Syntax: Meta-Programming Helpers

Version 9.0.0.2

October 27, 2025

Contents

1 Parsing and Specifying Syntax

The syntax/parse library provides a framework for writing macros and processing syntax. The library provides a powerful language of syntax patterns, used by the pattern-matching form syntax-parse and the specification form define-syntax-class. Macros that use syntax-parse automatically generate error messages based on descriptions and messages embedded in the macro's syntax patterns.

```
(require syntax/parse)
```
package: base

1.1 Introduction

This section provides an introduction to writing robust macros with syntax-parse and syntax classes.

As a running example we use the following task: write a macro named mylet that has the same syntax and behavior as Racket's let form. The macro should produce good error messages when used incorrectly.

Here is the specification of mylet's syntax:

```
(mylet ([var-id rhs-expr] ...) body ...+)
(mylet loop-id ([var-id rhs-expr] ...) body ...+)
```

For simplicity, we handle only the first case for now. We return to the second case later in the introduction.

The macro can be implemented very simply using define-syntax-rule:

```
> (define-syntax-rule (mylet ([var rhs] ...) body ...)
    ((lambda (var ...) body ...) rhs ...))
```

When used correctly, the macro works, but it behaves very badly in the presence of errors. In some cases, the macro merely fails with an uninformative error message; in others, it blithely accepts illegal syntax and passes it along to lambda, with strange consequences:

```
> (mylet ([a 1] [b 2]) (+ a b))
3
> (mylet (b 2) (sub1 b))
mylet: use does not match pattern: (mylet ((var rhs) ...)
body ...)
  in: (mylet (b 2) (sub1 b))
> (mylet ([1 a]) (add1 a))
```

```
lambda: not an identifier, identifier with default, or
keyword
   at: 1
   in: (lambda (1) (add1 a))
> (mylet ([#:x 1] [y 2]) (* x y))
eval:1:0: arity mismatch;
 the expected number of arguments does not match the given
number
   expected: 0 plus an argument with keyword #:x
   given: 2
   arguments...:
    1
    2
```

These examples of illegal syntax are not to suggest that a typical programmer would make such mistakes attempting to use `mylet`. At least, not often, not after an initial learning curve. But macros are also used by inexpert programmers and as targets of other macros (or code generators), and many macros are far more complex than `mylet`. Macros must validate their syntax and report appropriate errors. Furthermore, the macro writer benefits from the *machine-checked* specification of syntax in the form of more readable, maintainable code.

We can improve the error behavior of the macro by using `syntax-parse`. First, we import `syntax-parse` into the transformer environment, since we will use it to implement a macro transformer.

```
> (require (for-syntax syntax/parse))
```

The following is the syntax specification above transliterated into a `syntax-parse` macro definition. It behaves no better than the version using `define-syntax-rule` above.

```
> (define-syntax (mylet stx)
    (syntax-parse stx
      [(_ ([var-id rhs-expr] ...) body ...+)
       #'((lambda (var-id ...) body ...) rhs-expr ...)]))
```

One minor difference is the use of `...+` in the pattern; `...` means match zero or more repetitions of the preceding pattern; `...+` means match one or more. Only `...` may be used in the template, however.

The first step toward validation and high-quality error reporting is annotating each of the macro's pattern variables with the syntax class that describes its acceptable syntax. In `mylet`, each variable must be an `identifier` (`id` for short) and each right-hand side must be an `expr` (expression). An annotated pattern variable is written by concatenating the pattern variable name, a colon character, and the syntax class name.

For an alternative to the “colon” syntax, see the `~var` pattern form.

```
> (define-syntax (mylet stx)
    (syntax-parse stx
      [(_ ((var:id rhs:expr) ...) body ...+)
       #'((lambda (var ...) body ...) rhs ...)]))
```

Note that the syntax class annotations do not appear in the template (i.e., `var`, not `var:id`).

The syntax class annotations are checked when we use the macro.

```
> (mylet ([a 1] [b 2]) (+ a b))
3
> (mylet (["a" 1]) (add1 a))
mylet: expected identifier
  at: "a"
  in: (mylet (("a" 1)) (add1 a))
```

The `expr` syntax class does not actually check that the term it matches is a valid expression—that would require calling that macro expander. Instead, `expr` just means not a keyword.

```
> (mylet ([a #:whoops]) 1)
mylet: expected expression
  at: #:whoops
  in: (mylet ((a #:whoops)) 1)
```

Also, `syntax-parse` knows how to report a few kinds of errors without any help:

```
> (mylet ([a 1 2]) (* a a))
mylet: unexpected term
  at: 2
  in: (mylet ((a 1 2)) (* a a))
```

There are other kinds of errors, however, that this macro does not handle gracefully:

```
> (mylet (a 1) (+ a 2))
mylet: bad syntax
  in: (mylet (a 1) (+ a 2))
```

It's too much to ask for the macro to respond, "This expression is missing a pair of parentheses around `(a 1)`." The pattern matcher is not that smart. But it can pinpoint the source of the error: when it encountered `a` it was expecting what we might call a "binding pair," but that term is not in its vocabulary yet.

To allow `syntax-parse` to synthesize better errors, we must attach *descriptions* to the patterns we recognize as discrete syntactic categories. One way of doing that is by defining new syntax classes:

Another way is the `~describe` pattern form.

```
> (define-syntax (mylet stx)

    (define-syntax-class binding
      #:description "binding pair"
      (pattern (var:id rhs:expr)))

    (syntax-parse stx
      [(_ (b:binding ...) body ...+)
       #'((lambda (b.var ...) body ...) b.rhs ...)]))
```

Note that we write `b.var` and `b.rhs` now. They are the nested attributes formed from the annotated pattern variable `b` and the attributes `var` and `rhs` of the syntax class `binding`.

Now the error messages can talk about "binding pairs."

```
> (mylet (a 1) (+ a 2))
mylet: expected binding pair
  at: a
  in: (mylet (a 1) (+ a 2))
```

Errors are still reported in more specific terms when possible:

```
> (mylet (["a" 1]) (+ a 2))
mylet: expected identifier
  at: "a"
  in: (mylet (("a" 1)) (+ a 2))
  parsing context:
   while parsing binding pair
    term: ("a" 1)
    location: eval:16:0
```

There is one other constraint on the legal syntax of `mylet`. The variables bound by the different binding pairs must be distinct. Otherwise the macro creates an illegal `lambda` form:

```
> (mylet ([a 1] [a 2]) (+ a a))
lambda: duplicate argument name
  at: a
  in: (lambda (a a) (+ a a))
```

Constraints such as the distinctness requirement are expressed as side conditions, thus:

```
> (define-syntax (mylet stx)
```

```
    (define-syntax-class binding
      #:description "binding pair"
      (pattern (var:id rhs:expr)))

    (syntax-parse stx
      [(_ (b:binding ...) body ...+)
       #:fail-when (check-duplicate-identifier
                    (syntax->list #'(b.var ...)))
                   "duplicate variable name"
       #'((lambda (b.var ...) body ...) b.rhs ...)]))
```

```
> (mylet ([a 1] [a 2]) (+ a a))
```

mylet: duplicate variable name
at: a
in: (mylet ((a 1) (a 2)) (+ a a))

The `#:fail-when` keyword is followed by two expressions: the condition and the error message. When the condition evaluates to anything but `#f`, the pattern fails. Additionally, if the condition evaluates to a syntax object, that syntax object is used to pinpoint the cause of the failure.

Syntax classes can have side conditions, too. Here is the macro rewritten to include another syntax class representing a "sequence of distinct binding pairs."

```
> (define-syntax (mylet stx)

    (define-syntax-class binding
      #:description "binding pair"
      (pattern (var:id rhs:expr)))

    (define-syntax-class distinct-bindings
      #:description "sequence of distinct binding pairs"
      (pattern (b:binding ...)
               #:fail-when (check-duplicate-identifier
                            (syntax->list #'(b.var ...)))
                           "duplicate variable name"
               #:with (var ...) #'(b.var ...)
               #:with (rhs ...) #'(b.rhs ...)))

    (syntax-parse stx
      [(_ bs:distinct-bindings body ...+)
       #'((lambda (bs.var ...) body ...) bs.rhs ...)]))
```

Here we've introduced the `#:with` clause. A `#:with` clause matches a pattern with a computed term. Here we use it to bind `var` and `rhs` as attributes of `distinct-bindings`.

By default, a syntax class only exports its patterns' pattern variables as attributes, not their nested attributes.

The alternative would be to explicitly declare the attributes of `distinct-bindings` to include the nested attributes `b.var` and `b.rhs`, using the `#:attribute` option. Then the macro would refer to `bs.b.var` and `bs.b.rhs`.

Alas, so far the macro only implements half of the functionality offered by Racket's `let`. We must add the "named-`let`" form. That turns out to be as simple as adding a new clause:

```
> (define-syntax (mylet stx)

    (define-syntax-class binding
      #:description "binding pair"
      (pattern (var:id rhs:expr)))

    (define-syntax-class distinct-bindings
      #:description "sequence of distinct binding pairs"
      (pattern (b:binding ...)
               #:fail-when (check-duplicate-identifier
                            (syntax->list #'(b.var ...)))
                           "duplicate variable name"
               #:with (var ...) #'(b.var ...)
               #:with (rhs ...) #'(b.rhs ...)))

    (syntax-parse stx
      [(_ bs:distinct-bindings body ...+)
       #'((lambda (bs.var ...) body ...) bs.rhs ...)]
      [(_ loop:id bs:distinct-bindings body ...+)
       #'(letrec ([loop (lambda (bs.var ...) body ...)])
           (loop bs.rhs ...))]))
```

We are able to reuse the `distinct-bindings` syntax class, so the addition of the "named-`let`" syntax requires only three lines.

But does adding this new case affect `syntax-parse`'s ability to pinpoint and report errors?

```
> (mylet ([a 1] [b 2]) (+ a b))
3
> (mylet (["a" 1]) (add1 a))
mylet: expected identifier
  at: "a"
  in: (mylet (("a" 1)) (add1 a))
  parsing context:
   while parsing binding pair
    term: ("a" 1)
    location: eval:23:0
   while parsing sequence of distinct binding pairs
    term: (("a" 1))
    location: eval:23:0
```

```
> (mylet ([a #:whoops]) 1)
mylet: expected expression
  at: #:whoops
  in: (mylet ((a #:whoops)) 1)
  parsing context:
   while parsing binding pair
    term: (a #:whoops)
    location: eval:24:0
   while parsing sequence of distinct binding pairs
    term: ((a #:whoops))
    location: eval:24:0
> (mylet ([a 1 2]) (* a a))
mylet: unexpected term
  at: 2
  in: (mylet ((a 1 2)) (* a a))
  parsing context:
   while parsing binding pair
    term: (a 1 2)
    location: eval:25:0
   while parsing sequence of distinct binding pairs
    term: ((a 1 2))
    location: eval:25:0
> (mylet (a 1) (+ a 2))
mylet: expected binding pair
  at: a
  in: (mylet (a 1) (+ a 2))
  parsing context:
   while parsing sequence of distinct binding pairs
    term: (a 1)
    location: eval:26:0
> (mylet ([a 1] [a 2]) (+ a a))
mylet: duplicate variable name
  at: a
  in: (mylet ((a 1) (a 2)) (+ a a))
  parsing context:
   while parsing sequence of distinct binding pairs
    term: ((a 1) (a 2))
    location: eval:27:0
```

The error reporting for the original syntax seems intact. We should verify that the named-`let` syntax is working, that `syntax-parse` is not simply ignoring that clause.

```
> (mylet loop ([a 1] [b 2]) (+ a b))
3
> (mylet loop (["a" 1]) (add1 a))
mylet: expected identifier
```

```
  at: "a"
  in: (mylet loop (("a" 1)) (add1 a))
  parsing context:
   while parsing binding pair
    term: ("a" 1)
    location: eval:29:0
   while parsing sequence of distinct binding pairs
    term: (("a" 1))
    location: eval:29:0
> (mylet loop ([a #:whoops]) 1)
mylet: expected expression
  at: #:whoops
  in: (mylet loop ((a #:whoops)) 1)
  parsing context:
   while parsing binding pair
    term: (a #:whoops)
    location: eval:30:0
   while parsing sequence of distinct binding pairs
    term: ((a #:whoops))
    location: eval:30:0
> (mylet loop ([a 1 2]) (* a a))
mylet: unexpected term
  at: 2
  in: (mylet loop ((a 1 2)) (* a a))
  parsing context:
   while parsing binding pair
    term: (a 1 2)
    location: eval:31:0
   while parsing sequence of distinct binding pairs
    term: ((a 1 2))
    location: eval:31:0
> (mylet loop (a 1) (+ a 2))
mylet: expected binding pair
  at: a
  in: (mylet loop (a 1) (+ a 2))
  parsing context:
   while parsing sequence of distinct binding pairs
    term: (a 1)
    location: eval:32:0
> (mylet loop ([a 1] [a 2]) (+ a a))
mylet: duplicate variable name
  at: a
  in: (mylet loop ((a 1) (a 2)) (+ a a))
  parsing context:
   while parsing sequence of distinct binding pairs
    term: ((a 1) (a 2))
```

location: eval:33:0

How does `syntax-parse` decide which clause the programmer was attempting, so it can use it as a basis for error reporting? After all, each of the bad uses of the named-`let` syntax are also bad uses of the normal syntax, and vice versa. And yet the macro does not produce errors like "`mylet`: expected sequence of distinct binding pairs at: `loop`."

The answer is that `syntax-parse` records a list of all the potential errors (including ones like `loop` not matching `distinct-binding`) along with the *progress* made before each error. Only the error with the most progress is reported.

For example, in this bad use of the macro,

```
> (mylet loop (["a" 1]) (add1 a))
mylet: expected identifier
  at: "a"
  in: (mylet loop (("a" 1)) (add1 a))
  parsing context:
   while parsing binding pair
    term: ("a" 1)
    location: eval:34:0
   while parsing sequence of distinct binding pairs
    term: (("a" 1))
    location: eval:34:0
```

there are two potential errors: expected `distinct-bindings` at `loop` and expected `identifier` at `"a"`. The second error occurs further in the term than the first, so it is reported.

For another example, consider this term:

```
> (mylet (["a" 1]) (add1 a))
mylet: expected identifier
  at: "a"
  in: (mylet (("a" 1)) (add1 a))
  parsing context:
   while parsing binding pair
    term: ("a" 1)
    location: eval:35:0
   while parsing sequence of distinct binding pairs
    term: (("a" 1))
    location: eval:35:0
```

Again, there are two potential errors: expected `identifier` at `(["a" 1])` and expected `identifier` at `"a"`. They both occur at the second term (or first argument, if you prefer), but the second error occurs deeper in the term. Progress is based on a left-to-right traversal of the syntax.

A final example: consider the following:

```
> (mylet ([a 1] [a 2]) (+ a a))
mylet: duplicate variable name
  at: a
  in: (mylet ((a 1) (a 2)) (+ a a))
  parsing context:
   while parsing sequence of distinct binding pairs
    term: ((a 1) (a 2))
    location: eval:36:0
```

There are two errors again: duplicate variable name at `([a 1] [a 2])` and expected `identifier` at `([a 1] [a 2])`. Note that as far as `syntax-parse` is concerned, the progress associated with the duplicate error message is the second term (first argument), not the second occurrence of `a`. That's because the check is associated with the entire `distinct-bindings` pattern. It would seem that both errors have the same progress, and yet only the first one is reported. The difference between the two is that the first error is from a *post-traversal* check, whereas the second is from a normal (i.e., pre-traversal) check. A post-traversal check is considered to have made more progress than a pre-traversal check of the same term; indeed, it also has greater progress than any failure *within* the term.

It is, however, possible for multiple potential errors to occur with the same progress. Here's one example:

```
> (mylet "not-even-close")
mylet: expected identifier or expected sequence of distinct
binding pairs
  at: "not-even-close"
  in: (mylet "not-even-close")
```

In this case `syntax-parse` reports both errors.

Even with all of the annotations we have added to our macro, there are still some misuses that defy `syntax-parse`'s error reporting capabilities, such as this example:

```
> (mylet)
mylet: expected more terms starting with sequence of
distinct binding pairs or identifier
  at: ()
  within: (mylet)
  in: (mylet)
```

The philosophy behind `syntax-parse` is that in these situations, a generic error such as "bad syntax" is justified. The use of `mylet` here is so far off that the only informative error message would include a complete recapitulation of the syntax of `mylet`. That is not the role of error messages, however; it is the role of documentation.

This section has provided an introduction to syntax classes, side conditions, and progress-ordered error reporting. But `syntax-parse` has many more features. Continue to the §1.2 "Examples" section for samples of other features in working code, or skip to the subsequent sections for the complete reference documentation.

1.2 Examples

This section provides an extended introduction to `syntax/parse` as a series of worked examples.

1.2.1 Phases and Reusable Syntax Classes

As demonstrated in the §1.1 "Introduction", the simplest place to define a syntax class is within the macro definition that uses it. But that limits the scope of the syntax class to the one client macro, and it makes for very large macro definitions. Creating reusable syntax classes requires some awareness of the Racket phase level separation. A syntax class defined immediately within a module cannot be used by macros in the same module; it is defined at the wrong phase.

```
> (module phase-mismatch-mod racket
    (require syntax/parse (for-syntax syntax/parse))
    (define-syntax-class foo
      (pattern (a b c)))
    (define-syntax (macro stx)
      (syntax-parse stx
        [(_ f:foo) #'(+ f.a f.b f.c)])))
syntax-parse: not defined as syntax class
  at: foo
  in: (syntax-parse stx ((_ f:foo) (syntax (+ f.a f.b
f.c))))
```

In the module above, the syntax class `foo` is defined at phase level 0. The reference to `foo` within `macro`, however, is at phase level 1, being the implementation of a macro transformer. (Needing to require `syntax/parse` twice, once normally and once `for-syntax` is a common warning sign of phase level incompatibility.)

The phase level mismatch is easily remedied by putting the syntax class definition within a `begin-for-syntax` block:

```
> (module phase-ok-mod racket
    (require (for-syntax syntax/parse))
    (begin-for-syntax
     (define-syntax-class foo
```

```
      (pattern (a b c))))
   (define-syntax (macro stx)
     (syntax-parse stx
       [(_ f:foo) #'(+ f.a f.b f.c)])))
```

In the revised module above, `foo` is defined at phase 1, so it can be used in the implementation of the macro.

An alternative to `begin-for-syntax` is to define the syntax class in a separate module and require that module `for-syntax`.

```
> (module stxclass-mod racket
    (require syntax/parse)
    (define-syntax-class foo
      (pattern (a b c)))
    (provide foo))
> (module macro-mod racket
    (require (for-syntax syntax/parse
                         'stxclass-mod))
    (define-syntax (macro stx)
      (syntax-parse stx
        [(_ f:foo) #'(+ f.a f.b f.c)]))
    (provide macro))
> (require 'macro-mod)
> (macro (1 2 3))
6
```

If a syntax class refers to literal identifiers, or if it computes expressions via syntax templates, then the module containing the syntax class must generally require `for-template` the bindings referred to in the patterns and templates.

```
> (module arith-keywords-mod racket
    (define-syntax plus (syntax-rules ()))
    (define-syntax times (syntax-rules ()))
    (provide plus times))
> (module arith-stxclass-mod racket
    (require syntax/parse
             (for-template 'arith-keywords-mod
                           racket))
    (define-syntax-class arith
      #:literals (plus times)
      (pattern n:nat
               #:with expr #'n)
      (pattern (plus a:arith b:arith)
               #:with expr #'(+ a.expr b.expr))
```

```
      (pattern (times a:arith b:arith)
               #:with expr #'(* a.expr b.expr)))
    (provide arith))
> (module arith-macro-mod racket
    (require (for-syntax syntax/parse
                         'arith-stxclass-mod)
             'arith-keywords-mod)
    (define-syntax (arith-macro stx)
      (syntax-parse stx
        [(_ a:arith)
         #'(values 'a.expr a.expr)]))
    (provide arith-macro
             (all-from-out 'arith-keywords-mod)))
> (require 'arith-macro-mod)
> (arith-macro (plus 1 (times 2 3)))
'(+ 1 (* 2 3))
7
```

In 'arith-stxclass-mod, the module 'arith-keywords-mod must be required for-template because the keywords are used in phase-0 expressions. Likewise, the module racket must be required for-template because the syntax class contains syntax templates involving + and * (and, in fact, the implicit #%app syntax). All of these identifiers (the keywords plus and times; the procedures + and *; and the implicit syntax #%app) must be bound at "absolute" phase level 0. Since the module 'arith-stxclass-mod is required with a phase level offset of 1 (that is, for-syntax), it must compensate with a phase level offset of -1, or for-template.

1.2.2 Optional Keyword Arguments

This section explains how to write a macro that accepts (simple) optional keyword arguments. We use the example mycond, which is like Racket's cond except that it takes an optional keyword argument that controls what happens if none of the clauses match.

Optional keyword arguments are supported via head patterns. Unlike normal patterns, which match one term, head patterns can match a variable number of subterms in a list. Some important head-pattern forms are ~seq, ~or*, and ~optional.

Here's one way to do it:

```
> (define-syntax (mycond stx)
    (syntax-parse stx
      [(mycond (~or* (~seq #:error-on-fallthrough who:expr)
                     (~seq))
               clause ...)
```

```
        (with-syntax ([error? (if (attribute who) #'#t #'#f)]
                      [who (or (attribute who) #'#f)])
          #'(mycond* error? who clause ...))]))
> (define-syntax mycond*
    (syntax-rules ()
      [(mycond* error? who [question answer] . clauses)
       (if question answer (mycond* error? who . clauses))]
      [(mycond* #t who)
       (error who "no clauses matched")]
      [(mycond* #f _)
       (void)]))
```

We cannot simply write `#'who` in the macro's right-hand side, because the `who` attribute does not receive a value if the keyword argument is omitted. Instead we must first check the attribute using `(attribute who)`, which produces `#f` if matching did not assign a value to the attribute.

```
> (mycond [(even? 13) 'blue]
          [(odd? 4) 'red])
> (mycond #:error-on-fallthrough 'myfun
          [(even? 13) 'blue]
          [(odd? 4) 'red])
```

myfun: no clauses matched

There's a simpler way of writing the `~or*` pattern above:

```
(~optional (~seq #:error-on-fallthrough who:expr))
```

Optional Arguments with `~?`

The `~?` template form provides a compact alternative to explicitly testing attribute values. Here's one way to do it:

```
> (define-syntax (mycond stx)
    (syntax-parse stx
      [(mycond (~optional (~seq #:error-on-fallthrough who:expr))
               clause ...)
       #'(mycond* (~? (~@ #t who) (~@ #f #f)) clause ...)]))
```

If `who` matched, then the `~?` subtemplate splices in the two terms `#t who` into the enclosing template (`~@` is the template splicing form). Otherwise, it splices in `#f #f`.

Here's an alternative definition that re-uses Racket's `cond` macro:

```
> (define-syntax (mycond stx)
```

```
(syntax-parse stx
  [(mycond (~optional (~seq #:error-on-fallthrough who:expr))
           clause ...)
   #'(cond clause ... (~? [else (error who "no clause
matched")] (~@)))]))
```

In this version, we optionally insert an `else` clause at the end to signal the error; otherwise we use `cond`'s fall-through behavior (that is, returning `(void)`).

If the second subtemplate of a `~?` template is `(~@)`—that is, it produces no terms at all—the second subtemplate can be omitted.

Optional Arguments with `define-splicing-syntax-class`

Yet another way is to introduce a splicing syntax class, which is like an ordinary syntax class but for head patterns.

```
> (define-syntax (mycond stx)

    (define-splicing-syntax-class maybe-fallthrough-option
      (pattern (~seq #:error-on-fallthrough who:expr)
               #:with error? #'#t)
      (pattern (~seq)
               #:with error? #'#f
               #:with who #'#f))

    (syntax-parse stx
      [(mycond fo:maybe-fallthrough-option clause ...)
       #'(mycond* fo.error? fo.who clause ...)]))
```

Defining a splicing syntax class also makes it easy to eliminate the case analysis we did before using `attribute` by defining `error?` and `who` as attributes within both of the syntax class's variants. This is possible to do in the inline pattern version too, using `~and` and `~parse`, but it is less convenient. Splicing syntax classes also closely parallel the style of grammars in macro documentation.

1.2.3 Variants with Uniform Meanings

Syntax classes not only validate syntax, they also extract some measure of meaning from it. From the perspective of meaning, there are essentially two kinds of syntax class. In the first, all of the syntax class's variants have the same kind of meaning. In the second, variants may have different kinds of meaning. This section discusses the first kind, syntax classes with uniform meanings. The next section discusses §1.2.4 "Variants with Varied Meanings".

In other words, some syntax classes' meanings are products and others' meanings are sums.

If all of a syntax class's variants express the same kind of information, that information can be cleanly represented via attributes, and it can be concisely processed using ellipses.

One example of a syntax class with uniform meaning: the `init-decl` syntax of the `class` macro. Here is the specification of `init-decl`:

```
      init-decl = id
                | (maybe-renamed)
                | (maybe-renamed default-expr)

  maybe-renamed = id
                | (internal-id external-id)
```

The `init-decl` syntax class has three variants, plus an auxiliary syntax class that has two variants of its own. But all forms of `init-decl` ultimately carry just three pieces of information: an internal name, an external name, and a default configuration of some sort. The simpler syntactic variants are just abbreviations for the full information.

The three pieces of information determine the syntax class's attributes. It is useful to declare the attributes explicitly using the `#:attributes` keyword; the declaration acts both as in-code documentation and as a check on the variants.

```
(define-syntax-class init-decl
  #:attributes (internal external default)
  __)
```

Next we fill in the syntactic variants, deferring the computation of the attributes:

```
(define-syntax-class init-decl
  #:attributes (internal external default)
  (pattern ???:id
           __)
  (pattern (???:maybe-renamed)
           __)
  (pattern (???:maybe-renamed ???:expr)
           __))
```

We perform a similar analysis of `maybe-renamed`:

```
(define-syntax-class maybe-renamed
  #:attributes (internal external)
  (pattern ???:id
           __)
  (pattern (???:id ???:id)
           __))
```

Here's one straightforward way of matching syntactic structure with attributes for `maybe-renamed`:

```
(define-syntax-class maybe-renamed
  #:attributes (internal external)
  (pattern internal:id
           #:with external #'internal)
  (pattern (internal:id external:id)))
```

Given that definition of `maybe-renamed`, we can fill in most of the definition of `init-decl`:

```
(define-syntax-class init-decl
  #:attributes (internal external default)
  (pattern internal:id
           #:with external #'internal
           #:with default ???)
  (pattern (mr:maybe-renamed)
           #:with internal #'mr.internal
           #:with external #'mr.external
           #:with default ???)
  (pattern (mr:maybe-renamed default0:expr)
           #:with internal #'mr.internal
           #:with external #'mr.external
           #:with default ???))
```

At this point we realize we have not decided on a representation for the default configuration. In fact, it is an example of syntax with varied meanings (aka sum or disjoint union). The following section discusses representation options in greater detail; for the sake of completeness, we present one of them here.

There are two kinds of default configuration. One indicates that the initialization argument is optional, with a default value computed from the given expression. The other indicates that the initialization argument is mandatory. We represent the variants as a (syntax) list containing the default expression and as the empty (syntax) list, respectively. More precisely:

```
(define-syntax-class init-decl
  #:attributes (internal external default)
  (pattern internal:id
           #:with external #'internal
           #:with default #'())
  (pattern (mr:maybe-renamed)
           #:with internal #'mr.internal
           #:with external #'mr.external
           #:with default #'())
  (pattern (mr:maybe-renamed default0:expr)
           #:with internal #'mr.internal
```

```
          #:with external #'mr.external
          #:with default #'(default0)))
```

Another way to look at this aspect of syntax class design is as the algebraic factoring of sums-of-products (concrete syntax variants) into products-of-sums (attributes and abstract syntax variants). The advantages of the latter form are the "dot" notation for data extraction, avoiding or reducing additional case analysis, and the ability to concisely manipulate sequences using ellipses.

1.2.4 Variants with Varied Meanings

As explained in the previous section, the meaning of a syntax class can be uniform, or it can be varied; that is, different instances of the syntax class can carry different kinds of information. This section discusses the latter kind of syntax class.

A good example of a syntax class with varied meanings is the `for-clause` of the `for` family of special forms.

```
for-clause = [id seq-expr]
           | [(id ...) seq-expr]
           | #:when guard-expr
```

The first two variants carry the same kind of information; both consist of identifiers to bind and a sequence expression. The third variant, however, means something totally different: a condition that determines whether to continue the current iteration of the loop, plus a change in scoping for subsequent `seq-expr`s. The information of a `for-clause` must be represented in a way that a client macro can do further case analysis to distinguish the "bind variables from a sequence" case from the "skip or continue this iteration and enter a new scope" case.

This section discusses two ways of representing varied kinds of information.

Syntactic Normalization

One approach is based on the observation that the syntactic variants already constitute a representation of the information they carry. So why not adapt that representation, removing redundancies and eliminating simplifying the syntax to make subsequent re-parsing trivial.

```
(define-splicing-syntax-class for-clause
  #:attributes (norm)
  (pattern [var:id seq:expr]
           #:with norm #'[(var) seq])
  (pattern [(var:id ...) seq:expr]
           #:with norm #'[(var ...) seq])
  (pattern (~seq #:when guard:expr)
```

```
             #:with norm #'[#:when guard]))
```

First, note that since the #:when variant consists of two separate terms, we define for-clause as a splicing syntax class. Second, that kind of irregularity is just the sort of thing we'd like to remove so we don't have to deal with it again later. Thus we represent the normalized syntax as a single term beginning with either a sequence of identifiers (the first two cases) or the keyword #:when (the third case). The two normalized cases are easy to process and easy to tell apart. We have also taken the opportunity to desugar the first case into the second.

A normalized syntactic representation is most useful when the subsequent case analysis is performed by syntax-parse or a similar form.

Non-syntax-valued Attributes

When the information carried by the syntax is destined for complicated processing by Racket code, it is often better to parse it into an intermediate representation using idiomatic Racket data structures, such as lists, hashes, structs, and even objects.

Thus far we have only used syntax pattern variables and the #:with keyword to bind attributes, and the values of the attributes have always been syntax. To bind attributes to values other than syntax, use the #:attr keyword.

```
; A ForClause is either
;  - (bind-clause (listof identifier) syntax)
;  - (when-clause syntax)
(struct bind-clause (vars seq-expr))
(struct when-clause (guard))

(define-splicing-syntax-class for-clause
  #:attributes (ast)
  (pattern [var:id seq:expr]
           #:attr ast (bind-clause (list #'var) #'seq))
  (pattern [(var:id ...) seq:expr]
           #:attr ast (bind-clause (syntax->list #'(var ...))
                                   #'seq))
  (pattern (~seq #:when guard:expr)
           #:attr ast (when-clause #'guard)))
```

Be careful! If we had used #:with instead of #:attr, a value produced by the right-hand side would be coerced to a syntax object before being matched against the pattern ast.

Attributes with non-syntax values cannot be used in syntax templates. Use the attribute form to get the value of an attribute.

1.2.5 More Keyword Arguments

This section shows how to express the syntax of `struct`'s optional keyword arguments using `syntax-parse` patterns.

The part of `struct`'s syntax that is difficult to specify is the sequence of struct options. Let's get the easy part out of the way first.

```
> (define-splicing-syntax-class maybe-super
    (pattern (~seq super:id))
    (pattern (~seq)))
> (define-syntax-class field-option
    (pattern #:mutable)
    (pattern #:auto))
> (define-syntax-class field
    (pattern field:id
             #:with (option ...) '())
    (pattern [field:id option:field-option ...]))
```

Given those auxiliary syntax classes, here is a first approximation of the main pattern, including the struct options:

```
(struct name:id super:maybe-super (field:field ...)
  (~alt (~seq #:mutable)
        (~seq #:super super-expr:expr)
        (~seq #:inspector inspector:expr)
        (~seq #:auto-value auto:expr)
        (~seq #:guard guard:expr)
        (~seq #:property prop:expr prop-val:expr)
        (~seq #:transparent)
        (~seq #:prefab)
        (~seq #:constructor-name constructor-name:id)
        (~seq #:extra-constructor-name extra-constructor-name:id)
        (~seq #:omit-define-syntaxes)
        (~seq #:omit-define-values))
  ...)
```

The fact that `expr` does not match keywords helps in the case where the programmer omits a keyword's argument; instead of accepting the next keyword as the argument expression, `syntax-parse` reports that an expression was expected.

There are two main problems with the pattern above:

- There's no way to tell whether a zero-argument keyword like `#:mutable` was seen.

- Some options, like #:mutable, should appear at most once.

The first problem can be remedied using ~and patterns to bind a pattern variable to the keyword itself, as in this sub-pattern:

```
(~seq (~and #:mutable mutable-kw))
```

The second problem can be solved using *repetition constraints*:

```
(struct name:id super:maybe-super (field:field ...)
  (~alt (~optional (~seq (~and #:mutable mutable-kw)))
        (~optional (~seq #:super super-expr:expr))
        (~optional (~seq #:inspector inspector:expr))
        (~optional (~seq #:auto-value auto:expr))
        (~optional (~seq #:guard guard:expr))
        (~seq #:property prop:expr prop-val:expr)
        (~optional (~seq (~and #:transparent transparent-kw)))
        (~optional (~seq (~and #:prefab prefab-kw)))
        (~optional (~seq #:constructor-name constructor-name:id))
        (~optional
         (~seq #:extra-constructor-name extra-constructor-
name:id))
        (~optional
         (~seq (~and #:omit-define-syntaxes omit-def-stxs-kw)))
        (~optional (~seq (~and #:omit-define-values omit-def-vals-
kw))))
  ...)
```

The ~optional repetition constraint indicates that an alternative can appear at most once. (There is a ~once form that means it must appear exactly once.) In struct's keyword options, only #:property may occur any number of times.

There are still some problems, though. Without additional help, ~optional does not report particularly good errors. We must give it the language to use, just as we had to give descriptions to sub-patterns via syntax classes. Also, some related options are mutually exclusive, such as #:inspector, #:transparent, and #:prefab.

```
(struct name:id super:maybe-super (field:field ...)
  (~alt (~optional
         (~or* (~seq #:inspector inspector:expr)
               (~seq (~and #:transparent transparent-kw))
               (~seq (~and #:prefab prefab-kw)))
         #:name "#:inspector, #:transparent, or #:prefab option")
        (~optional (~seq (~and #:mutable mutable-kw))
                   #:name "#:mutable option")
```

```
         (~optional (~seq #:super super-expr:expr)
                    #:name "#:super option")
         (~optional (~seq #:auto-value auto:expr)
                    #:name "#:auto-value option")
         (~optional (~seq #:guard guard:expr)
                    #:name "#:guard option")
         (~seq #:property prop:expr prop-val:expr)
         (~optional (~seq #:constructor-name constructor-name:id)
                    #:name "#:constructor-name option")
         (~optional
           (~seq #:extra-constructor-name extra-constructor-
name:id)
           #:name "#:extra-constructor-name option")
         (~optional (~seq (~and #:omit-define-syntaxes omit-def-
stxs-kw))
                    #:name "#:omit-define-syntaxes option")
         (~optional (~seq (~and #:omit-define-values omit-def-vals-
kw))
                    #:name "#:omit-define-values option"))
  ...)
```

Here we have grouped the three incompatible options together under a single `~optional` constraint. That means that at most one of any of those options is allowed. We have given names to the optional clauses. See `~optional` for other customization options.

Note that there are other constraints that we have not represented in the pattern. For example, `#:prefab` is also incompatible with both `#:guard` and `#:property`. Repetition constraints cannot express arbitrary incompatibility relations. The best way to handle such constraints is with a side condition using `#:fail-when`.

1.2.6 Contracts on Macro Sub-expressions

Just as procedures often expect certain kinds of values as arguments, macros often have expectations about the expressions they are given. And just as procedures express those expectations via contracts, so can macros, using the `expr/c` syntax class.

For example, here is a macro `myparameterize` that behaves like `parameterize` but enforces the `parameter?` contract on the parameter expressions.

```
> (define-syntax (myparameterize stx)
    (syntax-parse stx
      [(_ ((p v:expr) ...) body:expr)
       #:declare p (expr/c #'parameter?
                           #:name "parameter argument")
       #'(parameterize ([p.c v] ...) body)]))
```

```
> (myparameterize ([current-input-port
                    (open-input-string "(1 2 3)")])
    (read))
'(1 2 3)
> (myparameterize (['whoops 'something])
    'whatever)
```

myparameterize: contract violation
expected: parameter?
given: 'whoops
in: parameter?
macro argument contract on parameter argument
contract from: 'program
blaming: (quote program)
(assuming the contract is correct)
at: eval:3:0

Important: Make sure when using `expr/c` to use the `c` attribute. If the macro above had used `p` in the template, the expansion would have used the raw, unchecked expressions. The `expr/c` syntax class does not change how pattern variables are bound; it only computes an attribute that represents the checked expression.

The previous example shows a macro applying a contract on an argument, but a macro can also apply a contract to an expression that it produces. In that case, it should use `#:arg? #f` to indicate that the macro, not the calling context, is responsible for expression produced.

```
; BUG: rationals not closed under inversion
> (define-syntax (invert stx)
    (syntax-parse stx
      [(_ e)
       #:declare e (expr/c #'rational?)
       #:with result #'(/ 1 e.c)
       #:declare result (expr/c #'rational? #:arg? #f)
       #'result.c]))
> (invert 4)
1/4
> (invert 'abc)
```

invert: contract violation
expected: rational?
given: 'abc
in: rational?
macro argument contract
contract from: 'program
blaming: (quote program)
(assuming the contract is correct)
at: eval:6:0

```
> (invert 0.0)
```

```
invert: contract violation
  expected: rational?
  given: +inf.0
  in: rational?
      macro result contract
  contract from: 'program
  blaming: (quote program)
   (assuming the contract is correct)
  at: eval:4:0
```

The following example shows a macro that uses a contracted expression at a different phase level. The macro's *ref* argument is used as a "compile-time expression"—more precisely, it is used as an expression at a phase level one higher than the use of the macro itself. That is because the macro places the expression in the right-hand side of a `define-syntax` form. The macro uses `expr/c` with a `#:phase` argument to ensure that *ref* produces an identifier when used as a compile-time expression.

```
> (define-syntax (define-alias stx)
    (syntax-parse stx
      [(_ name:id ref)
       #:declare ref (expr/c #'identifier?
                             #:phase (add1 (syntax-local-phase-
level)))
       #'(define-syntax name (make-rename-transformer ref.c))]))
> (define-alias plus #'+)
> (define-alias zero 0)
define-alias: contract violation
  expected: identifier?
  given: 0
  in: identifier?
      macro argument contract
  contract from: 'program
  blaming: (quote program)
   (assuming the contract is correct)
  at: eval:10:0
```

1.3 Parsing Syntax

This section describes `syntax-parse`, the `syntax/parse` library's facility for parsing syntax. Both `syntax-parse` and the specification facility, syntax classes, use a common language of syntax patterns, which is described in detail in §1.5 "Syntax Patterns".

Two parsing forms are provided: `syntax-parse` and `syntax-parser`.

```
(syntax-parse stx-expr parse-option ... clause ...+)

      parse-option = #:context context-expr
                   | #:literals (literal ...)
                   | #:datum-literals (datum-literal ...)
                   | #:literal-sets (literal-set ...)
                   | #:track-literals
                   | #:conventions (convention-id ...)
                   | #:local-conventions (convention-rule ...)
                   | #:disable-colon-notation

           literal = literal-id
                   | (pattern-id literal-id)
                   | (pattern-id literal-id #:phase phase-expr)

     datum-literal = literal-id
                   | (pattern-id literal-id)

       literal-set = literal-set-id
                   | (literal-set-id literal-set-option ...)

literal-set-option = #:at context-id
                   | #:phase phase-expr

            clause = (syntax-pattern pattern-directive ... body ...+)

  stx-expr : syntax?
  context-expr : (or/c syntax? symbol? #f
                       (list/c symbol? syntax?))
  phase-expr : (or/c exact-integer? #f)
```

Evaluates *stx-expr*, which should produce a syntax object, and matches it against the *clause*s in order. If some clause's pattern matches, its attributes are bound to the corresponding subterms of the syntax object and that clause's side conditions and `expr` is evaluated. The result is the result of `expr`.

Each clause consists of a syntax pattern, an optional sequence of pattern directives, and a non-empty sequence of body forms.

If the syntax object fails to match any of the patterns (or all matches fail the corresponding clauses' side conditions), a syntax error is raised.

The following options are supported:

```
#:context context-expr

  context-expr : (or/c syntax? symbol? #f
                       (list/c symbol? syntax?))
```

When present, *context-expr* is used in reporting parse failures; otherwise *stx-expr* is used. If *context-expr* evaluates to `(list who context-stx)`, then *who* appears in the error message as the form raising the error, and *context-stx* is used as the term. If *context-expr* evaluates to a symbol, it is used as *who* and *stx-expr* (the syntax to be destructured) is used as *context-stx*. If *context-expr* evaluates to a syntax object, it is used as *context-stx* and *who* is inferred as with `raise-syntax-error`.

The `current-syntax-context` parameter is also set to the syntax object *context-stx*.

Examples:

```
> (syntax-parse #'(a b 3)
    [(x:id ...) 'ok])
a: expected identifier
  at: 3
  in: (a b 3)
> (syntax-parse #'(a b 3)
    #:context #'(lambda (a b 3) (+ a b))
    [(x:id ...) 'ok])
lambda: expected identifier
  at: 3
  in: (lambda (a b 3) (+ a b))
> (syntax-parse #'(a b 3)
    #:context 'check-id-list
    [(x:id ...) 'ok])
check-id-list: expected identifier
  at: 3
  in: (a b 3)
```

```
#:literals (literal ...)

literal = literal-id
        | (pattern-id literal-id)
        | (pattern-id literal-id #:phase phase-expr)

  phase-expr : (or/c exact-integer? #f)
```

The `#:literals` option specifies identifiers that should be treated as literals rather than pattern variables. An entry in the literals list has two components: the identifier used within the pattern to signify the positions to be

Unlike `syntax-case`, `syntax-parse` requires all literals to have a binding. To match identifiers by their symbolic names, use `#:datum-literals` or the `~datum` pattern form instead.

matched (*pattern-id*), and the identifier expected to occur in those positions (*literal-id*). If the entry is a single identifier, that identifier is used for both purposes.

If the `#:phase` option is given, then the literal is compared at phase *phase-expr*. Specifically, the binding of the *literal-id* at phase *phase-expr* must match the input's binding at phase *phase-expr*.

In other words, the *syntax-pattern*s are interpreted as if each occurrence of *pattern-id* were replaced with the following pattern:

```
(~literal literal-id #:phase phase-expr)
```

```
#:datum-literals (datum-literal ...)

datum-literal = literal-id
              | (pattern-id literal-id)
```

Like `#:literals`, but the literals are matched as symbols instead of as identifiers.

In other words, the *syntax-pattern*s are interpreted as if each occurrence of *pattern-id* were replaced with the following pattern:

```
(~datum literal-id)
```

```
#:literal-sets (literal-set ...)

       literal-set = literal-set-id
                   | (literal-set-id literal-set-option ...)

literal-set-option = #:at lctx
                   | #:phase phase-expr

  phase-expr : (or/c exact-integer? #f)
```

Many literals can be declared at once via one or more literal sets, imported with the `#:literal-sets` option. See literal sets for more information.

If the `#:at` keyword is given, the lexical context of the *lctx* term is used to determine which identifiers in the patterns are treated as literals; this option is useful primarily for macros that generate `syntax-parse` expressions.

```
#:track-literals
```

If specified, each final *body* expression is further constrained to produce a single value, which must be a syntax object, and its `'disappeared-use` syntax property is automatically extended to include literals matched as part of pattern-matching. Literals are automatically tracked from uses of `#:literals`, `#:literal-sets`, or `~literal`, but they can also be manually tracked using `syntax-parse-state-cons!`. The property is added or extended in the same way as a property added by `syntax-parse-track-literals`.

Due to the way the *body* forms are wrapped, specifying this option means the final *body* form will no longer be in tail position with respect to the enclosing `syntax-parse` form.

Added in version 6.90.0.29 of package `base`.

```
#:conventions (conventions-id ...)
```

Imports conventions that give default syntax classes to pattern variables that do not explicitly specify a syntax class.

```
#:local-conventions (convention-rule ...)
```

Uses the conventions specified. The advantage of `#:local-conventions` over `#:conventions` is that local conventions can be in the scope of syntax-class parameter bindings. See the section on conventions for examples.

```
#:disable-colon-notation
```

Suppresses the "colon notation" for annotated pattern variables.

Examples:

```
> (syntax-parse #'(a b c)
    [(x:y ...) 'ok])
syntax-parse: not defined as syntax class
  at: y
  in: (syntax-parse (syntax (a b c)) ((x:y ...) (quote ok)))
> (syntax-parse #'(a b c) #:disable-colon-notation
    [(x:y ...) 'ok])
'ok
```

```
(syntax-parser parse-option ... clause ...+)
```

Like `syntax-parse`, but produces a matching procedure. The procedure accepts a single argument, which should be a syntax object.

```
(define/syntax-parse syntax-pattern pattern-directive ... stx-expr)

  stx-expr : syntax?
```

Definition form of `syntax-parse`. That is, it matches the syntax object result of *`stx-expr`* against *`syntax-pattern`* and creates pattern variable definitions for the attributes of *`syntax-pattern`*.

Examples:

```
> (define/syntax-parse ((~seq kw:keyword arg:expr) ...)
    #'(#:a 1 #:b 2 #:c 3))
> #'(kw ...)
#<syntax:eval:7:0 (#:a #:b #:c)>
```

Compare with `define/with-syntax`, a similar definition form that uses the simpler `syntax-case` patterns.

1.4 Specifying Syntax with Syntax Classes

Syntax classes provide an abstraction mechanism for syntax patterns. Built-in syntax classes are supplied that recognize basic classes such as `identifier` and `keyword`. Programmers can compose basic syntax classes to build specifications of more complex syntax, such as lists of distinct identifiers and formal arguments with keywords. Macros that manipulate the same syntactic structures can share syntax class definitions.

```
(define-syntax-class name-id stxclass-option ...
  stxclass-variant ...+)
(define-syntax-class (name-id . kw-formals) stxclass-option ...
  stxclass-variant ...+)
```

```
 stxclass-option = #:attributes (attr-arity-decl ...)
                 | #:auto-nested-attributes
                 | #:description description-expr
                 | #:opaque
                 | #:commit
                 | #:no-delimit-cut
                 | #:literals (literal-entry ...)
                 | #:datum-literals (datum-literal-entry ...)
                 | #:literal-sets (literal-set ...)
                 | #:conventions (convention-id ...)
                 | #:local-conventions (convention-rule ...)
                 | #:disable-colon-notation

 attr-arity-decl = attr-name-id
                 | (attr-name-id depth)

stxclass-variant = (pattern syntax-pattern pattern-directive ...)

  description-expr : (or/c string? #f)
```

Defines `name-id` as a *syntax class*, which encapsulates one or more single-term patterns.

A syntax class may have formal parameters, in which case they are bound as variables in the body. Syntax classes support optional arguments and keyword arguments using the same syntax as `lambda`. The body of the syntax-class definition contains a non-empty sequence of `pattern` variants.

The following options are supported:

```
#:attributes (attr-arity-decl ...)

attr-arity-decl = attr-id
                | (attr-id depth)
```

Declares the attributes of the syntax class. An attribute arity declaration consists of the attribute name and optionally its ellipsis depth (zero if not explicitly specified).

If the attributes are not explicitly listed, they are inferred as the set of all pattern variables occurring in every variant of the syntax class. Pattern variables that occur at different ellipsis depths are not included, nor are nested attributes from annotated pattern variables.

```
#:auto-nested-attributes
```

Deprecated. This option cannot be combined with `#:attributes`.

Declares the attributes of the syntax class as the set of all pattern variables and nested attributes from annotated pattern variables occurring in every variant of the syntax class. Only syntax classes defined strictly before the enclosing syntax class are used to compute the nested attributes; pattern variables annotated with not-yet-defined syntax classes contribute no nested attributes for export. Note that with this option, reordering syntax-class definitions may change the attributes they export.

```
#:description description-expr

  description-expr : (or/c string? #f)
```

The `description` argument is evaluated in a scope containing the syntax class's parameters. If the result is a string, it is used in error messages involving the syntax class. For example, if a term is rejected by the syntax class, an error of the form `"expected description"` may be synthesized. If the result is `#f`, the syntax class is skipped in the search for a description to report.

If the option is not given, the name of the syntax class is used instead.

```
#:opaque
```

Indicates that errors should not be reported with respect to the internal structure of the syntax class.

```
#:commit
```

Directs the syntax class to "commit" to the first successful match. When a variant succeeds, all choice points within the syntax class are discarded. See also `~commit`.

```
#:no-delimit-cut
```

By default, a cut (`~!`) within a syntax class only discards choice points within the syntax class. That is, the body of the syntax class acts as though it is wrapped in a `~delimit-cut` form. If `#:no-delimit-cut` is specified, a cut may affect choice points of the syntax class's calling context (another syntax class's patterns or a `syntax-parse` form).

It is an error to use both `#:commit` and `#:no-delimit-cut`.

```
#:literals (literal-entry ...)
```

```
#:datum-literals (datum-literal-entry ...)
```

```
#:literal-sets (literal-set ...)
```

```
#:conventions (convention-id ...)
```

Declares the literals and conventions that apply to the syntax class's variant patterns and their immediate `#:with` clauses. Patterns occurring within subexpressions of the syntax class (for example, on the right-hand side of a `#:fail-when` clause) are not affected.

```
#:local-conventions (convention-rule ...)
```

```
#:disable-colon-notation
```

These options have the same meaning as in `syntax-parse`.

Each variant of a syntax class is specified as a separate `pattern`-form whose syntax pattern is a single-term pattern.

```
(define-splicing-syntax-class name-id stxclass-option ...
  stxclass-variant ...+)
(define-splicing-syntax-class (name-id . kw-formals) stxclass-option ...
  stxclass-variant ...+)
```

Defines `name-id` as a *splicing syntax class*, analogous to a syntax class but encapsulating head patterns rather than single-term patterns.

The options are the same as for `define-syntax-class`.

Each variant of a splicing syntax class is specified as a separate `pattern`-form whose syntax pattern is a head pattern.

```
(pattern syntax-pattern pattern-directive ...)
```

Used to indicate a variant of a syntax class or splicing syntax class. The variant accepts syntax matching the given syntax pattern with the accompanying pattern directives.

When used within `define-syntax-class`, *`syntax-pattern`* should be a single-term pattern; within `define-splicing-syntax-class`, it should be a head pattern.

The attributes of the variant are the attributes of the pattern together with all attributes bound by `#:with` clauses, including nested attributes produced by syntax classes associated with the pattern variables.

```
this-syntax
```

When used as an expression within a syntax-class definition or `syntax-parse` expression, evaluates to the syntax object or syntax pair being matched.

Examples:

```
> (define-syntax-class one (pattern _ #:attr s this-syntax))
> (syntax-parse #'(1 2 3) [(1 o:one _) (attribute o.s)])
#<syntax:eval:3:0 2>
> (syntax-parse #'(1 2 3) [(1 . o:one) (attribute o.s)])
'(#<syntax:eval:4:0 2> #<syntax:eval:4:0 3>)
> (define-splicing-syntax-class two (pattern (~seq _ _) #:attr s this-
syntax))
> (syntax-parse #'(1 2 3) [(t:two 3) (attribute t.s)])
#<syntax:eval:6:0 (1 2 3)>
> (syntax-parse #'(1 2 3) [(1 t:two) (attribute t.s)])
'(#<syntax:eval:7:0 2> #<syntax:eval:7:0 3>)
```

Raises an error when used as an expression outside of a syntax-class definition or `syntax-parse` expression.

```
prop:syntax-class
 : (struct-type-property/c (or/c identifier?
                                 (-> any/c identifier?)))
```

A structure type property to identify structure types that act as an alias for a syntax class or splicing syntax class. The property value must be an identifier or a procedure of one argument.

When a transformer is bound to an instance of a struct with this property, then it may be used as a syntax class or splicing syntax class in the same way as the bindings created by `define-syntax-class` or `define-splicing-syntax-class`. If the value of the property is an identifier, then it should be bound to a syntax class or splicing syntax class, and the binding will be treated as an alias for the referenced syntax class. If the value of the property is a procedure, then it will be applied to the value with the `prop:syntax-class` property to obtain an identifier, which will then be used as in the former case.

Examples:

```
> (begin-for-syntax
    (struct expr-and-stxclass (expr-id stxclass-id)
      #:property prop:procedure
      (lambda (this stx) ((set!-transformer-procedure
                           (make-variable-like-transformer
                            (expr-and-stxclass-expr-id this)))
                          stx))
      #:property prop:syntax-class
      (lambda (this) (expr-and-stxclass-stxclass-id this))))
> (define-syntax is-id? (expr-and-stxclass #'identifier? #'id))
> (is-id? #'x)
#t
> (syntax-parse #'x
    [x:is-id? #t]
    [_ #f])
#t
```

Added in version 7.2.0.4 of package `base`.

1.4.1 Pattern Directives

Both the parsing forms and syntax class definition forms support *pattern directives* for annotating syntax patterns and specifying side conditions. The grammar for pattern directives follows:

```
pattern-directive = #:declare pvar-id stxclass maybe-role
                  | #:post action-pattern
                  | #:and action-pattern
                  | #:with syntax-pattern stx-expr
                  | #:attr attr-arity-decl expr
                  | #:fail-when condition-expr message-expr
                  | #:fail-unless condition-expr message-expr
                  | #:when condition-expr
                  | #:do [def-or-expr ...]
                  | #:undo [def-or-expr ...]
                  | #:cut
```

```
#:declare pvar-id stxclass maybe-role

  stxclass = syntax-class-id
           | (syntax-class-id arg ...)

maybe-role =
           | #:role role-expr
```

Associates `pvar-id` with a syntax class and possibly a role, equivalent to replacing each occurrence of `pvar-id` in the pattern with `(~var pvar-id stxclass maybe-role)`. The second form of `stxclass` allows the use of parameterized syntax classes, which cannot be expressed using the "colon" notation. The `arg`s are evaluated in the scope where the `pvar-id` occurs in the pattern. Keyword arguments are supported, using the same syntax as in `#%app`.

If a `#:with` directive appears between the main pattern (e.g., in a `syntax-parse` or `define-syntax-class` clause) and a `#:declare`, then only pattern variables from the `#:with` pattern may be declared.

Examples:

```
> (syntax-parse #'P
    [x
     #:declare x id
     #'x])
#<syntax:eval:12:0 P>
> (syntax-parse #'L
    [x
     #:with y #'x
     #:declare x id
     #'x])
syntax-parse: identifier in #:declare clause does not appear
in pattern;
 this #:declare clause affects only the preceding #:with
pattern
   at: x
   in: (syntax-parse (syntax L) (x #:with y (syntax x)
#:declare x id (syntax x)))
> (syntax-parse #'T
    [x
     #:with y #'x
     #:declare y id
     #'x])
#<syntax:eval:14:0 T>
```

```
#:post action-pattern
```

Executes the given action pattern as a "post-traversal check" after matching the main pattern. That is, the following are equivalent:

```
main-pattern #:post action-pattern
main-pattern #:and (~post action-pattern)
(~and main-pattern (~post action-pattern))
```

```
#:and action-pattern
```

Like `#:post` except that no `~post` wrapper is added. That is, the following are equivalent:

```
main-pattern #:and action-pattern
(~and main-pattern action-pattern)
```

```
#:with syntax-pattern stx-expr
```

Evaluates the *stx-expr* in the context of all previous attribute bindings and matches it against the pattern. If the match succeeds, the pattern's attributes are added to environment for the evaluation of subsequent side conditions. If the `#:with` match fails, the matching process backtracks. Since a syntax object may match a pattern in several ways, backtracking may cause the same clause to be tried multiple times before the next clause is reached.

If the value of *stx-expr* is not a syntax object, it is implicitly converted to a syntax object. If the the conversion would produce *3D syntax*—that is, syntax that contains unwritable values such as procedures, non-prefab structures, etc—then an exception is raised instead.

Equivalent to `#:post (~parse syntax-pattern stx-expr)`.

Examples:

```
> (syntax-parse #'(1 2 3)
    [(a b c)
     #:with rev #'(c b a)
     #'rev])
#<syntax:eval:15:0 (3 2 1)>
> (syntax-parse #'(['x "Ex."] ['y "Why?"] ['z "Zee!"])
    [([stuff ...] ...)
     #:with h #'(hash stuff ... ...)
     #'h])
#<syntax:eval:16:0 (hash (quote x) "Ex." (quote y)
"Why?" (quote z) "Zee!")>
```

```
#:attr attr-arity-decl expr
```

Evaluates the *expr* in the context of all previous attribute bindings and binds it to the given attribute. The value of *expr* need not be, or even contain, syntax—see `attribute` for details.

Equivalent to `#:and (~bind attr-arity-decl expr)`.

Examples:

```
> (syntax-parse #'("do" "mi")
    [(a b)
     #:attr rev #'(b a)
     #'rev])
#<syntax:eval:17:0 ("mi" "do")>
> (syntax-parse #'(1 2)
    [(a:number b:number)
     #:attr sum (+ (syntax-e #'a) (syntax-e #'b))
     (attribute sum)])
3
```

The `#:attr` directive is often used in syntax classes:

Examples:

```
> (define-syntax-class ab-sum
    (pattern (a:number b:number)
      #:attr sum (+ (syntax-e #'a) (syntax-e #'b))))
> (syntax-parse #'(1 2)
    [x:ab-sum
     (attribute x.sum)])
3
```

```
#:fail-when condition-expr message-expr

  message-expr : (or/c string? #f)
```

Evaluates the *condition-expr* in the context of all previous attribute bindings. If the value is any true value (not `#f`), the matching process backtracks (with the given message); otherwise, it continues. If the value of the condition expression is a syntax object, it is indicated as the cause of the error.

If the *message-expr* produces a string it is used as the failure message; otherwise the failure is reported in terms of the enclosing descriptions.

Equivalent to `#:post (~fail #:when condition-expr message-expr)`.

Examples:

```
> (syntax-parse #'(m 4)
    [(m x:number)
     #:fail-when (even? (syntax-e #'x))
     "expected an odd number"
     #'x])
m: expected an odd number
  at: (m 4)
  in: (m 4)
> (syntax-parse #'(m 4)
    [(m x:number)
     #:fail-when (and (even? (syntax-e #'x)) #'x)
     "expected an odd number"
     #'x])
m: expected an odd number
  at: 4
  in: (m 4)
```

```
#:fail-unless condition-expr message-expr

  message-expr : (or/c string? #f)
```

Like `#:fail-when` with the condition negated.

Equivalent to `#:post (~fail #:unless condition-expr message-expr)`.

Example:

```
> (syntax-parse #'(m 5)
    [(m x:number)
     #:fail-unless (even? (syntax-e #'x))
     "expected an even number"
     #'x])
m: expected an even number
  at: (m 5)
  in: (m 5)
```

```
#:when condition-expr
```

Evaluates the *condition-expr* in the context of all previous attribute bindings. If the value is `#f`, the matching process backtracks. In other words, `#:when` is like `#:fail-unless` without the message argument.

Equivalent to `#:post (~fail #:unless condition-expr #f)`.

Example:

```
> (syntax-parse #'(m 5)
    [(m x:number)
     #:when (even? (syntax-e #'x))
     #'x])
m: bad syntax
  in: (m 5)
```

```
#:do [defn-or-expr ...]
```

Takes a sequence of definitions and expressions, which may be intermixed, and evaluates them in the scope of all previous attribute bindings. The names bound by the definitions are in scope in the expressions of subsequent patterns and clauses.

There is currently no way to bind attributes using a `#:do` block. It is an error to shadow an attribute binding with a definition in a `#:do` block.

Equivalent to `#:and (~do defn-or-expr ...)`.

```
#:undo [defn-or-expr ...]
```

Has no effect when initially matched, but if backtracking returns to a point *before* the `#:undo` directive, the `defn-or-expr`s are executed. See `~undo` for an example.

Equivalent to `#:and (~undo defn-or-expr ...)`.

```
#:cut
```

Eliminates backtracking choice points and commits parsing to the current branch at the current point.

Equivalent to `#:and ~!`.

1.4.2 Pattern Variables and Attributes

An *attribute* is a name bound by a syntax pattern. An attribute can be a pattern variable itself, or it can be a nested attribute bound by an annotated pattern variable. The name of a nested attribute is computed by concatenating the pattern variable name with the syntax class's exported attribute's name, separated by a dot (see the example below).

Attributes can be used in three ways: with the `attribute` form; inside syntax templates via `syntax`, `quasisyntax`, etc; and inside `datum` templates. Attribute names cannot be used directly as expressions; that is, attributes are not variables.

A *syntax-valued attribute* is an attribute whose value is a syntax object or list of the appropriate ellipsis depth. That is, an attribute with ellipsis depth 0 is syntax-valued if its value is `syntax?`; an attribute with ellipsis depth 1 is syntax-valued if its value is `(listof syntax?)`; an attribute with ellipsis depth 2 is syntax-valued if its value is `(listof (listof syntax?))`; and so on. The value is considered syntax-valued if it contains promises that when completely forced produces a suitable syntax object or list. Syntax-valued attributes can be used within `syntax`, `quasisyntax`, etc as part of a syntax template. If an attribute is used inside a syntax template but it is not syntax-valued, an error is signaled.

There are uses for non-syntax-valued attributes. A non-syntax-valued attribute can be used to return a parsed representation of a subterm or the results of an analysis on the subterm. A non-syntax-valued attribute must be bound using the `#:attr` directive or a `~bind` pattern; `#:with` and `~parse` will convert the right-hand side to a (possibly 3D) syntax object.

Example:

```
> (define-syntax-class table
    (pattern ((key value) ...)
             #:attr hashtable
                    (for/hash ([k (syntax->datum #'(key ...))]
                               [v (syntax->datum #'(value ...))])
                      (values k v))
             #:attr [sorted-kv 1]
                    (delay
                     (printf "sorting!\n")
                     (sort (syntax->list #'((key value) ...))
                           <
                           #:key (lambda (kv) (cadr (syntax-
>datum kv)))))))
```

The `table` syntax class provides four attributes: `key`, `value`, `hashtable`, and `sorted-kv`. The `hashtable` attribute has ellipsis depth 0 and the rest have depth 1; `key`, `value`, and `sorted-kv` are syntax-valued, but `hashtable` is not. The `sorted-kv` attribute's value is a promise; it will be automatically forced if used in a template.

Syntax-valued attributes can be used in syntax templates:

```
> (syntax-parse #'((a 3) (b 2) (c 1))
    [t:table
     #'(t.key ...)])
#<syntax:eval:26:0 (a b c)>
> (syntax-parse #'((a 3) (b 2) (c 1))
    [t:table
     #'(t.sorted-kv ...)])
sorting!
#<syntax:eval:27:0 ((c 1) (b 2) (a 3))>
```

But non-syntax-valued attributes cannot:

```
> (syntax-parse #'((a 3) (b 2) (c 1))
    [t:table
     #'t.hashtable])
t.hashtable: attribute contains non-syntax value
  value: '#hash((a . 3) (b . 2) (c . 1))
  in: t.hashtable
```

The `attribute` form gets the value of an attribute, whether it is syntax-valued or not.

```
> (syntax-parse #'((a 1) (b 2) (c 3))
    [t:table
     (attribute t.hashtable)])
'#hash((a . 1) (b . 2) (c . 3))
> (syntax-parse #'((a 3) (b 2) (c 1))
    [t:table
     (attribute t.sorted-kv)])
#<promise:sorted-kv326>
```

Every attribute has an associated *ellipsis depth* that determines how it can be used in a syntax template (see the discussion of ellipses in `syntax`). For a pattern variable, the ellipsis depth is the number of ellipses the pattern variable "occurs under" in the pattern. An attribute bound by `#:attr` has depth 0 unless declared otherwise. For a nested attribute the depth is the sum of the annotated pattern variable's depth and the depth of the attribute exported by the syntax class.

Consider the following code:

```
(define-syntax-class quark
  (pattern (a b ...)))
(syntax-parse some-term
  [(x (y:quark ...) ... z:quark)
   some-code])
```

The syntax class `quark` exports two attributes: `a` at depth 0 and `b` at depth 1. The `syntax-parse` pattern has three pattern variables: `x` at depth 0, `y` at depth 2, and `z` at depth 0. Since `y` and `z` are annotated with the `quark` syntax class, the pattern also binds the following nested attributes: `y.a` at depth 2, `y.b` at depth 3, `z.a` at depth 0, and `z.b` at depth 1.

An attribute's ellipsis nesting depth is *not* a guarantee that it is syntax-valued or has any list structure. In particular, `~or*` and `~optional` patterns may result in attributes with fewer than expected levels of list nesting, and `#:attr` and `~bind` can be used to bind attributes to arbitrary values.

Example:

```
> (syntax-parse #'(a b 3)
    [(~or* (x:id ...) _)
     (attribute x)])
#f
```

```
(attribute attr-id)
```

Returns the value associated with the attribute named *attr-id*. If *attr-id* is not bound as an attribute, a syntax error is raised.

Attributes and `datum`

The `datum` form is another way, in addition to `syntax` and `attribute`, of using syntax pattern variables and attributes. Unlike `syntax`, `datum` does not require attributes to be syntax-valued. Wherever the `syntax` form would create syntax objects based on its template (as opposed to reusing syntax objects bound by pattern variables), the `datum` form creates plain S-expressions.

Continuing the `table` example from above, we can use `datum` with the `key` attribute as follows:

```
> (syntax-parse #'((a 1) (b 2) (c 3))
    [t:table (datum (t.key ...))])
'(#<syntax:eval:32:0 a> #<syntax:eval:32:0 b> #<syntax:eval:32:0
c>)
```

A `datum` template may contain multiple pattern variables combined within some S-expression structure:

```
> (syntax-parse #'((a 1) (b 2) (c 3))
    [t:table (datum ([t.key t.value] ...))])
'((#<syntax:eval:33:0 a> #<syntax:eval:33:0 1>)
  (#<syntax:eval:33:0 b> #<syntax:eval:33:0 2>)
  (#<syntax:eval:33:0 c> #<syntax:eval:33:0 3>))
```

A `datum` template can use the `~@` and `~?` template forms:

```
> (syntax-parse #'((a 1) (b 2) (c 3))
    [t:table (datum ((~@ t.key t.value) ...))])
'(#<syntax:eval:34:0 a>
  #<syntax:eval:34:0 1>
  #<syntax:eval:34:0 b>
  #<syntax:eval:34:0 2>
  #<syntax:eval:34:0 c>
  #<syntax:eval:34:0 3>)
```

```
> (syntax-parse #'((a 56) (b 71) (c 13))
    [t:table (datum ((~@ . t.sorted-kv) ...))])
sorting!
'(#<syntax:eval:35:0 c>
  #<syntax:eval:35:0 13>
  #<syntax:eval:35:0 a>
  #<syntax:eval:35:0 56>
  #<syntax:eval:35:0 b>
  #<syntax:eval:35:0 71>)
> (syntax-parse #'( ((a 1) (b 2) (c 3)) ((d 4) (e 5)))
    [(t1:table (~or* t2:table #:nothing))
     (datum (t1.key ... (~? (~@ t2.key ...))))])
'(#<syntax:eval:36:0 a>
  #<syntax:eval:36:0 b>
  #<syntax:eval:36:0 c>
  #<syntax:eval:36:0 d>
  #<syntax:eval:36:0 e>)
> (syntax-parse #'( ((a 1) (b 2) (c 3)) #:nothing)
    [(t1:table (~or* t2:table #:nothing))
     (datum (t1.key ... (~? (~@ t2.key ...))))])
'(#<syntax:eval:37:0 a> #<syntax:eval:37:0 b> #<syntax:eval:37:0
c>)
```

However, unlike for `syntax`, a value of `#f` only signals a template failure to `~?` if a list is needed for ellipsis iteration, as in the previous example; it does not cause a failure when it occurs as a leaf. Contrast the following:

```
> (syntax-parse #'( ((a 1) (b 2) (c 3)) #:nothing)
    [(t1:table (~or* t2:table #:nothing))
     #'(~? t2 skipped)])
#<syntax:eval:38:0 skipped>
> (syntax-parse #'( ((a 1) (b 2) (c 3)) #:nothing)
    [(t1:table (~or* t2:table #:nothing))
     (datum (~? t2 skipped))])
#f
```

The `datum` form is also useful for accessing non-syntax-valued attributes. Compared to `attribute`, `datum` has the following advantage: The use of ellipses in `datum` templates provides a visual reminder of the list structure of their results. For example, if the pattern is `(t:table ...)`, then both `(attribute t.hashtable)` and `(datum (t.hashtable ...))` produce a `(listof hash?)`, but the ellipses make it more apparent.

Changed in version 7.8.0.9 of package `base`: Added support for syntax pattern variables and attributes to `datum`.

1.5 Syntax Patterns

The grammar of *syntax patterns* used by `syntax/parse` facilities is given in the following table. There are four main kinds of syntax pattern:

- single-term patterns, abbreviated *S-pattern*
- head patterns, abbreviated *H-pattern*
- ellipsis-head patterns, abbreviated *EH-pattern*
- action patterns, abbreviated *A-pattern*

A fifth kind, list patterns (abbreviated *L-pattern*), is just a syntactically restricted subset of single-term patterns.

When a special form in this manual refers to *syntax-pattern* (eg, the description of the `syntax-parse` special form), it means specifically single-term pattern.

```
S-pattern = pvar-id
          | pvar-id:syntax-class-id
          | pvar-id:literal-id
          | literal-id
          | (~var^s- id)
          | (~var^s+ id syntax-class-id maybe-role)
          | (~var^s+ id (syntax-class-id arg ...) maybe-role)
          | (~literal literal-id maybe-phase)
          | atomic-datum
          | (~datum datum)
          | (H-pattern . S-pattern)
          | (A-pattern . S-pattern)
          | (EH-pattern ... . S-pattern)
          | (H-pattern ...+ . S-pattern)
          | (~and^s proper-S/A-pattern ...+)
          | (~or*^s S-pattern ...+)
          | (~not S-pattern)
          | #(pattern-part ...)
          | #s(prefab-struct-key pattern-part ...)
          | #&S-pattern
          | (~rest S-pattern)
          | (~describe^s maybe-opaque maybe-role expr S-pattern)
          | (~commit^s S-pattern)
          | (~delimit-cut^s S-pattern)
          | (~post^s S-pattern)
          | A-pattern
```

```
       L-pattern = ()
                 | (A-pattern . L-pattern)
                 | (H-pattern . L-pattern)
                 | (EH-pattern ... . L-pattern)
                 | (H-pattern ...+ . L-pattern)
                 | (~rest L-pattern)

       H-pattern = pvar-id:splicing-syntax-class-id
                 | (~var^h id splicing-syntax-class-id maybe-role)
                 | (~var^h id (splicing-syntax-class-id arg ...)
                          maybe-role)
                 | (~seq . L-pattern)
                 | (~and^h proper-H/A-pattern ...+)
                 | (~or*^h H-pattern ...+)
                 | (~optional^h H-pattern maybe-optional-option)
                 | (~describe^h maybe-opaque maybe-role expr H-pattern)
                 | (~commit^h H-pattern)
                 | (~delimit-cut^h H-pattern)
                 | (~post^h H-patter)
                 | (~peek H-pattern)
                 | (~peek-not H-pattern)
                 | proper-S-pattern

      EH-pattern = (~alt EH-pattern ...)
                 | (~once H-pattern once-option ...)
                 | (~optional^eh H-pattern optional-option ...)
                 | (~between H min-number max-number between-option)
                 | H-pattern

       A-pattern = ~!
                 | (~bind [attr-arity-decl expr] ...)
                 | (~fail maybe-fail-condition maybe-message-expr)
                 | (~parse S-pattern stx-expr)
                 | (~and^a A-pattern ...+)
                 | (~post^a A-pattern)
                 | (~do defn-or-expr ...)
                 | (~undo defn-or-expr ...)

proper-S-pattern = a S-pattern that is not a A-pattern

proper-H-pattern = a H-pattern that is not a S-pattern
```

The following pattern keywords can be used in multiple pattern variants:

`~var`

One of `~var`$^{\text{s-}}$, `~var`$^{\text{s+}}$, or `~var`$^{\text{h}}$.

`~and`

One of `~and`$^{\text{s}}$, `~and`$^{\text{h}}$, or `~and`$^{\text{a}}$:

- `~and`$^{\text{a}}$ if all of the conjuncts are action patterns
- `~and`$^{\text{h}}$ if any of the conjuncts is a proper head pattern
- `~and`$^{\text{s}}$ otherwise

`~or*`

One of `~or*`$^{\text{s}}$ or `~or*`$^{\text{h}}$:

- `~or*`$^{\text{h}}$ if any of the disjuncts is a proper head pattern
- `~or*`$^{\text{s}}$ otherwise

`~or`

Behaves like `~or*`$^{\text{s}}$, `~or*`$^{\text{h}}$, or `~alt`:

- like `~alt` if the pattern occurs directly before ellipses (`...`) or immediately within another `~alt` pattern
- like `~or*`$^{\text{h}}$ if any of the disjuncts is a proper head pattern
- like `~or*`$^{\text{s}}$ otherwise

The context-sensitive interpretation of `~or` is a design mistake and a common source of confusion. Use `~alt` and `~or*` instead.

`~describe`

One of `~describe`$^{\text{s}}$ or `~describe`$^{\text{h}}$:

- `~describe`$^{\text{h}}$ if the subpattern is a proper head pattern

- `~describe`^{s} otherwise

`~commit`

One of `~commit`^{s} or `~commit`^{h}:

- `~commit`^{h} if the subpattern is a proper head pattern
- `~commit`^{s} otherwise

`~delimit-cut`

One of `~delimit-cut`^{s} or `~delimit-cut`^{h}:

- `~delimit-cut`^{h} if the subpattern is a proper head pattern
- `~delimit-cut`^{s} otherwise

`~post`

One of `~post`^{s}, `~post`^{h}, or `~post`^{a}:

- `~post`^{a} if the subpattern is an action pattern
- `~post`^{h} if the subpattern is a proper head pattern
- `~post`^{s} otherwise

`~optional`

One of `~optional`^{h} or `~optional`^{eh}:

- `~optional`^{eh} if it is an immediate disjunct of an `~alt` pattern
- `~optional`^{h} otherwise

1.5.1 Single-term Patterns

A *single-term pattern* (abbreviated *S-pattern*) is a pattern that describes a single term. These are like the traditional patterns used in `syntax-rules` and `syntax-case`, but with additional variants that make them more expressive.

"Single-term" does not mean "atomic"; a single-term pattern can have complex structure, and it can match terms that have many parts. For example, `(17 ...)` is a single-term pattern that matches any term that is a proper list of repeated `17` numerals.

A *proper single-term pattern* is one that is not an action pattern.

The *list patterns* (for "list pattern") are single-term patterns having a restricted structure that guarantees that they match only terms that are proper lists.

Here are the variants of single-term pattern:

id

An identifier can be either a pattern variable, an annotated pattern variable, or a literal:

- If *id* is the "pattern" name of an entry in the literals list, it is a literal pattern that behaves like `(~literal id)`.

  Examples:

```
> (syntax-parse #'(define x 12)
    #:literals (define)
    [(define var:id body:expr) 'ok])
'ok
> (syntax-parse #'(lambda x 12)
    #:literals (define)
    [(define var:id body:expr) 'ok])
lambda: expected the identifier `define'
  at: lambda
  in: (lambda x 12)
> (syntax-parse #'(define x 12)
    #:literals ([def define])
    [(def var:id body:expr) 'ok])
'ok
> (syntax-parse #'(lambda x 12)
    #:literals ([def define])
    [(def var:id body:expr) 'ok])
lambda: expected the identifier `define'
  at: lambda
  in: (lambda x 12)
```

- If *id* is of the form *pvar-id:syntax-class-id* (that is, two names joined by a colon character), it is an annotated pattern variable, and the pattern is equivalent to (~var *pvar-id* *syntax-class-id*).

  Examples:

  ```
  > (syntax-parse #'a
      [var:id (syntax-e #'var)])
  'a
  > (syntax-parse #'12
      [var:id (syntax-e #'var)])
  ?: expected identifier
    at: 12
    in: 12
  > (define-syntax-class two
      #:attributes (x y)
      (pattern (x y)))
  > (syntax-parse #'(a b)
      [t:two (syntax->datum #'(t t.x t.y))])
  '((a b) a b)
  > (syntax-parse #'(a b)
      [t
       #:declare t two
       (syntax->datum #'(t t.x t.y))])
  '((a b) a b)
  ```

  Note that an *id* of the form *:syntax-class-id* is legal; see the discussion of a ~var^{s+} form with a zero-length *pvar-id*.

- If *id* is of the form *pvar-id:literal-id*, where *literal-id* is in the literals list, then it is equivalent to (~and (~var *pvar-id*) literal-id).

  Examples:

  ```
  > (require (only-in racket/base [define def]))
  > (syntax-parse #'(def x 7)
      #:literals (define)
      [(d:define var:id body:expr) #'d])
  #<syntax:eval:11:0 def>
  ```

- Otherwise, *id* is a pattern variable, and the pattern is equivalent to (~var *id*).

(~var *pvar-id*)

A *pattern variable*. If *pvar-id* has no syntax class (by #:convention), the pattern variable matches anything. The pattern variable is bound to the matched subterm, unless the pattern variable is the wildcard (_), in which case no binding occurs.

If *pvar-id* does have an associated syntax class, it behaves like an annotated pattern variable with the implicit syntax class inserted.

```
(~var pvar-id syntax-class-use maybe-role)

syntax-class-use = syntax-class-id
                 | (syntax-class-id arg ...)

      maybe-role =
                 | #:role role-expr

  role-expr : (or/c string? #f)
```

An *annotated pattern variable*. The pattern matches only terms accepted by `syntax-class-id` (parameterized by the `arg`s, if present).

In addition to binding `pvar-id`, an annotated pattern variable also binds *nested attributes* from the syntax class. The names of the nested attributes are formed by prefixing `pvar-id.` (that is, `pvar-id` followed by a "dot" character) to the name of the syntax class's attribute.

If `pvar-id` is _, no attributes are bound. If `pvar-id` is the zero-length identifier (`||`), then `pvar-id` is not bound, but the nested attributes of `syntax-class-use` are bound without prefixes.

If `role-expr` is given and evaluates to a string, it is combined with the syntax class's description in error messages.

Examples:

```
> (syntax-parse #'a
    [(~var var id) (syntax-e #'var)])
'a
> (syntax-parse #'12
    [(~var var id) (syntax-e #'var)])
?: expected identifier
  at: 12
  in: 12
> (define-syntax-class two
    #:attributes (x y)
    (pattern (x y)))
> (syntax-parse #'(a b)
    [(~var t two) (syntax->datum #'(t t.x t.y))])
'((a b) a b)
> (define-syntax-class (nat-less-than n)
    (pattern x:nat #:when (< (syntax-e #'x) n)))
```

```
> (syntax-parse #'(1 2 3 4 5)
    [((~var small (nat-less-than 4)) ... large:nat ...)
     (list #'(small ...) #'(large ...))])
'(#<syntax:eval:17:0 (1 2 3)> #<syntax:eval:17:0 (4 5)>)
> (syntax-parse #'(m a b 3)
    [(_ (~var x id #:role "variable") ...) 'ok])
m: expected identifier for variable
  at: 3
  in: (m a b 3)
```

```
(~literal literal-id maybe-phase)

maybe-phase =
            | #:phase phase-expr
```

A *literal* identifier pattern. Matches any identifier `free-identifier=?` to *`literal-id`*.

Examples:

```
> (syntax-parse #'(define x 12)
    [((~literal define) var:id body:expr) 'ok])
'ok
> (syntax-parse #'(lambda x 12)
    [((~literal define) var:id body:expr) 'ok])
lambda: expected the identifier `define'
  at: lambda
  in: (lambda x 12)
```

The identifiers are compared at the phase given by *`phase-expr`*, if it is given, or `(syntax-local-phase-level)` otherwise.

```
atomic-datum
```

Numbers, strings, booleans, keywords, and the empty list match as literals.

Examples:

```
> (syntax-parse #'(a #:foo bar)
    [(x #:foo y) (syntax->datum #'y)])
'bar
> (syntax-parse #'(a foo bar)
    [(x #:foo y) (syntax->datum #'y)])
a: expected the literal #:foo
  at: foo
  in: (a foo bar)
```

```
(~datum datum)
```

Matches syntax whose S-expression contents (obtained by `syntax->datum`) is `equal?` to the given *`datum`*.

Examples:

```
> (syntax-parse #'(a #:foo bar)
    [(x (~datum #:foo) y) (syntax->datum #'y)])
'bar
> (syntax-parse #'(a foo bar)
    [(x (~datum #:foo) y) (syntax->datum #'y)])
a: expected the literal #:foo
  at: foo
  in: (a foo bar)
```

The `~datum` form is useful for recognizing identifiers symbolically, in contrast to the `~literal` form, which recognizes them by binding.

Examples:

```
> (define-syntax (is-define? stx)
    (syntax-parse stx
      [(is-define? id)
       (syntax-parse #'id
         [(~literal define) #''yes]
         [(~datum   define) #''not-really]
         [_                 #''not-even-close])]))
> (is-define? define)
'yes
> (let ([define 42])
    (is-define? define))
'not-really
> (is-define? something-else)
'not-even-close
```

```
(H-pattern . S-pattern)
```

Matches any term that can be decomposed into a list prefix matching *`H-pattern`* and a suffix matching *`S-pattern`*.

Note that the pattern may match terms that are not even improper lists; if the head pattern can match a zero-length head, then the whole pattern matches whatever the tail pattern accepts.

The first pattern can be a single-term pattern, in which case the whole pattern matches any pair whose first element matches the first pattern and whose rest matches the second.

See head patterns for more information.

`(A-pattern . S-pattern)`

Performs the actions specified by `A-pattern`, then matches any term that matches `S-pattern`.

Pragmatically, one can throw an action pattern into any list pattern. Thus, `(x y z)` is a pattern matching a list of three terms, and `(x y ~! z)` is a pattern matching a list of three terms, with a cut performed after the second one. In other words, action patterns "don't take up space."

See action patterns for more information.

`(EH-pattern ... . S-pattern)`

Matches any term that can be decomposed into a list head matching some number of repetitions of the `EH-pattern` alternatives (subject to its repetition constraints) followed by a list tail matching `S-pattern`.

In other words, the whole pattern matches either the second pattern (which need not be a list) or a term whose head matches one of the alternatives of the first pattern and whose tail recursively matches the whole sequence pattern.

See ellipsis-head patterns for more information.

`(H-pattern ...+ . S-pattern)`

Like an ellipses (`...`) pattern, but requires at least one occurrence of the head pattern to be present.

That is, the following patterns are equivalent:

- `(H ...+ . S)`
- `((~between H 1 +inf.0) ... . S)`

Examples:

```
> (syntax-parse #'(1 2 3)
    [(n:nat ...+) 'ok])
'ok
> (syntax-parse #'()
    [(n:nat ...+) 'ok]
    [_ 'none])
'none
```

```
(~and S/A-pattern ...)
```

Matches any term that matches all of the subpatterns.

The subpatterns can contain a mixture of single-term patterns and action patterns, but must contain at least one single-term pattern.

Attributes bound in subpatterns are available to subsequent subpatterns. The whole pattern binds all of the subpatterns' attributes.

One use for ~and-patterns is preserving a whole term (including its lexical context, source location, etc) while also examining its structure. Syntax classes are useful for the same purpose, but ~and can be lighter weight.

Examples:

```
> (define-syntax (import stx)
    (raise-syntax-error #f "illegal use of import" stx))
> (define (check-imports stx) ....)
> (syntax-parse #'(m (import one two))
   #:literals (import)
   [(_ (~and import-clause (import i ...)))
    (let ([bad (check-imports
                (syntax->list #'(i ...)))])
      (when bad
        (raise-syntax-error
         #f "bad import" #'import-clause bad))
      'ok)])
'ok
```

```
(~or* S-pattern ...)
```

Matches any term that matches one of the included patterns. The alternatives are tried in order.

The whole pattern binds *all* of the subpatterns' attributes. An attribute that is not bound by the "chosen" subpattern has a value of #f. The same attribute may be bound by multiple subpatterns, and if it is bound by all of the subpatterns, it is sure to have a value if the whole pattern matches.

Examples:

```
> (syntax-parse #'a
    [(~or* x:id y:nat) (values (attribute x) (attribute y))])
#<syntax:eval:34:0 a>
#f
> (syntax-parse #'(a 1)
    [(~or* (x:id y:nat) (x:id)) (values #'x (attribute y))])
```

```
#<syntax:eval:35:0 a>
#<syntax:eval:35:0 1>
> (syntax-parse #'(b)
    [(~or* (x:id y:nat) (x:id)) (values #'x (attribute y))])
#<syntax:eval:36:0 b>
#f
```

```
(~not S-pattern)
```

Matches any term that does not match the subpattern. None of the subpattern's attributes are bound outside of the ~not-pattern.

Example:

```
> (syntax-parse #'(x y z => u v)
    #:literals (=>)
    [((~and before (~not =>)) ... => after ...)
     (list #'(before ...) #'(after ...))])
'(#<syntax:eval:37:0 (x y z)> #<syntax:eval:37:0 (u v)>)
```

```
#(pattern-part ...)
```

Matches a term that is a vector whose elements, when considered as a list, match the single-term pattern corresponding to `(pattern-part ...)`.

Examples:

```
> (syntax-parse #'#(1 2 3)
    [#(x y z) (syntax->datum #'z)])
3
> (syntax-parse #'#(1 2 3)
    [#(x y ...) (syntax->datum #'(y ...))])
'(2 3)
> (syntax-parse #'#(1 2 3)
    [#(x ~rest y) (syntax->datum #'y)])
'(2 3)
```

```
#s(prefab-struct-key pattern-part ...)
```

Matches a term that is a prefab struct whose key is exactly the given key and whose sequence of fields, when considered as a list, match the single-term pattern corresponding to `(pattern-part ...)`.

Examples:

```
> (syntax-parse #'#s(point 1 2 3)
    [#s(point x y z) 'ok])
'ok
> (syntax-parse #'#s(point 1 2 3)
    [#s(point x y ...) (syntax->datum #'(y ...))])
'(2 3)
> (syntax-parse #'#s(point 1 2 3)
    [#s(point x ~rest y) (syntax->datum #'y)])
'(2 3)
```

`#&S-pattern`

Matches a term that is a box whose contents matches the inner single-term pattern.

Example:

```
> (syntax-parse #'#&5
    [#&n:nat 'ok])
'ok
```

`(~rest S-pattern)`

Matches just like `S-pattern`. The `~rest` pattern form is useful in positions where improper ("dotted") lists are not allowed by the reader, such as vector and structure patterns (see above).

Examples:

```
> (syntax-parse #'(1 2 3)
    [(x ~rest y) (syntax->datum #'y)])
'(2 3)
> (syntax-parse #'#(1 2 3)
    [#(x ~rest y) (syntax->datum #'y)])
'(2 3)
```

```
(~describe maybe-role maybe-opaque expr S-pattern)

maybe-opaque =
             | #:opaque

  maybe-role =
             | #:role role-expr
```

```
expr : (or/c string? #f)
role-expr : (or/c string? #f)
```

The `~describe` pattern form annotates a pattern with a description, a string expression that is evaluated in the scope of all prior attribute bindings. If parsing the inner pattern fails, then the description is used to synthesize the error message. A `~describe` pattern does not influence backtracking.

If `#:opaque` is given, failure information from within *S-pattern* is discarded and the error is reported solely in terms of the description given.

If *role-expr* is given and produces a string, its value is combined with the description in error messages.

Examples:

```
> (syntax-parse #'(m 1)
    [(_ (~describe "id pair" (x:id y:id))) 'ok])
m: expected id pair
  at: 1
  in: (m 1)
> (syntax-parse #'(m (a 2))
    [(_ (~describe "id pair" (x:id y:id))) 'ok])
m: expected identifier
  at: 2
  in: (m (a 2))
  parsing context:
   while parsing id pair
    term: (a 2)
    location: eval:48:0
> (syntax-parse #'(m (a 2))
    [(_ (~describe #:opaque "id pair" (x:id y:id))) 'ok])
m: expected id pair
  at: (a 2)
  in: (m (a 2))
> (syntax-parse #'(m 1)
    [(_ (~describe #:role "formals" "id
pair" (x y))) 'ok])
m: expected id pair for formals
  at: 1
  in: (m 1)
```

```
(~commit S-pattern)
```

The `~commit` pattern form affects backtracking in two ways:

- If the pattern succeeds, then all choice points created within the subpattern are discarded, and a failure *after* the `~commit` pattern backtracks only to choice points *before* the `~commit` pattern, never one *within* it.

- A cut (`~!`) within a `~commit` pattern only eliminates choice-points created within the `~commit` pattern. In this sense, it acts just like `~delimit-cut`.

`(~delimit-cut S-pattern)`

The `~delimit-cut` pattern form affects backtracking in the following way:

- A cut (`~!`) within a `~delimit-cut` pattern only eliminates choice-points created within the `~delimit-cut` pattern.

`(~post S-pattern)`

Marks failures within the subpattern as occurring in a "post-order check"; that is, they are considered to have made greater progress than a normal failure.

`A-pattern`

An action pattern is considered a single-term pattern when there is no ambiguity; it matches any term.

1.5.2 Head Patterns

A *head pattern* (abbreviated `H-pattern`) is a pattern that describes some number of terms that occur at the head of some list (possibly an improper list). A head pattern's usefulness comes from being able to match heads of different lengths, such as optional forms like keyword arguments.

A *proper head pattern* is a head pattern that is not a single-term pattern.

Here are the variants of head pattern:

`pvar-id:splicing-syntax-class-id`

Equivalent to `(~var pvar-id splicing-syntax-class-id)`.

`(~var pvar-id splicing-syntax-class-use maybe-role)`

```
splicing-syntax-class-use = splicing-syntax-class-id
                          | (splicing-syntax-class-id arg ...)

               maybe-role =
                          | #:role role-expr

  role-expr : (or/c string? #f)
```

Pattern variable annotated with a splicing syntax class. Similar to a normal annotated pattern variable, except matches a head pattern.

```
(~seq . L-pattern)
```

Matches a sequence of terms whose elements, if put in a list, would match *L-pattern*.

Example:

```
> (syntax-parse #'(1 2 3 4)
    [((~seq 1 2 3) 4) 'ok])
'ok
```

See also the section on ellipsis-head patterns for more interesting examples of `~seq`.

```
(~and H-pattern ...)
```

Like the single-term pattern version, `~and`[s], but matches a sequence of terms instead.

Example:

```
> (syntax-parse #'(#:a 1 #:b 2 3 4 5)
    [((~and (~seq (~seq k:keyword e:expr) ...)
            (~seq keyword-stuff ...))
      positional-stuff ...)
     (syntax->datum #'((k ...) (e ...) (keyword-
stuff ...)))])
'((#:a #:b) (1 2) (#:a 1 #:b 2))
```

The head pattern variant of `~and` requires that all of the subpatterns be proper head patterns (not single-term patterns). This is to prevent typos like the following, a variant of the previous example with the second `~seq` omitted:

Examples:

```
> (syntax-parse #'(#:a 1 #:b 2 3 4 5)
    [((~and (~seq (~seq k:keyword e:expr) ...)
            (keyword-stuff ...))
      positional-stuff ...)
     (syntax->datum #'((k ...) (e ...) (keyword-
stuff ...)))])
syntax-parse: single-term pattern not allowed after head
pattern
  at: (keyword-stuff ...)
  in: (syntax-parse (syntax (#:a 1 #:b 2 3 4 5)) (((~and
(~seq (~seq k:keyword e:expr) ...) (keyword-stuff ...))
positional-stuff ...) (syntax->datum (syntax ((k ...) (e
...) (keyword-stuff ...))))))
; If the example above were allowed, it would be equiva-
lent to this:
> (syntax-parse #'(#:a 1 #:b 2 3 4 5)
    [((~and (~seq (~seq k:keyword e:expr) ...)
            (~seq (keyword-stuff ...)))
      positional-stuff ...)
     (syntax->datum #'((k ...) (e ...) (keyword-
stuff ...)))])
?: bad syntax
  in: (#:a 1 #:b 2 3 4 5)
```

```
(~or* H-pattern ...)
```

Like the single-term pattern version, ~or*[S], but matches a sequence of terms instead.

Examples:

```
> (syntax-parse #'(m #:foo 2 a b c)
    [(_ (~or* (~seq #:foo x) (~seq)) y:id ...)
     (attribute x)])
#<syntax:eval:55:0 2>
> (syntax-parse #'(m a b c)
    [(_ (~or* (~seq #:foo x) (~seq)) y:id ...)
     (attribute x)])
#f
```

```
(~optional H-pattern maybe-optional-option)

maybe-optional-option =
                      | #:defaults ([attr-arity-decl expr] ...)

      attr-arity-decl = attr-id
                      | (attr-id depth)
```

Matches either the given head subpattern or an empty sequence of terms. If the #:defaults option is given, the subsequent attribute bindings are used if the subpattern does not match. The default attributes must be a subset of the subpattern's attributes.

Examples:

```
> (syntax-parse #'(m #:foo 2 a b c)
    [(_ (~optional (~seq #:foo x) #:defaults ([x #'#f])) y:id ...)
     (attribute x)])
#<syntax:eval:57:0 2>
> (syntax-parse #'(m a b c)
    [(_ (~optional (~seq #:foo x) #:defaults ([x #'#f])) y:id ...)
     (attribute x)])
#<syntax:eval:58:0 #f>
> (syntax-parse #'(m a b c)
    [(_ (~optional (~seq #:foo x)) y:id ...)
     (attribute x)])
#f
> (syntax-parse #'(m #:syms a b c)
    [(_ (~optional (~seq #:nums n:nat ...) #:defaults ([(n 1) null]))
        (~optional (~seq #:syms s:id ...) #:defaults ([(s 1) null])))
     #'((n ...) (s ...))])
#<syntax:eval:60:0 (() (a b c))>
```

```
(~describe expr H-pattern)
```

Like the single-term pattern version, ~describe^s, but matches a head pattern instead.

```
(~commit H-pattern)
```

Like the single-term pattern version, ~commit^s, but matches a head pattern instead.

```
(~delimit-cut H-pattern)
```

Like the single-term pattern version, ~delimit-cut^s, but matches a head pattern instead.

```
(~post H-pattern)
```

Like the single-term pattern version, ~post^s, but matches a head pattern instead.

```
(~peek H-pattern)
```

Matches the *H-pattern* but then resets the matching position, so the `~peek` pattern consumes no input. Used to look ahead in a sequence.

Examples:

```
> (define-splicing-syntax-class nf-id ; non-final id
    (pattern (~seq x:id (~peek another:id))))
> (syntax-parse #'(a b c 1 2 3)
    [(n:nf-id ... rest ...)
     (printf "nf-ids are ~s\n" (syntax-
>datum #'(n.x ...)))
     (printf "rest is ~s\n" (syntax-
>datum #'(rest ...)))])
nf-ids are (a b)
rest is (c 1 2 3)
```

```
(~peek-not H-pattern)
```

Like `~peek`, but succeeds if the subpattern fails and fails if the subpattern succeeds. On success, the `~peek-not` resets the matching position, so the pattern consumes no input. Used to look ahead in a sequence. None of the subpattern's attributes are bound outside of the `~peek-not`-pattern.

Examples:

```
> (define-splicing-syntax-class final ; final term
    (pattern (~seq x (~peek-not _))))
> (syntax-parse #'(a b c)
    [((~alt f:final other) ...)
     (printf "finals are ~s\n" (syntax-
>datum #'(f.x ...)))
     (printf "others are ~s\n" (syntax-
>datum #'(other ...)))])
finals are (c)
others are (a b)
```

```
S-pattern
```

Matches a sequence of one element, which must be a term matching *S-pattern*.

1.5.3 Ellipsis-head Patterns

An *ellipsis-head pattern* (abbreviated *EH-pattern*) is pattern that describes some number of terms, like a head pattern, but also places constraints on the number of times it occurs in a repetition. They are useful for matching, for example, keyword arguments where the keywords may come in any order. Multiple alternatives are grouped together via `~alt`.

Examples:

```
> (define parser1
    (syntax-parser
     [((~alt (~once (~seq #:a x) #:name "#:a keyword")
             (~optional (~seq #:b y) #:name "#:b keyword")
             (~seq #:c z)) ...)
      'ok]))
> (parser1 #'(#:a 1))
'ok
> (parser1 #'(#:b 2 #:c 3 #:c 25 #:a 'hi))
'ok
> (parser1 #'(#:a 1 #:a 2))
?: too many occurrences of #:a keyword
  at: ()
  within: (#:a 1 #:a 2)
  in: (#:a 1 #:a 2)
```

The pattern requires exactly one occurrence of the `#:a` keyword and argument, at most one occurrence of the `#:b` keyword and argument, and any number of `#:c` keywords and arguments. The “pieces” can occur in any order.

Here are the variants of ellipsis-head pattern:

```
(~alt EH-pattern ...)
```

Matches if any of the inner *EH-pattern* alternatives match.

```
(~once H-pattern once-option ...)

once-option = #:name name-expr
            | #:too-few too-few-message-expr
            | #:too-many too-many-message-expr

  name-expr : (or/c string? #f)
  too-few-message-expr : (or/c string? #f)
  too-many-message-expr : (or/c string? #f)
```

Matches if the inner *H-pattern* matches. This pattern must be matched exactly once in the match of the entire repetition sequence.

If the pattern is not matched in the repetition sequence, then the ellipsis pattern fails with the message either *too-few-message-expr* or `"missing required occurrence of name-expr"`.

If the pattern is chosen more than once in the repetition sequence, then the ellipsis pattern fails with the message either *too-many-message-expr* or `"too many occurrences of name-expr"`.

```
(~optional H-pattern optional-option ...)

optional-option = #:name name-expr
                | #:too-many too-many-message-expr
                | #:defaults ([attr-id expr] ...)

  name-expr : (or/c string? #f)
  too-many-message-expr : (or/c string? #f)
```

Matches if the inner *H-pattern* matches. This pattern may be used at most once in the match of the entire repetition.

If the pattern is matched more than once in the repetition sequence, then the ellipsis pattern fails with the message either *too-many-message-expr* or `"too many occurrences of name-expr"`.

If the `#:defaults` option is given, the following attribute bindings are used if the subpattern does not match at all in the sequence. The default attributes must be a subset of the subpattern's attributes.

```
(~between H-pattern min-number max-number between-option ...)

between-option = #:name name-expr
               | #:too-few too-few-message-expr
               | #:too-many too-many-message-expr

  name-expr : (or/c string? #f)
  too-few-message-expr : (or/c string? #f)
  too-many-message-expr : (or/c string? #f)
```

Matches if the inner *H-pattern* matches. This pattern must be matched at least *min-number* and at most *max-number* times in the entire repetition.

If the pattern is matched too few times, then the ellipsis pattern fails with the message either *too-few-message-expr* or `"too few occurrences of name-expr"`, when *name-expr* is provided.

If the pattern is chosen too many times, then the ellipsis pattern fails with the message either *too-many-message-expr* or "too many occurrences of *name-expr*", when *name-expr* is provided.

1.5.4 Action Patterns

An *action pattern* (abbreviated *A-pattern*) does not describe any syntax; rather, it has an effect such as the binding of attributes or the modification of the matching process.

~!

The *cut* operator, written `~!`, eliminates backtracking choice points and commits parsing to the current branch of the pattern it is exploring.

Common opportunities for cut-patterns come from recognizing special forms based on keywords. Consider the following expression:

```
> (syntax-parse #'(define-values a 123)
    #:literals (define-values define-syntaxes)
    [(define-values (x:id ...) e) 'define-values]
    [(define-syntaxes (x:id ...) e) 'define-syntaxes]
    [e 'expression])
'expression
```

Given the ill-formed term `(define-values a 123)`, `syntax-parse` tries the first clause, fails to match `a` against the pattern `(x:id ...)`, and then backtracks to the second clause and ultimately the third clause, producing the value `'expression`. But the term is not an expression; it is an ill-formed use of `define-values`. The proper way to write the `syntax-parse` expression follows:

```
> (syntax-parse #'(define-values a 123)
    #:literals (define-values define-syntaxes)
    [(define-values ~! (x:id ...) e) 'define-values]
    [(define-syntaxes ~! (x:id ...) e) 'define-syntaxes]
    [e 'expression])
define-values: bad syntax
  in: (define-values a 123)
```

Now, given the same term, `syntax-parse` tries the first clause, and since the keyword `define-values` matches, the cut-pattern commits to the current pattern, eliminating the choice points for the second and third clauses. So when the clause fails to match, the `syntax-parse` expression raises an error.

The effect of a `~!` pattern is delimited by the nearest enclosing `~delimit-cut` or `~commit` pattern. If there is no enclosing `~describe` pattern but the cut occurs within a syntax class definition, then only choice points within the syntax class definition are discarded. A `~!` pattern is not allowed within a `~not` pattern unless there is an intervening `~delimit-cut` or `~commit` pattern.

```
(~bind [attr-arity-decl expr] ...)

attr-arity-decl = attr-name-id
                | (attr-name-id depth)
```

Evaluates the `expr`s and binds them to the given `attr-id`s as attributes.

```
(~fail maybe-fail-condition maybe-message-expr)

maybe-fail-condition =
                     | #:when condition-expr
                     | #:unless condition-expr

  maybe-message-expr =
                     | message-expr

  message-expr : (or/c string? #f)
```

If the condition is absent, or if the `#:when` condition evaluates to a true value, or if the `#:unless` condition evaluates to `#f`, then the pattern fails with the given message. If the message is omitted, the default value `#f` is used, representing “no message.”

Fail patterns can be used together with cut patterns to recognize specific ill-formed terms and address them with custom failure messages.

```
(~parse S-pattern stx-expr)
```

Evaluates `stx-expr` and matches it against `S-pattern`. If `stx-expr` does not produce a syntax object, the value is implicitly converted to a syntax object, unless the conversion would produce 3D syntax, in which case an exception is raised instead.

```
(~and A-pattern ...+)
```

Performs the actions of each `A-pattern`.

```
(~do defn-or-expr ...)
```

Takes a sequence of definitions and expressions, which may be intermixed, and evaluates them in the scope of all previous attribute bindings. The names bound by the definitions are in scope in the expressions of subsequent patterns and clauses.

There is currently no way to bind attributes using a `~do` pattern. It is an error to shadow an attribute binding with a definition in a `~do` block.

Example:

```
> (syntax-parse #'(1 2 3)
    [(a b (~do (printf "a was ~s\n" #'a)) c:id) 'ok])
a was #<syntax:eval:71:0 1>
?: expected identifier
  at: 3
  in: (1 2 3)
```

```
(~undo defn-or-expr ...)
```

Has no effect when initially matched, but if backtracking returns to a point *before* the `~undo` pattern, the *`defn-or-expr`*s are executed. They are evaluated in the scope of all previous attribute bindings.

Use `~do` paired with `~undo` to perform side effects and then unwind them if the enclosing pattern is later discarded.

Examples:

```
> (define total 0)
> (define-syntax-class nat/add
    (pattern (~and n:nat
                   (~do (printf "adding ~s\n" (syntax-
e #'n))
                        (set! total (+ total (syntax-
e #'n))))
                   (~undo (printf "subtracting
~s\n" (syntax-e #'n))
                          (set! total (- total (syntax-
e #'n)))))))
> (syntax-parse #'(1 2 3)
    [(x:nat/add ...) 'ok])
adding 1
adding 2
adding 3
```

```
'ok
> total
6
> (set! total 0)
> (syntax-parse #'(1 2 3 bad)
    [(x:nat/add ...) 'ok]
    [_ 'something-else])
adding 1
adding 2
adding 3
subtracting 3
subtracting 2
subtracting 1
'something-else
> total
0
```

```
(~post A-pattern)
```

Like the single-term pattern version, `~post`S, but contains only action patterns.

1.5.5 Pattern Expanders

The grammar of syntax patterns is extensible through the use of *pattern expanders*, which allow the definition of new pattern forms by rewriting them into existing pattern forms.

As a convention to avoid ambiguity, pattern expander names normally begin with a ~ character.

```
(pattern-expander proc) → pattern-expander?
  proc : (-> syntax? syntax?)
```

Returns a pattern expander that uses *proc* to transform the pattern.

Example:

```
> (define-syntax ~maybe
    (pattern-expander
     (lambda (stx)
       (syntax-case stx ()
         [(~maybe pat ...)
          #'(~optional (~seq pat ...))]))))
```

```
prop:pattern-expander
 : (struct-type-property/c (-> pattern-expander? (-> syntax? syntax?)))
```

A structure type property to identify structure types that act as pattern expanders like the ones created by pattern-expander.

```
(begin-for-syntax
  (struct thing (proc pattern-expander)
    #:property prop:procedure (struct-field-index proc)
    #:property prop:pattern-expander (λ (this) (thing-pattern-
expander this))))
(define-syntax ~maybe
  (thing
   (lambda (stx) .... macro behavior ....)
   (lambda (stx) .... pattern-expander behavior ....)))
```

```
(pattern-expander? v) → boolean?
  v : any/c
```

Returns #t if *v* is a pattern expander, #f otherwise.

```
(syntax-local-syntax-parse-pattern-introduce stx) → syntax?
  stx : 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.

1.6 Defining Simple Macros

```
(require syntax/parse/define)        package: base
```

The syntax/parse/define library provides for-syntax all of syntax/parse, as well as providing some new forms.

```
(define-syntax-parse-rule (macro-id . pattern) pattern-directive ...
  template)
```

Defines a macro named *macro-id*; equivalent to the following:

```
(define-syntax (macro-id stx)
  (syntax-parse stx
    #:track-literals
    [((~var macro-id id) . pattern) pattern-
directive ... #'template]))
```

Examples:

```
> (define-syntax-parse-rule (fn x:id rhs:expr) (lambda (x) rhs))
> ((fn x x) 17)
17
> (fn 1 2)
fn: expected identifier
  at: 1
  in: (fn 1 2)
> (define-syntax-parse-rule (fn2 x y rhs)
    #:declare x id
    #:declare y id
    #:declare rhs expr
    (lambda (x y) rhs))
> ((fn2 a b (+ a b)) 3 4)
7
> (fn2 a #:b 'c)
fn2: expected identifier
  at: #:b
  in: (fn2 a #:b (quote c))
```

Added in version 7.9.0.22 of package `base`.

```
(define-syntax-parser macro-id parse-option ... clause ...+)
```

Defines a macro named *macro-id*; equivalent to:

```
(define-syntax macro-id
  (syntax-parser parse-option ... clause ...))
```

Examples:

```
> (define-syntax-parser fn3
    [(fn3 x:id rhs:expr)
     #'(lambda (x) rhs)]
    [(fn3 x:id y:id rhs:expr)
     #'(lambda (x y) rhs)])
> ((fn3 x x) 17)
17
> ((fn3 a b (+ a b)) 3 4)
7
> (fn3 1 2)
fn3: expected identifier
  at: 1
  in: (fn3 1 2)
> (fn3 a #:b 'c)
fn3: expected expression or expected identifier
```

at: #:b
in: (fn3 a #:b (quote c))

```
(define-simple-macro (macro-id . pattern) pattern-directive ...
  template)
```

NOTE: This macro is deprecated; use `define-syntax-parse-rule`, instead.

Re-exports `define-syntax-parse-rule` for backward-compatibility.

Changed in version 6.12.0.3 of package `base`: Changed pattern head to `(~var macro-id id)` from `macro-id`, allowing tilde-prefixed identifiers or identifiers containing colons to be used as `macro-id` without producing a syntax error.
Changed in version 6.90.0.29: Changed to always use the `#:track-literals` `syntax-parse` option.

1.7 Literal Sets and Conventions

Sometimes the same literals are recognized in a number of different places. The most common example is the literals for fully expanded programs, which are used in many analysis and transformation tools. Specifying literals individually is burdensome and error-prone. As a remedy, `syntax/parse` offers *literal sets*. A literal set is defined via `define-literal-set` and used via the `#:literal-set` option of `syntax-parse`.

```
(define-literal-set id maybe-phase maybe-imports maybe-datum-literals
  (literal ...))

               literal = literal-id
                       | (pattern-id literal-id)

           maybe-phase =
                       | #:for-template
                       | #:for-syntax
                       | #:for-label
                       | #:phase phase-level

  maybe-datum-literals =
                       | #:datum-literals (datum-literal ...)

         maybe-imports =
                       | #:literal-sets (imported-litset-id ...)
```

Defines `id` as a literal set. Each `literal` can have a separate `pattern-id` and `literal-id`. The `pattern-id` determines what identifiers in the pattern are treated as literals. The *literal-id* determines what identifiers the literal matches. If the `#:literal-sets` option is present, the contents of the given *imported-litset-id*s are included.

Examples:

```
> (define-literal-set def-litset
    (define-values define-syntaxes))
> (syntax-parse #'(define-syntaxes (x) 12)
   #:literal-sets (def-litset)
   [(define-values (x:id ...) e:expr) 'v]
   [(define-syntaxes (x:id ...) e:expr) 's])
's
```

The literals in a literal set always refer to the bindings at phase *phase-level* *relative to the enclosing module*. If the `#:for-template` option is given, *phase-level* is `-1`; `#:for-syntax` means `1`, and `#:for-label` means `#f`. If no phase keyword option is given, then *phase-level* is `0`.

For example:

```
> (module common racket/base
    (define x 'something)
    (provide x))
> (module lits racket/base
    (require syntax/parse 'common)
    (define-literal-set common-lits (x))
    (provide common-lits))
```

In the literal set `common-lits`, the literal `x` always recognizes identifiers bound to the variable `x` defined in module `'common`.

The following module defines an equivalent literal set, but imports the `'common` module for-template instead:

```
> (module lits racket/base
    (require syntax/parse (for-template 'common))
    (define-literal-set common-lits #:for-template (x))
    (provide common-lits))
```

When a literal set is *used* with the `#:phase phase-expr` option, the literals' fixed bindings are compared against the binding of the input literal at the specified phase. Continuing the example:

```
> (require syntax/parse 'lits (for-syntax 'common))
> (syntax-parse #'x #:literal-sets ([common-lits #:phase 1])
    [x 'yes]
    [_ 'no])
```

```
'yes
```

The occurrence of `x` in the pattern matches any identifier whose binding at phase 1 is the `x` from module `'common`.

```
(literal-set->predicate litset-id)
```

Given the name of a literal set, produces a predicate that recognizes identifiers in the literal set. The predicate takes one required argument, an identifier *id*, and one optional argument, the phase *phase* at which to examine the binding of *id*; the *phase* argument defaults to `(syntax-local-phase-level)`.

Examples:

```
> (define kernel? (literal-set->predicate kernel-literals))
> (kernel? #'lambda)
#f
> (kernel? #'#%plain-lambda)
#t
> (kernel? #'define-values)
#t
> (kernel? #'define-values 4)
#f
```

```
(define-conventions name-id convention-rule ...)

convention-rule = (name-pattern syntax-class)

   name-pattern = exact-id
                | name-rx

   syntax-class = syntax-class-id
                | (syntax-class-id expr ...)
```

Defines *conventions* that supply default syntax classes for pattern variables. A pattern variable that has no explicit syntax class is checked against each *name-pattern*, and the first one that matches determines the syntax class for the pattern. If no *name-pattern* matches, then the pattern variable has no syntax class.

Examples:

```
> (define-conventions xyz-as-ids
    [x id] [y id] [z id])
> (syntax-parse #'(a b c 1 2 3)
    #:conventions (xyz-as-ids)
    [(x ... n ...) (syntax->datum #'(x ...))])
```

```
'(a b c)
> (define-conventions xn-prefixes
    [#rx"^x" id]
    [#rx"^n" nat])
> (syntax-parse #'(a b c 1 2 3)
    #:conventions (xn-prefixes)
    [(x0 x ... n0 n ...)
     (syntax->datum #'(x0 (x ...) n0 (n ...)))])
'(a (b c) 1 (2 3))
```

Local conventions, introduced with the `#:local-conventions` keyword argument of `syntax-parse` and syntax class definitions, may refer to local bindings:

Examples:

```
> (define-syntax-class (nat> bound)
    (pattern n:nat
             #:fail-unless (> (syntax-e #'n) bound)
                           (format "expected number > ~s" bound)))
> (define-syntax-class (natlist> bound)
    #:local-conventions ([N (nat> bound)])
    (pattern (N ...)))
> (define (parse-natlist> bound x)
    (syntax-parse x
      #:local-conventions ([NS (natlist> bound)])
      [NS 'ok]))
> (parse-natlist> 0 #'(1 2 3))
'ok
> (parse-natlist> 5 #'(8 6 4 2))
?: expected number > 5
  at: 4
  in: (8 6 4 2)
  parsing context:
   while parsing nat>
    term: 4
    location: eval:21:0
   while parsing natlist>
    term: (8 6 4 2)
    location: eval:21:0
```

1.8 Library Syntax Classes and Literal Sets

1.8.1 Syntax Classes

```
expr
```

Matches anything except a keyword literal (to distinguish expressions from the start of a keyword argument sequence). The term is not otherwise inspected, since it is not feasible to check if it is actually a valid expression.

```
identifier
boolean
char
keyword
number
integer
exact-integer
exact-nonnegative-integer
exact-positive-integer
regexp
byte-regexp
```

Match syntax satisfying the corresponding predicates.

```
string
bytes
```

As special cases, Racket's `string` and `bytes` bindings are also interpreted as syntax classes that recognize literal strings and bytes, respectively.

Added in version 6.9.0.4 of package `base`.

```
id
```

Alias for `identifier`.

```
nat
```

Alias for `exact-nonnegative-integer`.

```
str
```

Alias for `string`.

```
character
```

Alias for char.

```
(static predicate description) → syntax class
  predicate : (-> any/c any/c)
  description : (or/c string? #f)
```

The static syntax class matches an identifier that is bound in the syntactic environment to static information (see syntax-local-value) satisfying the given *predicate*. If the term does not match, the *description* argument is used to describe the expected syntax.

When used outside of the dynamic extent of a macro transformer (see syntax-transforming?), matching fails.

The attribute *value* contains the value the name is bound to.

If matching succeeds, static additionally adds the matched identifier to the current syntax-parse state under the key 'literals using syntax-parse-state-cons!, in the same way as identifiers matched using #:literals or ~literal.

Changed in version 6.90.0.29 of package base: Changed to add matched identifiers to the syntax-parse state under the key 'literals.

```
(expr/c  contract-expr
        [#:arg? arg?
         #:positive pos-blame
         #:negative neg-blame
         #:name expr-name
         #:macro macro-name
         #:context context
         #:phase phase])       → syntax class
  contract-expr : syntax?
  arg? : any/c = #t
  pos-blame : (or/c syntax? string? module-path-index? 'from-macro 'use-site 'unknown)
            = 'from-macro
  neg-blame : (or/c syntax? string? module-path-index? 'from-macro 'use-site 'unknown)
            = 'use-site
  expr-name : (or/c identifier? string? symbol?) = #f
  macro-name : (or/c identifier? string? symbol?) = #f
  context : (or/c syntax? #f) = determined automatically
  phase : exact-integer? = (syntax-local-phase-level)
```

Accepts an expression (expr) and computes an attribute c that represents the expression wrapped with the contract represented by *contract-expr*. Note that *contract-expr* is potentially evaluated each time the code generated by the macro is run; for the best performance, *contract-expr* should be a variable reference.

The positive blame represents the obligations of the macro imposing the contract—the ultimate user of `expr/c`. The contract's negative blame represents the obligations of the expression being wrapped. By default, the positive blame is inferred from the definition site of the macro (itself inferred from the *`context`* argument), and the negative blame is taken as the module currently being expanded, but both blame locations can be overridden. When *`arg?`* is `#t`, the term being matched is interpreted as an argument (that is, coming from the negative party); when *`arg?`* is `#f`, the term being matched is interpreted as a result of the macro (that is, coming from the positive party).

The *`pos-blame`* and *`neg-blame`* arguments are turned into blame locations as follows:

- If the argument is a string, it is used directly as the blame label.
- If the argument is syntax, its source location is used to produce the blame label.
- If the argument is a module path index, its resolved module path is used.
- If the argument is `'from-macro`, the macro is inferred from either the *`macro-name`* argument (if *`macro-name`* is an identifier) or the *`context`* argument, and the module where it is *defined* is used as the blame location. If neither an identifier *`macro-name`* nor a *`context`* argument is given, the location is `"unknown"`.
- If the argument is `'use-site`, the module being expanded is used.
- If the argument is `'unknown`, the blame label is `"unknown"`.

The *`macro-name`* argument is used to determine the macro's binding, if it is an identifier. If *`expr-name`* is given, *`macro-name`* is also included in the contract error message. If *`macro-name`* is omitted or `#f`, but *`context`* is a syntax object, then *`macro-name`* is determined from *`context`*.

If *`expr-name`* is not `#f`, it is used in the contract's error message to describe the expression the contract is applied to.

The *`context`* argument is used, when necessary, to infer the macro name for the negative blame party and the contract error message. The *`context`* should be either an identifier or a syntax pair with an identifier in operator position; in either case, that identifier is taken as the macro ultimately requesting the contract wrapping.

The *`phase`* argument must indicate the phase level at which the contracted expression will be evaluated. Using the contracted expression at a different phase level will cause a syntax error because it will contain introduced references bound in the wrong phase. In particular:

- Use the default value, `(syntax-local-phase-level)`, when the contracted expression will be evaluated at the same phase as the form currently being expanded. This is usually the case.

- Use (add1 (syntax-local-phase-level)) in cases such as the following: the contracted expression will be placed inside a begin-for-syntax form, used in the right-hand side of a define-syntax or let-syntax form, or passed to syntax-local-bind-syntaxes or syntax-local-eval.

Any phase level other than #f (the label phase level) is allowed, but phases other than (syntax-local-phase-level) and (add1 (syntax-local-phase-level)) may only be used when in the dynamic extent of a syntax transformer or while a module is being visited (see syntax-transforming?), otherwise exn:fail:contract? is raised.

See §1.2.6 "Contracts on Macro Sub-expressions" for examples.

Important: Make sure when using expr/c to use the c attribute. The expr/c syntax class does not change how pattern variables are bound; it only computes an attribute that represents the checked expression.

Changed in version 7.2.0.3 of package base: Added the #:arg? keyword argument and changed the default values and interpretation of the #:positive and #:negative arguments.
Changed in version 7.3.0.3: Added the #:phase keyword argument.

1.8.2 Literal Sets

```
kernel-literals
```

Literal set containing the identifiers for fully-expanded code (§1.2.3.1 "Fully Expanded Programs"). The set contains all of the forms listed by kernel-form-identifier-list, plus module, #%plain-module-begin, #%require, and #%provide.

Note that the literal-set uses the names #%plain-lambda and #%plain-app, not lambda and #%app.

1.8.3 Function Headers

```
(require syntax/parse/lib/function-header)
```
package: base

```
function-header
```

Matches a name and formals found in function header. It also supports the curried function shorthand.

```
name : syntax?
```

The name part in the function header.

`params : syntax?`

The list of parameters in the function header.

`formal`

Matches a single formal that can be used in a function header.

`name : syntax?`

The name part in the formal.

`kw : (or/c syntax? #f)`

The keyword part in the formal, if it exists.

`default : (or/c syntax? #f)`

The default expression part in the formal, if it exists.

`formals`

Matches a list of formals that would be used in a function header.

`params : syntax?`

The list of parameters in the formals.

```
> (syntax-parse #'(define ((foo x) y) 1)
    [(_ header:function-header body ...+) #'(header header.name header.params)])
#<syntax:eval:2:0 (((foo x) y) foo (x y))>
```

```
> (syntax-parse #'(lambda xs xs)
    [(_ fmls:formals body ...+) #'(fmls fmls.params)])
#<syntax:eval:3:0 (xs (xs))>
> (syntax-parse #'(lambda (x y #:kw [kw 42] . xs) xs)
    [(_ fmls:formals body ...+) #'(fmls fmls.params)])
#<syntax:eval:4:0 ((x y #:kw (kw 42) . xs) (x y kw xs))>
> (syntax-parse #'(lambda (x) x)
    [(_ (fml:formal) body ...+) #'(fml
                                   fml.name
                                   (~? fml.kw #f)
                                   (~? fml.default #f))])
#<syntax:eval:5:0 ((x) x #f #f)>
> (syntax-parse #'(lambda (#:kw [kw 42]) kw)
    [(_ (fml:formal) body ...+) #'(fml fml.name fml.kw fml.default)])
#<syntax:eval:6:0 ((#:kw (kw 42)) kw #:kw 42)>
```

```
formals-no-rest
```

Like `formals` but without dotted-tail identifier.

```
params : syntax?
```

The list of parameters.

1.9 Unwindable State

```
(syntax-parse-state-ref key [default]) → any/c
  key : any/c
  default : default/c = (lambda () (error ....))
(syntax-parse-state-set! key value) → void?
  key : any/c
  value : any/c
(syntax-parse-state-update! key
                            update
                           [default]) → void?
  key : any/c
  update : (-> any/c any/c)
  default : default/c = (lambda () (error ....))
(syntax-parse-state-cons! key value [default]) → void?
  key : any/c
  value : any/c
  default : default/c = null
```

Get or update the current syntax-parse state. Updates to the state are unwound when syntax-parse backtracks. Keys are compared using eq?.

The state can be updated only within ~do patterns (or #:do blocks). In addition, syntax-parse automatically adds identifiers that match literals (from ~literal patterns and literals declared with #:literals, but not from ~datum or #:datum-literals) under the key 'literals.

Examples:

```
> (define-syntax-class cond-clause
    #:literals (=> else)
    (pattern [test:expr => ~! answer:expr ...])
    (pattern [else answer:expr ...])
    (pattern [test:expr answer:expr ...]))
> (syntax-parse #'(cond [A => B] [else C])
    [(_ c:cond-clause ...) (syntax-parse-state-
ref 'literals null)])
'(#<syntax:eval:2:0 else> #<syntax:eval:2:0 =>>)
```

Added in version 6.11.0.4 of package base.

```
(syntax-parse-track-literals stx
                            [#:introduce? introduce?]) → syntax?
  stx : syntax?
  introduce? : any/c = #t
```

Add a 'disappeared-use syntax property to *stx* containing the information stored in the current syntax-parse state under the key 'literals. If *stx* already has a 'disappeared-use property, the added information is consed onto the property's current value.

Due to the way syntax-parse automatically adds identifiers that match literals to the state under the key 'literals, as described in the documentation for syntax-parse-state-ref, syntax-parse-track-literals can be used to automatically add any identifiers used as literals to the 'disappeared-use property.

If syntax-parse-track-literals is called within the dynamic extent of a syntax transformer (see syntax-transforming?), *introduce?* is not #f, and the value in the current syntax-parse state under the key 'literals is a list, then syntax-local-introduce is applied to any identifiers in the list before they are added to *stx*'s 'disappeared-use property.

Most of the time, it is unnecessary to call this function directly. Instead, the #:track-literals option should be provided to syntax-parse, which will automatically call syntax-parse-track-literals on syntax-valued results.

Examples:

```
> (define-syntax-class cond-clause
    #:literals (=> else)
    (pattern [test:expr => ~! answer:expr ...])
    (pattern [else answer:expr ...])
    (pattern [test:expr answer:expr ...]))
> (syntax-property
   (syntax-parse #'(cond [A => B] [else C])
     [(_ c:cond-clause ...) (syntax-parse-track-literals #'#f)])
   'disappeared-use)
'(#<syntax:eval:4:0 else> #<syntax:eval:4:0 =>>)
```

Added in version 6.90.0.29 of package `base`.

1.10 Configuring Error Reporting

```
(require syntax/parse/report-config)    package: base
```

Added in version 8.9.0.5 of package `base`.

```
(current-report-configuration) → report-configuration?
(current-report-configuration config) → void?
  config : report-configuration?
```

A parameter that determines parts error messages that are generated by `syntax-parse` for failed matches. When `syntax-parse` needs to report that a particular datum or literal identifier was expected, it consults the configuration in this parameter. This parameter is cross-phase persistent, which means that the parameter and its value are shared across phases.

A configuration is a hash table with the following keys:

- `'datum-to-what` — a procedure of one argument used to get a noun describing an expected datum, which appears in a pattern either with `~datum`, as "self-quoting," or so on. The procedure's argument is the datum value. The result must be either a string or a list containing two strings; if two strings are provided, the first is used when a singular noun is needed, and the second is used as a plural noun.

  The default configuration returns `'("literal symbol" "literal symbols")` for a symbol and `'("literal" "literals")` for any other datum value.

- `'datum-to-string` — a procedure of one argument, used to convert the datum value to a string that is included in the error message. The procedure's argument is the datum value, and the result must be a string.

  The default configuration formats a symbol value using `(format "`~s'" v)` any other datum value using `(format "~s" v)`.

- `'literal-to-what` — a procedure of one argument used to get a noun describing an expected literal identifier, which appears in a pattern with `~literal`, as declared with `#:literals`, or so on. The procedure's argument is an identifier when available, or a symbol when only simplified information has been preserved. The result must be either a string or a list containing two strings, like the result for a `'datum-to-what` procedure.

  The default configuration returns `'("identifier" "identifiers")`.

- `'literal-to-string` — a procedure of one argument, used to convert a literal identifier or symbol to a string that is included in the error message.

  The default configuration formats a symbol value using `(format "`~s'" v)`, and it formats an identifier the same after extracting its symbol with `syntax-e`.

Changed in version 8.15.0.4 of package `base`: Changed parameter to cross-phase persistent.

```
(report-configuration? v) → boolean?
  v : any/c
```

Checks whether `v` is an immutable hash table that maps each of the keys `'datum-to-what`, `'datum-to-string` `'identifier-to-what` and `'identifier-to-string` to a procedure that accepts one argument.

1.11 Debugging and Inspection Tools

```
(require syntax/parse/debug)        package: base
```

The following special forms are for debugging syntax classes.

```
(syntax-class-attributes syntax-class-id)
```

Returns a list of the syntax class's attributes. Each attribute entry consists of the attribute's name and ellipsis depth.

```
(syntax-class-arity syntax-class-id)
(syntax-class-keywords syntax-class-id)
```

Returns the syntax class's arity and keywords, respectively. Compare with `procedure-arity` and `procedure-keywords`.

```
(syntax-class-parse syntax-class-id stx-expr arg ...)

  stx-expr : syntax?
```

Runs the parser for the syntax class (parameterized by the arg-exprs) on the syntax object produced by *stx-expr*. On success, the result is a list of vectors representing the attribute bindings of the syntax class. Each vector contains the attribute name, depth, and associated value. On failure, the result is some internal representation of the failure.

```
(debug-parse stx-expr S-pattern ...+)

  stx-expr : syntax?
```

Tries to match *stx-expr* against the *S-pattern*s. If matching succeeds, the symbol `'success` is returned. Otherwise, an S-expression describing the failure is returned.

The failure S-expression shows both the raw set of failures (unsorted) and the failures with maximal progress. The maximal failures are divided into equivalence classes based on their progress (progress is a partial order); that is, failures within an equivalence class have the same progress and, in principle, pinpoint the same term as the problematic term. Multiple equivalence classes only arise from `~parse` patterns (or equivalently, `#:with` clauses) that match computed terms or `~fail` (`#:fail-when`, etc) clauses that allow a computed term to be pinpointed.

```
(debug-syntax-parse!) → void?
```

Installs a `syntax-parse` reporting handler that prints debugging information to the current error port when a `syntax-parse` error occurs.

Added in version 6.5.0.3 of package `base`.

1.12 Experimental

The following facilities are experimental.

1.12.1 Contracts for Macro Sub-expressions

```
(require syntax/parse/experimental/contract)     package: base
```

This module is deprecated; it reprovides `expr/c` for backward compatibility.

1.12.2 Contracts for Syntax Classes

```
(require syntax/parse/experimental/provide)     package: base
```

```
(provide-syntax-class/contract
  [syntax-class-id syntax-class-contract] ...)

syntax-class-contract = (syntax-class/c (mandatory-arg ...))
                      | (syntax-class/c (mandatory-arg ...)
                                        (optional-arg ...))

                  arg = contract-expr
                      | keyword contract-expr

  contract-expr : contract?
```

Provides the syntax class (or splicing syntax class) *syntax-class-id* with the given contracts imposed on its formal parameters.

```
syntax-class/c
```

Keyword recognized by `provide-syntax-class/contract`.

1.12.3 Reflection

```
(require syntax/parse/experimental/reflect)      package: base
```

A syntax class can be reified into a run-time value, and a reified syntax class can be used in a pattern via the `~reflect` and `~splicing-reflect` pattern forms.

```
(reify-syntax-class syntax-class-id)
```

Reifies the syntax class named *syntax-class-id* as a run-time value. The same form also handles splicing syntax classes. Syntax classes with the `#:no-delimit-cut` option cannot be reified.

```
(reified-syntax-class? x) → boolean?
  x : any/c
(reified-splicing-syntax-class? x) → boolean?
  x : any/c
```

Returns `#t` if *x* is a reified (normal) syntax class or a reified splicing syntax class, respectively.

```
(reified-syntax-class-attributes r)
 → (listof (list/c symbol? exact-nonnegative-integer?))
  r : (or/c reified-syntax-class? reified-splicing-syntax-class?)
```

Returns the reified syntax class's attributes.

```
(reified-syntax-class-arity r) → procedure-arity?
  r : (or/c reified-syntax-class? reified-splicing-syntax-class?)
(reified-syntax-class-keywords r)
 → (listof keyword?) (listof keyword?)
  r : (or/c reified-syntax-class? reified-splicing-syntax-class?)
```

Returns the reified syntax class's arity and keywords, respectively. Compare with `procedure-arity` and `procedure-keywords`.

```
(reified-syntax-class-curry r
                            arg ...
                            #:<kw> kw-arg ...)
 → (or/c reified-syntax-class? reified-splicing-syntax-class?)
  r : (or/c reified-syntax-class? reified-splicing-syntax-class?)
  arg : any/c
  kw-arg : any/c
```

Partially applies the reified syntax class to the given arguments. If more arguments are given than the reified syntax class accepts, an error is raised.

```
 S-pattern = ....
           | (~reflect var-id (reified-expr arg-expr ...) maybe-attrs)

 H-pattern = ....
           | (~splicing-reflect var-id (reified-expr arg-expr ...)
                                maybe-attrs)
```

```
(~reflect var-id (reified-expr arg-expr ...) maybe-attrs)

maybe-attrs =
            | #:attributes (attr-arity-decl ...)
```

Like `~var`, except that the syntax class position is an expression evaluating to a reified syntax object, not a syntax class name, and the attributes bound by the reified syntax class (if any) must be specified explicitly.

```
(~splicing-reflect var-id (reified-expr arg-expr ...) maybe-attrs)
```

Like `~reflect` but for reified splicing syntax classes.

Examples:

```
> (define-syntax-class (nat> x)
    #:description (format "natural number greater than ~s" x)
    #:attributes (diff)
    (pattern n:nat
             #:when (> (syntax-e #'n) x)
             #:with diff (- (syntax-e #'n) x)))
> (define-syntax-class (nat/mult x)
    #:description (format "natural number multiple of ~s" x)
    #:attributes (quot)
    (pattern n:nat
             #:when (zero? (remainder (syntax-e #'n) x))
             #:with quot (quotient (syntax-e #'n) x)))
> (define r-nat> (reify-syntax-class nat>))
> (define r-nat/mult (reify-syntax-class nat/mult))
> (define (partition/r stx r n)
    (syntax-parse stx
      [((~alt (~reflect yes (r n)) no) ...)
       #'((yes ...) (no ...))]))
> (partition/r #'(1 2 3 4 5) r-nat> 3)
#<syntax:eval:5:0 ((4 5) (1 2 3))>
> (partition/r #'(1 2 3 4 5) r-nat/mult 2)
#<syntax:eval:5:0 ((2 4) (1 3 5))>
> (define (bad-attrs r)
    (syntax-parse #'6
      [(~reflect x (r 3) #:attributes (diff))
       #'x.diff]))
> (bad-attrs r-nat>)
#<syntax 3>
> (bad-attrs r-nat/mult)
```

reflect-syntax-class: reified syntax-class is missing declared attribute `diff'

1.12.4 Procedural Splicing Syntax Classes

```
(require syntax/parse/experimental/splicing)
```
package: base

```
(define-primitive-splicing-syntax-class (name-id param-id ...)
  #:description description-expr
  #:attributes (attr-arity-decl ...)
  parser-expr)

  parser-expr : (-> syntax?
                    (->* () ((or/c string? #f)) any)
                    (cons/c exact-positive-integer? list?))
```

Defines a splicing syntax via a procedural parser.

The parser procedure is given two arguments, the syntax to parse and a failure procedure. To signal a successful parse, the parser procedure returns a list of N+1 elements, where N is the number of attributes declared by the splicing syntax class. The first element is the size of the prefix consumed. The rest of the list contains the values of the attributes.

To indicate failure, the parser calls the failure procedure with an optional message argument.

1.12.5 Ellipsis-head Alternative Sets

```
(require syntax/parse/experimental/eh)        package: base
```

Unlike single-term patterns and head patterns, ellipsis-head patterns cannot be encapsulated by syntax classes, since they describe not only sets of terms but also repetition constraints.

This module provides *ellipsis-head alternative sets*, reusable encapsulations of ellipsis-head patterns.

```
(define-eh-alternative-set name eh-alternative ...)

alternative = (pattern EH-pattern)
```

Defines `name` as an ellipsis-head alternative set. Using `name` (via `~eh-var`) in an ellipsis-head pattern is equivalent to including each of the alternatives in the pattern via `~alt`, except that the attributes bound by the alternatives are prefixed with the name given to `~eh-var`.

Unlike syntax classes, ellipsis-head alternative sets must be defined before they are referenced, and they do not delimit cuts (use `~delimit-cut` instead).

```
EH-pattern = ....
           | (~eh-var name eh-alternative-set-id)
```

```
(~eh-var name eh-alternative-set-id)
```

Includes the alternatives of `eh-alternative-set-id`, prefixing their attributes with `name`.

Examples:

```
> (define-eh-alternative-set options
    (pattern (~once (~seq #:a a:expr) #:name "#:a option"))
    (pattern (~seq #:b b:expr)))
```

```
> (define (parse/options stx)
    (syntax-parse stx
      [(_ (~eh-var s options) ...)
       #'(s.a (s.b ...))]))
> (parse/options #'(m #:a 1 #:b 2 #:b 3))
#<syntax:eval:12:0 (1 (2 3))>
> (parse/options #'(m #:a 1 #:a 2))
m: too many occurrences of #:a option
  at: ()
  within: (m #:a 1 #:a 2)
  in: (m #:a 1 #:a 2)
> (define (parse/more-options stx)
    (syntax-parse stx
      [(_ (~alt (~eh-var s options)
                (~seq #:c c1:expr c2:expr))
          ...)
       #'(s.a (s.b ...) ((c1 c2) ...))]))
> (parse/more-options #'(m #:a 1 #:b 2 #:c 3 4 #:c 5 6))
#<syntax:eval:15:0 (1 (2) ((3 4) (5 6)))>
> (define-eh-alternative-set ext-options
    (pattern (~eh-var s options))
    (pattern (~seq #:c c1 c2)))
> (syntax-parse #'(m #:a 1 #:b 2 #:c 3 4 #:c 5 6)
    [(_ (~eh-var x ext-options) ...)
     #'(x.s.a (x.s.b ...) ((x.c1 x.c2) ...))])
#<syntax:eval:18:0 (1 (2) ((3 4) (5 6)))>
```

1.12.6 Syntax Class Specialization

```
(require syntax/parse/experimental/specialize)
```
package: base

```
(define-syntax-class/specialize header syntax-class-use)

          header = id
                 | (id . kw-formals)

syntax-class-use = target-stxclass-id
                 | (target-stxclass-id arg ...)
```

Defines *id* as a syntax class with the same attributes, options (eg, `#:commit`, `#:no-delimit-cut`), and patterns as *target-stxclass-id* but with the given *arg*s supplied.

Examples:

```
> (define-syntax-class/specialize nat>10 (nat> 10))
> (syntax-parse #'(11 12) [(n:nat>10 ...) 'ok])
'ok
> (syntax-parse #'(8 9) [(n:nat>10 ...) 'ok])
?: expected natural number greater than 10
  at: 8
  in: (8 9)
```

1.12.7 Syntax Templates

```
(require syntax/parse/experimental/template)      package: base
```

```
(template tmpl)
(template/loc loc-expr tmpl)
(quasitemplate tmpl)
(quasitemplate/loc loc-expr tmpl)
```

Equivalent to `syntax`, `syntax/loc`, `quasisyntax`, and `quasisyntax/loc`, respectively.

```
(datum-template tmpl)
```

Equivalent to `datum`.

```
??
?@
```

Equivalent to `~?` and `~@`, respectively.

```
(define-template-metafunction metafunction-id expr)
(define-template-metafunction (metafunction-id . formals) body ...+)
```

Defines *`metafunction-id`* as a *template metafunction*. A metafunction application in a `syntax` or `template` expression is evaluated by applying the metafunction to the result of processing the “argument” part of the template.

Examples:

```
> (define-template-metafunction (join stx)
    (syntax-parse stx
      [(join (~optional (~seq #:lctx lctx)) a:id b:id ...)
       (datum->syntax (or (attribute lctx) #'a)
                      (string->symbol
                       (apply string-append
```

```
                                  (map symbol->string
                                       (syntax->datum #'(a b ...)))))
                          stx)]))
> (template (join a b c))
#<syntax:eval:23:0 abc>
> (with-syntax ([(x ...) #'(a b c)])
    (template ((x (join tmp- x)) ...)))
#<syntax:eval:24:0 ((a tmp-a) (b tmp-b) (c tmp-c))>
```

Metafunctions are useful for performing transformations in contexts where macro expansion does not occur, such as binding occurrences. For example:

```
> (syntax->datum
   (with-syntax ([name #'posn]
                 [(field ...) #'(x y)])
     (template (let-values ([((join name ?)
                              (join #:lctx name make- name)
                              (join name - field) ...)
                             (make-struct-type __)])
                 __))))
'(let-values (((posn? make-posn posn-x posn-y) (make-struct-type
___))) ___)
```

If `join` were defined as a macro, it would not be usable in the context above; instead, `let-values` would report an invalid binding list.

1.13 Minimal Library

```
(require syntax/parse/pre)        package: base
```

The `syntax/parse/pre` library is useful for accessing most syntax-parsing functionality while minimizing library dependencies. It provides most of `syntax/parse`, but omits these bindings:

```
expr/c
pattern-expander?
prop:syntax-class
pattern-expander
prop:pattern-expander
syntax-local-syntax-parse-pattern-introduce
```

In addition, the provided variant of `static` is a different binding that lacks explicit contract checks.

2 Syntax Object Helpers

2.1 Deconstructing Syntax Objects

```
(require syntax/stx)        package: base
```

```
(stx-null? v) → boolean?
  v : any/c
```

Returns `#t` if *v* is either the empty list or a syntax object representing the empty list (i.e., `syntax-e` on the syntax object returns the empty list).

Examples:

```
> (stx-null? null)
#t
> (stx-null? #'())
#t
> (stx-null? #'(a))
#f
```

```
(stx-pair? v) → boolean?
  v : any/c
```

Returns `#t` if *v* is either a pair or a syntax object representing a pair (see syntax pair).

Examples:

```
> (stx-pair? (cons #'a #'b))
#t
> (stx-pair? #'(a . b))
#t
> (stx-pair? #'())
#f
> (stx-pair? #'a)
#f
```

```
(stx-list? v) → boolean?
  v : any/c
```

Returns `#t` if *v* is a list, or if it is a sequence of pairs leading to a syntax object such that `syntax->list` would produce a list.

Examples:

```
> (stx-list? #'(a b c d))
#t
> (stx-list? #'((a b) (c d)))
#t
> (stx-list? #'(a b (c d)))
#t
> (stx-list? (list #'a #'b))
#t
> (stx-list? #'a)
#f
```

```
(stx->list stx-list) → (or/c list? #f)
  stx-list : stx-list?
```

Produces a list by flatting out a trailing syntax object using `syntax->list`.

Examples:

```
> (stx->list #'(a b c d))
'(#<syntax:eval:14:0 a>
  #<syntax:eval:14:0 b>
  #<syntax:eval:14:0 c>
  #<syntax:eval:14:0 d>)
> (stx->list #'((a b) (c d)))
'(#<syntax:eval:15:0 (a b)> #<syntax:eval:15:0 (c d)>)
> (stx->list #'(a b (c d)))
'(#<syntax:eval:16:0 a> #<syntax:eval:16:0 b> #<syntax:eval:16:0
(c d)>)
> (stx->list (list #'a #'b))
'(#<syntax:eval:17:0 a> #<syntax:eval:17:0 b>)
> (stx->list #'a)
#f
```

```
(stx-car v) → any
  v : stx-pair?
```

Takes the car of a syntax pair.

Examples:

```
> (stx-car #'(a b))
#<syntax:eval:19:0 a>
> (stx-car (list #'a #'b))
#<syntax:eval:20:0 a>
```

```
(stx-cdr v) → any
  v : stx-pair?
```

Takes the cdr of a syntax pair.

Examples:

```
> (stx-cdr #'(a b))
'(#<syntax:eval:21:0 b>)
> (stx-cdr (list #'a #'b))
'(#<syntax:eval:22:0 b>)
```

```
(stx-map proc stxl ...) → list?
  proc : procedure?
  stxl : stx-list?
```

Equivalent to `(map proc (stx->list stxl) ...)`.

Example:

```
> (stx-map (λ (id) (free-identifier=? id #'a)) #'(a b c d))
'(#t #f #f #f)
```

```
(module-or-top-identifier=? a-id b-id) → boolean?
  a-id : identifier?
  b-id : identifier?
```

Returns `#t` if *a-id* and *b-id* are `free-identifier=?`, or if *a-id* and *b-id* have the same name (as extracted by `syntax-e`) and *a-id* has no binding other than at the top level.

This procedure is useful in conjunction with `syntax-case*` to match procedure names that are normally bound by Racket. For example, the `include` macro uses this procedure to recognize `build-path`; using `free-identifier=?` would not work well outside of `module`, since the top-level `build-path` is a distinct variable from the `racket/base` export (though it's bound to the same procedure, initially).

2.2 Matching Fully-Expanded Expressions

```
(require syntax/kerncase)        package: base
```

```
(kernel-syntax-case stx-expr trans?-expr clause ...)
```

A syntactic form like `syntax-case*`, except that the literals are built-in as the names of the primitive Racket forms as exported by `racket/base`, including `letrec-syntaxes+values`; see §1.2.3.1 "Fully Expanded Programs".

The *`trans?-expr`* boolean expression replaces the comparison procedure, and instead selects simply between normal-phase comparisons or transformer-phase comparisons. The *`clause`*s are the same as in `syntax-case*`.

The primitive syntactic forms must have their normal bindings in the context of the `kernel-syntax-case` expression. Beware that `kernel-syntax-case` does not work in a module whose language provides different bindings for these primitive syntactic forms, such as `mzscheme` which does not provide the primitive `if` and `typed/racket` which does not provide the primitive `let-values` among others.

```
(kernel-syntax-case* stx-expr trans?-expr (extra-id ...) clause ...)
```

A syntactic form like `kernel-syntax-case`, except that it takes an additional list of extra literals that are in addition to the primitive Racket forms.

```
(kernel-syntax-case/phase stx-expr phase-expr clause ...)
```

Generalizes `kernel-syntax-case` to work at an arbitrary phase level, as indicated by *`phase-expr`*.

```
(kernel-syntax-case*/phase stx-expr phase-expr (extra-id ..)
  clause ...)
```

Generalizes `kernel-syntax-case*` to work at an arbitrary phase level, as indicated by *`phase-expr`*.

```
(kernel-form-identifier-list) → (listof identifier?)
```

Returns a list of identifiers that are bound normally, `for-syntax`, and `for-template` to the primitive Racket forms for expressions, internal-definition positions, and module-level and top-level positions. This function is useful for generating a list of stopping points to provide to `local-expand`.

In addition to the identifiers listed in §1.2.3.1 "Fully Expanded Programs", the list includes `letrec-syntaxes+values`, which is the core form for local expand-time binding and can appear in the result of `local-expand`.

Changed in version 6.90.0.27 of package `base`: Added `quote-syntax` and `#%plain-module-begin` to the list, which had previously been unintentionally missing.

2.3 Dictionaries with Identifier Keys

```
(require syntax/id-table)        package: base
```

This module provides two implementations of *identifier tables*: dictionaries with identifier keys that use identifier-specific comparisons instead of eq? or equal?. Identifier tables implement the racket/dict interface, and they are available in both mutable and immutable variants.

2.3.1 Dictionaries for free-identifier=?

A free-identifier table is a dictionary whose keys are compared using free-identifier=?. Free-identifier tables implement the dictionary interface of racket/dict, so all of the appropriate generic functions (dict-ref, dict-map, etc) can be used on free-identifier tables.

A caveat for using these tables is that a lookup can fail with unexpected results if the binding of an identifier changes between key-value insertion and the lookup.

For example, consider the following use:

```
> (define-syntax-rule (m)
    (begin
      (begin-for-syntax
        (define table (make-free-id-table))
        ; set table entry to #t
        (free-id-table-set! table #'x #t)
        ; sanity check, it's set to #t
        (displayln (free-id-table-ref table #'x #f)))

      (define x 'defined-now)

      (begin-for-syntax
        ; might expect to get #t, but prints #f
        (displayln (free-id-table-ref table #'x #f)))))
> (m)
#t
#f
```

The macro m expands to code that initializes an identifier table at compile-time and inserts a key-value pair for #'x and #t. The #'x identifier has no binding, however, until the definition (define x 'defined-now) is evaluated.

As a result, the lookup at the end of m will return #f instead of #t because the binding symbol for #'x changes after the initial key-value pair is put into the table. If the definition is evaluated *before* the initial insertion, both expressions will print #t.

```
(make-free-id-table [init-dict
                     #:phase phase]) → mutable-free-id-table?
  init-dict : dict? = null
  phase : (or/c exact-integer? #f) = (syntax-local-phase-level)
```

```
(make-immutable-free-id-table [init-dict
                                #:phase phase])
 → immutable-free-id-table?
  init-dict : dict? = null
  phase : (or/c exact-integer? #f) = (syntax-local-phase-level)
```

Produces a mutable free-identifier table or immutable free-identifier table, respectively. The dictionary uses `free-identifier=?` to compare keys, but also uses a hash table based on symbol equality to make the dictionary efficient in the common case.

The identifiers are compared at phase level *phase*. The default phase, `(syntax-local-phase-level)`, is generally appropriate for identifier tables used by macros, but code that analyzes fully-expanded programs may need to create separate identifier tables for each phase of the module.

The optional *init-dict* argument provides the initial mappings. It must be a dictionary, and its keys must all be identifiers. If the *init-dict* dictionary has multiple distinct entries whose keys are `free-identifier=?`, only one of the entries appears in the new id-table, and it is not specified which entry is picked.

```
(free-id-table? v) → boolean?
  v : any/c
```

Returns `#t` if *v* was produced by `make-free-id-table` or `make-immutable-free-id-table`, `#f` otherwise.

```
(mutable-free-id-table? v) → boolean?
  v : any/c
```

Returns `#t` if *v* was produced by `make-free-id-table`, `#f` otherwise.

```
(immutable-free-id-table? v) → boolean?
  v : any/c
```

Returns `#t` if *v* was produced by `make-immutable-free-id-table`, `#f` otherwise.

```
(free-id-table-ref table id [failure]) → any
  table : free-id-table?
  id : identifier?
  failure : any/c = (lambda () (raise (make-exn:fail .....)))
```

Like `hash-ref`. In particular, if *id* is not found, the *failure* argument is applied if it is a procedure, or simply returned otherwise.

```
(free-id-table-ref! table id failure) → any
  table : mutable-free-id-table?
  id : identifier?
  failure : any/c
```

Like hash-ref!.

Added in version 6.3.0.6 of package base.

```
(free-id-table-set! table id v) → void?
  table : mutable-free-id-table?
  id : identifier?
  v : any/c
```

Like hash-set!.

```
(free-id-table-set table id v) → immutable-free-id-table?
  table : immutable-free-id-table?
  id : identifier?
  v : any/c
```

Like hash-set.

```
(free-id-table-set*! table id v ... ...) → void?
  table : mutable-free-id-table?
  id : identifier?
  v : any/c
```

Like hash-set*!.

Added in version 6.3.0.6 of package base.

```
(free-id-table-set* table id v ... ...) → immutable-free-id-table?
  table : immutable-free-id-table?
  id : identifier?
  v : any/c
```

Like hash-set*.

Added in version 6.3.0.6 of package base.

```
(free-id-table-remove! table id) → void?
  table : mutable-free-id-table?
  id : identifier?
```

Like hash-remove!.

```
(free-id-table-remove table id) → immutable-free-id-table?
  table : immutable-free-id-table?
  id : identifier?
```

Like `hash-remove`.

```
(free-id-table-update!  table
                        id
                        updater
                       [failure]) → void?
  table : mutable-free-id-table?
  id : identifier?
  updater : (any/c . -> . any/c)
  failure : any/c = (lambda () (raise (make-exn:fail .....)))
```

Like `hash-update!`.

Added in version 6.3.0.6 of package `base`.

```
(free-id-table-update  table
                       id
                       updater
                      [failure]) → immutable-free-id-table?
  table : immutable-free-id-table?
  id : identifier?
  updater : (any/c . -> . any/c)
  failure : any/c = (lambda () (raise (make-exn:fail .....)))
```

Like `hash-update`.

Added in version 6.3.0.6 of package `base`.

```
(free-id-table-map table proc) → list?
  table : free-id-table?
  proc : (-> identifier? any/c any)
```

Like `hash-map`.

```
(free-id-table-keys table) → (listof identifier?)
  table : free-id-table?
```

Like `hash-keys`.

Added in version 6.3.0.3 of package `base`.

```
(free-id-table-values table) → (listof any/c)
  table : free-id-table?
```

Like `hash-values`.

Added in version 6.3.0.3 of package `base`.

```
(in-free-id-table table) → sequence?
  table : free-id-table?
```

Like `in-hash`.

Added in version 6.3.0.3 of package `base`.

```
(free-id-table-for-each table proc) → void?
  table : free-id-table?
  proc : (-> identifier? any/c any)
```

Like `hash-for-each`.

```
(free-id-table-count table) → exact-nonnegative-integer?
  table : free-id-table?
```

Like `hash-count`.

```
(free-id-table-iterate-first table) → id-table-iter?
  table : free-id-table?
(free-id-table-iterate-next table position) → id-table-iter?
  table : free-id-table?
  position : id-table-iter?
(free-id-table-iterate-key table position) → identifier?
  table : free-id-table?
  position : id-table-iter?
(free-id-table-iterate-value table
                             position) → identifier?
  table : bound-it-table?
  position : id-table-iter?
```

Like `hash-iterate-first`, `hash-iterate-next`, `hash-iterate-key`, and `hash-iterate-value`, respectively.

```
(id-table-iter? v) → boolean?
  v : any/c
```

Returns `#t` if *v* represents a position in an identifier table (free or bound, mutable or immutable), `#f` otherwise.

```
(free-id-table/c key-ctc
                 val-ctc
                [#:immutable immutable?]) → contract?
  key-ctc : flat-contract?
  val-ctc : chaperone-contract?
  immutable? : (or/c #t #f 'dont-care) = 'dont-care
```

Like `hash/c`, but for free-identifier tables. If *immutable?* is `#t`, the contract accepts only immutable identifier tables; if *immutable?* is `#f`, the contract accepts only mutable identifier tables.

2.3.2 Dictionaries for `bound-identifier=?`

A bound-identifier table is a dictionary whose keys are compared using `bound-identifier=?`. Bound-identifier tables implement the dictionary interface of `racket/dict`, so all of the appropriate generic functions (`dict-ref`, `dict-map`, etc) can be used on bound-identifier tables.

```
(make-bound-id-table [init-dict
                      #:phase phase]) → mutable-bound-id-table?
  init-dict : dict? = null
  phase : (or/c exact-integer? #f) = (syntax-local-phase-level)
(make-immutable-bound-id-table [init-dict
                                #:phase phase])
 → immutable-bound-id-table?
  init-dict : dict? = null
  phase : (or/c exact-integer? #f) = (syntax-local-phase-level)
(bound-id-table? v) → boolean?
  v : any/c
(mutable-bound-id-table? v) → boolean?
  v : any/c
(immutable-bound-id-table? v) → boolean?
  v : any/c
(bound-id-table-ref table id [failure]) → any
  table : bound-id-table?
  id : identifier?
  failure : any/c = (lambda () (raise (make-exn:fail .....)))
(bound-id-table-ref! table id failure) → any
  table : mutable-bound-id-table?
  id : identifier?
  failure : any/c
(bound-id-table-set! table id v) → void?
  table : mutable-bound-id-table?
  id : identifier?
  v : any/c
(bound-id-table-set table id v) → immutable-bound-id-table?
  table : immutable-bound-id-table?
  id : identifier?
  v : any/c
```

```
(bound-id-table-set*! table id v ... ...) → void?
  table : mutable-bound-id-table?
  id : identifier?
  v : any/c
(bound-id-table-set* table id v ... ...)
 → immutable-bound-id-table?
  table : immutable-bound-id-table?
  id : identifier?
  v : any/c
(bound-id-table-remove! table id) → void?
  table : mutable-bound-id-table?
  id : identifier?
(bound-id-table-remove table id) → immutable-bound-id-table?
  table : immutable-bound-id-table?
  id : identifier?
(bound-id-table-update! table
                        id
                        updater
                       [failure]) → void?
  table : mutable-bound-id-table?
  id : identifier?
  updater : (any/c . -> . any/c)
  failure : any/c = (lambda () (raise (make-exn:fail .....)))
(bound-id-table-update table
                       id
                       updater
                      [failure]) → immutable-bound-id-table?
  table : immutable-bound-id-table?
  id : identifier?
  updater : (any/c . -> . any/c)
  failure : any/c = (lambda () (raise (make-exn:fail .....)))
(bound-id-table-map table proc) → list?
  table : bound-id-table?
  proc : (-> identifier? any/c any)
(bound-id-table-keys table) → (listof identifier?)
  table : bound-id-table?
(bound-id-table-values table) → (listof any/c)
  table : bound-id-table?
(in-bound-id-table table) → sequence?
  table : bound-id-table?
(bound-id-table-for-each table proc) → void?
  table : bound-id-table?
  proc : (-> identifier? any/c any)
(bound-id-table-count table) → exact-nonnegative-integer?
  table : bound-id-table?
```

```
(bound-id-table-iterate-first table) → id-table-position?
  table : bound-id-table?
(bound-id-table-iterate-next table
                             position) → id-table-position?
  table : bound-id-table?
  position : id-table-position?
(bound-id-table-iterate-key table position) → identifier?
  table : bound-id-table?
  position : id-table-position?
(bound-id-table-iterate-value table
                              position) → identifier?
  table : bound-id-table?
  position : id-table-position?
(bound-id-table/c  key-ctc
                   val-ctc
                  [#:immutable immutable]) → contract?
  key-ctc : flat-contract?
  val-ctc : chaperone-contract?
  immutable : (or/c #t #f 'dont-care) = 'dont-care
```

Like the procedures for free-identifier tables (`make-free-id-table`, `free-id-table-ref`, etc), but for bound-identifier tables, which use `bound-identifier=?` to compare keys.

Changed in version 6.3.0.3 of package `base`: Added bound-id-table-keys, bound-id-table-values, in-bound-id-table.
Changed in version 6.3.0.6: Added bound-id-table-ref!, bound-id-table-set*, bound-id-table-set*!, bound-id-table-update!, and bound-id-table-update

2.4 Sets with Identifier Keys

```
(require syntax/id-set)        package: base
```

This module provides *identifier sets*: sets with identifier keys that use identifier-specific comparisons instead of the usual equality operators such as `eq?` or `equal?`.

This module implements two kinds of identifier sets: one via `free-identifier=?` and one via `bound-identifier=?`. Each are available in both mutable and immutable variants and implement the `gen:set`, `gen:stream`, `prop:sequence`, and `gen:equal+hash` generic interfaces.

Identifier sets are implemented using identifier tables, in the same way that hash sets are implemented with hash tables.

2.4.1 Sets for free-identifier=?

A free-identifier set is a set whose keys are compared using free-identifier=?. Free-identifier sets implement the gen:set interface, so all of the appropriate generic functions (e.g., set-add, set-map, etc) can be used on free-identifier sets.

```
(mutable-free-id-set [init-set
                      #:phase phase]) → mutable-free-id-set?
  init-set : generic-set? = null
  phase : (or/c exact-integer? #f) = (syntax-local-phase-level)
(immutable-free-id-set [init-set
                        #:phase phase]) → immutable-free-id-set?
  init-set : generic-set? = null
  phase : (or/c exact-integer? #f) = (syntax-local-phase-level)
```

Produces a mutable free-identifier set or immutable free-identifier set, respectively. The set uses free-identifier=? to compare keys.

The identifiers are compared at phase level *phase*. The default phase, (syntax-local-phase-level), is generally appropriate for identifier sets used by macros, but code that analyzes fully-expanded programs may need to create separate identifier sets for each phase of the module.

The optional *init-set* argument provides the initial set elements. It must be a set of identifiers. If the *init-set* set has multiple distinct entries whose keys are free-identifier=?, only one of the entries appears in the new id-set, and it is not specified which entry is picked.

```
(free-id-set? v) → boolean?
  v : any/c
```

Returns #t if *v* was produced by mutable-free-id-set or immutable-free-id-set, #f otherwise.

```
(mutable-free-id-set? v) → boolean?
  v : any/c
```

Returns #t if *v* was produced by mutable-free-id-set, #f otherwise.

```
(immutable-free-id-set? v) → boolean?
  v : any/c
```

Returns #t if *v* was produced by immutable-free-id-set, #f otherwise.

```
(free-id-set-empty? s) → boolean?
  s : free-id-set?
```

Like set-empty?.

```
(free-id-set-count s) → exact-nonnegative-integer?
  s : free-id-set?
```

Like set-count.

```
(free-id-set-member? s v) → boolean?
  s : free-id-set?
  v : identifier?
```

Like set-member?.

```
(free-id-set=? s1 s2) → boolean?
  s1 : free-id-set?
  s2 : free-id-set?
```

Like set=?.

```
(free-id-set-add s v) → immutable-free-id-set?
  s : immutable-free-id-set?
  v : identifier?
```

Like set-add.

```
(free-id-set-add! s v) → void?
  s : mutable-free-id-set?
  v : identifier?
```

Like set-add!.

```
(free-id-set-remove s v) → immutable-free-id-set?
  s : immutable-free-id-set?
  v : identifier?
```

Like set-remove.

```
(free-id-set-remove! s v) → void?
  s : mutable-free-id-set?
  v : identifier?
```

Like set-remove!.

```
(free-id-set-first s) → identifier?
  s : free-id-set?
```

Like `set-first`.

```
(free-id-set-rest s) → immutable-free-id-set?
  s : immutable-free-id-set?
```

Like `set-rest`.

```
(in-free-id-set s) → sequence?
  s : free-id-set?
```

Like `in-set`.

```
(free-id-set->stream s) → stream?
  s : free-id-set?
```

Like `set->stream`.

```
(free-id-set->list s) → list?
  s : free-id-set?
```

Like `set->list`.

```
(free-id-set-copy s) → free-id-set?
  s : free-id-set?
```

Like `set-copy`.

```
(free-id-set-copy-clear s) → free-id-set?
  s : free-id-set?
```

Like `set-copy-clear`.

```
(free-id-set-clear s) → immutable-free-id-set?
  s : immutable-free-id-set?
```

Like `set-clear`.

```
(free-id-set-clear! s) → void?
  s : mutable-free-id-set?
```

Like `set-clear!`.

```
(free-id-set-union s0 s ...) → immutable-free-id-set?
  s0 : immutable-free-id-set?
  s : free-id-set?
```

Like `set-union`.

```
(free-id-set-union! s0 s ...) → void?
  s0 : mutable-free-id-set?
  s : free-id-set?
```

Like `set-union!`.

```
(free-id-set-intersect s0 s ...) → immutable-free-id-set?
  s0 : immutable-free-id-set?
  s : free-id-set?
```

Like `set-intersect`.

```
(free-id-set-intersect! s0 s ...) → void?
  s0 : mutable-free-id-set?
  s : free-id-set?
```

Like `set-intersect!`.

```
(free-id-set-subtract s0 s ...) → immutable-free-id-set?
  s0 : immutable-free-id-set?
  s : free-id-set?
```

Like `set-subtract`.

```
(free-id-set-subtract! s0 s ...) → void?
  s0 : mutable-free-id-set?
  s : free-id-set?
```

Like `set-subtract!`.

```
(free-id-set-symmetric-difference s0 s ...)
 → immutable-free-id-set?
  s0 : immutable-free-id-set?
  s : free-id-set?
```

Like `set-symmetric-difference`.

```
(free-id-set-symmetric-difference! s0 s ...) → void?
  s0 : mutable-free-id-set?
  s : free-id-set?
```

Like `set-symmetric-difference!`.

```
(free-id-subset? s1 s2) → boolean?
  s1 : free-id-set?
  s2 : free-id-set?
```

Like `subset?`.

```
(free-id-proper-subset? s1 s2) → boolean?
  s1 : free-id-set?
  s2 : free-id-set?
```

Like `proper-subset?`.

```
(free-id-set-map s f) → list?
  s : free-id-set?
  f : (-> identifier? any/c)
```

Like `set-map`.

```
(free-id-set-for-each s f) → void?
  s : free-id-set?
  f : (-> identifier? any/c)
```

Like `set-for-each`.

```
(id-set/c elem-ctc
         [#:setidtype idsettype
          #:mutability mutability]) → contract?
  elem-ctc : flat-contract?
  idsettype : (or/c 'dont-care 'free 'bound) = 'dont-care
  mutability : (or/c 'dont-care 'mutable 'immutable)
             = 'immutable
```

Creates a contract for identifier sets. If *mutability* is `'immutable`, the contract accepts only immutable identifier sets; if *mutability* is `'mutable`, the contract accepts only mutable identifier sets.

```
(free-id-set/c elem-ctc
              [#:mutability mutability]) → contract?
  elem-ctc : flat-contract?
  mutability : (or/c 'dont-care 'mutable 'immutable)
             = 'immutable
```

Creates a contract for free-identifier sets. If *mutability* is `'immutable`, the contract accepts only immutable identifier sets; if *mutability* is `'mutable`, the contract accepts only mutable identifier sets.

2.4.2 Sets for `bound-identifier=?`

A bound-identifier set is a set whose keys are compared using `bound-identifier=?`. Bound-identifier sets implement the `gen:set` interface, so all of the appropriate generic functions (e.g., `set-add`, `set-map`, etc.) can be used on bound-identifier sets.

```
(mutable-bound-id-set [init-set
                       #:phase phase]) → mutable-bound-id-set?
  init-set : set? = null
  phase : (or/c exact-integer? #f) = (syntax-local-phase-level)
(immutable-bound-id-set [init-set
                         #:phase phase])
 → immutable-bound-id-set?
  init-set : set? = null
  phase : (or/c exact-integer? #f) = (syntax-local-phase-level)
(bound-id-set? v) → boolean?
  v : any/c
(mutable-bound-id-set? v) → boolean?
  v : any/c
(immutable-bound-id-set? v) → boolean?
  v : any/c
(bound-id-set-empty? s) → boolean?
  s : bound-id-set?
(bound-id-set-count s) → exact-nonnegative-integer?
  s : bound-id-set?
(bound-id-set-member? s v) → boolean?
  s : bound-id-set?
  v : identifier?
(bound-id-set=? s1 s2) → boolean?
  s1 : bound-id-set?
  s2 : bound-id-set?
(bound-id-set-add s v) → immutable-bound-id-set?
  s : immutable-bound-id-set?
  v : identifier?
(bound-id-set-add! s v) → void?
  s : mutable-bound-id-set?
  v : identifier?
(bound-id-set-remove s v) → immutable-bound-id-set?
  s : immutable-bound-id-set?
  v : identifier?
(bound-id-set-remove! s v) → void?
  s : mutable-bound-id-set?
  v : identifier?
(bound-id-set-first s) → identifier?
  s : bound-id-set?
```

```
(bound-id-set-rest s) → immutable-bound-id-set?
  s : immutable-bound-id-set?
(in-bound-id-set s) → sequence?
  s : bound-id-set?
(bound-id-set->stream s) → stream?
  s : bound-id-set?
(bound-id-set->list s) → list?
  s : bound-id-set?
(bound-id-set-copy s) → bound-id-set?
  s : bound-id-set?
(bound-id-set-copy-clear s) → bound-id-set?
  s : bound-id-set?
(bound-id-set-clear s) → immutable-bound-id-set?
  s : immutable-bound-id-set?
(bound-id-set-clear! s) → void?
  s : mutable-bound-id-set?
(bound-id-set-union s0 s ...) → immutable-bound-id-set?
  s0 : immutable-bound-id-set?
  s : bound-id-set?
(bound-id-set-union! s0 s ...) → void?
  s0 : mutable-bound-id-set?
  s : bound-id-set?
(bound-id-set-intersect s0 s ...) → immutable-bound-id-set?
  s0 : immutable-bound-id-set?
  s : bound-id-set?
(bound-id-set-intersect! s0 s ...) → void?
  s0 : mutable-bound-id-set?
  s : bound-id-set?
(bound-id-set-subtract s0 s ...) → immutable-bound-id-set?
  s0 : immutable-bound-id-set?
  s : bound-id-set?
(bound-id-set-subtract! s0 s ...) → void?
  s0 : mutable-bound-id-set?
  s : bound-id-set?
(bound-id-set-symmetric-difference s0 s ...)
 → immutable-bound-id-set?
  s0 : immutable-bound-id-set?
  s : bound-id-set?
(bound-id-set-symmetric-difference! s0
                                    s ...) → void?
  s0 : mutable-bound-id-set?
  s : bound-id-set?
(bound-id-subset? s1 s2) → boolean?
  s1 : bound-id-set?
  s2 : bound-id-set?
```

```
(bound-id-proper-subset? s1 s2) → boolean?
  s1 : bound-id-set?
  s2 : bound-id-set?
(bound-id-set-map s f) → list?
  s : bound-id-set?
  f : (-> identifier? any/c)
(bound-id-set-for-each s f) → void?
  s : bound-id-set?
  f : (-> identifier? any/c)
(bound-id-set/c  elem-ctc
                [#:mutability mutability]) → contract?
  elem-ctc : flat-contract?
  mutability : (or/c 'dont-care 'mutable 'immutable)
             = 'immutable
```

Like the procedures for free-identifier sets (e.g., `immutable-free-id-set`, `free-id-set-add`, etc.), but for bound-identifier sets, which use `bound-identifier=?` to compare keys.

2.5 Hashing on `bound-identifier=?` and `free-identifier=?`

This library is for backwards-compatibility. Do not use it for new libraries; use `syntax/id-table` instead.

```
(require syntax/boundmap)        package: base
```

```
(make-bound-identifier-mapping) → bound-identifier-mapping?
(bound-identifier-mapping? v) → boolean?
  v : any/c
(bound-identifier-mapping-get  bound-map
                               id
                              [failure-thunk]) → any
  bound-map : bound-identifier-mapping?
  id : identifier?
  failure-thunk : (-> any)
                = (lambda () (raise (make-exn:fail ....)))
(bound-identifier-mapping-put! bound-map
                               id
                               v)          → void?
  bound-map : bound-identifier-mapping?
  id : identifier?
  v : any/c
```

```
(bound-identifier-mapping-for-each bound-map
                                   proc)      → void?
  bound-map : bound-identifier-mapping?
  proc : (identifier? any/c . -> . any)
(bound-identifier-mapping-map bound-map
                              proc)      → (listof any?)
  bound-map : bound-identifier-mapping?
  proc : (identifier? any/c . -> . any)
```

Similar to make-bound-id-table, bound-id-table?, bound-id-table-ref, bound-id-table-set!, bound-id-table-for-each, and bound-id-table-map, respectively.

```
(make-free-identifier-mapping) → free-identifier-mapping?
(free-identifier-mapping? v) → boolean?
  v : any/c
(free-identifier-mapping-get  free-map
                              id
                             [failure-thunk]) → any
  free-map : free-identifier-mapping?
  id : identifier?
  failure-thunk :  (-> any)
                = (lambda () (raise (make-exn:fail ....)))
(free-identifier-mapping-put! free-map id v) → void?
  free-map : free-identifier-mapping?
  id : identifier?
  v : any/c
(free-identifier-mapping-for-each free-map
                                  proc)     → void?
  free-map : free-identifier-mapping?
  proc : (identifier? any/c . -> . any)
(free-identifier-mapping-map free-map proc) → (listof any?)
  free-map : free-identifier-mapping?
  proc : (identifier? any/c . -> . any)
```

Similar to make-free-id-table, free-id-table?, free-id-table-ref, free-id-table-set!, free-id-table-for-each, and free-id-table-map, respectively.

```
(make-module-identifier-mapping) → module-identifier-mapping?
(module-identifier-mapping? v) → boolean?
  v : any/c
(module-identifier-mapping-get  module-map
                                id
                               [failure-thunk]) → any
  module-map : module-identifier-mapping?
  id : identifier?
  failure-thunk :  (-> any)
                = (lambda () (raise (make-exn:fail ....)))
```

```
(module-identifier-mapping-put! module-map
                                id
                                v)          → void?
  module-map : module-identifier-mapping?
  id : identifier?
  v : any/c
(module-identifier-mapping-for-each module-map
                                    proc)      → void?
  module-map : module-identifier-mapping?
  proc : (identifier? any/c . -> . any)
(module-identifier-mapping-map module-map
                               proc)      → (listof any?)
  module-map : module-identifier-mapping?
  proc : (identifier? any/c . -> . any)
```

The same as `make-free-identifier-mapping`, etc.

2.6 Rendering Syntax Objects with Formatting

```
(require syntax/to-string)      package: base
```

```
(syntax->string stx-list) → string?
  stx-list : (and/c syntax? stx-list?)
```

Builds a string with newlines and indenting according to the source locations in *`stx-list`*; the outer pair of parens are not rendered from *`stx-list`*.

2.7 Computing the Free Variables of an Expression

```
(require syntax/free-vars)      package: base
```

```
(free-vars  expr-stx
           [insp
            #:module-bound? module-bound?])
 → (listof identifier?)
  expr-stx : syntax?
  insp : inspector? = mod-decl-insp
  module-bound? : any/c = #f
```

Returns a list of free `lambda`- and `let`-bound identifiers in *`expr-stx`* in the order in which each identifier first appears within *`expr-stx`*. The expression must be fully expanded (see §1.2.3.1 "Fully Expanded Programs" and `expand`).

The inspector *insp* is used to disarm *expr-stx* and sub-expressions before extracting identifiers. The default *insp* is the declaration-time inspector of the syntax/free-vars module.

If *module-bound?* is non-false, the list of free variables also includes free module-bound identifiers.

Examples:

```
> (require (for-syntax racket/base syntax/parse syntax/free-vars))
> (define-syntax (print-body-free-vars stx)
    (syntax-parse stx
      #:literals (lambda)
      [(_ (~and lam (lambda (a ...) b ...)))
       (define expanded-body (local-expand #'lam 'expression '()))
       (syntax-parse expanded-body
         #:literals (#%plain-lambda)
         [(#%plain-lambda (arg ...) body)
          (displayln (free-vars #'body))
          expanded-body])]))
> (lambda (x) (print-body-free-vars (lambda (y) x)))
(#<syntax:eval:3:0 x>)
#<procedure>
```

2.8 Replacing Lexical Context

```
(require syntax/strip-context)
```
package: base

```
(strip-context form) → any/c
  form : any/c
```

Removes all lexical context from syntax objects within *form*, preserving source-location information and properties.

Typically, *form* is a syntax object, and then the result is also a syntax object. Otherwise, pairs, vectors, boxes, hash tables, and prefab structures are traversed (and copied for the result) to find syntax objects. Graph structure is not preserved in the result, and cyclic data structures will cause strip-context to never return.

Changed in version 7.7.0.10 of package base: Repaired to traverse hash tables in stx.

```
(replace-context ctx-stx form) → any/c
  ctx-stx : (or/c syntax? #f)
  form : any/c
```

Uses the lexical context of *ctx-stx* to replace the lexical context of all parts of all syntax objects in *form*, preserving source-location information and properties of those syntax objects.

Syntax objects are found in *form* the same as in `strip-context`.

Changed in version 7.7.0.10 of package `base`: Repaired to traverse hash tables in `stx`.

2.9 Helpers for Processing Keyword Syntax

The `syntax/keyword` module contains procedures for parsing keyword options in macros.

```
(require syntax/keyword)        package: base
```

keyword-table = `(dict-of keyword (listof check-procedure))`

A keyword-table is a dictionary (`dict?`) mapping keywords to lists of check-procedures. (Note that an association list is a suitable dictionary.) The keyword's arity is the length of the list of procedures.

Example:

```
> (define my-keyword-table
    (list (list '#:a check-identifier)
          (list '#:b check-expression check-expression)))
```

check-procedure = `(syntax syntax -> any)`

A check procedure consumes the syntax to check and a context syntax object for error reporting and either raises an error to reject the syntax or returns a value as its parsed representation.

Example:

```
> (define (check-stx-string stx context-stx)
    (unless (string? (syntax-e stx))
      (raise-syntax-error #f "expected string" context-stx stx))
    stx)
```

options = `(listof (list keyword syntax-keyword any ...))`

Parsed options are represented as an list of option entries. Each entry contains the keyword, the syntax of the keyword (for error reporting), and the list of parsed values returned by the keyword's list of check procedures. The list contains the parsed options in the order they appeared in the input, and a keyword that occurs multiple times in the input occurs multiple times in the options list.

```
(parse-keyword-options  stx
                        table
                       [#:context ctx
                        #:no-duplicates? no-duplicates?
                        #:incompatible incompatible
                        #:on-incompatible incompatible-handler
                        #:on-too-short too-short-handler
                        #:on-not-in-table not-in-table-handler])
 → options any/c
  stx : syntax?
  table : keyword-table
  ctx : (or/c #f syntax?) = #f
  no-duplicates? : boolean? = #f
  incompatible : (listof (listof keyword?)) = '()
  incompatible-handler : (-> keyword? keyword?
                             options syntax? syntax?
                             (values options syntax?))
                       = (lambda (....) (error ....))
  too-short-handler : (-> keyword? options syntax? syntax?
                          (values options syntax?))
                    = (lambda (....) (error ....))
  not-in-table-handler : (-> keyword? options syntax? syntax?
                             (values options syntax?))
                       = (lambda (....) (error ....))
```

Parses the keyword options in the syntax *stx* (*stx* may be an improper syntax list). The keyword options are described in the *table* association list. Each entry in *table* should be a list whose first element is a keyword and whose subsequent elements are procedures for checking the arguments following the keyword. The keyword's arity (number of arguments) is determined by the number of procedures in the entry. Only fixed-arity keywords are supported.

Parsing stops normally when the syntax list does not have a keyword at its head (it may be empty, start with a non-keyword term, or it may be a non-list syntax object). Two values are returned: the parsed options and the rest of the syntax (generally either a syntax object or a list of syntax objects).

A variety of errors and exceptional conditions can occur during the parsing process. The following keyword arguments determine the behavior in those situations.

The `#:context` *ctx* argument is used to report all errors in parsing syntax. In addition, *ctx* is passed as the final argument to all provided handler procedures. Macros using `parse-keyword-options` should generally pass the syntax object for the whole macro use as *ctx*.

If *no-duplicates?* is a non-false value, then duplicate keyword options are not allowed. If a duplicate is seen, the keyword's associated check procedures are not called and an incompatibility is reported.

The *incompatible* argument is a list of incompatibility entries, where each entry is a list of *at least two* keywords. If any keyword in the entry occurs after any other keyword in the entry, an incompatibility is reported.

Note that including a keyword in an incompatibility entry does not prevent it from occurring multiple times. To disallow duplicates of some keywords (as opposed to all keywords), include those keywords in the *incompatible* list as being incompatible with themselves. That is, include them twice:

```
; Disallow duplicates of only the #:foo keyword
(parse-keyword-options .... #:incompatible '((#:foo #:foo)))
```

When an *incompatibility* occurs, the *incompatible-handler* is tail-called with the two keywords causing the incompatibility (in the order that they occurred in the syntax list, so the keyword triggering the incompatibility occurs second), the syntax list starting with the occurrence of the second keyword, and the context (*ctx*). If the incompatibility is due to a duplicate, the two keywords are the same.

When a keyword is not followed by enough arguments according to its arity in *table*, the *too-short-handler* is tail-called with the keyword, the options parsed thus far, the syntax list starting with the occurrence of the keyword, and *ctx*.

When a keyword occurs in the syntax list that is not in *table*, the *not-in-table-handler* is tail-called with the keyword, the options parsed thus far, the syntax list starting with the occurrence of the keyword, and *ctx*.

Handlers typically escape—all of the default handlers raise errors—but if they return, they should return two values: the parsed options and a syntax object; these are returned as the results of `parse-keyword-options`.

Examples:

```
> (parse-keyword-options
   #'(#:transparent #:property p (lambda (x) (f x)))
   (list (list '#:transparent)
         (list '#:inspector check-expression)
         (list '#:property check-expression check-expression)))
'((#:transparent #<syntax:eval:4:0 #:transparent>)
  (#:property
   #<syntax:eval:4:0 #:property>
   #<syntax:eval:4:0 p>
   #<syntax:eval:4:0 (lambda (x) (f x))>))
'()
> (parse-keyword-options
```

```
#'(#:transparent #:inspector (make-inspector))
(list (list '#:transparent)
      (list '#:inspector check-expression)
      (list '#:property check-expression check-expression))
#:context #'define-struct
#:incompatible '((#:transparent #:inspector)
                 (#:inspector #:inspector)
                 (#:inspector #:inspector)))
```

define-struct: #:inspector option not allowed after #:transparent option

```
(parse-keyword-options/eol
 stx
 table
[#:context ctx
 #:no-duplicates? no-duplicates?
 #:incompatible incompatible
 #:on-incompatible incompatible-handler
 #:on-too-short too-short-handler
 #:on-not-in-table not-in-table-handler
 #:on-not-eol not-eol-handler])
→ options
stx : syntax?
table : keyword-table
ctx : (or/c #f syntax?) = #f
no-duplicates? : boolean? = #f
incompatible : (listof (list keyword? keyword?)) = '()
incompatible-handler : (-> keyword? keyword?
                           options syntax? syntax?
                           (values options syntax?))
                     = (lambda (....) (error ....))
too-short-handler : (-> keyword? options syntax? syntax?
                        (values options syntax?))
                  = (lambda (....) (error ....))
not-in-table-handler : (-> keyword? options syntax? syntax?
                           (values options syntax?))
                     = (lambda (....) (error ....))
not-eol-handler : (-> options syntax? syntax?
                      options)
                = (lambda (....) (error ....))
```

Like `parse-keyword-options`, but checks that there are no terms left over after parsing all of the keyword options. If there are, *not-eol-handler* is tail-called with the options parsed thus far, the leftover syntax, and *ctx*.

```
(options-select options keyword) → (listof list?)
  options : options
  keyword : keyword?
```

Selects the values associated with one keyword from the parsed options. The resulting list has as many items as there were occurrences of the keyword, and each element is a list whose length is the arity of the keyword.

```
(options-select-row options
                    keyword
                    #:default default) → any
  options : options
  keyword : keyword?
  default : any/c
```

Like `options-select`, except that the given keyword must occur either zero or one times in *options*. If the keyword occurs, the associated list of parsed argument values is returned. Otherwise, the *default* list is returned.

```
(options-select-value options
                      keyword
                      #:default default) → any
  options : options
  keyword : keyword?
  default : any/c
```

Like `options-select`, except that the given keyword must occur either zero or one times in *options*. If the keyword occurs, the associated list of parsed argument values must have exactly one element, and that element is returned. If the keyword does not occur in *options*, the *default* value is returned.

```
(check-identifier stx ctx) → identifier?
  stx : syntax?
  ctx : (or/c #f syntax?)
```

A check-procedure that accepts only identifiers.

```
(check-expression stx ctx) → syntax?
  stx : syntax?
  ctx : (or/c #f syntax?)
```

A check-procedure that accepts any non-keyword term. It does not actually check that the term is a valid expression.

```
((check-stx-listof check) stx ctx) → (listof any/c)
  check : check-procedure
  stx : syntax?
  ctx : (or/c #f syntax?)
```

Lifts a check-procedure to accept syntax lists of whatever the original procedure accepted.

```
(check-stx-string stx ctx) → syntax?
  stx : syntax?
  ctx : (or/c #f syntax?)
```

A check-procedure that accepts syntax strings.

```
(check-stx-boolean stx ctx) → syntax?
  stx : syntax?
  ctx : (or/c #f syntax?)
```

A check-procedure that accepts syntax booleans.

3 Datum Pattern Matching

```
(require syntax/datum)        package: base
```

The syntax/datum library provides forms that implement the pattern and template language of syntax-case, but for matching and constructing datum values instead of syntax.

For most pattern-matching purposes, racket/match is a better choice than syntax/datum. The syntax/datum library is useful mainly for its template support (i.e., datum) and, to a lesser extent, its direct correspondence to syntax-case patterns.

```
(datum-case datum-expr (literal-id ...)
  clause ...)
(datum template)
```

Like syntax-case and syntax, but *datum-expr* in datum-case should produce a datum (i.e., plain S-expression) instead of a syntax object to be matched in *clause*s, and datum similarly produces a datum. Pattern variables bound in each *clause* of datum-case (or syntax-case, see below) are accessible via datum instead of syntax. When a *literal-id* appears in a *clause*'s pattern, it matches the corresponding symbol (using eq?).

Using datum-case and datum is similar to converting the input to syntax-case using datum->syntax and then wrapping each use of syntax with syntax->datum, but datum-case and datum do not create intermediate syntax objects, and they do not destroy existing syntax objects within the S-expression structure of *datum-expr*.

Example:

```
> (datum-case '(1 "x" -> y) (->)
    [(a ... -> b) (datum (b (+ a) ...))])
'(y (+ 1) (+ "x"))
```

The datum form also cooperates with syntax pattern variables such as those bound by syntax-case and attributes bound by syntax-parse (see §1.4.2 "Pattern Variables and Attributes" for more information about attributes). As one consequence, datum provides a convenient way of getting the list of syntax objects bound to a syntax pattern variable of depth 1. For example, the following expressions are equivalent, except that the datum version avoids creating and eliminating a superfluous syntax object wrapper:

```
> (with-syntax ([(x ...) #'(a b c)])
    (syntax->list #'(x ...)))
'(#<syntax:eval:3:0 a> #<syntax:eval:3:0 b> #<syntax:eval:3:0 c>)
> (with-syntax ([(x ...) #'(a b c)])
    (datum (x ...)))
'(#<syntax:eval:4:0 a> #<syntax:eval:4:0 b> #<syntax:eval:4:0 c>)
```

A template can also use multiple syntax or datum pattern variables and datum constants, and it can use the `~@` and `~?` template forms:

```
> (with-syntax ([(x ...) #'(a b c)])
    (with-datum ([(y ...) (list 1 2 3)])
      (datum ([x -> y] ...))))
'((#<syntax:eval:5:0 a> -> 1)
  (#<syntax:eval:5:0 b> -> 2)
  (#<syntax:eval:5:0 c> -> 3))
> (with-syntax ([(x ...) #'(a b c)])
    (with-datum ([(y ...) (list 1 2 3)])
      (datum ((~@ x y) ...))))
'(#<syntax:eval:6:0 a> 1 #<syntax:eval:6:0 b> 2 #<syntax:eval:6:0
c> 3)
```

See §1.4.2.1 "Attributes and `datum`" for examples of `~?` with `datum`.

If a datum variable is used in a syntax template, a compile-time error is raised.

Changed in version 7.8.0.9 of package `base`: Changed `datum` to cooperate with `syntax-case`, `syntax-parse`, etc.

```
(with-datum ([pattern datum-expr] ...)
  body ...+)
```

Analogous to `with-syntax`, but for `datum-case` and `datum` instead of `syntax-case` and `syntax`.

Example:

```
> (with-datum ([(a ...) '(1 2 3)]
               [(b ...) '("x" "y" "z")])
    (datum ((a b) ...)))
'((1 "x") (2 "y") (3 "z"))
```

```
(define/with-datum pattern datum-expr)
```

The definition form of `with-datum`. Analogous to `define/with-syntax`, but for `datum-case` and `datum`.

Examples:

```
> (define/with-datum ((x y) ...)
    '((a 1) (b 2) (c 3)))
> (datum ((x ...)
          (y ...)))
```

```
'((a b c) (1 2 3))
```

Added in version 8.2.0.8 of package base.

```
(quasidatum template)
(undatum expr)
(undatum-splicing expr)
```

Analogous to quasisyntax, unsyntax, and unsyntax-splicing.

Example:

```
> (with-datum ([(a ...) '(1 2 3)])
    (quasidatum ((undatum (- 1 1)) a ... (undatum (+ 2 2)))))
'(0 1 2 3 4)
```

4 Module-Processing Helpers

4.1 Reading Module Source Code

```
(require syntax/modread)        package: base
```

```
(with-module-reading-parameterization thunk) → any
  thunk : (-> any)
```

Calls *thunk* with all reader parameters reset to their default values.

```
(check-module-form stx
                   expected-module-sym
                   source-v)             → (or/c syntax? #f)
  stx : (or/c syntax? eof-object?)
  expected-module-sym : symbol?
  source-v : (or/c string? #f)
```

Inspects *stx* to check whether evaluating it will declare a module—at least if `module` is bound in the top-level to Racket's `module`. The syntax object *stx* can contain a compiled expression. Also, *stx* can be an end-of-file, on the grounds that `read-syntax` can produce an end-of-file.

The *expected-module-sym* argument is currently ignored. In previous versions, the module form *stx* was obliged to declare a module who name matched *expected-module-sym*.

If *stx* can declare a module in an appropriate top-level, then the `check-module-form` procedure returns a syntax object that certainly will declare a module (adding explicit context to the leading `module` if necessary) in any top-level. Otherwise, if *source-v* is not `#f`, a suitable exception is raised using the `write` form of the source in the message; if *source-v* is `#f`, `#f` is returned.

If *stx* is eof or eof wrapped as a syntax object, then an error is raised or `#f` is returned.

4.2 Getting Module Compiled Code

```
(require syntax/modcode)        package: base
```

```
(get-module-code path
                 [#:submodule-path submodule-path
                  #:sub-path compiled-subdir0
                  compiled-subdir
                  #:roots roots
                  #:compile compile-proc0
                  compile-proc
                  #:extension-handler ext-proc0
                  ext-proc
                  #:notify notify-proc
                  #:source-reader read-syntax-proc
                  #:rkt-try-ss? rkt-try-ss?
                  #:choose choose-proc])          → any
  path : path-string?
  submodule-path : (listof symbol?) = '()
  compiled-subdir0 : (or/c (and/c path-string? relative-path?)
                           (listof (and/c path-string? relative-path?)))
                   = (use-compiled-file-paths)
  compiled-subdir : (or/c (and/c path-string? relative-path?)
                          (listof (and/c path-string? relative-path?)))
                  = compiled-subdir0
  roots : (listof (or/c path-string? 'same))
        = (current-compiled-file-roots)
  compile-proc0 : (any/c . -> . any) = compile
  compile-proc : (any/c . -> . any) = compile-proc0
  ext-proc0 : (or/c #f (path? boolean? . -> . any)) = #f
  ext-proc : (or/c #f (path? boolean? . -> . any)) = ext-proc0
  notify-proc : (any/c . -> . any) = void
  read-syntax-proc : (any/c input-port? . -> . (or/c syntax? eof-object?))
                   = read-syntax
  rkt-try-ss? : boolean? = #t
  choose-proc : (or/c (-> path? path? path?         = #f
                          (or/c 'src 'zo 'so #f))
                      #f)
```

Returns a compiled expression for the declaration of the module specified by `path` and `submodule-path`, where `submodule-path` is empty for a root module or a list for a submodule.

The `roots`, `compiled-subdir`, `choose-proc`, and `rkt-try-ss?` and `submodule-path` arguments determine which file is consulted to find the compiled code. If the default values are provided, then this function uses the same logic as the default value of `current-load/use-compiled`. In more detail:

- If `submodule-path` is not the empty list, then the compiled code will never be located in a dynamic extension; instead the original source or a "zo" file will be used.

- The *rkt-try-ss?* argument defaults to #t. If it is not #f, then if *path* ends in ".rkt", then the corresponding file ending in ".ss" will be tried as well.
- The *choose-proc* argument is called with the original source file (which might have had its ending changed, c.f. the *rkt-try-ss?* argument) and two other paths that end with "zo" and "so" (but they are not necessarily the paths that get-module-code uses). If the *choose-proc* returns 'src, then compiled files are not used. If it returns any other result, the result is ignored. In previous versions of this function, the *choose-proc* offered more control over which file was used but it no longer does; the current interface is kept for backwards compatibility.
- The *compiled-subdir* argument defaults to (use-compiled-file-paths); it specifies the sub-directories to search for a compiled version of the module. If *compiled-subdir* is a list, then the first directory that contains a file with an appropriate name is used as the compiled file.
- The *roots* list specifies a compiled-file search path in the same way as the current-compiled-file-roots parameter; it defaults to the current value of current-compiled-file-roots.

The *compile-proc* argument defaults to compile. This procedure is used to compile module source if an already-compiled version is not available. If *submodule-path* is not '(), then *compile-proc* must return a compiled module form.

The *ext-proc* argument defaults to #f. If it is not #f, it must be a procedure of two arguments that is called when a native-code version of *path* should be used. In that case, the arguments to *ext-proc* are the path for the dynamic extension, and a boolean indicating whether the extension is a _loader file (#t) or not (#f).

If a dynamic extension is preferred or is the only file that exists, it is supplied to *ext-proc* when *ext-proc* is #f, or an exception is raised (to report that an extension file cannot be used) when *ext-proc* is #f.

If *notify-proc* is supplied, it is called for the file (source, ".zo" or dynamic extension) that is chosen.

If *read-syntax-proc* is provided, it is used to read the module from a source file (but not from a bytecode file).

Changed in version 6.90.0.7 of package base: Use (default-compiled-sub-path) for the default value of *compiled-subdir*.
Changed in version 8.6.0.12: Generalize the #:sub-path argument, change #:sub-path's default to (use-compiled-file-paths), and pay less attention to the #:choose argument.

```
(get-module-path  path
                  #:submodule? submodule?
                 [#:sub-path compiled-subdir0
                  compiled-subdir
                  #:roots roots
                  #:rkt-try-ss? rkt-try-ss?
                  #:choose choose-proc])
 → path? (or/c 'src 'zo 'so)
  path : path-string?
  submodule? : boolean?
  compiled-subdir0 :  (or/c (and/c path-string? relative-path?)
                            (listof (and/c path-string? relative-path?)))
                   = (use-compiled-file-paths)
  compiled-subdir :  (or/c (and/c path-string? relative-path?)
                           (listof (and/c path-string? relative-path?)))
                  = compiled-subdir0
  roots :  (listof (or/c path-string? 'same))
         = (current-compiled-file-roots)
  rkt-try-ss? : boolean? = #t
  choose-proc : any/c = #f
```

Produces two values. The first is the path of the latest source or compiled file for the module specified by *path*; this result is the path of the file that `get-module-code` would read to produce a compiled module expression. The second value is `'src`, `'zo`, or `'so`, depending on whether the first value represents a Racket source file, a compiled bytecode file, or a native library file.

The *compiled-subdir*, *roots*, *choose-proc*, and *rkt-try-ss?* arguments are interpreted the same as by `get-module-code`.

The *submodule?* argument represents whether the desired module is a submodule of the one specified by *path*. When *submodule?* is true, the result path never refers to a dynamic extension and the result symbol is never `'so`, as native libraries cannot provide submodules.

Changed in version 6.90.0.7 of package `base`: Use `(default-compiled-sub-path)` for the default value of *compiled-subdir*.
Changed in version 8.6.0.12: Generalize the `#:sub-path` argument, change `#:sub-path`'s default to `(use-compiled-file-paths)`, and pay less attention to the `#:choose` argument.

```
(default-compiled-sub-path) → path-string?
```

If `(use-compiled-file-paths)` is not `'()`, returns the first element of the list. Otherwise, returns `"compiled"`.

This function used to provide the default for the `#:sub-path` argument to `get-module-code` and `get-module-path`, but it is no longer used by this library.

Added in version 6.90.0.7 of package base.

```
(get-metadata-path  path
                   [#:roots roots]
                    sub-path ...+) → path?
  path : path-string?
  roots : (listof (or/c path-string? 'same))
        = (current-compiled-file-roots)
  sub-path : (or/c path-string? 'same)
```

Constructs the path used to store compilation metadata for a source file stored in the directory *path*. The argument *roots* specifies the possible root directories to consider and to search for an existing file. The *sub-path* arguments specify the subdirectories and filename of the result relative to the chosen root. For example, the compiled ".zo" file for "/path/to/source.rkt" might be stored in (get-metadata-path (build-path "/path/to") "compiled" "source_rkt.zo").

```
(moddep-current-open-input-file)
 → (path-string? . -> . input-port?)
(moddep-current-open-input-file proc) → void?
  proc : (path-string? . -> . input-port?)
```

A parameter whose value is used like open-input-file to read a module source or ".zo" file.

```
(struct exn:get-module-code exn:fail (path)
    #:extra-constructor-name make-exn:get-module-code)
  path : path?
```

An exception structure type for exceptions raised by get-module-code.

4.3 Resolving Module Paths to File Paths

```
(require syntax/modresolve)       package: base
```

```
(resolve-module-path  module-path-v
                     [rel-to-path-v])
 → (or/c path? symbol?
         (cons/c 'submod (cons/c (or/c path? symbol?) (listof symbol?))))
  module-path-v : module-path?
  rel-to-path-v : (or/c #f path-string? (-> any)) = #f
```

Resolves a module path to filename path. The module path is resolved relative to *rel-to-path-v* if it is a path string (assumed to be for a file), to the directory result of calling the thunk if it is a thunk, or to the current directory otherwise.

When *module-path-v* refers to a module using a collection-based path, resolution invokes the current module name resolver, but without loading the module even if it is not declared. Beware that concurrent resolution in namespaces that share a module registry can create race conditions when loading modules; see also namespace-call-with-registry-lock.

```
(resolve-module-path-index  module-path-index
                           [rel-to-path-v])
 → (or/c path? symbol?
        (cons/c 'submod (cons/c (or/c path? symbol?) (listof symbol?))))
  module-path-index : module-path-index?
  rel-to-path-v : (or/c #f path-string? (-> any)) = #f
```

Like resolve-module-path but the input is a module path index; in this case, the *rel-to-path-v* base is used where the module path index contains the "self" index. If *module-path-index* depends on the "self" module path index, then an exception is raised unless *rel-to-path-v* is a path string.

See module-path-index-resolve.

Examples:

```
> (resolve-module-path-index
   (module-path-index-join 'racket #f))
#<path:/Users/robby/snapshot/racket/racket/collects/racket/main.rkt>
> (resolve-module-path-index
   (module-path-index-join "apple.rkt" #f))
#<path:/Users/robby/snapshot/racket/build/docs/share/pkgs/racket-
doc/syntax/apple.rkt>
> (resolve-module-path-index
   (module-path-index-join '(submod "." test) #f)
   (string->path "banana.rkt"))
'(submod #<path:banana.rkt> test)
```

4.4 Simplifying Module Paths

```
(require syntax/modcollapse)        package: base
```

```
(collapse-module-path module-path-v
                      rel-to-module-path-v) → module-path?
  module-path-v : module-path?
  rel-to-module-path-v : (or/c module-path?
                               (-> module-path?))
```

Returns a "simplified" module path by combining *module-path-v* with *rel-to-module-path-v*, where the latter must have one of the following forms: a `'(lib ....)` or symbol module path; a `'(file ....)` module path; a `'(planet ....)` module path; a path;

`'(quote symbol)`; a `'(submod base symbol ...)` module path where *base* would be allowed; or a thunk to generate one of those.

The result can be a path if *module-path-v* contains a path element that is needed for the result, or if *rel-to-module-path-v* is a non-string path that is needed for the result. Similarly, the result can be `'submod` wrapping a path. Otherwise, the result is a module path (in the sense of `module-path?`) that is not a plain filesystem path.

When the result is a `'lib` or `'planet` module path, it is normalized so that equivalent module paths are represented by `equal?` results. When the result is a `'submod` module path, it contains only symbols after the base module path, and the base is normalized in the case of a `'lib` or `'planet` base.

Examples:

```
> (collapse-module-path "m.rkt"  '(lib "n/main.rkt"))
'(lib "n/m.rkt")
> (collapse-module-path '(submod "." x)  '(lib "n/main.rkt"))
'(submod (lib "n/main.rkt") x)
> (collapse-module-path '(submod "." x)  '(submod (lib "n/main.rkt") y))
'(submod (lib "n/main.rkt") y x)
```

```
(collapse-module-path-index module-path-index
                            rel-to-module-path-v)
 → module-path?
  module-path-index : module-path-index?
  rel-to-module-path-v : (or/c module-path?
                               (-> module-path?))
(collapse-module-path-index module-path-index)
 → (or/c module-path? #f)
  module-path-index : module-path-index?
```

Like `collapse-module-path` when given two arguments, but the input is a module path index; in this case, the *rel-to-module-path-v* base is used where the module path index contains the “self” index (see `module-path-index-split`).

When given a single argument, `collapse-module-path-index` returns a module path that is relative if the given module path index is relative, except that it returns `#f` if its argument is the “self” module path index. A resulting module path is not necessarily normalized.

Changed in version 6.1.1.8 of package `base`: Added the one-argument variant for collapsing a relative module path index.
Changed in version 6.9.0.5: Added support for the “self” module path index as the only argument, which meant extending the result contract to include `#f`

4.5 Inspecting Modules and Module Dependencies

```
(require syntax/moddep)        package: base
```

Re-exports `syntax/modread`, `syntax/modcode`, `syntax/modcollapse`, and `syntax/modresolve`, in addition to the following:

```
(show-import-tree  module-path-v
                  [#:dag? dag?
                   #:path-to path-to-module-path-v
                   #:show show])                    → void?
  module-path-v : module-path?
  dag? : any/c = #f
  path-to-module-path-v : (or/c #f module-path?) = #f
  show : (string? any/c string? (or/c #f exact-integer?) . -> . any)
       = (lambda (indent path require-mode phase)
           (printf "~a~a~a ~a\n" indent path require-mode phase))
```

A debugging aid that prints (by default) the import hierarchy starting from a given module path. Supply an alternate *show* function to handle each path instead of having it printed; the second argument is a result of `resolved-module-path-name`.

If *dag?* is true, then a module is passed to *show* only the first time is encountered in the hierarchy at a given phase.

If *path-to-module-path-v* is a module path, then only the spines of the tree that reach *path-to-module-path-v* are shown.

Changed in version 6.12.0.4 of package `base`: Added the `#:dag?` and `#:path-to` arguments.
Changed in version 7.0.0.10: Added the `#:show` argument.

4.6 Wrapping Module-Body Expressions

```
(require syntax/wrap-modbeg)        package: base
```

Added in version 6.0.0.1 of package `base`.

```
(make-wrapping-module-begin  wrap-form
                            [module-begin-form])
 → (syntax? . -> . syntax?)
  wrap-form : syntax?
  module-begin-form : syntax? = #'#%plain-module-begin
```

Provided `for-syntax`.

Constructs a function that is suitable for use as a `#%module-begin` replacement, particularly to replace the facet of `#%module-begin` that wraps each top-level expression to print the expression's result(s).

The function takes a syntax object and returns a syntax object using *`module-begin-form`*. Assuming that *`module-begin-form`* resembles `#%plain-module-begin`, each top-level expression *`expr`* will be wrapped as `(`*`wrap-form expr`*`)`, while top-level declarations (such as `define-values` and `require` forms) are left as-is. Expressions are detected after macro expansion and `begin` splicing, and expansion is interleaved with declaration processing as usual.

5 Macro Transformer Helpers

5.1 Extracting Inferred Names

```
(require syntax/name)      package: base
```

```
(syntax-local-infer-name stx [use-local?]) → any/c
  stx : syntax?
  use-local? : any/c = #t
```

Similar to `syntax-local-name`, except that *`stx`* is checked for an `'inferred-name` property (which overrides any inferred name). If neither `syntax-local-name` nor `'inferred-name` produce a name, or if the `'inferred-name` property value is `#<void>`, then a name is constructed from the source-location information in *`stx`*, if any. If no name can be constructed, the result is `#f`.

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

If *`use-local?`* is `#f`, then `syntax-local-name` is not used. Provide *`use-local?`* as `#f` to construct a name for a syntax object that is not an expression currently being expanded.

5.2 Support for `local-expand`

```
(require syntax/context)      package: base
```

```
(build-expand-context v) → list?
  v : (or/c symbol? list?)
```

Returns a list suitable for use as a context argument to `local-expand` for an internal-definition context. The *`v`* argument represents the immediate context for expansion. The context list builds on `(syntax-local-context)` if it is a list.

```
(generate-expand-context [liberal-definitions?]) → list?
  liberal-definitions? : boolean? = #f
```

Calls `build-expand-context` with a generated unique value. When *`liberal-definitions?`* is true, the value is an instance of a structure type with a true value for the `prop:liberal-define-context` property.

5.3 Parsing `define`-like Forms

`(require syntax/define)` package: `base`

```
(normalize-definition  defn-stx
                       lambda-id-stx
                      [check-context?
                       opt+kws?])      → identifier? syntax?
  defn-stx : syntax?
  lambda-id-stx : identifier?
  check-context? : boolean? = #t
  opt+kws? : boolean? = #f
```

Takes a definition form whose shape is like `define` (though possibly with a different name) and returns two values: the defined identifier and the right-hand side expression.

To generate the right-hand side, this function may need to insert uses of `lambda`. The *`lambda-id-stx`* argument provides a suitable `lambda` identifier.

If the definition is ill-formed, a syntax error is raised. If *`check-context?`* is true, then a syntax error is raised if `(syntax-local-context)` indicates that the current context is an expression context. The default value of *`check-context?`* is `#t`.

If *`opt+kws?`* is `#t`, then arguments of the form `[id expr]`, `keyword id`, and `keyword [id expr]` are allowed, and they are preserved in the expansion.

```
(normalize-definition/mk-rhs defn-stx
                             lambda-id-stx
                             check-context?
                             opt+kws?
                             err-no-body?)
 → identifier? (-> syntax? syntax?) syntax?
  defn-stx : syntax?
  lambda-id-stx : identifier?
  check-context? : boolean?
  opt+kws? : boolean?
  err-no-body? : boolean?
```

The helper for `normalize-definition` that produces three values: the defined identifier, a function that takes the syntax of the body and produces syntax that has the expected binding structure, and finally the right-hand side expression that `normalize-definition` gives to the previous function.

If *`err-no-body?`* is true, then there must be a right-hand side expression or else it is a syntax error. The *`err-no-body?`* argument is true for uses of `normalize-definition`.

Added in version 6.1.1.8 of package `base`.

5.4 Flattening begin Forms

```
(require syntax/flatten-begin)        package: base
```

```
(flatten-begin stx) → (listof syntax?)
  stx : syntax?
```

Extracts the sub-expressions from *stx*, assuming that it is a `begin` form. Reports an error if *stx* does not have the right shape (i.e., a syntax list). The resulting syntax objects have annotations transferred from *stx* using `syntax-track-origin`.

Examples:

```
> (flatten-begin #'(begin 1 2 3))
'(#<syntax:eval:2:0 1> #<syntax:eval:2:0 2> #<syntax:eval:2:0 3>)
> (flatten-begin #'(begin (begin 1 2) 3))
'(#<syntax:eval:3:0 (begin 1 2)> #<syntax:eval:3:0 3>)
> (flatten-begin #'(+ (- 1 2) 3))
'(#<syntax:eval:4:0 (- 1 2)> #<syntax:eval:4:0 3>)
```

```
(flatten-all-begins stx) → (listof syntax?)
  stx : syntax?
```

Extracts the sub-expressions from a `begin` form and recursively flattens `begin` forms nested in the original one. An error will be reported if *stx* is not a `begin` form. The resulting syntax objects have annotations transferred from *stx* using `syntax-track-origin`.

Examples:

```
> (flatten-all-begins #'(begin 1 2 3))
'(#<syntax:eval:5:0 1> #<syntax:eval:5:0 2> #<syntax:eval:5:0 3>)
> (flatten-all-begins #'(begin (begin 1 2) 3))
'(#<syntax:eval:6:0 1> #<syntax:eval:6:0 2> #<syntax:eval:6:0 3>)
```

Added in version 6.1.0.3 of package `base`.

5.5 Expanding define-struct-like Forms

```
(require syntax/struct)        package: base
```

```
(parse-define-struct stx orig-stx) → identifier?
                                     (or/c identifier? #f)
                                     (listof identifier?)
                                     syntax?
```

```
stx : syntax?
orig-stx : syntax?
```

Parses *stx* as a `define-struct` form, but uses *orig-stx* to report syntax errors (under the assumption that *orig-stx* is the same as *stx*, or that they at least share sub-forms). The result is four values: an identifier for the struct type name, a identifier or `#f` for the super-name, a list of identifiers for fields, and a syntax object for the inspector expression.

```
(build-struct-names  name-id
                     field-ids
                    [#:constructor-name ctr-name]
                     omit-sel?
                     omit-set?
                    [src-stx])
 → (listof identifier?)
  name-id : identifier?
  field-ids : (listof identifier?)
  ctr-name : (or/c identifier? #f) = #f
  omit-sel? : boolean?
  omit-set? : boolean?
  src-stx : (or/c syntax? #f) = #f
```

Generates the names bound by `define-struct` given an identifier for the struct type name and a list of identifiers for the field names. The result is a list of identifiers:

- `struct:`*`name-id`*
- *`ctr-name`*, or `make-`*`name-id`* if *`ctr-name`* is `#f`
- *`name-id?`*
- *`name-id-field`*, for each *`field`* in *`field-ids`*.
- `set-`*`name-id-field`*`!` (getter and setter names alternate).
- ....

If *`omit-sel?`* is true, then the selector names are omitted from the result list. If *`omit-set?`* is true, then the setter names are omitted from the result list.

The default *`src-stx`* is `#f`; it is used to provide a source location to the generated identifiers.

```
(build-struct-generation  name-id
                          field-ids
                         [#:constructor-name ctr-name]
                          omit-sel?
                          omit-set?
                         [super-type
                          prop-value-list
                          immutable-k-list])
 → (listof identifier?)
  name-id : identifier?
  field-ids : (listof identifier?)
  ctr-name : (or/c identifier? #f) = #f
  omit-sel? : boolean?
  omit-set? : boolean?
  super-type : any/c = #f
  prop-value-list : list? = '(list)
  immutable-k-list : list? = '(list)
```

Takes the same arguments as `build-struct-names` and generates an S-expression for code using `make-struct-type` to generate the structure type and return values for the identifiers created by `build-struct-names`. The optional *super-type*, *prop-value-list*, and *immutable-k-list* parameters take S-expressions that are used as the corresponding argument expressions to `make-struct-type`.

```
(build-struct-generation*  all-name-ids
                           name-id
                           field-ids
                          [#:constructor-name ctr-name]
                           omit-sel?
                           omit-set?
                          [super-type
                           prop-value-list
                           immutable-k-list])
 → (listof identifier?)
  all-name-ids : (listof identifier?)
  name-id : identifier?
  field-ids : (listof identifier?)
  ctr-name : (or/c identifier? #f) = #f
  omit-sel? : boolean?
  omit-set? : boolean?
  super-type : any/c = #f
  prop-value-list : list? = '(list)
  immutable-k-list : list? = '(list)
```

Like `build-struct-generation`, but given the names produced by `build-struct-names`, instead of re-generating them.

```
(build-struct-expand-info  name-id
                           field-ids
                          [#:omit-constructor? no-ctr?
                           #:constructor-name ctr-name
                           #:omit-struct-type? no-type?]
                           omit-sel?
                           omit-set?
                           base-name
                           base-getters
                           base-setters)              → any
  name-id : identifier?
  field-ids : (listof identifier?)
  no-ctr? : any/c = #f
  ctr-name : (or/c identifier? #f) = #f
  no-type? : any/c = #f
  omit-sel? : boolean?
  omit-set? : boolean?
  base-name : (or/c identifier? boolean?)
  base-getters : (listof (or/c identifier? #f))
  base-setters : (listof (or/c identifier? #f))
```

Takes mostly the same arguments as `build-struct-names`, plus a parent identifier/`#t`/`#f` and a list of accessor and mutator identifiers (possibly ending in `#f`) for a parent type, and generates an S-expression for expansion-time code to be used in the binding for the structure name.

If *no-ctr?* is true, then the constructor name is omitted from the expansion-time information. Similarly, if *no-type?* is true, then the structure-type name is omitted.

A `#t` for the *base-name* means no super-type, `#f` means that the super-type (if any) is unknown, and an identifier indicates the super-type identifier.

```
(struct-declaration-info? v) → boolean?
  v : any/c
```

Returns `#t` if `x` has the shape of expansion-time information for structure type declarations, `#f` otherwise. See §5.7 "Structure Type Transformer Binding".

```
(generate-struct-declaration  orig-stx
                              name-id
                              super-id-or-false
                              field-id-list
                              current-context
                              make-make-struct-type
                             [omit-sel?
                              omit-set?])          → syntax?
```

```
orig-stx : syntax?
name-id : identifier?
super-id-or-false : (or/c identifier? #f)
field-id-list : (listof identifier?)
current-context : any/c
make-make-struct-type : procedure?
omit-sel? : boolean? = #f
omit-set? : boolean? = #f
```

This procedure implements the core of a `define-struct` expansion.

The `generate-struct-declaration` procedure is called by a macro expander to generate the expansion, where the *`name-id`*, *`super-id-or-false`*, and *`field-id-list`* arguments provide the main parameters. The *`current-context`* argument is normally the result of `syntax-local-context`. The *`orig-stx`* argument is used for syntax errors. The optional *`omit-sel?`* and *`omit-set?`* arguments default to `#f`; a `#t` value suppresses definitions of field selectors or mutators, respectively.

The `make-struct-type` procedure is called to generate the expression to actually create the struct type. Its arguments are *`orig-stx`*, `name-id-stx`, `defined-name-stxes`, and `super-info`. The first two are as provided originally to `generate-struct-declaration`, the third is the set of names generated by `build-struct-names`, and the last is super-struct info obtained by resolving *`super-id-or-false`* when it is not `#f`, `#f` otherwise.

The result should be an expression whose values are the same as the result of `make-struct-type`. Thus, the following is a basic *`make-make-struct-type`*:

```
(lambda (orig-stx name-stx defined-name-stxes super-info)
  #`(make-struct-type '#,name-stx
                       #,(and super-info (list-ref super-info 0))
                       #,(/ (- (length defined-name-stxes) 3) 2)
                       0 #f))
```

but an actual *`make-make-struct-type`* will likely do more.

5.6 Resolving `include`-like Paths

```
(require syntax/path-spec)        package: base
```

```
(resolve-path-spec path-spec-stx
                   source-stx
                   expr-stx)      → complete-path?
  path-spec-stx : syntax?
  source-stx : syntax?
  expr-stx : syntax?
```

Resolves the syntactic path specification *path-spec-stx* as for `include`.

The *source-stx* specifies a syntax object whose source-location information determines relative-path resolution. The *expr-stx* is used for reporting syntax errors.

5.7 Controlling Syntax Templates

```
(require syntax/template)      package: base
```

```
(transform-template template-stx
                    #:save save-proc
                    #:restore-stx restore-proc-stx
                   [#:leaf-save leaf-save-proc
                    #:leaf-restore-stx leaf-restore-proc-stx
                    #:leaf-datum-stx leaf-datum-proc-stx
                    #:pvar-save pvar-save-proc
                    #:pvar-restore-stx pvar-restore-stx
                    #:cons-stx cons-proc-stx
                    #:ellipses-end-stx ellipses-end-stx
                    #:constant-as-leaf? constant-as-leaf?])
 → syntax?
  template-stx : syntax?
  save-proc : (syntax? . -> . any/c)
  restore-proc-stx : syntax?
  leaf-save-proc : (syntax? . -> . any/c) = save-proc
  leaf-restore-proc-stx : syntax? = #'(lambda (data stx) stx)
  leaf-datum-proc-stx : syntax? = #'(lambda (v) v)
  pvar-save-proc : (identifier? . -> . any/c) = (lambda (x) #f)
  pvar-restore-stx : syntax? = #'(lambda (d stx) stx)
  cons-proc-stx : syntax? = #'cons
  ellipses-end-stx : syntax? = #'values
  constant-as-leaf? : boolean? = #f
```

Produces an representation of an expression similar to `` #`(syntax #,template-stx) ``, but functions like *save-proc* can collect information that might otherwise be lost by `syntax` (such as properties when the syntax object is marshaled within bytecode), and run-time functions like the one specified by *restore-proc-stx* can use the saved information or otherwise process the syntax object that is generated by the template.

The *save-proc* is applied to each syntax object in the representation of the original template (i.e., in *template-stx*). If *constant-as-leaf?* is `#t`, then *save-proc* is applied only to syntax objects that contain at least one pattern variable in a sub-form. The result of *save-proc* is provided back as the first argument to *restore-proc-stx*, which indicates a function with a contract `(-> any/c syntax any/c any/c)`; the second argument to *restore-proc-stx* is the syntax object that syntax generates, and the last argument is a datum that have been processed recursively (by functions such as *restore-proc-stx*) and that normally would be converted back to a syntax object using the second argument's context, source, and properties. Note that *save-proc* works at expansion time (with respect to the template form), while *restore-proc-stx* indicates a function that is called at run time (for the template form), and the data that flows from *save-proc* to *restore-proc-stx* crosses phases via quote.

The *leaf-save-proc* and *leaf-restore-proc-stx* procedures are analogous to *save-proc* and *restore-proc-stx*, but they are applied to leaves, so there is no third argument for recursively processed sub-forms. The function indicated by *leaf-restore-proc-stx* should have the contract (-> any/c syntax? any/c).

The *leaf-datum-proc-stx* procedure is applied to leaves that are not syntax objects, which can happen because pairs and the empty list are not always individually wrapped as syntax objects. The function should have the contract (-> any/c any/c). When *constant-as-leaf?* is #f, the only possible argument to the procedure is null.

The pvar-save and *pvar-restore-stx* procedures are analogous to *save-proc* and *restore-proc-stx*, but they are applied to pattern variables. The *pvar-restore-stx* procedure should have the contract (-> any/c syntax? any/c), where the second argument corresponds to the substitution of the pattern variable.

The *cons-proc-stx* procedure is used to build intermediate pairs, including pairs passed to *restore-proc-stx* and pairs that do not correspond to syntax objects.

The *ellipses-end-stx* procedure is an extra filter on the syntax object that follows a sequence of ... ellipses in the template. The procedure should have the contract (-> any/c any/c).

The following example illustrates a use of transform-template to implement a syntax/shape form that preserves the 'paren-shape property from the original template, even if the template code is marshaled within bytecode.

```
(define-for-syntax (get-shape-prop stx)
  (syntax-property stx 'paren-shape))

(define (add-shape-prop v stx datum)
  (syntax-property (datum->syntax stx datum stx stx stx)
                   'paren-shape
                   v))

(define-syntax (syntax/shape stx)
  (syntax-case stx ()
    [(_ tmpl)
     (transform-template #'tmpl
                         #:save get-shape-prop
```

```
                    #:restore-stx #'add-shape-prop)]))
```

5.8 Creating Macro Transformers

```
(require syntax/transformer)      package: base
```

```
(make-variable-like-transformer  reference-stx
                                [setter-stx])
 → (and/c set!-transformer? (-> syntax? syntax?))
  reference-stx : (or/c syntax? (-> identifier? syntax?))
  setter-stx : (or/c syntax? (-> syntax? syntax?) #f) = #f
```

Creates a transformer that replaces references to the macro identifier with *reference-stx*. Uses of the macro in operator position are interpreted as an application with *reference-stx* as the function and the arguments as given. If the *reference-stx* is a procedure, it is applied to the macro identifier.

If the macro identifier is used as the target of a `set!` form, then the `set!` form expands into the application of *setter-stx* to the `set!` expression's right-hand side, if *setter-stx* is syntax; otherwise, the identifier is considered immutable and a syntax error is raised. If *setter-stx* is a procedure, it is applied to the entire `set!` expression.

Examples:

```
> (define the-box (box add1))
> (define-syntax op
    (make-variable-like-transformer
     #'(unbox the-box)
     #'(lambda (v) (set-box! the-box v))))
> (op 5)
6
> (set! op 0)
> op
0
```

Added in version 6.3 of package `base`.

```
(make-expression-transformer transformer)
 → (-> syntax? syntax?)
  transformer : (-> syntax? syntax?)
```

Creates a transformer derived from *transformer* that ensures it expands in an expression context. When invoked in an expression context, it calls *transformer*. When invoked in any other context, the new transformer wraps the argument syntax with `#%expression`.

Added in version 7.7.0.9 of package `base`.

5.9 Applying Macro Transformers

```
(require syntax/apply-transformer)      package: base
```

```
(local-apply-transformer transformer
                         stx
                         context
                         intdef-ctx) → syntax?
  transformer : (or/c (-> syntax? syntax?) set!-transformer?)
  stx : syntax?
  context : (or/c 'expression 'top-level 'module 'module-begin list?)
  intdef-ctx : (or/c internal-definition-context?
                     #f
                     (listof internal-definition-context?))
```

For backwards compatibility only; `syntax-local-apply-transformer` is preferred.

Applies *transformer* as a syntax transformer to *stx* in the current expansion context. The result is similar to expanding a use of an identifier bound as a syntax transformer bound to *transformer* with `local-expand`, except that expansion is guaranteed to stop after applying a single macro transformation (assuming *transformer* does not explicitly force further recursive expansion).

Unlike simply applying *transformer* to *stx* directly, using `local-apply-transformer` introduces the appropriate use-site scope and macro-introduction scope that would be added by the expander.

The *context* and *intdef-ctx* arguments are treated the same way as the corresponding arguments to `local-expand`.

Added in version 6.90.0.29 of package `base`.

6 Reader Helpers

6.1 Raising exn:fail:read

```
(require syntax/readerr)        package: base
```

```
(raise-read-error  msg-string
                   source
                   line
                   col
                   pos
                   span
                  [#:extra-srclocs extra-srclocs]) → any
  msg-string : string?
  source : any/c
  line : (or/c exact-positive-integer? #f)
  col : (or/c exact-nonnegative-integer? #f)
  pos : (or/c exact-positive-integer? #f)
  span : (or/c exact-nonnegative-integer? #f)
  extra-srclocs : (listof srcloc?) = '()
```

Creates and raises an `exn:fail:read` exception, using *`msg-string`* as the base error message.

Source-location information is added to the error message using the last five arguments and the *`extra-srclocs`* (if the `error-print-source-location` parameter is set to `#t`). The *`source`* argument is an arbitrary value naming the source location—usually a file path string. Each of the *`line`*, *`pos`* arguments is `#f` or a positive exact integer representing the location within *`source`* (as much as known), *`col`* is a non-negative exact integer for the source column (if known), and *`span`* is `#f` or a non-negative exact integer for an item range starting from the indicated position.

The usual location values should point at the beginning of whatever it is you were reading, and the span usually goes to the point the error was discovered.

```
(raise-read-eof-error msg-string
                      source
                      line
                      col
                      pos
                      span)      → any
  msg-string : string?
  source : any/c
  line : (or/c exact-positive-integer? #f)
  col : (or/c exact-nonnegative-integer? #f)
```

```
pos : (or/c exact-positive-integer? #f)
span : (or/c exact-nonnegative-integer? #f)
```

Like `raise-read-error`, but raises `exn:fail:read:eof` instead of `exn:fail:read`.

6.2 Module Reader

```
(require syntax/module-reader)    package: base
```

See also §17.3 "Defining new `#lang` Languages" in *The Racket Guide*.

The `syntax/module-reader` library provides support for defining `#lang` readers. It is normally used as a module language, though it may also be `required` to get `make-meta-reader`. It provides all of the bindings of `racket/base` other than `#%module-begin`.

```
(#%module-begin module-path)
(#%module-begin module-path reader-option ... form ....)
(#%module-begin             reader-option ... form ....)

reader-option = #:read        read-expr
              | #:read-syntax read-syntax-expr
              | #:whole-body-readers? whole?-expr
              | #:wrapper1    wrapper1-expr
              | #:wrapper2    wrapper2-expr
              | #:module-wrapper module-wrapper-expr
              | #:language    lang-expr
              | #:info        info-expr
              | #:interaction-info interaction-info-expr
              | #:language-info language-info-expr
```

```
read-expr : (input-port? . -> . any/c)
read-syntax-expr : (any/c input-port? . -> . any/c)
whole?-expr : any/c
wrapper1-expr : (or/c ((-> any/c) . -> . any/c)
                      ((-> any/c) boolean? . -> . any/c))
wrapper2-expr : (or/c (input-port? (input-port? . -> . any/c)
                        . -> . any/c)
                      (input-port? (input-port? . -> . any/c)
                       boolean? . -> . any/c))
module-wrapper-expr : (or/c ((-> any/c) . -> . any/c)
                            ((-> any/c) boolean? . -> . any/c))
info-expr : (symbol? any/c (symbol? any/c . -> . any/c) . -> . any/c)
interaction-info-expr : (or/c (symbol? any/c . -> . any/c) #f)
language-info-expr : (or/c (vector/c module-path? symbol? any/c) #f)
lang-expr : (or/c module-path?
                  (and/c syntax? (compose module-path? syntax->datum))
                  procedure?)
```

In its simplest form, the body of a module written with `syntax/module-reader` contains just a module path, which is used in the language position of read modules. For example, a module *something*`/lang/reader` implemented as

```
(module reader syntax/module-reader
  module-path)
```

creates a reader such that a module source

```
#lang something
....
```

is read as

```
(module name-id module-path
  (#%module-begin ....))
```

where *name-id* is derived from the source input port's name in the same way as for `#lang s-exp`.

Keyword-based *reader-option*s allow further customization, as listed below. Additional *form*s are as in the body of `racket/base` module; they can import bindings and define identifiers used by the *reader-option*s.

- `#:read` and `#:read-syntax` (both or neither must be supplied) specify alternate readers for parsing the module body—replacements `read` and `read-syntax`, respectively. Normally, the replacements for `read` and `read-syntax` are applied repeatedly to the module source until `eof` is produced, but see also `#:whole-body-readers?`.

  Unless `#:whole-body-readers?` specifies a true value, the repeated use of `read` or `read-syntax` is `parameterize`d to set `read-accept-lang` to `#f`, which disables nested uses of `#lang`.

  See also `#:wrapper1` and `#:wrapper2`, which support simple parameterization of readers rather than wholesale replacement.

- `#:whole-body-readers?` specified as true indicates that the `#:read` and `#:read-syntax` functions each produce a list of S-expressions or syntax objects for the module content, so that each is applied just once to the input stream.

  If the resulting list contains a single form that starts with the symbol `'#%module-begin` (or a syntax object whose datum is that symbol), then the first item is used as the module body; otherwise, a `'#%module-begin` (symbol or identifier) is added to the beginning of the list to form the module body.

- `#:wrapper1` specifies a function that controls the dynamic context in which the `read` and `read-syntax` functions are called. A `#:wrapper1`-specified function must accept a thunk, and it normally calls the thunk to produce a result while `parameterize`-ing the call. Optionally, a `#:wrapper1`-specified function can accept a boolean that indicates whether it is used in `read` (`#f`) or `read-syntax` (`#t`) mode.

  For example, a language like `racket/base` but with case-insensitive reading of symbols and identifiers can be implemented as

  ```
  (module reader syntax/module-reader
    racket/base
    #:wrapper1 (lambda (t)
                 (parameterize ([read-case-sensitive #f])
                   (t))))
  ```

  Using a readtable, you can implement languages that are extensions of plain S-expressions.

- `#:wrapper2` is like `#:wrapper1`, but a `#:wrapper2`-specified function receives the input port to be read, and the function that it receives accepts an input port (usually, but not necessarily the same input port). A `#:wrapper2`-specified function can optionally accept an boolean that indicates whether it is used in `read` (`#f`) or `read-syntax` (`#t`) mode.

- `#:module-wrapper` specifies a function that controls the dynamic context in which the overall `module` form is produced, including calls to the `read` and `read-syntax` functions and to any `#:wrapper1` and `#:wrapper2` functions. The `#:module-wrapper`-specified function must accept a thunk, and it can optionally accept a boolean that indicates whether it is used in `read` (`#f`) or `read-syntax` (`#t`) mode.

While a `#:wrapper1`-specified or `#:wrapper2`-specified function sees only individual forms within the read module, a `#:module-wrapper`-specified function sees the entire result `module` form (via the result of its thunk argument).

- `#:info` specifies an implementation of reflective information that is used by external tools to manipulate the *source* of modules in the language *`something`*. For example, DrRacket uses information from `#:info` to determine the style of syntax coloring that it should use for editing a module's source.

  The `#:info` specification should be a function of three arguments: a symbol indicating the kind of information requested (as defined by external tools), a default value that normally should be returned if the symbol is not recognized, and a default-filtering function that takes the first two arguments and returns a result.

  The expression after `#:info` is placed into a context where `language-module` and `language-data` are bound. The `language-module` identifier is bound to the *`module-path`* that is used for the read module's language as written directly or as determined through `#:language`. The `language-data` identifier is bound to the second result from `#:language`, or `#f` by default.

  The default-filtering function passed to the `#:info` function is intended to provide support for information that `syntax/module-reader` can provide automatically. Currently, it recognizes only the `'module-language` key, for which it returns `language-module`; it returns the given default value for any other key.

  In the case of the DrRacket syntax-coloring example, DrRacket supplies `'color-lexer` as the symbol argument, and it supplies `#f` as the default. The default-filtering argument (i.e., the third argument to the `#:info` function) currently just returns the default for `'color-lexer`.

- `#:interaction-info` specifies an implementation of reflective information that is used by external tools for interactive evaluation. When *`interaction-info-expr`* produces a function, the function accepts two arguments: a symbol a symbol indicating the kind of information requested (as defined by external tools), and a default value that normally should be returned if the symbol is not recognized.

  As long as *`interaction-info-expr`* is specified, *`module-path`* is specified, or *`lang-expr`* is literally a `quote` form, then `get-interaction-info` is defined an exported. Normally, interaction information is used by setting `current-interaction-info` to a vector with the enclosing module's path as its first element and `'get-interaction-info` as its second element. The third element of the vector is an argument to `get-interaction-info`, and it is bound as `language-data` in *`interaction-info-expr`*.

  If *`interaction-info-expr`* is omitted or is literally `#f`, and if *`module-path`* is specified or *`lang-expr`* is literally a `quote` form, then `get-interaction-info` is automatically defined to use the same implementation as `#:info`, but instantiated with `language-module` as `#f`.

- `#:language-info` specifies an implementation of reflective information that is used by external tools to manipulate the module in the language *`something`* in its *expanded*, *compiled*, or *declared* form (as opposed to source). For example, when

Racket starts a program, it uses information attached to the main module to initialize the run-time environment.

Submodules are normally a better way to implement reflective information, instead of `#:language-info`. For example, when Racket starts a program, it also checks for a `configure-runtime` submodule of the main module to initialize the run-time environment. The `#:language-info` mechanism pre-dates submodules.

Since the expanded/compiled/declared form exists at a different time than when the source is read, a `#:language-info` specification is a vector that indicates an implementation of the reflective information, instead of a direct implementation as a function like `#:info`. The first element of the vector is a module path, the second is a symbol corresponding to a function exported from the module, and the last element is a value to be passed to the function. The last value in the vector must be one that can be written with `write` and read back with `read`. When the exported function indicated by the first two vector elements is called with the value from the last vector element, the result should be a function or two arguments: a symbol and a default value. The symbol and default value are used as for the `#:info` function (but without an extra default-filtering function).

The value specified by `#:language-info` is attached to the `module` form that is parsed from source through the `'module-language` syntax property. See `module` for more information.

The expression after `#:language-info` is placed into a context where `language-module` are `language-data` are bound, the same as for `#:info`.

In the case of the Racket run-time configuration example, Racket uses the `#:language-info` vector to obtain a function, and then it passes `'configure-runtime` to the function to obtain information about configuring the runtime environment. See also §18.1.5 "Language Run-Time Configuration".

- `#:language` allows the language of the read `module` to be computed dynamically and based on the program source, instead of using a constant *`module-path`*. (Either `#:language` or *`module-path`* must be provided, but not both.)

  This value of the `#:language` option can be either a module path (possibly as a syntax object) that is used as a module language, or it can be a procedure. If it is a procedure it can accept either

  - 0 arguments;
  - 1 argument: an input port; or
  - 5 arguments: an input port, a syntax object whose datum is a module path for the enclosing module as it was referenced through `#lang` or `#reader`, a starting line number (positive exact integer) or `#f`, a column number (non-negative exact integer) or `#f`, and a position number (positive exact integer) or `#f`.

  The result can be either

  - a single value, which is a module path or a syntax object whose datum is a module path, to be used like *`module-path`*; or

- two values, where the first is like a single-value result and the second can be any value.

The second result, which defaults to #f if only a single result is produced, is made available to the #:info and #:module-info functions through the language-data binding. For example, it can be a specification derived from the input stream that changes the module's reflective information (such as the syntax-coloring mode or the output-printing styles).

As another example, the following reader defines a "language" that ignores the contents of the file, and simply reads files as if they were empty:

```
(module ignored syntax/module-reader
  racket/base
  #:wrapper1 (lambda (t) (t) '()))
```

Note that the wrapper still performs the read, otherwise the module loader would complain about extra expressions.

As a more useful example, the following module language is similar to at-exp, where the first datum in the file determines the actual language (which means that the library specification is effectively ignored):

```
(module reader syntax/module-reader
  -ignored-
  #:wrapper2
  (lambda (in rd stx?)
    (let* ([lang (read in)]
           [mod  (parameterize ([current-readtable
                                 (make-at-readtable)])
                   (rd in))]
           [mod  (if stx? mod (datum->syntax #f mod))]
           [r (syntax-case mod ()
                [(module name lang* . body)
                 (with-syntax ([lang (datum->syntax
                                      #'lang* lang #'lang*)])
                   (syntax/loc mod (module name lang . body)))])])
      (if stx? r (syntax->datum r))))
  (require scribble/reader))
```

The ability to change the language position in the resulting module expression can be useful in cases such as the above, where the base language module is chosen based on the input. To make this more convenient, you can omit the *module-path* and instead specify it via a #:language expression. This expression can evaluate to a datum or syntax object that is used as a language, or it can evaluate to a thunk. In the latter case, the thunk is invoked to obtain such a datum before reading the module body begins, in a dynamic extent where current-input-port is the source input. A syntax object is converted using syntax->datum when a datum is needed (for read instead of read-syntax). Using #:language, the last example above can be written more concisely:

```
(module reader syntax/module-reader
  #:language read
  #:wrapper2 (lambda (in rd stx?)
               (parameterize ([current-readtable
                               (make-at-readtable)])
                 (rd in)))
  (require scribble/reader))
```

For such cases, however, the alternative reader constructor make-meta-reader implements a more tightly controlled reading of the module language.

Changed in version 6.3 of package base: Added the #:module-reader option.

```
(make-meta-reader  self-sym
                   path-desc-str
                  [#:read-spec read-spec]
                   module-path-parser
                   convert-read
                   convert-read-syntax
                   convert-get-info)
 → procedure? procedure? procedure?
  self-sym : symbol?
  path-desc-str : string?
  read-spec : (input-port? . -> . any/c) = (lambda (in) ....)
  module-path-parser : (any/c . -> . (or/c module-path? #f
                                          (vectorof module-path?)))
  convert-read : (procedure? . -> . procedure?)
  convert-read-syntax : (procedure? . -> . procedure?)
  convert-get-info : (procedure? . -> . procedure?)
```

Generates procedures suitable for export as read (see read and #lang), read-syntax (see read-syntax and #lang), and get-info (see read-language and #lang), respectively, where the procedures chains to another language that is specified in an input stream.

The at-exp, reader, and planet languages are implemented using this function.

The generated functions expect a target language description in the input stream that is provided to *read-spec*. The default *read-spec* extracts a non-empty sequence of bytes after one or more space and tab bytes, stopping at the first whitespace byte or end-of-file (whichever is first), and it produces either such a byte string or #f. If *read-spec* produces #f, a reader exception is raised, and *path-desc-str* is used as a description of the expected language form in the error message.

The reader language supplies read for *read-spec*. The at-exp and planet languages use the default *read-spec*.

The result of *read-spec* is converted to a module path using *module-path-parser*. If *module-path-parser* produces a vector of module paths, they are tried in order using `module-declared?`. If *module-path-parser* produces `#f`, a reader exception is raised in the same way as when *read-spec* produces a `#f`. The `planet` languages supply a *module-path-parser* that converts a byte string to a module path. Lang-extensions like `at-exp` use `lang-reader-module-paths` as this argument.

If loading the module produced by *module-path-parser* succeeds, then the loaded module's `read`, `read-syntax`, or `get-info` export is passed to *convert-read*, *convert-read-syntax*, or *convert-get-info*, respectively. See §1.3.18 "Reading via an Extension" for information on the protocol of `read` and `read-syntax`.

The `at-exp` language supplies *convert-read* and *convert-read-syntax* to add @-expression support to the current readtable before chaining to the given procedures.

The procedures generated by `make-meta-reader` are not meant for use with the `syntax/module-reader` language; they are meant to be exported directly.

```
(lang-reader-module-paths bstr)
 → (or/c #f (vectorof module-path?))
  bstr : bytes?
```

To be used as the third argument to `make-meta-reader` in lang-extensions like `at-exp`. On success, it returns a vector of module paths, one of which should point to the reader module for the `#lang` *bstr* language. These paths are `(submod base-path reader)` and `base-path/lang/reader`.

```
(wrap-read-all mod-path
               in
               read
               mod-path-stx
               src
               line
               col
               pos)          → any/c
  mod-path : module-path?
  in : input-port?
  read : (input-port . -> . any/c)
  mod-path-stx : syntax?
  src : (or/c syntax? #f)
  line : number?
  col : number?
  pos : number?
```

This function is deprecated; the `syntax/module-reader` *language can be adapted using the various keywords to arbitrary readers; please use it instead.*

Repeatedly calls *read* on *in* until an end of file, collecting the results in order into *lst*, and derives a *name-id* from `(object-name in)` in the same way as `#lang s-exp`. The last five arguments are used to construct the syntax object for the language position of the module. The result is roughly

```
`(module ,name-id ,mod-path ,@lst)
```

7 Parsing `for` Bodies

```
(require syntax/for-body)
```
package: `base`

The `syntax/for-body` module provides a helper function for `for`-like syntactic forms that wrap the body of the form while expanding to another `for`-like form, and the wrapper should apply only after the last `#:break` or `#:final` clause in the body.

```
(split-for-body stx body-stxes) → syntax?
  stx : syntax?
  body-stxes : syntax?
```

The *`body-stxes`* argument must have the form `(pre-body ... post-body ...)`, and it is rewritten into `((pre-body ...) (post-body ...))` such that `(post-body ...)` is as large as possible without containing a `#:break` or `#:final` clause.

The *`stx`* argument is used only for reporting syntax errors.

Use `split-for-body` instead of assuming that the last form in a `for`-like form's body can be wrapped separately. In particular, the last form might contain definitions that need to be spliced in the same definition context as earlier forms to create mutually-recursive definitions.

8 Unsafe `for` Clause Transforms

```
(require syntax/unsafe/for-transform)       package: base
```

The `syntax/unsafe/for-transform` module provides a helper function that gives access to the sequence transformers defined by `define-sequence-syntax`. This is what the `for` forms use and enables faster sequence traversal than what the sequence interface provides.

The output may use unsafe operations.

```
(expand-for-clause* orig-stx clause) → syntax?
  orig-stx : syntax?
  clause : syntax?
```

Expands a `for` clause of the form `[(x ...) seq-expr]`, where `x` are identifiers, to:

```
(([(outer-id ...) outer-expr] ...)
 outer-check
 ([loop-id loop-expr] ...)
 pos-guard
 ([[(inner-id ...) inner-expr] ...)
 inner-check
 pre-guard
 post-guard
 (loop-arg ...))
```

which can then be spliced into the appropriate iterations. See `:do-in` for more information.

The result may use unsafe operations.

The first argument *`orig-stx`* is used only for reporting syntax errors.

Added in version 8.10.0.3 of package `base`.

```
(expand-for-clause orig-stx clause) → syntax?
  orig-stx : syntax?
  clause : syntax?
```

Like `expand-for-clause*`, but the result omits a `inner-check` part:

```
(([(outer-id ...) outer-expr] ...)
 outer-check
 ([loop-id loop-expr] ...)
 pos-guard
 ([[(inner-id ...) inner-expr] ...)
 pre-guard
```

```
post-guard
(loop-arg ...))
```

If a clause expands to a `inner-check` clauses that is not ignorable, `expand-for-clause` reports an error. An ignorable clause is `(void)` or a `begin` form wrapping ignorable clauses.

9 Source Locations

There are two libraries in this collection for dealing with source locations; one for manipulating representations of them, and the other for quoting the location of a particular piece of source code.

9.1 Representations

```
(require syntax/srcloc)    package: base
```

This module defines utilities for manipulating representations of source locations, including both `srcloc` structures and all the values accepted by `datum->syntax`'s third argument: syntax objects, lists, vectors, and `#f`.

```
(source-location? x) → boolean?
  x : any/c
(source-location-list? x) → boolean?
  x : any/c
(source-location-vector? x) → boolean?
  x : any/c
```

These functions recognize valid source location representations. The first, `source-location?`, recognizes `srcloc` structures, syntax objects, lists, and vectors with appropriate structure, as well as `#f`. The latter predicates recognize only valid lists and vectors, respectively.

Examples:

```
> (source-location? #f)
#t
> (source-location? #'here)
#t
> (source-location? (make-srcloc 'here 1 0 1 0))
#t
> (source-location? (make-srcloc 'bad 1 #f 1 0))
#f
> (source-location? (list 'here 1 0 1 0))
#t
> (source-location? (list* 'bad 1 0 1 0 'tail))
#f
> (source-location? (vector 'here 1 0 1 0))
#t
> (source-location? (vector 'bad 0 0 0 0))
#f
```

```
(check-source-location! name x) → void?
  name : symbol?
  x : any/c
```

This procedure checks that its input is a valid source location. If it is, the procedure returns `(void)`. If it is not, `check-source-location!` raises a detailed error message in terms of *name* and the problem with *x*.

Examples:

```
> (check-source-location! 'this-example #f)
> (check-source-location! 'this-example #'here)
> (check-source-location! 'this-example (make-
srcloc 'here 1 0 1 0))
> (check-source-location! 'this-example (make-
srcloc 'bad 1 #f 1 0))
this-example: expected a source location with line number
and column number both numeric or both #f; got 1 and #f
respectively: (srcloc 'bad 1 #f 1 0)
> (check-source-location! 'this-example (list 'here 1 0 1 0))
> (check-source-location! 'this-example (list* 'bad 1 0 1 0 'tail))
this-example: expected a source location (a list of 5
elements); got an improper list: '(bad 1 0 1 0 . tail)
> (check-source-location! 'this-example (vector 'here 1 0 1 0))
> (check-source-location! 'this-example (vector 'bad 0 0 0 0))
this-example: expected a source location with a positive
line number or #f (second element); got line number 0:
'#(bad 0 0 0 0)
```

```
(build-source-location loc ...) → srcloc?
  loc : source-location?
(build-source-location-list loc ...) → source-location-list?
  loc : source-location?
(build-source-location-vector loc ...) → source-location-vector?
  loc : source-location?
(build-source-location-syntax loc ...) → syntax?
  loc : source-location?
```

These procedures combine multiple (zero or more) source locations, merging locations within the same source and reporting `#f` for locations that span sources. They also convert the result to the desired representation: `srcloc`, list, vector, or syntax object, respectively.

Examples:

```
> (build-source-location)
```

```
(srcloc #f #f #f #f #f)
> (build-source-location-list)
'(#f #f #f #f #f)
> (build-source-location-vector)
'#(#f #f #f #f #f)
> (build-source-location-syntax)
#<syntax ()>
> (build-source-location #f)
(srcloc #f #f #f #f #f)
> (build-source-location-list #f)
'(#f #f #f #f #f)
> (build-source-location-vector #f)
'#(#f #f #f #f #f)
> (build-source-location-syntax #f)
#<syntax ()>
> (build-source-location (list 'here 1 2 3 4))
(srcloc 'here 1 2 3 4)
> (build-source-location-list (make-srcloc 'here 1 2 3 4))
'(here 1 2 3 4)
> (build-source-location-vector (make-srcloc 'here 1 2 3 4))
'#(here 1 2 3 4)
> (build-source-location-syntax (make-srcloc 'here 1 2 3 4))
#<syntax:here:1:2 ()>
> (build-source-location (list 'here 1 2 3 4) (vector 'here 5 6 7 8))
(srcloc 'here 1 2 3 12)
> (build-source-location-list (make-srcloc 'here 1 2 3 4) (vector 'here 5 6 7 8))
'(here 1 2 3 12)
> (build-source-location-vector (make-srcloc 'here 1 2 3 4) (vector 'here 5 6 7 8))
'#(here 1 2 3 12)
> (build-source-location-syntax (make-srcloc 'here 1 2 3 4) (vector 'here 5 6 7 8))
#<syntax:here:1:2 ()>
> (build-source-location (list 'here 1 2 3 4) (vector 'there 5 6 7 8))
(srcloc #f #f #f #f #f)
> (build-source-location-list (make-srcloc 'here 1 2 3 4) (vector 'there 5 6 7 8))
'(#f #f #f #f #f)
> (build-source-location-vector (make-srcloc 'here 1 2 3 4) (vector 'there 5 6 7 8))
'#(#f #f #f #f #f)
> (build-source-location-syntax (make-srcloc 'here 1 2 3 4) (vector 'there 5 6 7 8))
#<syntax ()>
```

```
(source-location-known? loc) → boolean?
  loc : source-location?
```

This predicate reports whether a given source location contains more information than simply #f.

Examples:

```
> (source-location-known? #f)
#f
> (source-location-known? (make-srcloc #f #f #f #f #f))
#f
> (source-location-known? (make-srcloc 'source 1 2 3 4))
#t
> (source-location-known? (list #f #f #f #f #f))
#f
> (source-location-known? (vector 'source #f #f #f #f))
#t
> (source-location-known? (datum->syntax #f null #f))
#f
> (source-location-known? (datum->syntax #f null (list 'source #f #f #f #f)))
#t
```

```
(source-location-source loc) → any/c
  loc : source-location?
(source-location-line loc) → (or/c exact-positive-integer? #f)
  loc : source-location?
(source-location-column loc)
 → (or/c exact-nonnegative-integer? #f)
  loc : source-location?
(source-location-position loc)
 → (or/c exact-positive-integer? #f)
  loc : source-location?
(source-location-span loc)
 → (or/c exact-nonnegative-integer? #f)
  loc : source-location?
```

These accessors extract the fields of a source location.

Examples:

```
> (source-location-source #f)
#f
> (source-location-line (make-srcloc 'source 1 2 3 4))
1
> (source-location-column (list 'source 1 2 3 4))
2
> (source-location-position (vector 'source 1 2 3 4))
3
> (source-location-span (datum->syntax #f null (list 'source 1 2 3 4)))
4
```

```
(source-location-end loc)
 → (or/c exact-nonnegative-integer? #f)
  loc : source-location?
```

This accessor produces the end position of a source location (the sum of its position and span, if both are numbers) or `#f`.

Examples:

```
> (source-location-end #f)
#f
> (source-location-end (make-srcloc 'source 1 2 3 4))
7
> (source-location-end (list 'source 1 2 3 #f))
#f
> (source-location-end (vector 'source 1 2 #f 4))
#f
```

```
(update-source-location loc
                        #:source source
                        #:line line
                        #:column column
                        #:position position
                        #:span span)        → source-location?
  loc : source-location?
  source : any/c
  line : (or/c exact-nonnegative-integer? #f)
  column : (or/c exact-positive-integer? #f)
  position : (or/c exact-nonnegative-integer? #f)
  span : (or/c exact-positive-integer? #f)
```

Produces a modified version of *loc*, replacing its fields with *source*, *line*, *column*, *position*, and/or *span*, if given.

Examples:

```
> (update-source-location #f #:source 'here)
'(here #f #f #f #f)
> (update-source-location (list 'there 1 2 3 4) #:line 20 #:column 79)
'(there 20 79 3 4)
> (update-source-location (vector 'everywhere 1 2 3 4) #:position #f #:span #f)
'#(everywhere 1 2 #f #f)
```

```
(source-location->string loc) → string?
  loc : source-location?
(source-location->prefix loc) → string?
  loc : source-location?
```

These procedures convert source locations to strings for use in error messages. The first produces a string describing the source location; the second appends `": "` to the string if it is non-empty.

Examples:

```
> (source-location->string (make-srcloc 'here 1 2 3 4))
"here:1:2"
> (source-location->string (make-srcloc 'here #f #f 3 4))
"here::3-7"
> (source-location->string (make-srcloc 'here #f #f #f #f))
"here"
> (source-location->string (make-srcloc #f 1 2 3 4))
":1:2"
> (source-location->string (make-srcloc #f #f #f 3 4))
"::3-7"
> (source-location->string (make-srcloc #f #f #f #f #f))
""
> (source-location->prefix (make-srcloc 'here 1 2 3 4))
"here:1:2: "
> (source-location->prefix (make-srcloc 'here #f #f 3 4))
"here::3-7: "
> (source-location->prefix (make-srcloc 'here #f #f #f #f))
"here: "
> (source-location->prefix (make-srcloc #f 1 2 3 4))
":1:2: "
> (source-location->prefix (make-srcloc #f #f #f 3 4))
"::3-7: "
> (source-location->prefix (make-srcloc #f #f #f #f #f))
""
```

Changed in version 8.1.0.5 of package `base`: Changed format to separate a line and column with `:` instead of `.`

9.2 Source Location Utilities

```
(require syntax/location)
```
package: `base`

```
(syntax-source-directory stx) → (or/c path? #f)
  stx : syntax?
```

```
(syntax-source-file-name stx) → (or/c path? #f)
  stx : syntax?
```

These produce the directory and file name, respectively, of the path with which `stx` is associated, or `#f` if `stx` is not associated with a path.

Examples:

```
(define loc
  (list (build-path "/tmp" "dir" "somewhere.rkt")
        #f #f #f #f))
(define stx1 (datum->syntax #f 'somewhere loc))
> (syntax-source-directory stx1)
#<path:/tmp/dir/>
> (syntax-source-file-name stx1)
#<path:somewhere.rkt>
(define stx2 (datum->syntax #f 'nowhere #f))
> (syntax-source-directory stx2)
#f
> (syntax-source-directory stx2)
#f
```

Added in version 6.3 of package `base`.

9.2.1 Quoting

The following macros evaluate to various aspects of their own source location.

Note: The examples below illustrate the use of these macros and the representation of their output. However, due to the mechanism by which they are generated, each example is considered a single character and thus does not have realistic line, column, and character positions.

Furthermore, the examples illustrate the use of source location quoting inside macros, and the difference between quoting the source location of the macro definition itself and quoting the source location of the macro's arguments.

```
(quote-srcloc)
(quote-srcloc form)
(quote-srcloc form #:module-source expr)
```

Quotes the source location of `form` as a `srcloc` structure, using the location of the whole `(quote-srcloc)` expression if no `expr` is given. Uses relative directories for paths found within the collections tree, the user's collections directory, or the PLaneT cache.

Examples:

```
> (quote-srcloc)
(srcloc 'eval 2 0 2 1)
> (define-syntax (not-here stx) #'(quote-srcloc))
> (not-here)
(srcloc 'eval 3 0 3 1)
> (not-here)
(srcloc 'eval 3 0 3 1)
> (define-syntax (here stx) #`(quote-srcloc #,stx))
> (here)
(srcloc 'eval 7 0 7 1)
> (here)
(srcloc 'eval 8 0 8 1)
```

```
(quote-source-file)
(quote-source-file form)
(quote-line-number)
(quote-line-number form)
(quote-column-number)
(quote-column-number form)
(quote-character-position)
(quote-character-position form)
(quote-character-span)
(quote-character-span form)
```

Quote various fields of the source location of *form*, or of the whole macro application if no *form* is given.

You can achieve an effect similar to `__FILE__` from Perl or Ruby and `__file__` from Python by using `quote-source-file`.

Examples:

```
> (list (quote-source-file)
        (quote-line-number)
        (quote-column-number)
        (quote-character-position)
        (quote-character-span))
'(eval 2 0 2 1)
> (define-syntax (not-here stx)
    #'(list (quote-source-file)
            (quote-line-number)
            (quote-column-number)
            (quote-character-position)
            (quote-character-span)))
> (not-here)
'(eval 3 0 3 1)
> (not-here)
'(eval 3 0 3 1)
```

```
> (define-syntax (here stx)
    #`(list (quote-source-file #,stx)
            (quote-line-number #,stx)
            (quote-column-number #,stx)
            (quote-character-position #,stx)
            (quote-character-span #,stx)))
> (here)
'(eval 7 0 7 1)
> (here)
'(eval 8 0 8 1)
```

```
(quote-srcloc-string)
(quote-srcloc-string form)
(quote-srcloc-prefix)
(quote-srcloc-prefix form)
```

Quote the result of `source-location->string` or `source-location->prefix`, respectively, applied to the source location of *form*, or of the whole macro application if no *form* is given.

Examples:

```
> (list (quote-srcloc-string)
        (quote-srcloc-prefix))
'("eval:2:0" "eval:2:0: ")
> (define-syntax (not-here stx)
    #'(list (quote-srcloc-string)
            (quote-srcloc-prefix)))
> (not-here)
'("eval:3:0" "eval:3:0: ")
> (not-here)
'("eval:3:0" "eval:3:0: ")
> (define-syntax (here stx)
    #`(list (quote-srcloc-string #,stx)
            (quote-srcloc-prefix #,stx)))
> (here)
'("eval:7:0" "eval:7:0: ")
> (here)
'("eval:8:0" "eval:8:0: ")
```

```
(quote-module-name submod-path-element ...)
```

Quotes the name of the module in a form suitable for printing, but not necessarily as a valid module path. See `quote-module-path` for constructing quoted module paths.

Returns a path, symbol, list, or `'top-level`, where `'top-level` is produced when used outside of a module. A list corresponds to a submodule in the same format as the result of `variable-reference->module-name`. Any given *`submod-path-element`*s (as in a `submod` form) are added to form a result submodule path.

To produce a name suitable for use in printed messages, apply `path->relative-string/library` when the result is a path.

Examples:

```
> (module A racket
    (require syntax/location)
    (define-syntax-rule (name) (quote-module-name))
    (define a-name (name))
    (module+ C
      (require syntax/location)
      (define c-name (quote-module-name))
      (define c-name2 (quote-module-name ".."))
      (provide c-name c-name2))
    (provide (all-defined-out)))
> (require 'A)
> a-name
'A
> (require (submod 'A C))
> c-name
'(A C)
> c-name2
'(A C "..")
> (module B racket
    (require syntax/location)
    (require 'A)
    (define b-name (name))
    (provide (all-defined-out)))
> (require 'B)
> b-name
'B
> (quote-module-name)
'top-level
> (current-namespace (module->namespace ''A))
> (quote-module-name)
'A
```

```
(quote-module-path submod-path-element ...)
```

Quotes the name of the module in which the form is compiled. When possible, the result is a valid module path suitable for use by `dynamic-require` and similar functions.

Builds the result using quote, a path, submod, or 'top-level, where 'top-level is produced when used outside of a module. Any given *submod-path-element*s (as in a submod form) are added to form a result submodule path.

Examples:

```
> (module A racket
    (require syntax/location)
    (define-syntax-rule (path) (quote-module-path))
    (define a-path (path))
    (module+ C
      (require syntax/location)
      (define c-path (quote-module-path))
      (define c-path2 (quote-module-path ".."))
      (provide c-path c-path2))
    (provide (all-defined-out)))
> (require 'A)
> a-path
''A
> (require (submod 'A C))
> c-path
'(submod 'A C)
> c-path2
'(submod 'A C "..")
> (module B racket
    (require syntax/location)
    (require 'A)
    (define b-path (path))
    (provide (all-defined-out)))
> (require 'B)
> b-path
''B
> (quote-module-path)
'top-level
> (current-namespace (module->namespace ''A))
> (quote-module-path)
''A
```

10 Preserving Source Locations

```
(require syntax/quote)
```
package: base

The syntax/quote module provides support for quoting syntax so that its source locations are preserved in marshaled bytecode form.

```
(quote-syntax/keep-srcloc datum)
(quote-syntax/keep-srcloc #:source source-expr datum)
```

Like (quote-syntax *datum*), but the source locations of *datum* are preserved. If a *source-expr* is provided, then it is used in place of a syntax-source value for each syntax object within *datum*.

Unlike a quote-syntax form, the results of evaluating the expression multiple times are not necessarily eq?.

11 Non-Module Compilation And Expansion

```
(require syntax/toplevel)    package: base
```

```
(expand-syntax-top-level-with-compile-time-evals stx) → syntax?
  stx : syntax?
```

Expands `stx` as a top-level expression, and evaluates its compile-time portion for the benefit of later expansions.

The expander recognizes top-level `begin` expressions, and interleaves the evaluation and expansion of the `begin` body, so that compile-time expressions within the `begin` body affect later expansions within the body. (In other words, it ensures that expanding a `begin` is the same as expanding separate top-level expressions.)

The `stx` should have a context already, possibly introduced with `namespace-syntax-introduce`.

```
(expand-top-level-with-compile-time-evals stx) → syntax?
  stx : syntax?
```

Like `expand-syntax-top-level-with-compile-time-evals`, but `stx` is first given context by applying `namespace-syntax-introduce` to it.

```
(expand-syntax-top-level-with-compile-time-evals/flatten stx)
 → (listof syntax?)
  stx : syntax?
```

Like `expand-syntax-top-level-with-compile-time-evals`, except that it returns a list of syntax objects, none of which have a `begin`. These syntax objects are the flattened out contents of any `begin`s in the expansion of `stx`.

```
(eval-compile-time-part-of-top-level stx) → void?
  stx : syntax?
```

Evaluates expansion-time code in the fully expanded top-level expression represented by `stx` (or a part of it, in the case of `begin` expressions). The expansion-time code might affect the compilation of later top-level expressions. For example, if `stx` is a `require` expression, then `namespace-require/expansion-time` is used on each require specification in the form. Normally, this function is used only by `expand-top-level-with-compile-time-evals`.

```
(eval-compile-time-part-of-top-level/compile stx)
 → (listof compiled-expression?)
  stx : syntax?
```

Like `eval-compile-time-part-of-top-level`, but the result is compiled code.

12 Trusting Standard Recertifying Transformers

```
(require syntax/trusted-xforms)
```
package: `base`

The `syntax/trusted-xforms` library has no exports. It exists only to require other modules that perform syntax transformations, where the other transformations must use `syntax-disarm` or `syntax-arm`. An application that wishes to provide a less powerful code inspector to a sub-program should generally attach `syntax/trusted-xforms` to the sub-program's namespace so that things like the class system from `racket/class` work properly.

13 Attaching Documentation to Exports

```
(require syntax/docprovide)        package: base
```

NOTE: This library is deprecated; use `scribble/srcdoc`, instead.

```
(provide-and-document doc-label-id doc-row ...)

doc-row = (section-string (name type-datum doc-string ...) ...)
        | (all-from prefix-id module-path doc-label-id)
        | (all-from-except prefix-id module-path doc-label-id id ...)

   name = id
        | (local-name-id external-name-id)
```

A form that exports names and records documentation information.

The *doc-label-id* identifier is used as a key for accessing the documentation through `lookup-documentation`. The actual documentation is organized into "rows", each with a section title.

A `row` has one of the following forms:

- `(section-string (name type-datum doc-string ...) ...)`

  Creates a documentation section whose title is *section-string*, and provides/documents each *name*. The *type-datum* is arbitrary, for use by clients that call `lookup-documentation`. The *doc-string*s are also arbitrary documentation information, usually concatenated by clients.

  A *name* is either an identifier or a renaming sequence `(local-name-id extenal-name-id)`.

  Multiple `row`s with the same section name will be merged in the documentation output. The final order of sections matches the order of the first mention of each section.

- `(all-from prefix-id module-path doc-label-id)`
- `(all-from-except prefix-id module-path doc-label-id id ...)`

  Merges documentation and provisions from the specified module into the current one; the *prefix-id* is used to prefix the imports into the current module (so they can be re-exported). If *id*s are provided, the specified *id*s are not re-exported and their documentation is not merged.

```
(lookup-documentation module-path-v
                      label-sym)     → any
  module-path-v : module-path?
  label-sym : symbol?
```

Returns documentation for the specified module and label. The *module-path-v* argument is a quoted module path, like the argument to dynamic-require. The *label-sym* identifies a set of documentation using the symbol as a label identifier in provide-and-document.

14 Contracts for Macro Subexpressions

```
(require syntax/contract)      package: base
```

This library provides a procedure `wrap-expr/c` for applying contracts to macro subexpressions.

```
(wrap-expr/c  contract-expr
              expr
             [#:arg? arg?
              #:positive pos-blame
              #:negative neg-blame
              #:name expr-name
              #:macro macro-name
              #:context context
              #:phase phase])      → syntax?
  contract-expr : syntax?
  expr : syntax?
  arg? : any/c = #t
  pos-blame : (or/c syntax? string? module-path-index?
                    'from-macro 'use-site 'unknown)
            = 'from-macro
  neg-blame : (or/c syntax? string? module-path-index?
                    'from-macro 'use-site 'unknown)
            = 'use-site
  expr-name : (or/c identifier? symbol? string? #f) = #f
  macro-name : (or/c identifier? symbol? string? #f) = #f
  context : (or/c syntax? #f) = (current-syntax-context)
  phase : exact-integer? = (syntax-local-phase-level)
```

Returns a syntax object representing an expression that applies the contract represented by *contract-expr* to the value produced by *expr*.

The other arguments have the same meaning as for `expr/c`.

Examples:

```
> (define-syntax (myparameterize1 stx)
    (syntax-case stx ()
      [(_ ([p v]) body)
       (with-syntax ([cp (wrap-expr/c
                           #'parameter? #'p
                           #:name "the parameter argument"
                           #:context stx)])
         #'(parameterize ([cp v]) body))]))
> (myparameterize1 ([current-input-port
```

```
                        (open-input-string "(1 2 3)")])
    (read))
'(1 2 3)
> (myparameterize1 (['whoops 'something])
    'whatever)
myparameterize1: contract violation
  expected: parameter?
  given: 'whoops
  in: parameter?
      macro argument contract on the parameter argument
  contract from: top-level
  blaming: top-level
   (assuming the contract is correct)
  at: eval:4:0
> (module mod racket
    (require (for-syntax syntax/contract))
    (define-syntax (app stx)
      (syntax-case stx ()
        [(app f arg)
         (with-syntax ([cf (wrap-expr/c
                            #'(-> number? number?)
                            #'f
                            #:name "the function argument"
                            #:context stx)])
           #'(cf arg))]))
    (provide app))
> (require 'mod)
> (app add1 5)
6
> (app add1 'apple)
app: broke its own contract
  promised: number?
  produced: 'apple
  in: the 1st argument of
      (-> number? number?)
      macro argument contract on the function argument
  contract from: 'mod
  blaming: (quote mod)
   (assuming the contract is correct)
  at: eval:8:0
> (app (lambda (x) 'pear) 5)
app: contract violation
  expected: number?
  given: 'pear
  in: the range of
      (-> number? number?)
```

```
        macro argument contract on the function argument
  contract from: 'mod
  blaming: top-level
   (assuming the contract is correct)
  at: eval:9:0
```

Added in version 6.3 of package `base`.
Changed in version 7.2.0.3: Added the `#:arg?` keyword argument and changed the default values and interpretation of the `#:positive` and `#:negative` arguments.
Changed in version 7.3.0.3: Added the `#:phase` keyword argument.

15 Macro Testing

```
(require syntax/macro-testing)    package: base
```

```
(phase1-eval ct-expr maybe-quote maybe-catch?)

 maybe-quote =
             | #:quote quote-id

maybe-catch? =
             | #:catch? catch?
```

Evaluates *ct-expr* at compile time and quotes the result using *quote-id*, which defaults to `quote`. Another suitable argument for *quote-id* is `quote-syntax`.

If *catch?* is `#t`, then if the evaluation of *ct-expr* raises a compile-time exception, it is caught and converted to a run-time exception.

Examples:

```
> (struct point (x y))
> (phase1-eval (extract-struct-info (syntax-local-value #'point)))
'(struct:point point point? (point-y point-x) (#f #f) #t)
> (phase1-eval (extract-struct-info (syntax-local-value #'point))
               #:quote quote-syntax)
#<syntax (struct:point point point? (point-y point-x) (#f #f) #t)>
```

Added in version 6.3 of package `base`.

```
(convert-compile-time-error expr)
```

Equivalent to `(#%expression expr)` except if expansion of *expr* causes a compile-time exception to be raised; in that case, the compile-time exception is converted to a run-time exception raised when the expression is evaluated.

Use `convert-compile-time-error` to write tests for compile-time error checking like syntax errors:

Examples:

```
> (check-exn #rx"missing an \"else\" expression"
             (lambda () (convert-compile-time-error (if 1 2))))
> (check-exn #rx"missing formals and body"
             (lambda () (convert-compile-time-error (lambda))))
```

```
--------------------
FAILURE
name:           check-exn
location:       eval:6:0
params:         '(#rx"missing formals and body" #<procedure>)
message:        "Wrong exception raised"
exn-message:    "eval:6:0: lambda: bad syntax\n   in: (lambda)"
exn:
  #(struct:exn:fail:syntax "eval:6:0: lambda: bad syntax\n in: (lambda)"
#<continuation-mark-set> (#<syntax:eval:6:0 (lambda)>))
--------------------
```

Without the use of `convert-compile-time-error`, the checks above would not be executed because the test program would not compile.

Added in version 6.3 of package `base`.

```
(convert-syntax-error expr)
```

Like `convert-compile-time-error`, but only catches compile-time `exn:fail:syntax?` exceptions and sets `error-print-source-location` to `#f` around the expansion of *`expr`* to make the message easier to match exactly.

Example:

```
> (check-exn #rx"^lambda: bad syntax$"
             (lambda () (convert-syntax-error (lambda))))
```

Added in version 6.3 of package `base`.

16 Internal-Definition Context Helpers

```
(require syntax/intdef)
```

package: `base`

Added in version 6.3.0.4 of package `base`.

```
(internal-definition-context-track intdef-ctx
                                   stx)        → syntax?
  intdef-ctx : internal-definition-context?
  stx : syntax?
```

Adjusts the syntax properties of *stx* to record that parts of *stx* were expanded via *intdef-ctx*.

Specifically, the identifiers produced by `(internal-definition-context-binding-identifiers intdef-ctx)` are added to the `'disappeared-binding` property of *stx*.

Index