td>443</td>
</tr>
<tr>
<td>dict-count</td>
<td>443</td>
</tr>
<tr>
<td>dict-empty?</td>
<td>442</td>
</tr>
<tr>
<td>dict-for-each</td>
<td>442</td>
</tr>
<tr>
<td>dict-has-key?</td>
<td>438</td>
</tr>
<tr>
<td>dict-implements/c</td>
<td>431</td>
</tr>
<tr>
<td>dict-implements?</td>
<td>430</td>
</tr>
<tr>
<td>dict-iterate-contract</td>
<td>447</td>
</tr>
<tr>
<td>dict-iterate-first</td>
<td>436</td>
</tr>
<tr>
<td>dict-iterate-key</td>
<td>437</td>
</tr>
<tr>
<td>dict-iterate-next</td>
<td>437</td>
</tr>
<tr>
<td>dict-iterate-value</td>
<td>438</td>
</tr>
<tr>
<td>dict-key-contract</td>
<td>447</td>
</tr>
<tr>
<td>dict-keys</td>
<td>444</td>
</tr>
<tr>
<td>dict-map</td>
<td>442</td>
</tr>
<tr>
<td>dict-mutable?</td>
<td>431</td>
</tr>
<tr>
<td>dict-ref</td>
<td>434</td>
</tr>
<tr>
<td>dict-ref!</td>
<td>440</td>
</tr>
<tr>
<td>dict-remove</td>
<td>436</td>
</tr>
<tr>
<td>dict-remove!</td>
<td>436</td>
</tr>
<tr>
<td>dict-set</td>
<td>435</td>
</tr>
</tbody>
</table>
IEEE floating-point numbers, 179
if, 130
if/c, 694
imag-part, 203
Immutable Cyclic Data, 363
immutable?, 176
impersonate-async-channel, 790
impersonate-box, 1060
impersonate-channel, 1062
impersonate-continuation-mark-key, 1064
impersonate-generics, 516
impersonate-hash, 1061
impersonate-hash-set, 468
impersonate-procedure, 1054
impersonate-procedure*, 1057
impersonate-prompt-tag, 1063
impersonate-struct, 1058
impersonate-vector, 1059
impersonate-vector*, 1060
impersonator, 1051
Impersonator Constructors, 1053
Impersonator contracts, 610
Impersonator Properties, 1074
impersonator-properties, 1052
impersonator-property-accessor, 1074
impersonator-property-descriptor, 1074
impersonator-property-predicate, 1074
impersonator-contract?, 689
impersonator-ephemeron, 1054
impersonator-of?, 1053
impersonator-prop:application-mark, 1075
impersonator-prop:blame, 680
impersonator-prop:collapsible, 698
impersonator-prop:contracted, 679
impersonator-property-accessor-procedure?, 1074
impersonator-property?, 1074
impersonator?, 1052
Impersonators and Chaperones, 1051
implementation?, 586
implementation?/c, 578
Implementations, 76
implemented generic method, 513
Implementing Equality for Custom Types, 173
implements, 531
implicit-made-explicit properties, 36
implies, 178
import, 864
import-local-id, 864
import-mode, 864
import-orig-mode, 864
import-orig-stx, 864
import-req-mode, 864
import-source, 865
import-source-mod-path-stx, 865
import-source-mode, 865
import-source?, 865
import-src-mod-path, 864
import-src-sym, 864
import?, 864
Importing and Exporting: require and provide, 89
Importing Modules Lazily: lazy-require, 165
in-bytes, 401
in-bytes-lines, 402
in-combinations, 360
in-cycle, 408
in-dict, 445
in-dict-keys, 445
in-dict-pairs, 446
in-dict-values, 446
in-directory, 406
in-ephemeron-hash, 406
in-ephemeron-hash-keys, 406
in-ephemeron-hash-pairs, 406
in-ephemeron-hash-values, 406
in-extflvector, 235
in-flvector, 224
in-fxvector, 230
in-generator 426
in-hash 402
in-hash-keys 403
in-hash-pairs 404
in-hash-values 403
in-immutable-hash 405
in-immutable-hash-keys 405
in-immutable-hash-pairs 405
in-immutable-hash-values 405
in-immutable-set 456
in-inclusive-range 398
in-indexed 408
in-input-port-bytes 402
in-input-port-chars 402
in-lines 402
in-list 399
in-mlist 399
in-mutable-hash 404
in-mutable-hash-keys 404
in-mutable-hash-pairs 405
in-mutable-hash-values 405
in-mutable-set 456
in-naturals 399
in-parallel 409
in-permutations 360
in-port 402
in-powerset 360
in-producer 407
in-range 398
in-rearrangements 360
in-sequences 408
in-set 468
in-slice 417
in-stream 420
in-string 401
in-syntax 417
in-value 408
in-values*sequence 409
in-values-sequence 409
in-vector 400
in-weak-hash 405
in-weak-hash-keys 405
in-weak-hash-pairs 405
in-weak-hash-values 405
in-weak-set 456
include 885
include-at/relative-to 885
include-at/relative-to/reader 886
include/reader 886
inclusive-range 358
index-of 348
index-where 348
indexes-of 348
indexes-where 349
inexact number 179
inexact->exact 186
inexact-real? 183
inexact? 185
'infer 1210
Inferred Linking 599
Inferred Value Names 49
'inferred-name 49
infinite-generator 426
infinite? 218
infinity 179
infix 57
Information on Expanded Modules 882
inherit 544
inherit-field 538
inherit/inner 545
inherit/super 545
inheritance 531
Inherited and Superclass Methods 552
init 537
Init Libraries 1254
init-depend 597
'init-dir 1141
init-field 537
'init-file 1141
init-rest 538
Initialization 1253
Initialization Variables 548
initiate 397
inner 552
inode, 909
inode, 1148
Input and Output, 893
input port, 893
input ports, pattern matching, 293
input-port-append, 938
input-port?, 895
inside-edge scope, 43
inspect, 537
Inspecting Compiler Passes, 1279
inspector, 1081
inspector-superior?, 1081
inspector?, 1081
instance-data, 1115
instance-describe-variable!, 1116
instance-name, 1115
instance-set-variable-value!, 1116
instance-unset-variable!, 1116
instance-variable-names, 1115
instance-variable-value, 1115
instance?, 1115
instanceof/c, 575
instantiate, 557
instantiate-linklet, 1113
instantiates, 20
instantiation, 20
integer->char, 282
integer->integer-bytes, 213
integer-bytes->integer, 213
integer-in, 615
integer-length, 207
integer-sqrt, 197
integer-sqrt/remainder, 197
integer?, 181
integers, 179
Interacting with Evaluators, 1103
interaction port handler, 1032
Interaction Wrapper: #%top-interaction, 163
Interactive Help, 1267
Interactive Module Loading, 1268
interface, 532
interface*, 533
interface->method-names, 587
interface-extension?, 587
interface?, 583
Internal and External Names, 553
Internal Definitions, 42
internal-definition context, 37
Internal-Definition Limiting:
  #%stratified-body, 164
internal-definition-context-apply, 891
internal-definition-context-binding-identifiers, 851
internal-definition-context-introduce, 851
internal-definition-context-seal, 851
internal-definition-context?, 849
interned, 51
Introducing Bindings, 37
invariant-assertion, 662
invoke-unit, 597
invoke-unit/infer, 602
invoked, 593
Invoking Units, 597
IP_MULTICAST_LOOP, 1182
IP_MULTICAST_TTL, 1182
IP_TTL, 1181
is-a?, 585
is-a?/c, 578
Iteration and Comprehension Forms, 143
iteration pattern variable, 821
Iterations and Comprehensions: for, for/list, ..., 143
iterator, 596
JIT, 1035
julian/scaliger->string, 1205
julian/scalinger->string, 1205
Kernel Forms and Functions, 1280
keyword, 319
keyword->immutable-string, 321
keyword->string, 320
list->mutable-set, 455
list->mutable-seteq, 455
list->mutable-seteqv, 455
list->set, 455
list->seteq, 455
list->seteqv, 455
list->string, 240
list->vector, 368
list->weak-set, 455
list->weak-seteq, 455
list->weak-seteqv, 455
list-contract?, 689
list-prefix?, 352
list-ref, 325
list-set, 347
list-tail, 325
list-update, 347
list/c, 622
list?, 323
listen-port-number?, 1173
listof, 619
Literals: quote and #%datum, 113
'11, 287
'lm, 287
'lo, 287
load, 1027
load handler, 1025
load-extension, 1028
load-on-demand-enabled, 1035
load-relative, 1027
load-relative-extension, 1028
load/cd, 1027
load/use-compiled, 1030
Loading and Reloading Modules, 1269
local, 127
Local Binding Context, 41
local binding context, 41
Local Binding with Splicing Body, 873
Local Binding: let, let*, letrec, ..., 123
local bindings, 31
Local Definitions: local, 127
local variable, 19
local-expand, 844
local-expand/capture-lifts, 848
local-require, 99
local-transformer-expand, 847
local-transformer-expand/capture-lifts, 848
locale, 894
Locale-Specific String Operations, 247
locale-string-encoding, 280
Locating Paths, 1140
location, 18
Locations: #%variable-reference, 117
log, 200
log receiver, 1193
log-all-levels, 1196
log-debug, 1197
log-error, 1197
log-fatal, 1197
log-info, 1197
log-level-evt, 1196
log-level/c, 1198
log-level?, 1195
log-max-level, 1196
log-message, 1195
log-receiver?, 1198
log-warning, 1197
logger, 1193
logger-name, 1194
logger?, 1194
Logging, 1193
Logging Events, 1195
logical operators, 205
LOGNAME, 1140
'low-latency, 1210
Low-level Contract Boundaries, 663
"LPT1", 1136
"LPT2", 1136
"LPT3", 1136
"LPT4", 1136
"LPT5", 1136
"LPT6", 1136
"LPT7", 1136
make-fsemaphore, 799
make-fxvector, 229
make-generic, 563
make-generic-struct-type-property, 518
make-handle-get-preference-locked, 1166
make-hash, 381
make-hash-placeholder, 364
make-hasheq, 381
make-hasheq-placeholder, 364
make-hasheqv, 381
make-hasheqv-placeholder, 364
make-immutable-struct-type-property/generic, 517
make-import, 864
make-import-source, 865
make-input-port, 914
make-input-port/read-to-peek, 938
make-inspector, 1081
make-instance, 1115
make-interned-syntax-introducer, 861
make-keyword-procedure, 480
make-known-char-range-list, 287
make-limited-input-port, 940
make-list, 347
make-lock-file-name, 1168
make-log-receiver, 1198
make-logger, 1194
make-mixin-contract, 578
make-module-evaluator, 1088
make-none/c, 691
make-object, 557
make-output-port, 927
make-parameter, 793
make-parameter-rename-transformer, 872
make-parent-directory*, 1162
make-phantom-bytes, 1228
make-pipe, 913
make-pipe-with-specials, 940
make-placeholder, 364
make-plumber, 1086
make-polar, 203
make-prefab-struct, 521
make-proj-contract, 699
make-provide-pre-transformer, 867
make-provide-transformer, 867
make-pseudo-random-generator, 209
make-reader-graph, 365
make-readable, 991
make-rectangular, 203
make-rename-transformer, 842
make-require-transformer, 863
make-resolved-module-path, 1037
make-security-guard, 1075
make-semaphore, 786
make-serialize-info, 1008
make-set!-transformer, 840
make-shared-bytes, 269
make-shared-extflvector, 235
make-shared-flvector, 225
make-shared-fxvector, 231
make-sibling-inspector, 1081
make-special-comment, 997
make-srcloc, 744
make-string, 237
make-struct-field-accessor, 508
make-struct-field-mutator, 508
make-struct-info, 527
make-struct-type, 505
make-struct-type-property, 509
make-struct-type-property/generic, 517
make-syntax-delta-introducer, 861
make-syntax-introducer, 860
make-temporary-file, 1162
make-tentative-pretty-print-output-port, 989
make-thread-cell, 791
<table>
<thead>
<tr>
<th>Package Name</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>module-begin context</code></td>
<td>37</td>
</tr>
<tr>
<td><code>module-body-context</code></td>
<td>883</td>
</tr>
<tr>
<td><code>module-body-context-simple?</code></td>
<td>883</td>
</tr>
<tr>
<td><code>module-body-inside-context</code></td>
<td>883</td>
</tr>
<tr>
<td><code>module-compiled-cross-phase-persistent?</code></td>
<td>1046</td>
</tr>
<tr>
<td><code>module-compiled-exports</code></td>
<td>1044</td>
</tr>
<tr>
<td><code>module-compiled-imports</code></td>
<td>1044</td>
</tr>
<tr>
<td><code>module-compiled-indirect-exports</code></td>
<td>1045</td>
</tr>
<tr>
<td><code>module-compiled-language-info</code></td>
<td>1046</td>
</tr>
<tr>
<td><code>module-compiled-name</code></td>
<td>1043</td>
</tr>
<tr>
<td><code>module-compiled-submodules</code></td>
<td>1043</td>
</tr>
<tr>
<td><code>module-declared?</code></td>
<td>1048</td>
</tr>
<tr>
<td><code>module-indirect-for-meta-provides</code></td>
<td>883</td>
</tr>
<tr>
<td><code>module-indirect-provides</code></td>
<td>883</td>
</tr>
<tr>
<td><code>module-language</code></td>
<td>1046</td>
</tr>
<tr>
<td><code>module-language</code></td>
<td>87</td>
</tr>
<tr>
<td><code>module-level variable</code></td>
<td>20</td>
</tr>
<tr>
<td><code>module-path-index-join</code></td>
<td>1043</td>
</tr>
<tr>
<td><code>module-path-index-resolve</code></td>
<td>1042</td>
</tr>
<tr>
<td><code>module-path-index-split</code></td>
<td>1042</td>
</tr>
<tr>
<td><code>module-path-index-submodule</code></td>
<td>1042</td>
</tr>
<tr>
<td><code>module-path-index?</code></td>
<td>1042</td>
</tr>
<tr>
<td><code>module-path?</code></td>
<td>1037</td>
</tr>
<tr>
<td><code>module-predefined?</code></td>
<td>1050</td>
</tr>
<tr>
<td><code>module-provide-protected?</code></td>
<td>1021</td>
</tr>
<tr>
<td><code>module-syntax-provides</code></td>
<td>883</td>
</tr>
<tr>
<td><code>module-variable-provides</code></td>
<td>883</td>
</tr>
<tr>
<td>modules, re-define</td>
<td>25</td>
</tr>
<tr>
<td>modules, imports</td>
<td>89</td>
</tr>
<tr>
<td>modules, exports</td>
<td>89</td>
</tr>
<tr>
<td><code>modules</code></td>
<td>20</td>
</tr>
<tr>
<td>Modules and Module-Level Variables</td>
<td>20</td>
</tr>
<tr>
<td>Modules: module, module*, ...,</td>
<td>84</td>
</tr>
<tr>
<td><code>modulo</code></td>
<td>189</td>
</tr>
<tr>
<td>More File and Directory Utilities</td>
<td>1156</td>
</tr>
<tr>
<td>More Path Utilities</td>
<td>1130</td>
</tr>
<tr>
<td>More Port Constructors, Procedures, and Events</td>
<td>934</td>
</tr>
<tr>
<td><code>mpair?</code></td>
<td>365</td>
</tr>
<tr>
<td>Multi-in</td>
<td>112</td>
</tr>
<tr>
<td>Multiple Return Values</td>
<td>13</td>
</tr>
<tr>
<td>Multiple Values</td>
<td>722</td>
</tr>
<tr>
<td><code>multiple values</code></td>
<td>13</td>
</tr>
<tr>
<td><code>must-truncate</code></td>
<td>904</td>
</tr>
<tr>
<td><code>mutable list</code></td>
<td>365</td>
</tr>
<tr>
<td><code>mutable pair</code></td>
<td>365</td>
</tr>
<tr>
<td>Mutable Pair Constructors and Selectors,</td>
<td>365</td>
</tr>
<tr>
<td>Mutable Pairs and Lists</td>
<td>365</td>
</tr>
<tr>
<td><code>mutable-set</code></td>
<td>433</td>
</tr>
<tr>
<td><code>mutable-seteq</code></td>
<td>455</td>
</tr>
<tr>
<td><code>mutable-seteqv</code></td>
<td>455</td>
</tr>
<tr>
<td><code>mutator</code></td>
<td>398</td>
</tr>
<tr>
<td><code>nack-guard-ev</code></td>
<td>780</td>
</tr>
<tr>
<td><code>namespace</code></td>
<td>47</td>
</tr>
<tr>
<td><code>namespace-anchor-&gt;empty-namespace</code></td>
<td>1015</td>
</tr>
<tr>
<td><code>namespace-anchor-&gt;namespace</code></td>
<td>1015</td>
</tr>
<tr>
<td><code>namespace-anchor?</code></td>
<td>1015</td>
</tr>
<tr>
<td><code>namespace-attach-module</code></td>
<td>1018</td>
</tr>
<tr>
<td><code>namespace-attach-module-declaration</code></td>
<td>1020</td>
</tr>
<tr>
<td><code>namespace-base-phase</code></td>
<td>1015</td>
</tr>
<tr>
<td><code>namespace-call-with-registry-lock</code></td>
<td>1020</td>
</tr>
<tr>
<td><code>namespace-mapped-symbols</code></td>
<td>1017</td>
</tr>
<tr>
<td><code>namespace-module-identifier</code></td>
<td>1016</td>
</tr>
<tr>
<td><code>namespace-module-registry</code></td>
<td>1020</td>
</tr>
<tr>
<td><code>namespace-require</code></td>
<td>1017</td>
</tr>
<tr>
<td><code>namespace-require/constant</code></td>
<td>1018</td>
</tr>
<tr>
<td><code>namespace-require/copy</code></td>
<td>1018</td>
</tr>
<tr>
<td><code>namespace-require/expansion-time</code></td>
<td>1018</td>
</tr>
<tr>
<td><code>namespace-set-variable-value!</code></td>
<td>1016</td>
</tr>
<tr>
<td><code>namespace-symbol-&gt;identifier</code></td>
<td>1015</td>
</tr>
<tr>
<td><code>namespace-syntax-introduce</code></td>
<td>1021</td>
</tr>
<tr>
<td><code>namespace-undefine-variable!</code></td>
<td>1017</td>
</tr>
<tr>
<td><code>namespace-unprotect-module</code></td>
<td>1020</td>
</tr>
<tr>
<td><code>namespace-variable-value</code></td>
<td>1016</td>
</tr>
<tr>
<td><code>namespace?</code></td>
<td>1014</td>
</tr>
<tr>
<td>Namespaces</td>
<td>1014</td>
</tr>
</tbody>
</table>

1314
objects, 15
Objects and Imperative Update, 15
Obligation Information in Check Syntax, 685
odd?, 185
one-of/c, 617
only, 596
only-in, 91
only-meta-in, 93
'opaque, 879
open, 596
open-input-bytes, 910
open-input-file, 902
open-input-output-file, 905
open-input-string, 910
open-output-bytes, 903
open-output-file, 903
open-output-nowhere, 941
open-output-string, 911
Operating System, 1119
opt/c, 692
or, 132
or/c, 611
order-of-magnitude, 218
'orig-dir, 1143
'origin, 875
ormap, 328
'os, 1208
'os*, 1208
Other Randomness Utilities, 210
other-execute-bit, 1168
other-read-bit, 1168
other-write-bit, 1168
output port, 893
output-port?, 895
outside-edge scope, 43
overment, 541
overment*, 546
override, 540
override*, 546
override-final, 541
override-final*, 546
overriding, 531
packages, 1269
pair, 321
Pair Accessor Shorthands, 336
Pair Constructors and Selectors, 321
pair?, 322
Pairs and Lists, 321
parameter procedure, 28
parameter-procedure=?, 796
parameter/c, 626
parameter?, 796
parameterization, 28
parameterization?, 796
parameterize, 794
parameterize*, 795
parameterize-break, 768
Parameters, 28
Parameters, 793
Parameters, 28
Parametric Contracts, 648
parametric->/c, 648
'paren-shape, 57
parent internal-definition context, 849
parse, 34
parse-command-line, 1217
parsed, 34
Partial Expansion, 42
partial expansion, 42
partition, 357
Passing keyword arguments in dictionaries, 452
patched, 129
PATH, 1144
path, 1119
path element, 1133
path or string, 1119
path->bytes, 1121
path->complete-path, 1125
path->directory-path, 1125
path->string, 1120
path-add-extension, 1129
path-add-suffix, 1130
Regular expressions, 293
REL, 1137
'relative,1127
relative-in, 93
relative-path?, 1124
relocate-input-port, 943
relocate-output-port, 946
remainder, 189
remf, 362
remf*, 362
remove, 330
remove*, 331
remove-duplicates, 356
remq, 330
remq*, 331
remv, 331
remv*, 332
rename, 597
rename transformer, 40
call, 597
rename-contract, 693
call, 693
rename-file-or-directory, 1146
call, 1146
rename-in, 92
rename-inner, 545
rename-out, 102
rename-super, 545
rename-transformer, 843
rename-transformer?, 842
REPL, 1031
'replace, 904
replace-evt, 781
require, 89
require Macros, 138
require transformer, 863
require Transformers, 863
require-transformer, 864
reroot-path, 1130
reset, 758
reset-at, 759
reset0, 759
reset0-at, 759
resolve-path, 1125
resolved, 1041
resolved module path, 1037
resolved-module-path-name, 1037
resolved-module-path?, 1037
Resolving Module Names, 1037
rest, 344
'return, 953
'return-linefeed, 954
reverse, 326
root namespace, 1014
round, 192
'run-file, 1143
'running, 1185
Running Racket, 1253
Running Racket or G Racket, 1253
runtime-paths, 1156
runtime-require, 1126
s-exp, 66
s-exp->fasl, 1011
S-Expression Reader Language, 66
'same, 1123
sandbox-coverage-enabled, 1096
sandbox-error-output, 1095
sandbox-eval-handlers, 1101
sandbox-eval-limits, 1100
sandbox-exit-handler, 1099
sandbox-gui-available, 1097
sandbox-init-hook, 1094
sandbox-input, 1094
sandbox-make-code-inspector, 1102
sandbox-make-environment-variables, 1103
sandbox-make-inspector, 1102
sandbox-make-logger, 1102
sandbox-make-name-space, 1097
sandbox-make-plumber, 1102
sandbox-memory-limit, 1099
sandbox-name-space-spec, 1097
sandbox-network-guard, 1099
sandbox-output, 1095
sandbox-override-collection-paths, 1098
sandbox-path-permissions, 1098
set-box!, 378
set-box*, 378
set-clear, 462
set-clear!, 462
set-copy, 462
set-copy-clear, 462
set-count, 461
set-empty?, 461
set-eq?, 454
set-equal?, 454
set-eqv?, 454
set-eval-handler, 1104
set-eval-limits, 1104
set-field!, 561
set-first, 461
set-for-each, 468
set-implements/c, 457
set-implements?, 457
set-intersect, 464
set-intersect!, 464
set-map, 468
set-mcar!, 365
set-mcdr!, 366
set-member?, 460
set-mutable?, 454
set-phantom-bytes!, 1228
set-port-next-location!, 901
set-remove, 460
set-remove!, 461
set-rest, 461
set-subset?, 467
set-subtract, 464
set-subtract!, 465
set-symmetric-difference, 465
set-symmetric-difference!, 466
set-union, 463
set-union!, 463
set-weak?, 454
set/c, 457
set=, 466
set?, 454
seteq, 454
seteqv, 454
Sets, 453
seventh, 345
sgn, 216
sha1-bytes, 1012
sha224-bytes, 1012
sha256-bytes, 1013
shadowing, 31
shadows, 31
Shallow time, 1093
shared, 127
'shared, 1209
shared memory space, 802
shared-bytes, 269
shared-extflvector, 235
shared-flvector, 225
shared-fxvector, 230
shell-execute, 1186
ShellExecute, 1187
shift, 758
shift-at, 759
shift0, 759
shift0-at, 759
shrink-path-wrt, 1134
shuffle, 359
SIGHUP, 766
SIGINT, 766
signature, 593
signature-members, 607
SIGTERM, 766
Simple Subprocesses, 1188
simple-form-path, 1134
simplify-path, 1126
sin, 201
single-flonum-available?, 183
single-flonum?, 183
single-flonums, 179
Single-Signature Modules, 606
Single-Unit Modules, 606
sinh, 217
sixth, 345
'sk, 287
string port
String Ports
string->bytes/latin-1
string->bytes/locale
string->bytes/utf-8
string->immutable-string
string->keyword
string->list
string->number
string->path
string->path-element
string->symbol
string->unreadable-symbol
string-contains?
string-copy
string-copy!
string-downcase
string-environment-variable-name?
string-fill!
string-foldcase
string-join
string-len/c
string-length
string-locale-cicr?
string-locale-ci=
string-locale-cir?
string-locale-downcase
string-locale-upcase
string-locale<
string-locale<=
string-locale=>
string-locale>=
string-no-nuls?
string-normalize-nfc
string-normalize-nfd
string-normalize-nfkc
string-normalize-nfkd
string-normalize-spaces
string-port
string-prefix?
string-ref
string-replace
string-set!
string-split
string-suffix?
string-titlecase
string-trim
string-upcase
string-utf-8-length
string<=?
string<=?
string=?
string>=?
string=>?
strings, pattern matching
strings, parsing
strings, immutable
strings, concatenate
struct
struct*
struct->list
struct->vector
struct-accessor-procedure?
struct-auto-info-lists
struct-auto-info?
struct-constructor-procedure?
struct-copy
struct-field-index
struct-field-info-list
struct-field-info?
struct-guard/c
struct-info
syntax-property-remove, 877
syntax-property-symbol-keys, 877
syntax-protect, 880
syntax-rearm, 880
syntax-recertify, 834
syntax-rules, 824
syntax-serialize, 884
syntax-shift-phase-level, 833
syntax-source, 826
syntax-source-module, 827
syntax-span, 826
syntax-taint, 828
syntax-tainted?, 879
syntax-track-origin, 877
syntax-transforming-module-expression?, 859
syntax-transforming-with-lifts?, 859
syntax-transforming?, 859
syntax/c, 623
syntax/loc, 823
syntax?, 825
'sys-dir, 1142
system, 1188
system*, 1189
system/exit-code, 1189
system-big-endian?, 215
system-idle-evt, 781
system-language+country, 1210
system-library-subpath, 1211
system-path-convention-type, 1128
system-type, 1208
system/exit-code, 1189
tag, 597
Tail Position, 12
tail position, 12
'taint-mode, 879
'taint-mode, 879
tainted, 878
take, 349
take-common-prefix, 352
take-right, 351
takef, 350
takef-right, 352
tamper status, 878
tan, 207
'target-machine, 1209
TCP, 1168
TCP listener, 1169
TCP port, 1173
tcp-abandon-port, 1172
tcp-accept, 1171
tcp-accept-evt, 1172
tcp-accept-ready?, 1171
tcp-accept/enable-break, 1171
tcp-addresses, 1172
tcp-close, 1172
tcp-connect, 1170
tcp-connect/enable-break, 1171
tcp-listen, 1169
tcp-listener?, 1172
tcp-port?, 1173
TEMP, 1141
'temp-dir, 1141
template environment, 32
tentative-pretty-print-port-cancel, 990
tentative-pretty-print-port-transfer, 990
tenth, 346
terminal-port?, 897
'terminating-macro, 992
'text, 902
The Printer, 67
The Racket Reference, 1
The racket/load Language, 1036
The racket/repl Library, 1108
The Reader, 50
The Separate Compilation Guarantee, 21
the-unsupplied-arg, 647
third, 344
this, 536
this%, 536