How to Design Programs Languages

Version 9.4.0.3

October 31, 2026

The languages documented in this manual are provided by DrRacket to be used with the *How to Design Programs* book.

When programs in these languages are run in DrRacket, any part of the program that was not run is highlighted in orange and black. These colors are intended to give the programmer feedback about the parts of the program that have not been tested. To avoid seeing these colors, use `check-expect` to test your program. Of course, just because you see no colors, does not mean that your program has been fully tested; it simply means that each part of the program has been run (at least once).

While these languages are normally selected using the Choose Language dialog in DrRacket, they can also be accessed using the `#lang` language directive as the first line of code in DrRacket or other editors.

- Beginning Student `#lang htdp/bsl`
- Beginning Student with List Abbreviations `#lang htdp/bsl+`
- Intermediate Student `#lang htdp/isl`
- Intermediate Student with lambda `#lang htdp/isl+`
- Advanced Student `#lang htdp/asl`

Contents

1 Beginning Student **7**

1.1 Pre-defined Variables 8

1.2 Template Variables 9

1.3 Syntax 10

1.4 Signatures 19

1.4.1 Signature Forms 20

1.4.2 Struct Signatures 21

1.5 Pre-defined Functions 21

1.6 Numbers: Integers, Rationals, Reals, Complex, Exacts, Inexacts 22

1.7 Booleans 37

1.8 Symbols 38

1.9 Lists 39

1.10 Posns 51

1.11 Characters 52

1.12 Strings 57

1.13 Images 65

1.14 Misc 66

1.15 Signatures 69

2 Beginning Student with List Abbreviations **71**

2.1 Pre-defined Variables 73

2.2 Template Variables 73

2.3 Syntaxes for Beginning Student with List Abbreviations 74

2.4 Common Syntaxes 75

2.5 Signatures 84
2.5.1 Signature Forms 85
2.5.2 Struct Signatures 86
2.6 Pre-defined Functions 87
2.7 Numbers: Integers, Rationals, Reals, Complex, Exacts, Inexacts 87
2.8 Booleans 102
2.9 Symbols 103
2.10 Lists 104
2.11 Posns 116
2.12 Characters 117
2.13 Strings 122
2.14 Images 130
2.15 Misc 131
2.16 Signatures 134

3 Intermediate Student **136**

3.1 Pre-defined Variables 138
3.2 Template Variables 138
3.3 Syntax for Intermediate 139
3.4 Common Syntaxes 140
3.5 Signatures 150
3.5.1 Signature Forms 151
3.5.2 Struct Signatures 152
3.6 Pre-defined Functions 153
3.7 Numbers: Integers, Rationals, Reals, Complex, Exacts, Inexacts 153
3.8 Booleans 167

3.9 Symbols . . . 168
3.10 Lists . . . 169
3.11 Posns . . . 181
3.12 Characters . . . 182
3.13 Strings . . . 187
3.14 Images . . . 193
3.15 Misc . . . 193
3.16 Signatures . . . 196
3.17 Numbers (relaxed conditions) . . . 198
3.18 String (relaxed conditions) . . . 199
3.19 Posn . . . 202
3.20 Higher-Order Functions . . . 202

4 Intermediate Student with Lambda 210

4.1 Pre-defined Variables . . . 212
4.2 Template Variables . . . 212
4.3 Syntax for Intermediate with Lambda . . . 213
4.4 Common Syntaxes . . . 214
4.5 Pre-defined Functions . . . 224
4.6 Signatures . . . 224
4.6.1 Signature Forms . . . 225
4.6.2 Struct Signatures . . . 227
4.7 Numbers: Integers, Rationals, Reals, Complex, Exacts, Inexacts . . . 227
4.8 Booleans . . . 241
4.9 Symbols . . . 243
4.10 Lists . . . 243

4.11 Posns 255
4.12 Characters 256
4.13 Strings 261
4.14 Images 267
4.15 Misc 268
4.16 Signatures 270
4.17 Numbers (relaxed conditions) 272
4.18 String (relaxed conditions) 272
4.19 Posn 275
4.20 Higher-Order Functions 275
4.21 Numbers (relaxed conditions plus) 275
4.22 Higher-Order Functions (with Lambda) 277

5 Advanced Student **284**

5.1 Pre-defined Variables 287
5.2 Template Variables 287
5.3 Syntax for Advanced 288
5.4 Common Syntaxes 291
5.5 Pre-Defined Functions 301
5.6 Signatures 301
5.6.1 Signature Forms 302
5.6.2 Struct Signatures 303
5.7 Numbers: Integers, Rationals, Reals, Complex, Exacts, Inexacts 303
5.8 Booleans 318
5.9 Symbols 319
5.10 Lists 319

5.11 Posns 331
5.12 Characters 333
5.13 Strings 337
5.14 Images 343
5.15 Misc 344
5.16 Signatures 348
5.17 Numbers (relaxed conditions) 350
5.18 String (relaxed conditions) 350
5.19 Posn 352
5.20 Higher-Order Functions 353
5.21 Numbers (relaxed conditions plus) 353
5.22 Higher-Order Functions (with Lambda) 354
5.23 Reading and Printing 360
5.24 Vectors 363
5.25 Boxes 365
5.26 Hash Tables 366

1 Beginning Student

The grammar notation uses the notation `X` **...** (bold dots) to indicate that `X` may occur an arbitrary number of times (zero, one, or more). Separately, the grammar also defines `...` as an identifier to be used in templates.

See How to Design Programs/2e, Intermezzo 1 for an explanation of the Beginning Student Language.

```
                program = def-or-expr ...

          def-or-expr = definition
                      | expr
                      | test-case
                      | library-require
                      | signature-declaration

           definition = (define (name variable variable ...) expr)
                      | (define name expr)
                      | (define name (lambda (variable variable ...) expr))
                      | (define-struct name (name ...))

                expr = (name expr expr ...)
                     | (cond [expr expr] ... [expr expr])
                     | (cond [expr expr] ... [else expr])
                     | (if expr expr expr)
                     | (and expr expr expr ...)
                     | (or expr expr expr ...)
                     | name
                     | 'name
                     | '()
                     | number
                     | boolean
                     | string
                     | character
                     | (signature signature-form)

signature-declaration = (: name signature-form)

       signature-form = (enum expr ...)
                      | (mixed signature-form ...)
                      | (signature-form ... -> signature-form)
                      | (ListOf signature-form)
                      | signature-variable
                      | expr
```

```
signature-variable = %name

         test-case = (check-expect expr expr)
                   | (check-random expr expr)
                   | (check-within expr expr expr)
                   | (check-random-within expr expr expr)
                   | (check-member-of expr expr ...)
                   | (check-range expr expr expr)
                   | (check-satisfied expr name)
                   | (check-error expr expr)
                   | (check-error expr)

   library-require = (require string)
                   | (require (lib string string ...))
                   | (require (planet string package))

           package = (string string number number)
```

A *name* or a *variable* is a sequence of characters not including a space or one of the following:

```
" , ' ` ( ) [ ] { } | ; #
```

A *number* is a number such as `123`, `3/2`, or `5.5`.

A *boolean* is one of: `#true` or `#false`.

Alternative spellings for the `#true` constant are `#t`, `true`, and `#T`. Similarly, `#f`, `false`, or `#F` are also recognized as `#false`.

A *symbol* is a quote character followed by a name. A symbol is a value, just like `42`, `'()`, or `#false`.

A *string* is a sequence of characters enclosed by a pair of `"`. Unlike symbols, strings may be split into characters and manipulated by a variety of functions. For example, `"abcdef"`, `"This is a string"`, and `"This is a string with \" inside"` are all strings.

A *character* begins with `#\` and has the name of the character. For example, `#\a`, `#\b`, and `#\space` are characters.

In function calls, the function appearing immediately after the open parenthesis can be any functions defined with `define` or `define-struct`, or any one of the pre-defined functions.

1.1 Pre-defined Variables

```
empty : empty?
```

The empty list.

```
true : boolean?
```

The `#true` value.

```
false : boolean?
```

The `#false` value.

1.2 Template Variables

```
..
```

A placeholder for indicating that a function definition is a template.

```
...
```

A placeholder for indicating that a function definition is a template.

```
....
```

A placeholder for indicating that a function definition is a template.

```
.....
```

A placeholder for indicating that a function definition is a template.

```
......
```

A placeholder for indicating that a function definition is a template.

1.3 Syntax

```
(define (name variable variable ...) expression)
```

Defines a function named *name*. The *expression* is the body of the function. When the function is called, the values of the arguments are inserted into the body in place of the *variable*s. The function returns the value of that new expression.

The function name's cannot be the same as that of another function or variable.

```
(define name expression)
```

Defines a variable called *name* with the the value of *expression*. The variable name's cannot be the same as that of another function or variable, and *name* itself must not appear in *expression*.

```
(define name (lambda (variable variable ...) expression))
```

An alternate way to defining functions. The *name* is the name of the function, which cannot be the same as that of another function or variable.

A `lambda` cannot be used outside of this alternate syntax.

```
'name
(quote name)
```

A quoted *name* is a symbol. A symbol is a value, just like `0` or `'()`.

```
(define-struct structure-name (field-name ...))
```

Defines a new structure called *structure-name*. The structure's fields are named by the *field-name*s. After the `define-struct`, the following new functions are available:

- `make-`*structure-name* : takes a number of arguments equal to the number of fields in the structure, and creates a new instance of that structure.
- *structure-name-field-name* : takes an instance of the structure and returns the value in the field named by *field-name*.
- *structure-name*`?` : takes any value, and returns `#true` if the value is an instance of the structure.

The name of the new functions introduced by `define-struct` must not be the same as that of other functions or variables, otherwise `define-struct` reports an error.

```
(name expression expression ...)
```

Calls the function named *name*. The value of the call is the value of *name*'s body when every one of the function's variables are replaced by the values of the corresponding *expression*s.

The function named *name* must defined before it can be called. The number of argument *expression*s must be the same as the number of arguments expected by the function.

```
(cond [question-expression answer-expression] ...)
(cond [question-expression answer-expression]
      ...
      [else answer-expression])
```

Chooses a clause based on some condition. `cond` finds the first *question-expression* that evaluates to `#true`, then evaluates the corresponding *answer-expression*.

If none of the *question-expression*s evaluates to `#true`, `cond`'s value is the *answer-expression* of the `else` clause. If there is no `else`, `cond` reports an error. If the result of a *question-expression* is neither `#true` nor `#false`, `cond` also reports an error.

`else` cannot be used outside of `cond`.

```
(if question-expression
    then-answer-expression
    else-answer-expression)
```

When the value of the *question-expression* is `#true`, `if` evaluates the *then-answer-expression*. When the test is `#false`, `if` evaluates the *else-answer-expression*.

If the *question-expression* is neither `#true` nor `#false`, `if` reports an error.

```
(and expression expression expression ...)
```

Evaluates to `#true` if all the *expression*s are `#true`. If any *expression* is `#false`, the `and` expression evaluates to `#false` (and the expressions to the right of that expression are not evaluated.)

If any of the expressions evaluate to a value other than `#true` or `#false`, `and` reports an error.

```
(or expression expression expression ...)
```

Evaluates to `#true` as soon as one of the *expression*s is `#true` (and the expressions to the right of that expression are not evaluated.) If all of the *expression*s are `#false`, the `or` expression evaluates to `#false`.

If any of the expressions evaluate to a value other than `#true` or `#false`, `or` reports an error.

```
(check-expect expression expected-expression)
```

Checks that the first *expression* evaluates to the same value as the *expected-expression*.

```
(check-expect (fahrenheit->celsius 212) 100)
(check-expect (fahrenheit->celsius -40) -40)

(define (fahrenheit->celsius f)
  (* 5/9 (- f 32)))
```

A `check-expect` expression must be placed at the top-level of a student program. Also it may show up anywhere in the program, including ahead of the tested function definition. By placing `check-expect`s there, a programmer conveys to a future reader the intention behind the program with working examples, thus making it often superfluous to read the function definition proper. Syntax errors in `check-expect` (and all check forms) are intentionally delayed to run time so that students can write tests *without* necessarily writing complete function headers.

It is an error for `expr` or `expected-expr` to produce an inexact number or a function value. As for inexact numbers, it is *morally* wrong to compare them for plain equality. Instead one tests whether they are both within a small interval; see `check-within`. As for functions (see Intermediate and up), it is provably impossible to compare functions.

```
(check-random expression expected-expression)
```

Checks that the first *expression* evaluates to the same value as the *expected-expression*.

The form supplies the same random-number generator to both parts. If both parts request `random` numbers from the same interval in the same order, they receive the same random numbers.

Here is a simple example of where `check-random` is useful:

```
(define WIDTH 100)
(define HEIGHT (* 2 WIDTH))
```

```
(define-struct player (name x y))
; A Player is (make-player String Nat Nat)

; String -> Player

(check-random (create-randomly-placed-player "David Van Horn")
              (make-player "David Van Horn" (random WIDTH) (random HEIGHT)))

(define (create-randomly-placed-player name)
  (make-player name (random WIDTH) (random HEIGHT)))
```

Note how `random` is called on the same numbers in the same order in both parts of `check-random`. If the two parts call `random` for different intervals, they are likely to fail:

```
; String -> Player

(check-random (create-randomly-placed-player "David Van Horn")
              (make-player "David Van Horn" (random WIDTH) (random HEIGHT)))

(define (create-randomly-placed-player name)
  (a-helper-function name (random HEIGHT)))

; String Number -> Player
(define (a-helper-function name height)
   (make-player name (random WIDTH) height))
```

Because the argument to `a-helper-function` is evaluated first, `random` is first called for the interval [0,*HEIGHT*) and then for [0,*WIDTH*), that is, in a different order than in the preceding `check-random`.

It is an error for `expr` or `expected-expr` to produce a function value or an inexact number; see note on `check-expect` for details.

`(check-satisfied` *`expression predicate`*`)`

Checks that the first *`expression`* satisfies the named *`predicate`* (function of one argument). Recall that "satisfies" means "the function produces `#true` for the given value."

Here are simple examples for `check-satisfied`:

```
> (check-satisfied 1 odd?)
The test passed!


> (check-satisfied 1 even?)
```

```
Ran 1 test.
0 tests passed.
Check failures:

        Actual value | 1 | does not satisfy even?.

at line 3, column 0
```

In general `check-satisfied` empowers program designers to use defined functions to formulate test suites:

```
; [cons Number [List-of Number]] -> Boolean
; a function for testing htdp-sort

(check-expect (sorted? (list 1 2 3)) #true)
(check-expect (sorted? (list 2 1 3)) #false)

(define (sorted? l)
  (cond
    [(empty? (rest l)) #true]
    [else (and (<= (first l) (second l)) (sorted? (rest l)))]))

; [List-of Number] -> [List-of Number]
; create a sorted version of the given list of numbers

(check-satisfied (htdp-sort (list 1 2 0 3)) sorted?)

(define (htdp-sort l)
  (cond
    [(empty? l) l]
    [else (insert (first l) (htdp-sort (rest l)))]))

; Number [List-of Number] -> [List-of Number]
; insert x into l at proper place
; assume l is arranged in ascending order
; the result is sorted in the same way
(define (insert x l)
  (cond
    [(empty? l) (list x)]
    [else (if (<= x (first l)) (cons x l) (cons (first l) (insert x (rest l))))]))
```

And yes, the results of `htdp-sort` satisfy the `sorted?` predicate:

```
> (check-satisfied (htdp-sort (list 1 2 0 3)) sorted?)
The test passed!
```

```
(check-within expression expected-expression delta)
```

Checks whether the value of the *expression* expression is structurally equal to the value produced by the *expected-expression* expression; every number in the first expression must be within *delta* of the corresponding number in the second expression.

```
(define-struct roots (x sqrt))
; RT is [List-of (make-roots Number Number)]

(define (root-of a)
  (make-roots a (sqrt a)))

(define (roots-table xs)
  (cond
    [(empty? xs) '()]
    [else (cons (root-of (first xs)) (roots-table (rest xs)))]))
```

Due to the presence of inexact numbers in nested data, `check-within` is the correct choice for testing, and the test succeeds if *delta* is reasonably large:

Example:

```
> (check-within (roots-table (list 1.0 2.0 3.0))
                (list
                  (make-roots 1.0 1.0)
                  (make-roots 2  1.414)
                  (make-roots 3  1.713))
                0.1)
The test passed!
```

In contrast, when *delta* is small, the test fails:

Example:

```
> (check-within (roots-table (list 2.0))
                (list
                  (make-roots 2  1.414))
                #i1e-5)
Ran 1 test.
0 tests passed.
Check failures:
                      ┌──────────────────────────────────────────┐

        Actual value │ '((make-roots 2.0 1.4142135623730951)) │ is
not within 1e-5 of expected value │ '((make-roots 2 1.414)) │.
```

```
at line 5, column 0
```

It is an error for `expressions` or *`expected-expression`* to produce a function value; see note on `check-expect` for details.

If *`delta`* is not a number, `check-within` reports an error.

```
(check-random-within expression expected-expression delta)
```

Combines `check-random` and `check-within`. Like `check-random`, *`expression`* and *`expected-expression`* are evaluated with the same random-number sequence, so calls to `random` in both expressions draw the same numbers. Like `check-within`, the test then succeeds if every number in the value of *`expression`* is within *`delta`* of the corresponding number in the value of *`expected-expression`*.

Use `check-random-within` in place of `check-random` when the random computation produces inexact numbers, which `check-random` refuses to compare.

If *`delta`* is not a number, `check-random-within` reports an error.

Added in version 1.9 of package `htdp-lib`.

```
(check-error expression expected-error-message)
(check-error expression)
```

Checks that the *`expression`* reports an error, where the error messages matches the value of *`expected-error-message`*, if it is present.

Here is a typical beginner example that calls for a use of `check-error`:

```
(define sample-table
  '(("matthias" 10)
    ("matthew"  20)
    ("robby"    -1)
    ("shriram"  18)))

; [List-of [list String Number]] String -> Number
; determine the number associated with s in table

(define (lookup table s)
  (cond
    [(empty? table) (error (string-append s " not found"))]
```

```
     [else (if (string=? (first (first table)) s)
               (second (first table))
               (lookup (rest table)))]))
```

Consider the following two examples in this context:

Example:

```
> (check-expect (lookup sample-table "matthew") 20)
The test passed!
```

Example:

```
> (check-error (lookup sample-table "kathi") "kathi not found")
The test passed!
```

(check-member-of *expression expression expression ...*)

Checks that the value of the first *expression* is that of one of the following *expression*s.

```
; [List-of X] -> X
; pick a random element from the given list l
(define (pick-one l)
  (list-ref l (random (length l))))
```

Example:

```
> (check-member-of (pick-one '("a" "b" "c")) "a" "b" "c")
The test passed!
```

It is an error for any of expressions to produce a function value; see note on check-expect for details.

(check-range *expression low-expression high-expression*)

Checks that the value of the first *expression* is a number in between the value of the *low-expression* and the *high-expression*, inclusive.

A check-range form is best used to delimit the possible results of functions that compute inexact numbers:

```
(define EPSILON 0.001)

; [Real -> Real] Real -> Real
; what is the slope of f at x?
(define (differentiate f x)
  (slope f (- x EPSILON) (+ x EPSILON)))

; [Real -> Real] Real Real -> Real
(define (slope f left right)
  (/ (- (f right) (f left))
     2 EPSILON))

(check-range (differentiate sin 0) 0.99 1.0)
```

It is an error for *expression*, *low-expression*, or *high-expression* to produce a function value; see note on `check-expect` for details.

```
(require string)
```

Makes the definitions of the module specified by *string* available in the current module (i.e., the current file), where *string* refers to a file relative to the current file.

The *string* is constrained in several ways to avoid problems with different path conventions on different platforms: a `/` is a directory separator, `.` always means the current directory, `..` always means the parent directory, path elements can use only `a` through `z` (uppercase or lowercase), `0` through `9`, `-`, `_`, and `.`, and the string cannot be empty or contain a leading or trailing `/`.

```
(require module-name)
```

Accesses a file in an installed library. The library name is an identifier with the same constraints as for a relative-path string (though without the quotes), with the additional constraint that it must not contain a `..`

```
(require (lib string string ...))
```

Accesses a file in an installed library, making its definitions available in the current module (i.e., the current file). The first *string* names the library file, and the remaining *string*s name the collection (and sub-collection, and so on) where the file is installed. Each string is constrained in the same way as for the `(require` *string*`)` form.

```
(require (planet string (string string number number)))
(require (planet id))
(require (planet string))
```

Accesses a library that is distributed on the internet via the PLaneT server, making it definitions available in the current module (i.e., current file).

The full grammar for planet requires is given in §3.2 “Importing and Exporting: `require` and `provide`”, but the best place to find examples of the syntax is on the the PLaneT server, in the description of a specific package.

1.4 Signatures

Signatures do not have to be comment: They can also be part of the code. When a signature is attached to a function, DrRacket will check that program uses the function in accordance with the signature and display signature violations along with the test results.

A signature is a regular value, and is specified as a *signature form*, a special syntax that only works with `:` signature declarations and inside `signature` expressions.

```
(: name signature-form)
```

This attaches the signature specified by *signature-form* to the definition of *name*. There must be a definition of *name* somewhere in the program.

```
(: age Integer)
(define age 42)

(: area-of-square (Number -> Number))
(define (area-of-square len)
  (sqr len))
```

On running the program, Racket checks whether the signatures attached with `:` actually match the value of the variable. If they don’t, Racket reports *signature violation* along with test failures.

For example, this piece of code:

```
(: age Integer)
(define age "fortytwo")
```

Yields this output:

```
1 signature violation.

Signature violations:
        got "fortytwo" at line 2, column 12, signature at line 1,
column 7
```

Note that a signature violation does not stop the running program.

```
(signature signature-form)
```

This returns the signature described by *signature-form* as a value.

1.4.1 Signature Forms

Any expression can be a signature form, in which case the signature is the value returned by that expression. There are a few special signature forms, however:

In a signature form, any name that starts with a % is a *signature variable* that stands for any signature depending on how the signature is used.

Example:

```
(: same (%a -> %a))

(define (same x) x)
```

```
(input-signature-form ... -> output-signature-form)
```

This signature form describes a function with inputs described by the *input-signature-form*s and output described by *output-signature-form*.

```
(enum expr ...)
```

This signature describes an enumeration of the values returned by the *expr*s.

Example:

```
(: cute? ((enum "cat" "snake") -> Boolean))

(define (cute? pet)
  (cond
    [(string=? pet "cat") #t]
    [(string=? pet "snake") #f]))
```

```
(mixed signature-form ...)
```

This signature describes mixed data, i.e. an itemization where each of the cases has a signature described by a *signature-form*.

Example:

```
(define SIGS (signature (mixed Aim Fired)))
```

```
(ListOf signature-form)
```

This signature describes a list where the elements are described by *signature-form*.

```
(predicate expression)
```

This signature describes values through a predicate: *expression* must evaluate to a function of one argument that returns a boolean. The signature matches all values for which the predicate returns `#true`.

1.4.2 Struct Signatures

A `define-struct` form defines two additional names that can be used in signatures. For a struct called `struct`, these are `Struct` and `StructOf`. Note that these names are capitalized. In particular, a struct called `Struct`, will also define `Struct` and `StructOf`. Moreover, when forming the additional names, hyphens are removed, and each letter following a hyphen is capitalized - so a struct called `foo-bar` will define `FooBar` and `FooBarOf`.

`Struct` is a signature that describes struct values from this structure type. `StructOf` is a function that takes as input a signature for each field. It returns a signature describing values of this structure type, additionally describing the values of the fields of the value.

```
(define-struct pair [fst snd])

(: add-pair ((PairOf Number Number) -> Number))
(define (add-pair p)
  (+ (pair-fst p) (pair-snd p)))
```

1.5 Pre-defined Functions

The remaining subsections list those functions that are built into the programming language. All other functions are imported from a teachpack or must be defined in the program.

1.6 Numbers: Integers, Rationals, Reals, Complex, Exacts, Inexacts

```
(* x y z ...) → number
  x : number
  y : number
  z : number
```

Multiplies all numbers.

```
> (* 5 3)
15
> (* 5 3 2)
30
```

```
(+ x y z ...) → number
  x : number
  y : number
  z : number
```

Adds up all numbers.

```
> (+ 2/3 1/16)
35/48
> (+ 3 2 5 8)
18
```

```
(- x y ...) → number
  x : number
  y : number
```

Subtracts the second (and following) number(s) from the first ; negates the number if there is only one argument.

```
> (- 5)
-5
> (- 5 3)
2
> (- 5 3 1)
1
```

```
(/ x y z ...) → number
  x : number
  y : number
  z : number
```

Divides the first by the second (and all following) number(s).

```
> (/ 12 2)
6
> (/ 12 2 3)
2
```

```
(< x y z ...) → boolean?
  x : real
  y : real
  z : real
```

Compares two or more (real) numbers for less-than.

```
> (< 42 2/5)
#false
```

```
(<= x y z ...) → boolean?
  x : real
  y : real
  z : real
```

Compares two or more (real) numbers for less-than or equality.

```
> (<= 42 2/5)
#false
```

```
(= x y z ...) → boolean?
  x : number
  y : number
  z : number
```

Compares two or more numbers for equality.

```
> (= 42 2/5)
#false
```

```
(> x y z ...) → boolean?
  x : real
  y : real
  z : real
```

Compares two or more (real) numbers for greater-than.

```
> (> 42 2/5)
#true
```

```
(>= x y z ...) → boolean?
  x : real
  y : real
  z : real
```

Compares two or more (real) numbers for greater-than or equality.

```
> (>= 42 42)
#true
```

```
(abs x) → real
  x : real
```

Determines the absolute value of a real number.

```
> (abs -12)
12
```

```
(acos x) → number
  x : number
```

Computes the arccosine (inverse of cos) of a number.

```
> (acos 0)
#i1.5707963267948966
```

```
(add1 x) → number
  x : number
```

Increments the given number.

```
> (add1 2)
3
```

```
(angle x) → real
  x : number
```

Extracts the angle from a complex number.

```
> (angle (make-polar 3 4))
#i-2.2831853071795867
```

```
(asin x) → number
  x : number
```

Computes the arcsine (inverse of sin) of a number.

```
> (asin 0)
0
```

```
(atan x) → number
  x : number
```

Computes the arctangent of the given number:

```
> (atan 0)
0
> (atan 0.5)
#i0.46364760900080615
```

Also comes in a two-argument version where `(atan y x)` computes `(atan (/ y x))` but the signs of `y` and `x` determine the quadrant of the result and the result tends to be more accurate than that of the 1-argument version in borderline cases:

```
> (atan 3 4)
#i0.6435011087932844
> (atan -2 -1)
#i-2.0344439357957027
```

```
(ceiling x) → integer
  x : real
```

Determines the closest integer (exact or inexact) above a real number. See `round`.

```
> (ceiling 12.3)
#i13.0
```

```
(complex? x) → boolean?
  x : any/c
```

Determines whether some value is complex.

```
> (complex? 1-2i)
#true
```

```
(conjugate x) → number
  x : number
```

Flips the sign of the imaginary part of a complex number.

```
> (conjugate 3+4i)
3-4i
> (conjugate -2-5i)
-2+5i
> (conjugate (make-polar 3 4))
#i-1.960930862590836+2.270407485923785i
```

```
(cos x) → number
  x : number
```

Computes the cosine of a number (radians).

```
> (cos pi)
#i-1.0
```

```
(cosh x) → number
  x : number
```

Computes the hyperbolic cosine of a number.

```
> (cosh 10)
#i11013.232920103324
```

```
(current-seconds) → integer
```

Determines the current time in seconds elapsed (since a platform-specific starting date).

```
> (current-seconds)
1793426320
```

```
(denominator x) → integer
  x : rational?
```

Computes the denominator of a rational.

```
> (denominator 2/3)
3
```

```
e : real
```

Euler's number.

```
> e
#i2.718281828459045
```

```
(even? x) → boolean?
  x : integer
```

Determines if some integer (exact or inexact) is even or not.

```
> (even? 2)
#true
```

```
(exact->inexact x) → number
  x : number
```

Converts an exact number to an inexact one.

```
> (exact->inexact 12)
#i12.0
```

```
(exact? x) → boolean?
  x : number
```

Determines whether some number is exact.

```
> (exact? (sqrt 2))
#false
```

```
(exp x) → number
  x : number
```

Determines e raised to a number.

```
> (exp -2)
#i0.1353352832366127
```

```
(expt x y) → number
  x : number
  y : number
```

Computes the power of the first to the second number, which is to say, exponentiation.

```
> (expt 16 1/2)
4
> (expt 3 -4)
1/81
```

```
(floor x) → integer
  x : real
```

Determines the closest integer (exact or inexact) below a real number. See `round`.

```
> (floor 12.3)
#i12.0
```

```
(gcd x y ...) → integer
  x : integer
  y : integer
```

Determines the greatest common divisor of two integers (exact or inexact).

```
> (gcd 6 12 8)
2
```

```
(imag-part x) → real
  x : number
```

Extracts the imaginary part from a complex number.

```
> (imag-part 3+4i)
4
```

```
(inexact->exact x) → number
  x : number
```

Approximates an inexact number by an exact one.

```
> (inexact->exact 12.0)
12
```

```
(inexact? x) → boolean?
  x : number
```

Determines whether some number is inexact.

```
> (inexact? 1-2i)
#false
```

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

Looks up the character that corresponds to the given exact integer in the ASCII table (if any).

```
> (integer->char 42)
#\*
```

```
(integer-sqrt x) → complex
  x : integer
```

Computes the integer or imaginary-integer square root of an integer.

```
> (integer-sqrt 11)
3
> (integer-sqrt -11)
0+3i
```

```
(integer? x) → boolean?
  x : any/c
```

Determines whether some value is an integer (exact or inexact).

```
> (integer? (sqrt 2))
#false
```

```
(lcm x y ...) → integer
  x : integer
  y : integer
```

Determines the least common multiple of two integers (exact or inexact).

```
> (lcm 6 12 8)
24
```

```
(log x) → number
  x : number
```

Determines the base-e logarithm of a number.

```
> (log 12)
#i2.4849066497880004
```

```
(magnitude x) → real
  x : number
```

Determines the magnitude of a complex number.

```
> (magnitude (make-polar 3 4))
#i3.0000000000000004
```

```
(make-polar x y) → number
  x : real
  y : real
```

Creates a complex from a magnitude and angle.

```
> (make-polar 3 4)
#i-1.960930862590836-2.270407485923785i
```

```
(make-rectangular x y) → number
  x : real
  y : real
```

Creates a complex from a real and an imaginary part.

```
> (make-rectangular 3 4)
3+4i
```

```
(max x y ...) → real
  x : real
  y : real
```

Determines the largest number—aka, the maximum.

```
> (max 3 2 8 7 2 9 0)
9
```

```
(min x y ...) → real
  x : real
  y : real
```

Determines the smallest number—aka, the minimum.

```
> (min 3 2 8 7 2 9 0)
0
```

```
(modulo x y) → integer
  x : integer
  y : integer
```

Finds the remainder of the division of the first number by the second:

```
> (modulo 9 2)
1
> (modulo 3 -4)
-1
```

```
(negative? x) → boolean?
  x : real
```

Determines if some real number is strictly smaller than zero.

```
> (negative? -2)
#true
```

```
(number->string x) → string
  x : number
```

Converts a number to a string.

```
> (number->string 42)
"42"
```

```
(number->string-digits x p) → string
  x : number
  p : posint
```

Converts a number x to a string with the specified number of digits.

```
> (number->string-digits 0.9 2)
"0.9"
> (number->string-digits pi 4)
"3.1416"
```

```
(number? n) → boolean?
  n : any/c
```

Determines whether some value is a number:

```
> (number? "hello world")
#false
> (number? 42)
#true
```

```
(numerator x) → integer
  x : rational?
```

Computes the numerator of a rational.

```
> (numerator 2/3)
2
```

```
(odd? x) → boolean?
  x : integer
```

Determines if some integer (exact or inexact) is odd or not.

```
> (odd? 2)
#false
```

```
pi : real
```

The ratio of a circle's circumference to its diameter.

```
> pi
#i3.141592653589793
```

```
(positive? x) → boolean?
  x : real
```

Determines if some real number is strictly larger than zero.

```
> (positive? -2)
#false
```

```
(quotient x y) → integer
  x : integer
  y : integer
```

Divides the first integer—also called dividend—by the second—known as divisor—to obtain the quotient.

```
> (quotient 9 2)
4
> (quotient 3 4)
0
```

```
(random x) → natural?
  x : (and/c natural? positive?)
```

Generates a random natural number less than some given exact natural.

```
> (random 42)
36
```

```
(rational? x) → boolean?
  x : any/c
```

Determines whether some value is a rational number.

```
> (rational? 1)
#true
> (rational? -2.349)
#true
> (rational? #i1.23456789)
#true
> (rational? (sqrt -1))
#false
> (rational? pi)
#true
> (rational? e)
#true
> (rational? 1-2i)
#false
```

As the interactions show, the teaching languages considers many more numbers as rationals than expected. In particular, `pi` is a rational number because it is only a finite approximation

to the mathematical π. Think of `rational?` as a suggestion to think of these numbers as fractions.

```
(real-part x) → real
  x : number
```

Extracts the real part from a complex number.

```
> (real-part 3+4i)
3
```

```
(real? x) → boolean?
  x : any/c
```

Determines whether some value is a real number.

```
> (real? 1-2i)
#false
```

```
(remainder x y) → integer
  x : integer
  y : integer
```

Determines the remainder of dividing the first by the second integer (exact or inexact).

```
> (remainder 9 2)
1
> (remainder 3 4)
3
```

```
(round x) → integer
  x : real
```

Rounds a real number to an integer (rounds to even to break ties). See `floor` and `ceiling`.

```
> (round 12.3)
#i12.0
```

```
(sgn x) → (union 1 #i1.0 0 #i0.0 -1 #i-1.0)
  x : real
```

Determines the sign of a real number.

```
> (sgn -12)
-1
```

```
(sin x) → number
  x : number
```

Computes the sine of a number (radians).

```
> (sin pi)
#i1.2246467991473532e-16
```

```
(sinh x) → number
  x : number
```

Computes the hyperbolic sine of a number.

```
> (sinh 10)
#i11013.232874703393
```

```
(sqr x) → number
  x : number
```

Computes the square of a number.

```
> (sqr 8)
64
```

```
(sqrt x) → number
  x : number
```

Computes the square root of a number.

```
> (sqrt 9)
3
> (sqrt 2)
#i1.4142135623730951
```

```
(sub1 x) → number
  x : number
```

Decrements the given number.

```
> (sub1 2)
1
```

```
(tan x) → number
  x : number
```

Computes the tangent of a number (radians).

```
> (tan pi)
#i-1.2246467991473532e-16
```

```
(zero? x) → boolean?
  x : number
```

Determines if some number is zero or not.

```
> (zero? 2)
#false
```

1.7 Booleans

```
(boolean->string x) → string
  x : boolean?
```

Produces a string for the given boolean

```
> (boolean->string #false)
"#false"
> (boolean->string #true)
"#true"
```

```
(boolean=? x y) → boolean?
  x : boolean?
  y : boolean?
```

Determines whether two booleans are equal.

```
> (boolean=? #true #false)
#false
```

```
(boolean? x) → boolean?
  x : any/c
```

Determines whether some value is a boolean.

```
> (boolean? 42)
#false
> (boolean? #false)
#true
```

```
(false? x) → boolean?
  x : any/c
```

Determines whether a value is false.

```
> (false? #false)
#true
```

```
(not x) → boolean?
  x : boolean?
```

Negates a boolean value.

```
> (not #false)
#true
```

1.8 Symbols

```
(symbol->string x) → string
  x : symbol
```

Converts a symbol to a string.

```
> (symbol->string 'c)
"c"
```

```
(symbol=? x y) → boolean?
  x : symbol
  y : symbol
```

Determines whether two symbols are equal.

```
> (symbol=? 'a 'b)
#false
```

```
(symbol? x) → boolean?
  x : any/c
```

Determines whether some value is a symbol.

```
> (symbol? 'a)
#true
```

1.9 Lists

```
(append x y z ...) → list?
  x : list?
  y : list?
  z : list?
```

Creates a single list from several, by concatenation of the items.

```
> (append (cons 1 (cons 2 '())) (cons "a" (cons "b" empty)))
(list 1 2 "a" "b")
```

```
(assoc x l) → (union (listof any) #false)
  x : any/c
  l : (listof any)
```

Produces the first pair on `l` whose `first` is `equal?` to `x`; otherwise it produces `#false`.

```
> (assoc "hello" '(("world" 2) ("hello" 3) ("good" 0)))
(list "hello" 3)
```

```
(assq x l) → (union #false cons?)
  x : any/c
  l : list?
```

Determines whether some item is the first item of a pair in a list of pairs. (It compares the items with `eq?`.)

```
> a
(list (list 'a 22) (list 'b 8) (list 'c 70))
> (assq 'b a)
(list 'b 8)
```

```
(caaar x) → any/c
  x : list?
```

LISP-style selector: `(car (car (car x)))`.

```
> w
(list (list (list (list "bye") 3) #true) 42)
> (caaar w)
(list "bye")
```

```
(caadr x) → any/c
  x : list?
```

LISP-style selector: `(car (car (cdr x)))`.

```
> (caadr (cons 1 (cons (cons 'a '()) (cons (cons 'd '()) '()))))
'a
```

```
(caar x) → any/c
  x : list?
```

LISP-style selector: `(car (car x))`.

```
> y
(list (list (list 1 2 3) #false "world"))
> (caar y)
(list 1 2 3)
```

```
(cadar x) → any/c
  x : list?
```

LISP-style selector: `(car (cdr (car x)))`.

```
> w
(list (list (list (list "bye") 3) #true) 42)
> (cadar w)
#true
```

```
(cadddr x) → any/c
  x : list?
```

LISP-style selector: `(car (cdr (cdr (cdr x))))`.

```
> v
(list 1 2 3 4 5 6 7 8 9 'A)
> (cadddr v)
4
```

```
(caddr x) → any/c
  x : list?
```

LISP-style selector: `(car (cdr (cdr x)))`.

```
> x
(list 2 "hello" #true)
> (caddr x)
#true
```

```
(cadr x) → any/c
  x : list?
```

LISP-style selector: `(car (cdr x))`.

```
> x
(list 2 "hello" #true)
> (cadr x)
"hello"
```

```
(car x) → any/c
  x : cons?
```

Selects the first item of a non-empty list.

```
> x
(list 2 "hello" #true)
> (car x)
2
```

```
(cdaar x) → any/c
  x : list?
```

LISP-style selector: `(cdr (car (car x)))`.

```
> w
(list (list (list (list "bye") 3) #true) 42)
> (cdaar w)
(list 3)
```

```
(cdadr x) → any/c
  x : list?
```

LISP-style selector: `(cdr (car (cdr x)))`.

```
> (cdadr (list 1 (list 2 "a") 3))
(list "a")
```

```
(cdar x) → list?
  x : list?
```

LISP-style selector: `(cdr (car x))`.

```
> y
(list (list (list 1 2 3) #false "world"))
> (cdar y)
(list #false "world")
```

```
(cddar x) → any/c
  x : list?
```

LISP-style selector: `(cdr (cdr (car x)))`

```
> w
(list (list (list (list "bye") 3) #true) 42)
> (cddar w)
'()
```

```
(cdddr x) → any/c
  x : list?
```

LISP-style selector: `(cdr (cdr (cdr x)))`.

```
> v
(list 1 2 3 4 5 6 7 8 9 'A)
> (cdddr v)
(list 4 5 6 7 8 9 'A)
```

```
(cddr x) → list?
  x : list?
```

LISP-style selector: `(cdr (cdr x))`.

```
> x
(list 2 "hello" #true)
> (cddr x)
(list #true)
```

```
(cdr x) → any/c
  x : cons?
```

Selects the rest of a non-empty list.

```
> x
(list 2 "hello" #true)
> (cdr x)
(list "hello" #true)
```

```
(cons x y) → list?
  x : any/c
  y : list?
```

Constructs a list.

```
> (cons 1 '())
(cons 1 '())
```

```
(cons? x) → boolean?
  x : any/c
```

Determines whether some value is a constructed list.

```
> (cons? (cons 1 '()))
#true
> (cons? 42)
#false
```

```
(eighth x) → any/c
  x : list?
```

Selects the eighth item of a non-empty list.

```
> v
(list 1 2 3 4 5 6 7 8 9 'A)
> (eighth v)
8
```

```
(empty? x) → boolean?
  x : any/c
```

Determines whether some value is the empty list.

```
> (empty? '())
#true
> (empty? 42)
#false
```

```
(fifth x) → any/c
  x : list?
```

Selects the fifth item of a non-empty list.

```
> v
(list 1 2 3 4 5 6 7 8 9 'A)
> (fifth v)
5
```

```
(first x) → any/c
  x : cons?
```

Selects the first item of a non-empty list.

```
> x
(list 2 "hello" #true)
> (first x)
2
```

```
(fourth x) → any/c
  x : list?
```

Selects the fourth item of a non-empty list.

```
> v
(list 1 2 3 4 5 6 7 8 9 'A)
> (fourth v)
4
```

```
(length l) → natural?
  l : list?
```

Evaluates the number of items on a list.

```
> x
(list 2 "hello" #true)
> (length x)
3
```

```
(list x ...) → list?
  x : any/c
```

Constructs a list of its arguments.

```
> (list 1 2 3 4 5 6 7 8 9 0)
(cons 1 (cons 2 (cons 3 (cons 4 (cons 5 (cons 6 (cons 7 (cons 8
(cons 9 (cons 0 '()))))))))))
```

```
(list* x ... l) → list?
  x : any/c
  l : list?
```

Constructs a list by adding multiple items to a list.

```
> x
(list 2 "hello" #true)
> (list* 4 3 x)
(list 4 3 2 "hello" #true)
```

```
(list-ref x i) → any/c
  x : list?
  i : natural?
```

Extracts the indexed item from the list.

```
> v
(list 1 2 3 4 5 6 7 8 9 'A)
> (list-ref v 9)
'A
```

```
(list? x) → boolean?
  x : any/c
```

Checks whether the given value is a list.

```
> (list? 42)
#false
> (list? '())
#true
> (list? (cons 1 (cons 2 '())))
#true
```

```
(make-list i x) → list?
  i : natural?
  x : any/c
```

Constructs a list of *i* copies of *x*.

```
> (make-list 3 "hello")
(cons "hello" (cons "hello" (cons "hello" '())))
```

```
(member x l) → boolean?
  x : any/c
  l : list?
```

Determines whether some value is on the list (comparing values with equal?).

```
> x
(list 2 "hello" #true)
> (member "hello" x)
#true
```

```
(member? x l) → boolean?
  x : any/c
  l : list?
```

Determines whether some value is on the list (comparing values with equal?).

```
> x
(list 2 "hello" #true)
> (member? "hello" x)
#true
```

```
(memq x l) → boolean?
  x : any/c
  l : list?
```

Determines whether some value *x* is on some list *l*, using `eq?` to compare *x* with items on *l*.

```
> x
(list 2 "hello" #true)
> (memq (list (list 1 2 3)) x)
#false
```

```
(memq? x l) → boolean?
  x : any/c
  l : list?
```

Determines whether some value *x* is on some list *l*, using `eq?` to compare *x* with items on *l*.

```
> x
(list 2 "hello" #true)
> (memq? (list (list 1 2 3)) x)
#false
```

```
(memv x l) → (or/c #false list)
  x : any/c
  l : list?
```

Determines whether some value is on the list if so, it produces the suffix of the list that starts with x if not, it produces false. (It compares values with the eqv? predicate.)

```
> x
(list 2 "hello" #true)
> (memv (list (list 1 2 3)) x)
#false
```

```
null : list
```

Another name for the empty list

```
> null
'()
```

```
(null? x) → boolean?
  x : any/c
```

Determines whether some value is the empty list.

```
> (null? '())
#true
> (null? 42)
#false
```

```
(range start end step) → list?
  start : number
  end : number
  step : number
```

Constructs a list of numbers by *step*ping from *start* to *end*.

```
> (range 0 10 2)
(cons 0 (cons 2 (cons 4 (cons 6 (cons 8 '()))))
```

```
(remove x l) → list?
  x : any/c
  l : list?
```

Constructs a list like the given one, with the first occurrence of the given item removed (comparing values with equal?).

```
> x
(list 2 "hello" #true)
> (remove "hello" x)
(list 2 #true)
> hello-2
(list 2 "hello" #true "hello")
> (remove "hello" hello-2)
(list 2 #true "hello")
```

```
(remove-all x l) → list?
  x : any/c
  l : list?
```

Constructs a list like the given one, with all occurrences of the given item removed (comparing values with equal?).

```
> x
(list 2 "hello" #true)
> (remove-all "hello" x)
(list 2 #true)
> hello-2
(list 2 "hello" #true "hello")
> (remove-all "hello" hello-2)
(list 2 #true)
```

```
(rest x) → any/c
  x : cons?
```

Selects the rest of a non-empty list.

```
> x
(list 2 "hello" #true)
> (rest x)
(list "hello" #true)
```

```
(reverse l) → list
  l : list?
```

Creates a reversed version of a list.

```
> x
(list 2 "hello" #true)
> (reverse x)
(list #true "hello" 2)
```

```
(second x) → any/c
  x : list?
```

Selects the second item of a non-empty list.

```
> x
(list 2 "hello" #true)
> (second x)
"hello"
```

```
(seventh x) → any/c
  x : list?
```

Selects the seventh item of a non-empty list.

```
> v
(list 1 2 3 4 5 6 7 8 9 'A)
> (seventh v)
7
```

```
(sixth x) → any/c
  x : list?
```

Selects the sixth item of a non-empty list.

```
> v
(list 1 2 3 4 5 6 7 8 9 'A)
> (sixth v)
6
```

(third x) → any/c
  x : list?

Selects the third item of a non-empty list.

```
> x
(list 2 "hello" #true)
> (third x)
#true
```

1.10 Posns

(make-posn x y) → posn
  x : any/c
  y : any/c

Constructs a posn from two arbitrary values.

```
> (make-posn 3 3)
(make-posn 3 3)
> (make-posn "hello" #true)
(make-posn "hello" #true)
```

(posn-x p) → any/c
  p : posn

Extracts the x component of a posn.

```
> p
(make-posn 2 -3)
> (posn-x p)
2
```

(posn-y p) → any/c
  p : posn

Extracts the y component of a posn.

```
> p
(make-posn 2 -3)
> (posn-y p)
-3
```

```
(posn? x) → boolean?
  x : any/c
```

Determines if its input is a posn.

```
> q
(make-posn "bye" 2)
> (posn? q)
#true
> (posn? 42)
#false
```

1.11 Characters

```
(char->integer c) → integer
  c : char
```

Looks up the number that corresponds to the given character in the ASCII table (if any).

```
> (char->integer #\a)
97
> (char->integer #\z)
122
```

```
(char-alphabetic? c) → boolean?
  c : char
```

Determines whether a character represents an alphabetic character.

```
> (char-alphabetic? #\Q)
#true
```

```
(char-ci<=? c d e ...) → boolean?
  c : char
  d : char
  e : char
```

Determines whether the characters are ordered in an increasing and case-insensitive manner.

```
> (char-ci<=? #\b #\B)
#true
> (char<=? #\b #\B)
#false
```

```
(char-ci<? c d e ...) → boolean?
  c : char
  d : char
  e : char
```

Determines whether the characters are ordered in a strictly increasing and case-insensitive manner.

```
> (char-ci<? #\B #\c)
#true
> (char<? #\b #\B)
#false
```

```
(char-ci=? c d e ...) → boolean?
  c : char
  d : char
  e : char
```

Determines whether two characters are equal in a case-insensitive manner.

```
> (char-ci=? #\b #\B)
#true
```

```
(char-ci>=? c d e ...) → boolean?
  c : char
  d : char
  e : char
```

Determines whether the characters are sorted in a decreasing and case-insensitive manner.

```
> (char-ci>=? #\b #\C)
#false
> (char>=? #\b #\C)
#true
```

```
(char-ci>? c d e ...) → boolean?
  c : char
  d : char
  e : char
```

Determines whether the characters are sorted in a strictly decreasing and case-insensitive manner.

```
> (char-ci>? #\b #\B)
#false
> (char>? #\b #\B)
#true
```

```
(char-downcase c) → char
  c : char
```

Produces the equivalent lower-case character.

```
> (char-downcase #\T)
#\t
```

```
(char-lower-case? c) → boolean?
  c : char
```

Determines whether a character is a lower-case character.

```
> (char-lower-case? #\T)
#false
```

```
(char-numeric? c) → boolean?
  c : char
```

Determines whether a character represents a digit.

```
> (char-numeric? #\9)
#true
```

```
(char-upcase c) → char
  c : char
```

Produces the equivalent upper-case character.

```
> (char-upcase #\t)
#\T
```

```
(char-upper-case? c) → boolean?
  c : char
```

Determines whether a character is an upper-case character.

```
> (char-upper-case? #\T)
#true
```

```
(char-whitespace? c) → boolean?
  c : char
```

Determines whether a character represents space.

```
> (char-whitespace? #\tab)
#true
```

```
(char<=? c d e ...) → boolean?
  c : char
  d : char
  e : char
```

Determines whether the characters are ordered in an increasing manner.

```
> (char<=? #\a #\a #\b)
#true
```

```
(char<? x d e ...) → boolean?
  x : char
  d : char
  e : char
```

Determines whether the characters are ordered in a strictly increasing manner.

```
> (char<? #\a #\b #\c)
#true
```

```
(char=? c d e ...) → boolean?
  c : char
  d : char
  e : char
```

Determines whether the characters are equal.

```
> (char=? #\b #\a)
#false
```

```
(char>=? c d e ...) → boolean?
  c : char
  d : char
  e : char
```

Determines whether the characters are sorted in a decreasing manner.

```
> (char>=? #\b #\b #\a)
#true
```

```
(char>? c d e ...) → boolean?
  c : char
  d : char
  e : char
```

Determines whether the characters are sorted in a strictly decreasing manner.

```
> (char>? #\A #\z #\a)
#false
```

```
(char? x) → boolean?
  x : any/c
```

Determines whether a value is a character.

```
> (char? "a")
#false
> (char? #\a)
#true
```

1.12 Strings

```
(explode s) → (listof string)
  s : string
```

Translates a string into a list of 1-letter strings.

```
> (explode "cat")
(list "c" "a" "t")
```

```
(format f x ...) → string
  f : string
  x : any/c
```

Formats a string, possibly embedding values.

```
> (format "Dear Dr. ~a:" "Flatt")
"Dear Dr. Flatt:"
> (format "Dear Dr. ~s:" "Flatt")
"Dear Dr. \"Flatt\":"
```

```
(implode l) → string
  l : list?
```

Concatenates the list of 1-letter strings into one string.

```
> (implode (cons "c" (cons "a" (cons "t" '()))))
"cat"
```

```
(int->string i) → string
  i : integer
```

Converts an integer in [0,55295] or [57344 1114111] to a 1-letter string.

```
> (int->string 65)
"A"
```

```
(list->string l) → string
  l : list?
```

Converts a s list of characters into a string.

```
> (list->string (cons #\c (cons #\a (cons #\t '()))))
"cat"
```

(make-string *i* *c*) → string
  i : natural?
  c : char

Produces a string of length *i* from *c*.

```
> (make-string 3 #\d)
"ddd"
```

(replicate *i* *s*) → string
  i : natural?
  s : string

Replicates *s* *i* times.

```
> (replicate 3 "h")
"hhh"
```

(string *c* ...) → string?
  c : char

Builds a string of the given characters.

```
> (string #\d #\o #\g)
"dog"
```

(string->int *s*) → integer
  s : string

Converts a 1-letter string to an integer in [0,55295] or [57344, 1114111].

```
> (string->int "a")
97
```

```
(string->list s) → (listof char)
  s : string
```

Converts a string into a list of characters.

```
> (string->list "hello")
(list #\h #\e #\l #\l #\o)
```

```
(string->number s) → (union number #false)
  s : string
```

Converts a string into a number, produce false if impossible.

```
> (string->number "-2.03")
-2.03
> (string->number "1-2i")
1-2i
```

```
(string->symbol s) → symbol
  s : string
```

Converts a string into a symbol.

```
> (string->symbol "hello")
'hello
```

```
(string-alphabetic? s) → boolean?
  s : string
```

Determines whether all 'letters' in the string are alphabetic.

```
> (string-alphabetic? "123")
#false
> (string-alphabetic? "cat")
#true
```

```
(string-append s t z ...) → string
  s : string
  t : string
  z : string
```

Concatenates the characters of several strings.

```
> (string-append "hello" " " "world" " " "good bye")
"hello world good bye"
```

```
(string-ci<=? s t) → boolean?
  s : string
  t : string
```

Determines whether the strings are ordered in a lexicographically increasing and case-insensitive manner.

```
> (string-ci<=? "hello" "WORLD")
#true
```

```
(string-ci<? s t) → boolean?
  s : string
  t : string
```

Determines whether the strings are ordered in a lexicographically strictly increasing and case-insensitive manner.

```
> (string-ci<? "hello" "WORLD")
#true
```

```
(string-ci=? s t) → boolean?
  s : string
  t : string
```

Determines whether all strings are equal, character for character, regardless of case.

```
> (string-ci=?  "hello" "HellO")
#true
```

```
(string-ci>=? s t) → boolean?
  s : string
  t : string
```

Determines whether the strings are ordered in a lexicographically decreasing and case-insensitive manner.

```
> (string-ci>? "WORLD" "hello")
#true
```

```
(string-ci>? s t) → boolean?
  s : string
  t : string
```

Determines whether the strings are ordered in a lexicographically strictly decreasing and case-insensitive manner.

```
> (string-ci>?  "WORLD" "hello")
#true
```

```
(string-contains-ci? s t) → boolean?
  s : string
  t : string
```

Determines whether the first string appears in the second one without regard to the case of the letters.

```
> (string-contains-ci? "At" "caT")
#true
```

```
(string-contains? s t) → boolean?
  s : string
  t : string
```

Determines whether the first string appears literally in the second one.

```
> (string-contains? "at" "cat")
#true
```

```
(string-copy s) → string
  s : string
```

Copies a string.

```
> (string-copy "hello")
"hello"
```

```
(string-downcase s) → string
  s : string
```

Produces a string like the given one with all 'letters' as lower case.

```
> (string-downcase "CAT")
"cat"
> (string-downcase "cAt")
"cat"
```

```
(string-ith s i) → 1string?
  s : string
  i : natural?
```

Extracts the *i*th 1-letter substring from *s*.

```
> (string-ith "hello world" 1)
"e"
```

```
(string-length s) → nat
  s : string
```

Determines the length of a string.

```
> (string-length "hello world")
11
```

```
(string-lower-case? s) → boolean?
  s : string
```

Determines whether all 'letters' in the string are lower case.

```
> (string-lower-case? "CAT")
#false
```

```
(string-numeric? s) → boolean?
  s : string
```

Determines whether all 'letters' in the string are numeric.

```
> (string-numeric? "123")
#true
> (string-numeric? "1-2i")
#false
```

```
(string-ref s i) → char
  s : string
  i : natural?
```

Extracts the `i`th character from `s`.

```
> (string-ref "cat" 2)
#\t
```

```
(string-upcase s) → string
  s : string
```

Produces a string like the given one with all 'letters' as upper case.

```
> (string-upcase "cat")
"CAT"
> (string-upcase "cAt")
"CAT"
```

```
(string-upper-case? s) → boolean?
  s : string
```

Determines whether all 'letters' in the string are upper case.

```
> (string-upper-case? "CAT")
#true
```

```
(string-whitespace? s) → boolean?
  s : string
```

Determines whether all 'letters' in the string are white space.

```
> (string-whitespace? (string-append " " (string #\tab #\newline #\return)))
#true
```

```
(string<=? s t) → boolean?
  s : string
  t : string
```

Determines whether the strings are ordered in a lexicographically increasing manner.

```
> (string<=? "hello" "hello")
#true
```

```
(string<? s t) → boolean?
  s : string
  t : string
```

Determines whether the strings are ordered in a lexicographically strictly increasing manner.

```
> (string<? "hello" "world")
#true
```

```
(string=? s t) → boolean?
  s : string
  t : string
```

Determines whether all strings are equal, character for character.

```
> (string=? "hello" "world")
#false
> (string=? "bye" "bye")
#true
```

```
(string>=? s t) → boolean?
  s : string
  t : string
```

Determines whether the strings are ordered in a lexicographically decreasing manner.

```
> (string>=? "world" "hello")
#true
```

```
(string>? s t) → boolean?
  s : string
  t : string
```

Determines whether the strings are ordered in a lexicographically strictly decreasing manner.

```
> (string>? "world" "hello")
#true
```

```
(string? x) → boolean?
  x : any/c
```

Determines whether a value is a string.

```
> (string? "hello world")
#true
> (string? 42)
#false
```

```
(substring s i j) → string
  s : string
  i : natural?
  j : natural?
```

Extracts the substring starting at *i* up to *j* (or the end if *j* is not provided).

```
> (substring "hello world" 1 5)
"ello"
> (substring "hello world" 1 8)
"ello wo"
> (substring "hello world" 4)
"o world"
```

1.13 Images

```
(image=? i j) → boolean?
  i : image
  j : image
```

Determines whether two images are equal.

```
> c1
```

```
> (image=? (circle 5 "solid" "green") c1)
#false
> (image=? (circle 10 "solid" "green") c1)
#true
```

```
(image? x) → boolean?
  x : any/c
```

Determines whether a value is an image.

```
> c1
```

```
> (image? c1)
#true
```

1.14 Misc

```
(=~ x y eps) → boolean?
  x : number
  y : number
  eps : non-negative-real
```

Checks whether *x* and *y* are within *eps* of either other.

```
> (=~ 1.01 1.0 0.1)
#true
> (=~ 1.01 1.5 0.1)
#false
```

```
eof : eof-object?
```

A value that represents the end of a file:

```
> eof
#<eof>
```

```
(eof-object? x) → boolean?
  x : any/c
```

Determines whether some value is the end-of-file value.

```
> (eof-object? eof)
#true
> (eof-object? 42)
#false
```

```
(eq? x y) → boolean?
  x : any/c
  y : any/c
```

Determines whether two values are equivalent from the computer's perspective (intensional).

```
> (eq? (cons 1 '()) (cons 1 '()))
#false
> one
(list 1)
> (eq? one one)
#true
```

```
(equal? x y) → boolean?
  x : any/c
  y : any/c
```

Determines whether two values are structurally equal where basic values are compared with the eqv? predicate.

```
> (equal? (make-posn 1 2) (make-posn (- 2 1) (+ 1 1)))
#true
```

```
(equal~? x y z) → boolean?
  x : any/c
  y : any/c
  z : non-negative-real
```

Compares *x* and *y* like `equal?` but uses =~ in the case of numbers.

```
> (equal~? (make-posn 1.01 1.0) (make-posn 1.01 0.99) 0.2)
#true
```

```
(eqv? x y) → boolean?
  x : any/c
  y : any/c
```

Determines whether two values are equivalent from the perspective of all functions that can be applied to it (extensional).

```
> (eqv? (cons 1 '()) (cons 1 '()))
#false
> one
(list 1)
> (eqv? one one)
#true
```

```
(error x ...) → void?
  x : any/c
```

Signals an error, combining the given values into an error message. If any of the values' printed representations is too long, it is truncated and "..." is put into the string. If the first value is a symbol, it is suffixed with a colon and the result pre-pended on to the error message.

```
> zero
0
> (if (= zero 0) (error "can't divide by 0") (/ 1 zero))
can't divide by 0
```

```
(exit) → void
```

Evaluating `(exit)` terminates the running program.

```
(identity x) → any/c
  x : any/c
```

Returns x.

```
> (identity 42)
42
> (identity c1)
```

```
> (identity "hello")
"hello"
```

```
(struct? x) → boolean?
  x : any/c
```

Determines whether some value is a structure.

```
> (struct? (make-posn 1 2))
#true
> (struct? 43)
#false
```

1.15 Signatures

```
Any : signature?
```

Signature for any value.

```
Boolean : signature?
```

Signature for booleans.

```
Char : signature?
```

Signature for chararacters.

```
(ConsOf first-sig rest-sig) → signature?
  first-sig : signature?
  rest-sig : signature?
```

Signature for a cons pair.

```
EmptyList : signature?
```

Signature for the empty list.

```
False : signature?
```

Signature for just false.

```
Integer : signature?
```

Signature for integers.

```
Natural : signature?
```

Signature for natural numbers.

```
Number : signature?
```

Signature for arbitrary numbers.

```
Rational : signature?
```

Signature for rational numbers.

```
Real : signature?
```

Signature for real numbers.

```
String : signature?
```

Signature for strings.

```
Symbol : signature?
```

Signature for symbols.

```
True : signature?
```

Signature for just true.

2 Beginning Student with List Abbreviations

The grammar notation uses the notation `X` **...** (bold dots) to indicate that `X` may occur an arbitrary number of times (zero, one, or more). Separately, the grammar also defines `...` as an identifier to be used in templates.

See How to Design Programs/2e, Intermezzo 1 for an explanation of the Beginning Student Language.

See How to Design Programs/2e, Intermezzo 2 for an explanation of quoted lists.

```
                program = def-or-expr ...

           def-or-expr = definition
                       | expr
                       | test-case
                       | library-require
                       | signature-declaration

            definition = (define (name variable variable ...) expr)
                       | (define name expr)
                       | (define name (lambda (variable variable ...) expr))
                       | (define-struct name (name ...))

                 expr = (name expr expr ...)
                      | (prim-op expr ...)
                      | (cond [expr expr] ... [expr expr])
                      | (cond [expr expr] ... [else expr])
                      | (if expr expr expr)
                      | (and expr expr expr ...)
                      | (or expr expr expr ...)
                      | name
                      | 'quoted
                      | `quasiquoted
                      | '()
                      | number
                      | boolean
                      | string
                      | character

signature-declaration = (: name signature-form)

       signature-form = (enum expr ...)
                      | (mixed signature-form ...)
                      | (signature-form ... -> signature-form)
```

```
                    | (ListOf signature-form)
                    | signature-variable
                    | expr

signature-variable = %name

            quoted = name
                   | number
                   | string
                   | character
                   | (quoted ...)
                   | 'quoted
                   | `quoted
                   | ,quoted
                   | ,@quoted

       quasiquoted = name
                   | number
                   | string
                   | character
                   | (quasiquoted ...)
                   | 'quasiquoted
                   | `quasiquoted
                   | ,expr
                   | ,@expr

         test-case = (check-expect expr expr)
                   | (check-random expr expr)
                   | (check-within expr expr expr)
                   | (check-random-within expr expr expr)
                   | (check-member-of expr expr ...)
                   | (check-range expr expr expr)
                   | (check-satisfied expr name)
                   | (check-error expr expr)
                   | (check-error expr)

   library-require = (require string)
                   | (require (lib string string ...))
                   | (require (planet string package))

           package = (string string number number)
```

A *name* or a *variable* is a sequence of characters not including a space or one of the following:

```
" , ' ` ( ) [ ] { } | ; #
```

A *number* is a number such as 123, 3/2, or 5.5.

A *boolean* is one of: #true or #false.

Alternative spellings for the #true constant are #t, true, and #T. Similarly, #f, false, or #F are also recognized as #false.

A *symbol* is a quote character followed by a name. A symbol is a value, just like 42, '(), or #false.

A *string* is a sequence of characters enclosed by a pair of ". Unlike symbols, strings may be split into characters and manipulated by a variety of functions. For example, "abcdef", "This is a string", and "This is a string with \" inside" are all strings.

A *character* begins with #\ and has the name of the character. For example, #\a, #\b, and #\space are characters.

In function calls, the function appearing immediately after the open parenthesis can be any functions defined with define or define-struct, or any one of the pre-defined functions.

2.1 Pre-defined Variables

empty : empty?

The empty list.

true : boolean?

The #true value.

false : boolean?

The #false value.

2.2 Template Variables

. .

A placeholder for indicating that a function definition is a template.

. . .

A placeholder for indicating that a function definition is a template.

```
....
```

A placeholder for indicating that a function definition is a template.

```
.....
```

A placeholder for indicating that a function definition is a template.

```
......
```

A placeholder for indicating that a function definition is a template.

2.3 Syntaxes for Beginning Student with List Abbreviations

```
'name
'part
(quote name)
(quote part)
```

A quoted name is a symbol. A quoted part is an abbreviation for a nested lists.

Normally, this quotation is written with a `'`, like `'(apple banana)`, but it can also be written with `quote`, like `(quote (apple banana))`.

```
`name
`part
(quasiquote name)
(quasiquote part)
```

Like `quote`, but also allows escaping to expression "unquotes."

Normally, quasi-quotations are written with a backquote, `` ` ``, like `` `(apple ,(+ 1 2)) ``, but they can also be written with `quasiquote`, like `(quasiquote (apple ,(+ 1 2)))`.

```
,expression
(unquote expression)
```

Under a single quasiquote, `,`*expression* escapes from the quote to include an evaluated expression whose result is inserted into the abbreviated list.

Under multiple quasiquotes, `,`*expression* is really the literal `,`*expression*, decrementing the quasiquote count by one for *expression*.

Normally, an unquote is written with `,`, but it can also be written with `unquote`.

```
,@expression
(unquote-splicing expression)
```

Under a single quasiquote, `,@`*expression* escapes from the quote to include an evaluated expression whose result is a list to splice into the abbreviated list.

Under multiple quasiquotes, a splicing unquote is like an unquote; that is, it decrements the quasiquote count by one.

Normally, a splicing unquote is written with `,`, but it can also be written with `unquote-splicing`.

2.4 Common Syntaxes

The following syntaxes behave the same in the *Beginner with List Abbreviations* level as they did in the §1 "Beginning Student" level.

```
(define (name variable variable ...) expression)
```

Defines a function named *name*. The *expression* is the body of the function. When the function is called, the values of the arguments are inserted into the body in place of the *variable*s. The function returns the value of that new expression.

The function name's cannot be the same as that of another function or variable.

```
(define name expression)
```

Defines a variable called *name* with the the value of *expression*. The variable name's cannot be the same as that of another function or variable, and *name* itself must not appear in *expression*.

```
(define name (lambda (variable variable ...) expression))
```

An alternate way to defining functions. The *name* is the name of the function, which cannot be the same as that of another function or variable.

A `lambda` cannot be used outside of this alternate syntax.

```
(define-struct structure-name (field-name ...))
```

Defines a new structure called *structure-name*. The structure's fields are named by the *field-name*s. After the `define-struct`, the following new functions are available:

- `make-`*structure-name* : takes a number of arguments equal to the number of fields in the structure, and creates a new instance of that structure.
- *structure-name-field-name* : takes an instance of the structure and returns the value in the field named by *field-name*.
- *structure-name*`?` : takes any value, and returns `#true` if the value is an instance of the structure.

The name of the new functions introduced by `define-struct` must not be the same as that of other functions or variables, otherwise `define-struct` reports an error.

```
(name expression expression ...)
```

Calls the function named *name*. The value of the call is the value of *name*'s body when every one of the function's variables are replaced by the values of the corresponding *expression*s.

The function named *name* must defined before it can be called. The number of argument *expression*s must be the same as the number of arguments expected by the function.

```
(cond [question-expression answer-expression] ...)
(cond [question-expression answer-expression]
      ...
      [else answer-expression])
```

Chooses a clause based on some condition. `cond` finds the first *question-expression* that evaluates to `#true`, then evaluates the corresponding *answer-expression*.

If none of the *question-expression*s evaluates to `#true`, `cond`'s value is the *answer-expression* of the `else` clause. If there is no `else`, `cond` reports an error. If the result of a *question-expression* is neither `#true` nor `#false`, `cond` also reports an error.

`else` cannot be used outside of `cond`.

```
(if question-expression
    then-answer-expression
    else-answer-expression)
```

When the value of the *question-expression* is `#true`, `if` evaluates the *then-answer-expression*. When the test is `#false`, `if` evaluates the *else-answer-expression*.

If the *question-expression* is neither `#true` nor `#false`, `if` reports an error.

```
(and expression expression expression ...)
```

Evaluates to `#true` if all the *expression*s are `#true`. If any *expression* is `#false`, the `and` expression evaluates to `#false` (and the expressions to the right of that expression are not evaluated.)

If any of the expressions evaluate to a value other than `#true` or `#false`, `and` reports an error.

```
(or expression expression expression ...)
```

Evaluates to `#true` as soon as one of the *expression*s is `#true` (and the expressions to the right of that expression are not evaluated.) If all of the *expression*s are `#false`, the `or` expression evaluates to `#false`.

If any of the expressions evaluate to a value other than `#true` or `#false`, `or` reports an error.

```
(check-expect expression expected-expression)
```

Checks that the first *expression* evaluates to the same value as the *expected-expression*.

```
(check-expect (fahrenheit->celsius 212) 100)
(check-expect (fahrenheit->celsius -40) -40)

(define (fahrenheit->celsius f)
  (* 5/9 (- f 32)))
```

A `check-expect` expression must be placed at the top-level of a student program. Also it may show up anywhere in the program, including ahead of the tested function definition. By placing `check-expect`s there, a programmer conveys to a future reader the intention behind the program with working examples, thus making it often superfluous to read the function definition proper. Syntax errors in `check-expect` (and all check forms) are intentionally delayed to run time so that students can write tests *without* necessarily writing complete function headers.

It is an error for `expr` or `expected-expr` to produce an inexact number or a function value. As for inexact numbers, it is *morally* wrong to compare them for plain equality. Instead one

tests whether they are both within a small interval; see `check-within`. As for functions (see Intermediate and up), it is provably impossible to compare functions.

```
(check-random expression expected-expression)
```

Checks that the first *expression* evaluates to the same value as the *expected-expression*.

The form supplies the same random-number generator to both parts. If both parts request `random` numbers from the same interval in the same order, they receive the same random numbers.

Here is a simple example of where `check-random` is useful:

```
(define WIDTH 100)
(define HEIGHT (* 2 WIDTH))

(define-struct player (name x y))
; A Player is (make-player String Nat Nat)

; String -> Player

(check-random (create-randomly-placed-player "David Van Horn")
              (make-player "David Van Horn" (random WIDTH) (random HEIGHT)))

(define (create-randomly-placed-player name)
  (make-player name (random WIDTH) (random HEIGHT)))
```

Note how `random` is called on the same numbers in the same order in both parts of `check-random`. If the two parts call `random` for different intervals, they are likely to fail:

```
; String -> Player

(check-random (create-randomly-placed-player "David Van Horn")
              (make-player "David Van Horn" (random WIDTH) (random HEIGHT)))

(define (create-randomly-placed-player name)
  (a-helper-function name (random HEIGHT)))

; String Number -> Player
(define (a-helper-function name height)
   (make-player name (random WIDTH) height))
```

Because the argument to `a-helper-function` is evaluated first, `random` is first called for the interval [0,*HEIGHT*) and then for [0,*WIDTH*), that is, in a different order than in the preceding `check-random`.

It is an error for `expr` or `expected-expr` to produce a function value or an inexact number; see note on `check-expect` for details.

```
(check-satisfied expression predicate)
```

Checks that the first *expression* satisfies the named *predicate* (function of one argument). Recall that "satisfies" means "the function produces `#true` for the given value."

Here are simple examples for `check-satisfied`:

```
> (check-satisfied 1 odd?)
The test passed!


> (check-satisfied 1 even?)
Ran 1 test.
0 tests passed.
Check failures:

        Actual value ┌─┐ does not satisfy even?.
                     │1│
                     └─┘
at line 3, column 0
```

In general `check-satisfied` empowers program designers to use defined functions to formulate test suites:

```
; [cons Number [List-of Number]] -> Boolean
; a function for testing htdp-sort

(check-expect (sorted? (list 1 2 3)) #true)
(check-expect (sorted? (list 2 1 3)) #false)

(define (sorted? l)
  (cond
    [(empty? (rest l)) #true]
    [else (and (<= (first l) (second l)) (sorted? (rest l)))]))

; [List-of Number] -> [List-of Number]
; create a sorted version of the given list of numbers

(check-satisfied (htdp-sort (list 1 2 0 3)) sorted?)

(define (htdp-sort l)
  (cond
    [(empty? l) l]
```

```
    [else (insert (first l) (htdp-sort (rest l)))]))

; Number [List-of Number] -> [List-of Number]
; insert x into l at proper place
; assume l is arranged in ascending order
; the result is sorted in the same way
(define (insert x l)
  (cond
    [(empty? l) (list x)]
    [else (if (<= x (first l)) (cons x l) (cons (first l) (insert x (rest l))))]))
```

And yes, the results of `htdp-sort` satisfy the `sorted?` predicate:

```
> (check-satisfied (htdp-sort (list 1 2 0 3)) sorted?)
```

`(check-within expression expected-expression delta)`

Checks whether the value of the *`expression`* expression is structurally equal to the value produced by the *`expected-expression`* expression; every number in the first expression must be within *`delta`* of the corresponding number in the second expression.

```
(define-struct roots (x sqrt))
; RT is [List-of (make-roots Number Number)]

(define (root-of a)
  (make-roots a (sqrt a)))

(define (roots-table xs)
  (cond
    [(empty? xs) '()]
    [else (cons (root-of (first xs)) (roots-table (rest xs)))]))
```

Due to the presence of inexact numbers in nested data, `check-within` is the correct choice for testing, and the test succeeds if *`delta`* is reasonably large:

Example:

```
> (check-within (roots-table (list 1.0 2.0 3.0))
                (list
                  (make-roots 1.0 1.0)
                  (make-roots 2  1.414)
                  (make-roots 3  1.713))
                0.1)
The test passed!
```

In contrast, when *delta* is small, the test fails:

Example:

```
> (check-within (roots-table (list 2.0))
                (list
                  (make-roots 2  1.414))
                #i1e-5)
Ran 1 test.
0 tests passed.
Check failures:

        Actual value | '((make-roots 2.0 1.4142135623730951)) | is
not within 1e-5 of expected value | '((make-roots 2 1.414)) |.


at line 5, column 0
```

It is an error for `expressions` or *expected-expression* to produce a function value; see note on `check-expect` for details.

If *delta* is not a number, `check-within` reports an error.

```
(check-random-within expression expected-expression delta)
```

Combines `check-random` and `check-within`. Like `check-random`, *expression* and *expected-expression* are evaluated with the same random-number sequence, so calls to `random` in both expressions draw the same numbers. Like `check-within`, the test then succeeds if every number in the value of *expression* is within *delta* of the corresponding number in the value of *expected-expression*.

Use `check-random-within` in place of `check-random` when the random computation produces inexact numbers, which `check-random` refuses to compare.

If *delta* is not a number, `check-random-within` reports an error.

Added in version 1.9 of package `htdp-lib`.

```
(check-error expression expected-error-message)
(check-error expression)
```

Checks that the *expression* reports an error, where the error messages matches the value of *expected-error-message*, if it is present.

Here is a typical beginner example that calls for a use of `check-error`:

```
(define sample-table
  '(("matthias" 10)
    ("matthew"  20)
    ("robby"    -1)
    ("shriram"  18)))

; [List-of [list String Number]] String -> Number
; determine the number associated with s in table

(define (lookup table s)
  (cond
    [(empty? table) (error (string-append s " not found"))]
    [else (if (string=? (first (first table)) s)
              (second (first table))
              (lookup (rest table)))]))
```

Consider the following two examples in this context:

Example:

```
> (check-expect (lookup sample-table "matthew") 20)
The test passed!
```

Example:

```
> (check-error (lookup sample-table "kathi") "kathi not found")
The test passed!
```

(check-member-of *expression expression expression ...*)

Checks that the value of the first *expression* is that of one of the following *expression*s.

```
; [List-of X] -> X
; pick a random element from the given list l
(define (pick-one l)
  (list-ref l (random (length l))))
```

Example:

```
> (check-member-of (pick-one '("a" "b" "c")) "a" "b" "c")
The test passed!
```

It is an error for any of expressions to produce a function value; see note on check-expect for details.

```
(check-range expression low-expression high-expression)
```

Checks that the value of the first *expression* is a number in between the value of the *low-expression* and the *high-expression*, inclusive.

A `check-range` form is best used to delimit the possible results of functions that compute inexact numbers:

```
(define EPSILON 0.001)

; [Real -> Real] Real -> Real
; what is the slope of f at x?
(define (differentiate f x)
  (slope f (- x EPSILON) (+ x EPSILON)))

; [Real -> Real] Real Real -> Real
(define (slope f left right)
  (/ (- (f right) (f left))
     2 EPSILON))

(check-range (differentiate sin 0) 0.99 1.0)
```

It is an error for *expression*, *low-expression*, or *high-expression* to produce a function value; see note on `check-expect` for details.

```
(require string)
```

Makes the definitions of the module specified by *string* available in the current module (i.e., the current file), where *string* refers to a file relative to the current file.

The *string* is constrained in several ways to avoid problems with different path conventions on different platforms: a `/` is a directory separator, `.` always means the current directory, `..` always means the parent directory, path elements can use only `a` through `z` (uppercase or lowercase), `0` through `9`, `-`, `_`, and `.`, and the string cannot be empty or contain a leading or trailing `/`.

```
(require module-name)
```

Accesses a file in an installed library. The library name is an identifier with the same constraints as for a relative-path string (though without the quotes), with the additional constraint that it must not contain a `..`.

```
(require (lib string string ...))
```

Accesses a file in an installed library, making its definitions available in the current module (i.e., the current file). The first *string* names the library file, and the remaining *string*s name the collection (and sub-collection, and so on) where the file is installed. Each string is constrained in the same way as for the `(require string)` form.

```
(require (planet string (string string number number)))
(require (planet id))
(require (planet string))
```

Accesses a library that is distributed on the internet via the PLaneT server, making it definitions available in the current module (i.e., current file).

The full grammar for planet requires is given in §3.2 "Importing and Exporting: `require` and `provide`", but the best place to find examples of the syntax is on the the PLaneT server, in the description of a specific package.

2.5 Signatures

Signatures do not have to be comment: They can also be part of the code. When a signature is attached to a function, DrRacket will check that program uses the function in accordance with the signature and display signature violations along with the test results.

A signature is a regular value, and is specified as a *signature form*, a special syntax that only works with `:` signature declarations and inside `signature` expressions.

```
(: name signature-form)
```

This attaches the signature specified by *signature-form* to the definition of *name*. There must be a definition of *name* somewhere in the program.

```
(: age Integer)
(define age 42)

(: area-of-square (Number -> Number))
(define (area-of-square len)
  (sqr len))
```

On running the program, Racket checks whether the signatures attached with `:` actually match the value of the variable. If they don't, Racket reports *signature violation* along with test failures.

For example, this piece of code:

```
(: age Integer)
(define age "fortytwo")
```

Yields this output:

```
1 signature violation.

Signature violations:
        got "fortytwo" at line 2, column 12, signature at line 1,
column 7
```

Note that a signature violation does not stop the running program.

`(signature` *`signature-form`*`)`

This returns the signature described by *`signature-form`* as a value.

2.5.1 Signature Forms

Any expression can be a signature form, in which case the signature is the value returned by that expression. There are a few special signature forms, however:

In a signature form, any name that starts with a `%` is a *signature variable* that stands for any signature depending on how the signature is used.

Example:

```
(: same (%a -> %a))

(define (same x) x)
```

`(`*`input-signature-form ...`* `->` *`output-signature-form`*`)`

This signature form describes a function with inputs described by the *`input-signature-form`*s and output described by *`output-signature-form`*.

`(enum` *`expr ...`*`)`

This signature describes an enumeration of the values returned by the *`expr`*s.

Example:

```
(: cute? ((enum "cat" "snake") -> Boolean))

(define (cute? pet)
  (cond
    [(string=? pet "cat") #t]
    [(string=? pet "snake") #f]))
```

```
(mixed signature-form ...)
```

This signature describes mixed data, i.e. an itemization where each of the cases has a signature described by a *signature-form*.

Example:

```
(define SIGS (signature (mixed Aim Fired)))
```

```
(ListOf signature-form)
```

This signature describes a list where the elements are described by *signature-form*.

```
(predicate expression)
```

This signature describes values through a predicate: *expression* must evaluate to a function of one argument that returns a boolean. The signature matches all values for which the predicate returns `#true`.

2.5.2 Struct Signatures

A `define-struct` form defines two additional names that can be used in signatures. For a struct called `struct`, these are `Struct` and `StructOf`. Note that these names are capitalized. In particular, a struct called `Struct`, will also define `Struct` and `StructOf`. Moreover, when forming the additional names, hyphens are removed, and each letter following a hyphen is capitalized - so a struct called `foo-bar` will define `FooBar` and `FooBarOf`.

`Struct` is a signature that describes struct values from this structure type. `StructOf` is a function that takes as input a signature for each field. It returns a signature describing values of this structure type, additionally describing the values of the fields of the value.

```
(define-struct pair [fst snd])

(: add-pair ((PairOf Number Number) -> Number))
(define (add-pair p)
  (+ (pair-fst p) (pair-snd p)))
```

2.6 Pre-defined Functions

The remaining subsections list those functions that are built into the programming language. All other functions are imported from a teachpack or must be defined in the program.

2.7 Numbers: Integers, Rationals, Reals, Complex, Exacts, Inexacts

```
(* x y z ...) → number
  x : number
  y : number
  z : number
```

Multiplies all numbers.

```
> (* 5 3)
15
> (* 5 3 2)
30
```

```
(+ x y z ...) → number
  x : number
  y : number
  z : number
```

Adds up all numbers.

```
> (+ 2/3 1/16)
35/48
> (+ 3 2 5 8)
18
```

```
(- x y ...) → number
  x : number
  y : number
```

Subtracts the second (and following) number(s) from the first ; negates the number if there is only one argument.

```
> (- 5)
-5
```

```
> (- 5 3)
2
> (- 5 3 1)
1
```

```
(/ x y z ...) → number
  x : number
  y : number
  z : number
```

Divides the first by the second (and all following) number(s).

```
> (/ 12 2)
6
> (/ 12 2 3)
2
```

```
(< x y z ...) → boolean?
  x : real
  y : real
  z : real
```

Compares two or more (real) numbers for less-than.

```
> (< 42 2/5)
#false
```

```
(<= x y z ...) → boolean?
  x : real
  y : real
  z : real
```

Compares two or more (real) numbers for less-than or equality.

```
> (<= 42 2/5)
#false
```

```
(= x y z ...) → boolean?
  x : number
  y : number
  z : number
```

Compares two or more numbers for equality.

```
> (= 42 2/5)
#false
```

```
(> x y z ...) → boolean?
  x : real
  y : real
  z : real
```

Compares two or more (real) numbers for greater-than.

```
> (> 42 2/5)
#true
```

```
(>= x y z ...) → boolean?
  x : real
  y : real
  z : real
```

Compares two or more (real) numbers for greater-than or equality.

```
> (>= 42 42)
#true
```

```
(abs x) → real
  x : real
```

Determines the absolute value of a real number.

```
> (abs -12)
12
```

```
(acos x) → number
  x : number
```

Computes the arccosine (inverse of cos) of a number.

```
> (acos 0)
#i1.5707963267948966
```

```
(add1 x) → number
  x : number
```

Increments the given number.

```
> (add1 2)
3
```

```
(angle x) → real
  x : number
```

Extracts the angle from a complex number.

```
> (angle (make-polar 3 4))
#i-2.2831853071795867
```

```
(asin x) → number
  x : number
```

Computes the arcsine (inverse of sin) of a number.

```
> (asin 0)
0
```

```
(atan x) → number
  x : number
```

Computes the arctangent of the given number:

```
> (atan 0)
0
> (atan 0.5)
#i0.46364760900080615
```

Also comes in a two-argument version where `(atan y x)` computes `(atan (/ y x))` but the signs of `y` and `x` determine the quadrant of the result and the result tends to be more accurate than that of the 1-argument version in borderline cases:

```
> (atan 3 4)
#i0.6435011087932844
> (atan -2 -1)
#i-2.0344439357957027
```

```
(ceiling x) → integer
  x : real
```

Determines the closest integer (exact or inexact) above a real number. See `round`.

```
> (ceiling 12.3)
#i13.0
```

```
(complex? x) → boolean?
  x : any/c
```

Determines whether some value is complex.

```
> (complex? 1-2i)
#true
```

```
(conjugate x) → number
  x : number
```

Flips the sign of the imaginary part of a complex number.

```
> (conjugate 3+4i)
3-4i
> (conjugate -2-5i)
-2+5i
> (conjugate (make-polar 3 4))
#i-1.960930862590836+2.270407485923785i
```

```
(cos x) → number
  x : number
```

Computes the cosine of a number (radians).

```
> (cos pi)
#i-1.0
```

```
(cosh x) → number
  x : number
```

Computes the hyperbolic cosine of a number.

```
> (cosh 10)
#i11013.232920103324
```

```
(current-seconds) → integer
```

Determines the current time in seconds elapsed (since a platform-specific starting date).

```
> (current-seconds)
1793426324
```

```
(denominator x) → integer
  x : rational?
```

Computes the denominator of a rational.

```
> (denominator 2/3)
3
```

```
e : real
```

Euler's number.

```
> e
#i2.718281828459045
```

```
(even? x) → boolean?
  x : integer
```

Determines if some integer (exact or inexact) is even or not.

```
> (even? 2)
#true
```

```
(exact->inexact x) → number
  x : number
```

Converts an exact number to an inexact one.

```
> (exact->inexact 12)
#i12.0
```

```
(exact? x) → boolean?
  x : number
```

Determines whether some number is exact.

```
> (exact? (sqrt 2))
#false
```

```
(exp x) → number
  x : number
```

Determines e raised to a number.

```
> (exp -2)
#i0.1353352832366127
```

```
(expt x y) → number
  x : number
  y : number
```

Computes the power of the first to the second number, which is to say, exponentiation.

```
> (expt 16 1/2)
4
> (expt 3 -4)
1/81
```

```
(floor x) → integer
  x : real
```

Determines the closest integer (exact or inexact) below a real number. See `round`.

```
> (floor 12.3)
#i12.0
```

```
(gcd x y ...) → integer
  x : integer
  y : integer
```

Determines the greatest common divisor of two integers (exact or inexact).

```
> (gcd 6 12 8)
2
```

```
(imag-part x) → real
  x : number
```

Extracts the imaginary part from a complex number.

```
> (imag-part 3+4i)
4
```

```
(inexact->exact x) → number
  x : number
```

Approximates an inexact number by an exact one.

```
> (inexact->exact 12.0)
12
```

```
(inexact? x) → boolean?
  x : number
```

Determines whether some number is inexact.

```
> (inexact? 1-2i)
#false
```

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

Looks up the character that corresponds to the given exact integer in the ASCII table (if any).

```
> (integer->char 42)
#\*
```

```
(integer-sqrt x) → complex
  x : integer
```

Computes the integer or imaginary-integer square root of an integer.

```
> (integer-sqrt 11)
3
> (integer-sqrt -11)
0+3i
```

```
(integer? x) → boolean?
  x : any/c
```

Determines whether some value is an integer (exact or inexact).

```
> (integer? (sqrt 2))
#false
```

```
(lcm x y ...) → integer
  x : integer
  y : integer
```

Determines the least common multiple of two integers (exact or inexact).

```
> (lcm 6 12 8)
24
```

```
(log x) → number
  x : number
```

Determines the base-e logarithm of a number.

```
> (log 12)
#i2.4849066497880004
```

```
(magnitude x) → real
  x : number
```

Determines the magnitude of a complex number.

```
> (magnitude (make-polar 3 4))
#i3.0000000000000004
```

```
(make-polar x y) → number
  x : real
  y : real
```

Creates a complex from a magnitude and angle.

```
> (make-polar 3 4)
#i-1.960930862590836-2.270407485923785i
```

```
(make-rectangular x y) → number
  x : real
  y : real
```

Creates a complex from a real and an imaginary part.

```
> (make-rectangular 3 4)
3+4i
```

```
(max x y ...) → real
  x : real
  y : real
```

Determines the largest number—aka, the maximum.

```
> (max 3 2 8 7 2 9 0)
9
```

```
(min x y ...) → real
  x : real
  y : real
```

Determines the smallest number—aka, the minimum.

```
> (min 3 2 8 7 2 9 0)
0
```

```
(modulo x y) → integer
  x : integer
  y : integer
```

Finds the remainder of the division of the first number by the second:

```
> (modulo 9 2)
1
> (modulo 3 -4)
-1
```

```
(negative? x) → boolean?
  x : real
```

Determines if some real number is strictly smaller than zero.

```
> (negative? -2)
#true
```

```
(number->string x) → string
  x : number
```

Converts a number to a string.

```
> (number->string 42)
"42"
```

```
(number->string-digits x p) → string
  x : number
  p : posint
```

Converts a number x to a string with the specified number of digits.

```
> (number->string-digits 0.9 2)
"0.9"
> (number->string-digits pi 4)
"3.1416"
```

```
(number? n) → boolean?
  n : any/c
```

Determines whether some value is a number:

```
> (number? "hello world")
#false
> (number? 42)
#true
```

```
(numerator x) → integer
  x : rational?
```

Computes the numerator of a rational.

```
> (numerator 2/3)
2
```

```
(odd? x) → boolean?
  x : integer
```

Determines if some integer (exact or inexact) is odd or not.

```
> (odd? 2)
#false
```

```
pi : real
```

The ratio of a circle's circumference to its diameter.

```
> pi
#i3.141592653589793
```

```
(positive? x) → boolean?
  x : real
```

Determines if some real number is strictly larger than zero.

```
> (positive? -2)
#false
```

```
(quotient x y) → integer
  x : integer
  y : integer
```

Divides the first integer—also called dividend—by the second—known as divisor—to obtain the quotient.

```
> (quotient 9 2)
4
> (quotient 3 4)
0
```

```
(random x) → natural?
  x : (and/c natural? positive?)
```

Generates a random natural number less than some given exact natural.

```
> (random 42)
36
```

```
(rational? x) → boolean?
  x : any/c
```

Determines whether some value is a rational number.

```
> (rational? 1)
#true
> (rational? -2.349)
#true
> (rational? #i1.23456789)
#true
> (rational? (sqrt -1))
#false
> (rational? pi)
#true
> (rational? e)
#true
> (rational? 1-2i)
#false
```

As the interactions show, the teaching languages considers many more numbers as rationals than expected. In particular, `pi` is a rational number because it is only a finite approximation

to the mathematical π. Think of `rational?` as a suggestion to think of these numbers as fractions.

```
(real-part x) → real
  x : number
```

Extracts the real part from a complex number.

```
> (real-part 3+4i)
3
```

```
(real? x) → boolean?
  x : any/c
```

Determines whether some value is a real number.

```
> (real? 1-2i)
#false
```

```
(remainder x y) → integer
  x : integer
  y : integer
```

Determines the remainder of dividing the first by the second integer (exact or inexact).

```
> (remainder 9 2)
1
> (remainder 3 4)
3
```

```
(round x) → integer
  x : real
```

Rounds a real number to an integer (rounds to even to break ties). See `floor` and `ceiling`.

```
> (round 12.3)
#i12.0
```

```
(sgn x) → (union 1 #i1.0 0 #i0.0 -1 #i-1.0)
  x : real
```

Determines the sign of a real number.

```
> (sgn -12)
-1
```

```
(sin x) → number
  x : number
```

Computes the sine of a number (radians).

```
> (sin pi)
#i1.2246467991473532e-16
```

```
(sinh x) → number
  x : number
```

Computes the hyperbolic sine of a number.

```
> (sinh 10)
#i11013.232874703393
```

```
(sqr x) → number
  x : number
```

Computes the square of a number.

```
> (sqr 8)
64
```

```
(sqrt x) → number
  x : number
```

Computes the square root of a number.

```
> (sqrt 9)
3
> (sqrt 2)
#i1.4142135623730951
```

```
(sub1 x) → number
  x : number
```

Decrements the given number.

```
> (sub1 2)
1
```

```
(tan x) → number
  x : number
```

Computes the tangent of a number (radians).

```
> (tan pi)
#i-1.2246467991473532e-16
```

```
(zero? x) → boolean?
  x : number
```

Determines if some number is zero or not.

```
> (zero? 2)
#false
```

2.8 Booleans

```
(boolean->string x) → string
  x : boolean?
```

Produces a string for the given boolean

```
> (boolean->string #false)
"#false"
> (boolean->string #true)
"#true"
```

```
(boolean=? x y) → boolean?
  x : boolean?
  y : boolean?
```

Determines whether two booleans are equal.

```
> (boolean=? #true #false)
#false
```

```
(boolean? x) → boolean?
  x : any/c
```

Determines whether some value is a boolean.

```
> (boolean? 42)
#false
> (boolean? #false)
#true
```

```
(false? x) → boolean?
  x : any/c
```

Determines whether a value is false.

```
> (false? #false)
#true
```

```
(not x) → boolean?
  x : boolean?
```

Negates a boolean value.

```
> (not #false)
#true
```

2.9 Symbols

```
(symbol->string x) → string
  x : symbol
```

Converts a symbol to a string.

```
> (symbol->string 'c)
"c"
```

```
(symbol=? x y) → boolean?
  x : symbol
  y : symbol
```

Determines whether two symbols are equal.

```
> (symbol=? 'a 'b)
#false
```

```
(symbol? x) → boolean?
  x : any/c
```

Determines whether some value is a symbol.

```
> (symbol? 'a)
#true
```

2.10 Lists

```
(append x y z ...) → list?
  x : list?
  y : list?
  z : list?
```

Creates a single list from several, by concatenation of the items.

```
> (append (cons 1 (cons 2 '())) (cons "a" (cons "b" empty)))
(list 1 2 "a" "b")
```

```
(assoc x l) → (union (listof any) #false)
  x : any/c
  l : (listof any)
```

Produces the first pair on `l` whose `first` is `equal?` to `x`; otherwise it produces `#false`.

```
> (assoc "hello" '(("world" 2) ("hello" 3) ("good" 0)))
(list "hello" 3)
```

```
(assq x l) → (union #false cons?)
  x : any/c
  l : list?
```

Determines whether some item is the first item of a pair in a list of pairs. (It compares the items with `eq?`.)

```
> a
(list (list 'a 22) (list 'b 8) (list 'c 70))
> (assq 'b a)
(list 'b 8)
```

```
(caaar x) → any/c
  x : list?
```

LISP-style selector: `(car (car (car x)))`.

```
> w
(list (list (list (list "bye") 3) #true) 42)
> (caaar w)
(list "bye")
```

```
(caadr x) → any/c
  x : list?
```

LISP-style selector: `(car (car (cdr x)))`.

```
> (caadr (cons 1 (cons (cons 'a '()) (cons (cons 'd '()) '()))))
'a
```

```
(caar x) → any/c
  x : list?
```

LISP-style selector: `(car (car x))`.

```
> y
(list (list (list 1 2 3) #false "world"))
> (caar y)
(list 1 2 3)
```

```
(cadar x) → any/c
  x : list?
```

LISP-style selector: `(car (cdr (car x)))`.

```
> w
(list (list (list (list "bye") 3) #true) 42)
> (cadar w)
#true
```

```
(cadddr x) → any/c
  x : list?
```

LISP-style selector: `(car (cdr (cdr (cdr x))))`.

```
> v
(list 1 2 3 4 5 6 7 8 9 'A)
> (cadddr v)
4
```

```
(caddr x) → any/c
  x : list?
```

LISP-style selector: `(car (cdr (cdr x)))`.

```
> x
(list 2 "hello" #true)
> (caddr x)
#true
```

```
(cadr x) → any/c
  x : list?
```

LISP-style selector: `(car (cdr x))`.

```
> x
(list 2 "hello" #true)
> (cadr x)
"hello"
```

```
(car x) → any/c
  x : cons?
```

Selects the first item of a non-empty list.

```
> x
(list 2 "hello" #true)
> (car x)
2
```

```
(cdaar x) → any/c
  x : list?
```

LISP-style selector: `(cdr (car (car x)))`.

```
> w
(list (list (list (list "bye") 3) #true) 42)
> (cdaar w)
(list 3)
```

```
(cdadr x) → any/c
  x : list?
```

LISP-style selector: `(cdr (car (cdr x)))`.

```
> (cdadr (list 1 (list 2 "a") 3))
(list "a")
```

```
(cdar x) → list?
  x : list?
```

LISP-style selector: `(cdr (car x))`.

```
> y
(list (list (list 1 2 3) #false "world"))
> (cdar y)
(list #false "world")
```

```
(cddar x) → any/c
  x : list?
```

LISP-style selector: `(cdr (cdr (car x)))`

```
> w
(list (list (list (list "bye") 3) #true) 42)
> (cddar w)
'()
```

```
(cdddr x) → any/c
  x : list?
```

LISP-style selector: `(cdr (cdr (cdr x)))`.

```
> v
(list 1 2 3 4 5 6 7 8 9 'A)
> (cdddr v)
(list 4 5 6 7 8 9 'A)
```

```
(cddr x) → list?
  x : list?
```

LISP-style selector: `(cdr (cdr x))`.

```
> x
(list 2 "hello" #true)
> (cddr x)
(list #true)
```

```
(cdr x) → any/c
  x : cons?
```

Selects the rest of a non-empty list.

```
> x
(list 2 "hello" #true)
> (cdr x)
(list "hello" #true)
```

```
(cons x y) → list?
  x : any/c
  y : list?
```

Constructs a list.

```
> (cons 1 '())
(cons 1 '())
```

```
(cons? x) → boolean?
  x : any/c
```

Determines whether some value is a constructed list.

```
> (cons? (cons 1 '()))
#true
> (cons? 42)
#false
```

```
(eighth x) → any/c
  x : list?
```

Selects the eighth item of a non-empty list.

```
> v
(list 1 2 3 4 5 6 7 8 9 'A)
> (eighth v)
8
```

```
(empty? x) → boolean?
  x : any/c
```

Determines whether some value is the empty list.

```
> (empty? '())
#true
> (empty? 42)
#false
```

```
(fifth x) → any/c
  x : list?
```

Selects the fifth item of a non-empty list.

```
> v
(list 1 2 3 4 5 6 7 8 9 'A)
> (fifth v)
5
```

```
(first x) → any/c
  x : cons?
```

Selects the first item of a non-empty list.

```
> x
(list 2 "hello" #true)
> (first x)
2
```

```
(fourth x) → any/c
  x : list?
```

Selects the fourth item of a non-empty list.

```
> v
(list 1 2 3 4 5 6 7 8 9 'A)
> (fourth v)
4
```

```
(length l) → natural?
  l : list?
```

Evaluates the number of items on a list.

```
> x
(list 2 "hello" #true)
> (length x)
3
```

```
(list x ...) → list?
  x : any/c
```

Constructs a list of its arguments.

```
> (list 1 2 3 4 5 6 7 8 9 0)
(cons 1 (cons 2 (cons 3 (cons 4 (cons 5 (cons 6 (cons 7 (cons 8
(cons 9 (cons 0 '()))))))))))
```

```
(list* x ... l) → list?
  x : any/c
  l : list?
```

Constructs a list by adding multiple items to a list.

```
> x
(list 2 "hello" #true)
> (list* 4 3 x)
(list 4 3 2 "hello" #true)
```

```
(list-ref x i) → any/c
  x : list?
  i : natural?
```

Extracts the indexed item from the list.

```
> v
(list 1 2 3 4 5 6 7 8 9 'A)
> (list-ref v 9)
'A
```

```
(list? x) → boolean?
  x : any/c
```

Checks whether the given value is a list.

```
> (list? 42)
#false
> (list? '())
#true
> (list? (cons 1 (cons 2 '())))
#true
```

```
(make-list i x) → list?
  i : natural?
  x : any/c
```

Constructs a list of *i* copies of *x*.

```
> (make-list 3 "hello")
(cons "hello" (cons "hello" (cons "hello" '())))
```

```
(member x l) → boolean?
  x : any/c
  l : list?
```

Determines whether some value is on the list (comparing values with equal?).

```
> x
(list 2 "hello" #true)
> (member "hello" x)
#true
```

```
(member? x l) → boolean?
  x : any/c
  l : list?
```

Determines whether some value is on the list (comparing values with equal?).

```
> x
(list 2 "hello" #true)
> (member? "hello" x)
#true
```

```
(memq x l) → boolean?
  x : any/c
  l : list?
```

Determines whether some value *x* is on some list *l*, using `eq?` to compare *x* with items on *l*.

```
> x
(list 2 "hello" #true)
> (memq (list (list 1 2 3)) x)
#false
```

```
(memq? x l) → boolean?
  x : any/c
  l : list?
```

Determines whether some value *x* is on some list *l*, using `eq?` to compare *x* with items on *l*.

```
> x
(list 2 "hello" #true)
> (memq? (list (list 1 2 3)) x)
#false
```

```
(memv x l) → (or/c #false list)
  x : any/c
  l : list?
```

Determines whether some value is on the list if so, it produces the suffix of the list that starts with x if not, it produces false. (It compares values with the eqv? predicate.)

```
> x
(list 2 "hello" #true)
> (memv (list (list 1 2 3)) x)
#false
```

```
null : list
```

Another name for the empty list

```
> null
'()
```

```
(null? x) → boolean?
  x : any/c
```

Determines whether some value is the empty list.

```
> (null? '())
#true
> (null? 42)
#false
```

```
(range start end step) → list?
  start : number
  end : number
  step : number
```

Constructs a list of numbers by *step*ping from *start* to *end*.

```
> (range 0 10 2)
(cons 0 (cons 2 (cons 4 (cons 6 (cons 8 '()))))
```

```
(remove x l) → list?
  x : any/c
  l : list?
```

Constructs a list like the given one, with the first occurrence of the given item removed (comparing values with equal?).

```
> x
(list 2 "hello" #true)
> (remove "hello" x)
(list 2 #true)
> hello-2
(list 2 "hello" #true "hello")
> (remove "hello" hello-2)
(list 2 #true "hello")
```

```
(remove-all x l) → list?
  x : any/c
  l : list?
```

Constructs a list like the given one, with all occurrences of the given item removed (comparing values with equal?).

```
> x
(list 2 "hello" #true)
> (remove-all "hello" x)
(list 2 #true)
> hello-2
(list 2 "hello" #true "hello")
> (remove-all "hello" hello-2)
(list 2 #true)
```

```
(rest x) → any/c
  x : cons?
```

Selects the rest of a non-empty list.

```
> x
(list 2 "hello" #true)
> (rest x)
(list "hello" #true)
```

```
(reverse l) → list
  l : list?
```

Creates a reversed version of a list.

```
> x
(list 2 "hello" #true)
> (reverse x)
(list #true "hello" 2)
```

```
(second x) → any/c
  x : list?
```

Selects the second item of a non-empty list.

```
> x
(list 2 "hello" #true)
> (second x)
"hello"
```

```
(seventh x) → any/c
  x : list?
```

Selects the seventh item of a non-empty list.

```
> v
(list 1 2 3 4 5 6 7 8 9 'A)
> (seventh v)
7
```

```
(sixth x) → any/c
  x : list?
```

Selects the sixth item of a non-empty list.

```
> v
(list 1 2 3 4 5 6 7 8 9 'A)
> (sixth v)
6
```

(third x) → any/c
  x : list?

Selects the third item of a non-empty list.

```
> x
(list 2 "hello" #true)
> (third x)
#true
```

2.11 Posns

(make-posn x y) → posn
  x : any/c
  y : any/c

Constructs a posn from two arbitrary values.

```
> (make-posn 3 3)
(make-posn 3 3)
> (make-posn "hello" #true)
(make-posn "hello" #true)
```

(posn-x p) → any/c
  p : posn

Extracts the x component of a posn.

```
> p
(make-posn 2 -3)
> (posn-x p)
2
```

(posn-y p) → any/c
  p : posn

Extracts the y component of a posn.

```
> p
(make-posn 2 -3)
> (posn-y p)
-3
```

```
(posn? x) → boolean?
  x : any/c
```

Determines if its input is a posn.

```
> q
(make-posn "bye" 2)
> (posn? q)
#true
> (posn? 42)
#false
```

2.12 Characters

```
(char->integer c) → integer
  c : char
```

Looks up the number that corresponds to the given character in the ASCII table (if any).

```
> (char->integer #\a)
97
> (char->integer #\z)
122
```

```
(char-alphabetic? c) → boolean?
  c : char
```

Determines whether a character represents an alphabetic character.

```
> (char-alphabetic? #\Q)
#true
```

```
(char-ci<=? c d e ...) → boolean?
  c : char
  d : char
  e : char
```

Determines whether the characters are ordered in an increasing and case-insensitive manner.

```
> (char-ci<=? #\b #\B)
#true
> (char<=? #\b #\B)
#false
```

```
(char-ci<? c d e ...) → boolean?
  c : char
  d : char
  e : char
```

Determines whether the characters are ordered in a strictly increasing and case-insensitive manner.

```
> (char-ci<? #\B #\c)
#true
> (char<? #\b #\B)
#false
```

```
(char-ci=? c d e ...) → boolean?
  c : char
  d : char
  e : char
```

Determines whether two characters are equal in a case-insensitive manner.

```
> (char-ci=? #\b #\B)
#true
```

```
(char-ci>=? c d e ...) → boolean?
  c : char
  d : char
  e : char
```

Determines whether the characters are sorted in a decreasing and case-insensitive manner.

```
> (char-ci>=? #\b #\C)
#false
> (char>=? #\b #\C)
#true
```

```
(char-ci>? c d e ...) → boolean?
  c : char
  d : char
  e : char
```

Determines whether the characters are sorted in a strictly decreasing and case-insensitive manner.

```
> (char-ci>? #\b #\B)
#false
> (char>? #\b #\B)
#true
```

```
(char-downcase c) → char
  c : char
```

Produces the equivalent lower-case character.

```
> (char-downcase #\T)
#\t
```

```
(char-lower-case? c) → boolean?
  c : char
```

Determines whether a character is a lower-case character.

```
> (char-lower-case? #\T)
#false
```

```
(char-numeric? c) → boolean?
  c : char
```

Determines whether a character represents a digit.

```
> (char-numeric? #\9)
#true
```

```
(char-upcase c) → char
  c : char
```

Produces the equivalent upper-case character.

```
> (char-upcase #\t)
#\T
```

```
(char-upper-case? c) → boolean?
  c : char
```

Determines whether a character is an upper-case character.

```
> (char-upper-case? #\T)
#true
```

```
(char-whitespace? c) → boolean?
  c : char
```

Determines whether a character represents space.

```
> (char-whitespace? #\tab)
#true
```

```
(char<=? c d e ...) → boolean?
  c : char
  d : char
  e : char
```

Determines whether the characters are ordered in an increasing manner.

```
> (char<=? #\a #\a #\b)
#true
```

```
(char<? x d e ...) → boolean?
  x : char
  d : char
  e : char
```

Determines whether the characters are ordered in a strictly increasing manner.

```
> (char<? #\a #\b #\c)
#true
```

```
(char=? c d e ...) → boolean?
  c : char
  d : char
  e : char
```

Determines whether the characters are equal.

```
> (char=? #\b #\a)
#false
```

```
(char>=? c d e ...) → boolean?
  c : char
  d : char
  e : char
```

Determines whether the characters are sorted in a decreasing manner.

```
> (char>=? #\b #\b #\a)
#true
```

```
(char>? c d e ...) → boolean?
  c : char
  d : char
  e : char
```

Determines whether the characters are sorted in a strictly decreasing manner.

```
> (char>? #\A #\z #\a)
#false
```

```
(char? x) → boolean?
  x : any/c
```

Determines whether a value is a character.

```
> (char? "a")
#false
> (char? #\a)
#true
```

2.13 Strings

```
(explode s) → (listof string)
  s : string
```

Translates a string into a list of 1-letter strings.

```
> (explode "cat")
(list "c" "a" "t")
```

```
(format f x ...) → string
  f : string
  x : any/c
```

Formats a string, possibly embedding values.

```
> (format "Dear Dr. ~a:" "Flatt")
"Dear Dr. Flatt:"
> (format "Dear Dr. ~s:" "Flatt")
"Dear Dr. \"Flatt\":"
```

```
(implode l) → string
  l : list?
```

Concatenates the list of 1-letter strings into one string.

```
> (implode (cons "c" (cons "a" (cons "t" '()))))
"cat"
```

```
(int->string i) → string
  i : integer
```

Converts an integer in [0,55295] or [57344 1114111] to a 1-letter string.

```
> (int->string 65)
"A"
```

```
(list->string l) → string
  l : list?
```

Converts a s list of characters into a string.

```
> (list->string (cons #\c (cons #\a (cons #\t '()))))
"cat"
```

(make-string *i* *c*) → string
  i : natural?
  c : char

Produces a string of length *i* from *c*.

```
> (make-string 3 #\d)
"ddd"
```

(replicate *i* *s*) → string
  i : natural?
  s : string

Replicates *s* *i* times.

```
> (replicate 3 "h")
"hhh"
```

(string *c* ...) → string?
  c : char

Builds a string of the given characters.

```
> (string #\d #\o #\g)
"dog"
```

(string->int *s*) → integer
  s : string

Converts a 1-letter string to an integer in [0,55295] or [57344, 1114111].

```
> (string->int "a")
97
```

```
(string->list s) → (listof char)
  s : string
```

Converts a string into a list of characters.

```
> (string->list "hello")
(list #\h #\e #\l #\l #\o)
```

```
(string->number s) → (union number #false)
  s : string
```

Converts a string into a number, produce false if impossible.

```
> (string->number "-2.03")
-2.03
> (string->number "1-2i")
1-2i
```

```
(string->symbol s) → symbol
  s : string
```

Converts a string into a symbol.

```
> (string->symbol "hello")
'hello
```

```
(string-alphabetic? s) → boolean?
  s : string
```

Determines whether all 'letters' in the string are alphabetic.

```
> (string-alphabetic? "123")
#false
> (string-alphabetic? "cat")
#true
```

```
(string-append s t z ...) → string
  s : string
  t : string
  z : string
```

Concatenates the characters of several strings.

```
> (string-append "hello" " " "world" " " "good bye")
"hello world good bye"
```

```
(string-ci<=? s t) → boolean?
  s : string
  t : string
```

Determines whether the strings are ordered in a lexicographically increasing and case-insensitive manner.

```
> (string-ci<=? "hello" "WORLD")
#true
```

```
(string-ci<? s t) → boolean?
  s : string
  t : string
```

Determines whether the strings are ordered in a lexicographically strictly increasing and case-insensitive manner.

```
> (string-ci<? "hello" "WORLD")
#true
```

```
(string-ci=? s t) → boolean?
  s : string
  t : string
```

Determines whether all strings are equal, character for character, regardless of case.

```
> (string-ci=?  "hello" "HellO")
#true
```

```
(string-ci>=? s t) → boolean?
  s : string
  t : string
```

Determines whether the strings are ordered in a lexicographically decreasing and case-insensitive manner.

```
> (string-ci>? "WORLD" "hello")
#true
```

```
(string-ci>? s t) → boolean?
  s : string
  t : string
```

Determines whether the strings are ordered in a lexicographically strictly decreasing and case-insensitive manner.

```
> (string-ci>?  "WORLD" "hello")
#true
```

```
(string-contains-ci? s t) → boolean?
  s : string
  t : string
```

Determines whether the first string appears in the second one without regard to the case of the letters.

```
> (string-contains-ci? "At" "caT")
#true
```

```
(string-contains? s t) → boolean?
  s : string
  t : string
```

Determines whether the first string appears literally in the second one.

```
> (string-contains? "at" "cat")
#true
```

```
(string-copy s) → string
  s : string
```

Copies a string.

```
> (string-copy "hello")
"hello"
```

```
(string-downcase s) → string
  s : string
```

Produces a string like the given one with all 'letters' as lower case.

```
> (string-downcase "CAT")
"cat"
> (string-downcase "cAt")
"cat"
```

```
(string-ith s i) → 1string?
  s : string
  i : natural?
```

Extracts the *i*th 1-letter substring from *s*.

```
> (string-ith "hello world" 1)
"e"
```

```
(string-length s) → nat
  s : string
```

Determines the length of a string.

```
> (string-length "hello world")
11
```

```
(string-lower-case? s) → boolean?
  s : string
```

Determines whether all 'letters' in the string are lower case.

```
> (string-lower-case? "CAT")
#false
```

```
(string-numeric? s) → boolean?
  s : string
```

Determines whether all 'letters' in the string are numeric.

```
> (string-numeric? "123")
#true
> (string-numeric? "1-2i")
#false
```

```
(string-ref s i) → char
  s : string
  i : natural?
```

Extracts the *i*th character from *s*.

```
> (string-ref "cat" 2)
#\t
```

```
(string-upcase s) → string
  s : string
```

Produces a string like the given one with all 'letters' as upper case.

```
> (string-upcase "cat")
"CAT"
> (string-upcase "cAt")
"CAT"
```

```
(string-upper-case? s) → boolean?
  s : string
```

Determines whether all 'letters' in the string are upper case.

```
> (string-upper-case? "CAT")
#true
```

```
(string-whitespace? s) → boolean?
  s : string
```

Determines whether all 'letters' in the string are white space.

```
> (string-whitespace? (string-append " " (string #\tab #\newline #\return)))
#true
```

```
(string<=? s t) → boolean?
  s : string
  t : string
```

Determines whether the strings are ordered in a lexicographically increasing manner.

```
> (string<=? "hello" "hello")
#true
```

```
(string<? s t) → boolean?
  s : string
  t : string
```

Determines whether the strings are ordered in a lexicographically strictly increasing manner.

```
> (string<? "hello" "world")
#true
```

```
(string=? s t) → boolean?
  s : string
  t : string
```

Determines whether all strings are equal, character for character.

```
> (string=? "hello" "world")
#false
> (string=? "bye" "bye")
#true
```

```
(string>=? s t) → boolean?
  s : string
  t : string
```

Determines whether the strings are ordered in a lexicographically decreasing manner.

```
> (string>=? "world" "hello")
#true
```

```
(string>? s t) → boolean?
  s : string
  t : string
```

Determines whether the strings are ordered in a lexicographically strictly decreasing manner.

```
> (string>? "world" "hello")
#true
```

```
(string? x) → boolean?
  x : any/c
```

Determines whether a value is a string.

```
> (string? "hello world")
#true
> (string? 42)
#false
```

```
(substring s i j) → string
  s : string
  i : natural?
  j : natural?
```

Extracts the substring starting at *i* up to *j* (or the end if *j* is not provided).

```
> (substring "hello world" 1 5)
"ello"
> (substring "hello world" 1 8)
"ello wo"
> (substring "hello world" 4)
"o world"
```

2.14 Images

```
(image=? i j) → boolean?
  i : image
  j : image
```

Determines whether two images are equal.

```
> c1
```

```
> (image=? (circle 5 "solid" "green") c1)
#false
> (image=? (circle 10 "solid" "green") c1)
#true
```

```
(image? x) → boolean?
  x : any/c
```

Determines whether a value is an image.

```
> c1
```

```
> (image? c1)
#true
```

2.15 Misc

```
(=~ x y eps) → boolean?
  x : number
  y : number
  eps : non-negative-real
```

Checks whether *x* and *y* are within *eps* of either other.

```
> (=~ 1.01 1.0 0.1)
#true
> (=~ 1.01 1.5 0.1)
#false
```

```
eof : eof-object?
```

A value that represents the end of a file:

```
> eof
#<eof>
```

```
(eof-object? x) → boolean?
  x : any/c
```

Determines whether some value is the end-of-file value.

```
> (eof-object? eof)
#true
> (eof-object? 42)
#false
```

```
(eq? x y) → boolean?
  x : any/c
  y : any/c
```

Determines whether two values are equivalent from the computer's perspective (intensional).

```
> (eq? (cons 1 '()) (cons 1 '()))
#false
> one
(list 1)
> (eq? one one)
#true
```

```
(equal? x y) → boolean?
  x : any/c
  y : any/c
```

Determines whether two values are structurally equal where basic values are compared with the eqv? predicate.

```
> (equal? (make-posn 1 2) (make-posn (- 2 1) (+ 1 1)))
#true
```

```
(equal~? x y z) → boolean?
  x : any/c
  y : any/c
  z : non-negative-real
```

Compares *x* and *y* like `equal?` but uses =~ in the case of numbers.

```
> (equal~? (make-posn 1.01 1.0) (make-posn 1.01 0.99) 0.2)
#true
```

```
(eqv? x y) → boolean?
  x : any/c
  y : any/c
```

Determines whether two values are equivalent from the perspective of all functions that can be applied to it (extensional).

```
> (eqv? (cons 1 '()) (cons 1 '()))
#false
> one
(list 1)
> (eqv? one one)
#true
```

(error x ...) → void?
  x : any/c

Signals an error, combining the given values into an error message. If any of the values' printed representations is too long, it is truncated and "..." is put into the string. If the first value is a symbol, it is suffixed with a colon and the result pre-pended on to the error message.

```
> zero
0
> (if (= zero 0) (error "can't divide by 0") (/ 1 zero))
can't divide by 0
```

(exit) → void

Evaluating (exit) terminates the running program.

(identity x) → any/c
  x : any/c

Returns x.

```
> (identity 42)
42
> (identity c1)
```

```
> (identity "hello")
"hello"
```

(struct? x) → boolean?
  x : any/c

Determines whether some value is a structure.

```
> (struct? (make-posn 1 2))
#true
> (struct? 43)
#false
```

2.16 Signatures

```
Any : signature?
```

Signature for any value.

```
Boolean : signature?
```

Signature for booleans.

```
Char : signature?
```

Signature for chararacters.

```
(ConsOf first-sig rest-sig) → signature?
  first-sig : signature?
  rest-sig : signature?
```

Signature for a cons pair.

```
EmptyList : signature?
```

Signature for the empty list.

```
False : signature?
```

Signature for just false.

```
Integer : signature?
```

Signature for integers.

```
Natural : signature?
```

Signature for natural numbers.

```
Number : signature?
```

Signature for arbitrary numbers.

```
Rational : signature?
```

Signature for rational numbers.

```
Real : signature?
```

Signature for real numbers.

```
String : signature?
```

Signature for strings.

```
Symbol : signature?
```

Signature for symbols.

```
True : signature?
```

Signature for just true.

3 Intermediate Student

The grammar notation uses the notation `X` **...** (bold dots) to indicate that `X` may occur an arbitrary number of times (zero, one, or more). Separately, the grammar also defines `...` as an identifier to be used in templates.

```
                program = def-or-expr ...

          def-or-expr = definition
                      | expr
                      | test-case
                      | library-require
                      | signature-declaration

           definition = (define (name variable variable ...) expr)
                      | (define name expr)
                      | (define name (lambda (variable variable ...) expr))
                      | (define-struct name (name ...))

                expr = (local [definition ...] expr)
                     | (letrec ([name expr-for-let] ...) expr)
                     | (let ([name expr-for-let] ...) expr)
                     | (let* ([name expr-for-let] ...) expr)
                     | (name expr expr ...)
                     | (cond [expr expr] ... [expr expr])
                     | (cond [expr expr] ... [else expr])
                     | (if expr expr expr)
                     | (and expr expr expr ...)
                     | (or expr expr expr ...)
                     | (time expr)
                     | name
                     | 'quoted
                     | `quasiquoted
                     | '()
                     | number
                     | boolean
                     | string
                     | character
                     | (signature signature-form)

          expr-for-let = (lambda (variable variable ...) expr)
                       | expr

signature-declaration = (: name signature-form)
```

```
signature-form = (enum expr ...)
               | (mixed signature-form ...)
               | (signature-form ... -> signature-form)
               | (ListOf signature-form)
               | signature-variable
               | expr

signature-variable = %name

quoted = name
       | number
       | string
       | character
       | (quoted ...)
       | 'quoted
       | `quoted
       | ,quoted
       | ,@quoted

quasiquoted = name
            | number
            | string
            | character
            | (quasiquoted ...)
            | 'quasiquoted
            | `quasiquoted
            | ,expr
            | ,@expr

test-case = (check-expect expr expr)
          | (check-random expr expr)
          | (check-within expr expr expr)
          | (check-random-within expr expr expr)
          | (check-member-of expr expr ...)
          | (check-range expr expr expr)
          | (check-satisfied expr expr)
          | (check-error expr expr)
          | (check-error expr)

library-require = (require string)
                | (require (lib string string ...))
                | (require (planet string package))

package = (string string number number)
```

A *name* or a *variable* is a sequence of characters not including a space or one of the

following:

```
" , ' ` ( ) [ ] { } | ; #
```

A *number* is a number such as `123`, `3/2`, or `5.5`.

A *boolean* is one of: `#true` or `#false`.

Alternative spellings for the `#true` constant are `#t`, `true`, and `#T`. Similarly, `#f`, `false`, or `#F` are also recognized as `#false`.

A *symbol* is a quote character followed by a name. A symbol is a value, just like `42`, `'()`, or `#false`.

A *string* is a sequence of characters enclosed by a pair of `"`. Unlike symbols, strings may be split into characters and manipulated by a variety of functions. For example, `"abcdef"`, `"This is a string"`, and `"This is a string with \" inside"` are all strings.

A *character* begins with `#\` and has the name of the character. For example, `#\a`, `#\b`, and `#\space` are characters.

In function calls, the function appearing immediately after the open parenthesis can be any functions defined with `define` or `define-struct`, or any one of the pre-defined functions.

3.1 Pre-defined Variables

`empty` : `empty?`

The empty list.

`true` : `boolean?`

The `#true` value.

`false` : `boolean?`

The `#false` value.

3.2 Template Variables

. .

A placeholder for indicating that a function definition is a template.

```
...
```

A placeholder for indicating that a function definition is a template.

```
....
```

A placeholder for indicating that a function definition is a template.

```
.....
```

A placeholder for indicating that a function definition is a template.

```
......
```

A placeholder for indicating that a function definition is a template.

3.3 Syntax for Intermediate

```
(local [definition ...] expression)
```

Groups related definitions for use in *expression*. Each *definition* can be either a `define` or a `define-struct`.

When evaluating `local`, each *definition* is evaluated in order, and finally the body *expression* is evaluated. Only the expressions within the `local` (including the right-hand-sides of the *definition*s and the *expression*) may refer to the names defined by the *definition*s. If a name defined in the `local` is the same as a top-level binding, the inner one "shadows" the outer one. That is, inside the `local`, any references to that name refer to the inner one.

```
(letrec ([name expr-for-let] ...) expression)
```

Like `local`, but with a simpler syntax. Each *name* defines a variable (or a function) with the value of the corresponding *expr-for-let*. If *expr-for-let* is a `lambda`, `letrec` defines a function, otherwise it defines a variable.

```
(let* ([name expr-for-let] ...) expression)
```

Like `letrec`, but each *name* can only be used in *expression*, and in *expr-for-let*s occuring after that *name*.

```
(let ([name expr-for-let] ...) expression)
```

Like `letrec`, but the defined *name*s can be used only in the last *expression*, not the *expr-for-let*s next to the *name*s.

```
(time expression)
```

Measures the time taken to evaluate *expression*. After evaluating *expression*, `time` prints out the time taken by the evaluation (including real time, time taken by the CPU, and the time spent collecting free memory). The value of `time` is the same as that of *expression*.

3.4 Common Syntaxes

The following syntaxes behave the same in the *Intermediate* level as they did in the §2 "Beginning Student with List Abbreviations" level.

```
'name
'part
(quote name)
(quote part)
```

A quoted name is a symbol. A quoted part is an abbreviation for a nested lists.

Normally, this quotation is written with a `'`, like `'(apple banana)`, but it can also be written with `quote`, like `(quote (apple banana))`.

```
`name
`part
(quasiquote name)
(quasiquote part)
```

Like `quote`, but also allows escaping to expression "unquotes."

Normally, quasi-quotations are written with a backquote, `` ` ``, like `` `(apple ,(+ 1 2)) ``, but they can also be written with `quasiquote`, like `(quasiquote (apple ,(+ 1 2)))`.

```
,expression
(unquote expression)
```

Under a single quasiquote, `,expression` escapes from the quote to include an evaluated expression whose result is inserted into the abbreviated list.

Under multiple quasiquotes, `,expression` is really the literal `,expression`, decrementing the quasiquote count by one for `expression`.

Normally, an unquote is written with `,`, but it can also be written with `unquote`.

```
,@expression
(unquote-splicing expression)
```

Under a single quasiquote, `,@expression` escapes from the quote to include an evaluated expression whose result is a list to splice into the abbreviated list.

Under multiple quasiquotes, a splicing unquote is like an unquote; that is, it decrements the quasiquote count by one.

Normally, a splicing unquote is written with `,`, but it can also be written with `unquote-splicing`.

```
(define (name variable variable ...) expression)
```

Defines a function named `name`. The `expression` is the body of the function. When the function is called, the values of the arguments are inserted into the body in place of the `variable`s. The function returns the value of that new expression.

The function name's cannot be the same as that of another function or variable.

```
(define name expression)
```

Defines a variable called `name` with the the value of `expression`. The variable name's cannot be the same as that of another function or variable, and `name` itself must not appear in `expression`.

```
(define name (lambda (variable variable ...) expression))
```

An alternate way to defining functions. The `name` is the name of the function, which cannot be the same as that of another function or variable.

A `lambda` cannot be used outside of this alternate syntax.

```
(define-struct structure-name (field-name ...))
```

Defines a new structure called *structure-name*. The structure's fields are named by the *field-name*s. After the `define-struct`, the following new functions are available:

- `make-`*structure-name* : takes a number of arguments equal to the number of fields in the structure, and creates a new instance of that structure.
- *structure-name-field-name* : takes an instance of the structure and returns the value in the field named by *field-name*.
- *structure-name*`?` : takes any value, and returns `#true` if the value is an instance of the structure.

The name of the new functions introduced by `define-struct` must not be the same as that of other functions or variables, otherwise `define-struct` reports an error.

```
(name expression expression ...)
```

Calls the function named *name*. The value of the call is the value of *name*'s body when every one of the function's variables are replaced by the values of the corresponding *expression*s.

The function named *name* must defined before it can be called. The number of argument *expression*s must be the same as the number of arguments expected by the function.

```
(cond [question-expression answer-expression] ...)
(cond [question-expression answer-expression]
      ...
      [else answer-expression])
```

Chooses a clause based on some condition. `cond` finds the first *question-expression* that evaluates to `#true`, then evaluates the corresponding *answer-expression*.

If none of the *question-expression*s evaluates to `#true`, `cond`'s value is the *answer-expression* of the `else` clause. If there is no `else`, `cond` reports an error. If the result of a *question-expression* is neither `#true` nor `#false`, `cond` also reports an error.

`else` cannot be used outside of `cond`.

```
(if question-expression
    then-answer-expression
    else-answer-expression)
```

When the value of the *question-expression* is `#true`, `if` evaluates the *then-answer-expression*. When the test is `#false`, `if` evaluates the *else-answer-expression*.

If the *question-expression* is neither `#true` nor `#false`, `if` reports an error.

```
(and expression expression expression ...)
```

Evaluates to `#true` if all the *expression*s are `#true`. If any *expression* is `#false`, the `and` expression evaluates to `#false` (and the expressions to the right of that expression are not evaluated.)

If any of the expressions evaluate to a value other than `#true` or `#false`, `and` reports an error.

```
(or expression expression expression ...)
```

Evaluates to `#true` as soon as one of the *expression*s is `#true` (and the expressions to the right of that expression are not evaluated.) If all of the *expression*s are `#false`, the `or` expression evaluates to `#false`.

If any of the expressions evaluate to a value other than `#true` or `#false`, `or` reports an error.

```
(check-expect expression expected-expression)
```

Checks that the first *expression* evaluates to the same value as the *expected-expression*.

```
(check-expect (fahrenheit->celsius 212) 100)
(check-expect (fahrenheit->celsius -40) -40)

(define (fahrenheit->celsius f)
  (* 5/9 (- f 32)))
```

A `check-expect` expression must be placed at the top-level of a student program. Also it may show up anywhere in the program, including ahead of the tested function definition. By placing `check-expect`s there, a programmer conveys to a future reader the intention behind the program with working examples, thus making it often superfluous to read the function definition proper. Syntax errors in `check-expect` (and all check forms) are intentionally delayed to run time so that students can write tests *without* necessarily writing complete function headers.

It is an error for `expr` or `expected-expr` to produce an inexact number or a function value. As for inexact numbers, it is *morally* wrong to compare them for plain equality. Instead one

tests whether they are both within a small interval; see `check-within`. As for functions (see Intermediate and up), it is provably impossible to compare functions.

```
(check-random expression expected-expression)
```

Checks that the first *expression* evaluates to the same value as the *expected-expression*.

The form supplies the same random-number generator to both parts. If both parts request `random` numbers from the same interval in the same order, they receive the same random numbers.

Here is a simple example of where `check-random` is useful:

```
(define WIDTH 100)
(define HEIGHT (* 2 WIDTH))

(define-struct player (name x y))
; A Player is (make-player String Nat Nat)

; String -> Player

(check-random (create-randomly-placed-player "David Van Horn")
              (make-player "David Van Horn" (random WIDTH) (random HEIGHT)))

(define (create-randomly-placed-player name)
  (make-player name (random WIDTH) (random HEIGHT)))
```

Note how `random` is called on the same numbers in the same order in both parts of `check-random`. If the two parts call `random` for different intervals, they are likely to fail:

```
; String -> Player

(check-random (create-randomly-placed-player "David Van Horn")
              (make-player "David Van Horn" (random WIDTH) (random HEIGHT)))

(define (create-randomly-placed-player name)
  (a-helper-function name (random HEIGHT)))

; String Number -> Player
(define (a-helper-function name height)
   (make-player name (random WIDTH) height))
```

Because the argument to `a-helper-function` is evaluated first, `random` is first called for the interval [0,*HEIGHT*) and then for [0,*WIDTH*), that is, in a different order than in the preceding `check-random`.

It is an error for `expr` or `expected-expr` to produce a function value or an inexact number; see note on `check-expect` for details.

```
(check-satisfied expression predicate)
```

Checks that the first *expression* satisfies the named *predicate* (function of one argument). Recall that "satisfies" means "the function produces `#true` for the given value."

Here are simple examples for `check-satisfied`:

```
> (check-satisfied 1 odd?)
The test passed!


> (check-satisfied 1 even?)
Ran 1 test.
0 tests passed.
Check failures:

        Actual value ┌─┐ does not satisfy even?.
                     │1│
                     └─┘
at line 3, column 0
```

In general `check-satisfied` empowers program designers to use defined functions to formulate test suites:

```
; [cons Number [List-of Number]] -> Boolean
; a function for testing htdp-sort

(check-expect (sorted? (list 1 2 3)) #true)
(check-expect (sorted? (list 2 1 3)) #false)

(define (sorted? l)
  (cond
    [(empty? (rest l)) #true]
    [else (and (<= (first l) (second l)) (sorted? (rest l)))]))

; [List-of Number] -> [List-of Number]
; create a sorted version of the given list of numbers

(check-satisfied (htdp-sort (list 1 2 0 3)) sorted?)

(define (htdp-sort l)
  (cond
    [(empty? l) l]
```

```
    [else (insert (first l) (htdp-sort (rest l)))]))

; Number [List-of Number] -> [List-of Number]
; insert x into l at proper place
; assume l is arranged in ascending order
; the result is sorted in the same way
(define (insert x l)
  (cond
    [(empty? l) (list x)]
    [else (if (<= x (first l)) (cons x l) (cons (first l) (insert x (rest l))))]))
```

And yes, the results of `htdp-sort` satisfy the `sorted?` predicate:

```
> (check-satisfied (htdp-sort (list 1 2 0 3)) sorted?)
```

`(check-within expression expected-expression delta)`

Checks whether the value of the *`expression`* expression is structurally equal to the value produced by the *`expected-expression`* expression; every number in the first expression must be within *`delta`* of the corresponding number in the second expression.

```
(define-struct roots (x sqrt))
; RT is [List-of (make-roots Number Number)]

(define (root-of a)
  (make-roots a (sqrt a)))

(define (roots-table xs)
  (cond
    [(empty? xs) '()]
    [else (cons (root-of (first xs)) (roots-table (rest xs)))]))
```

Due to the presence of inexact numbers in nested data, `check-within` is the correct choice for testing, and the test succeeds if *`delta`* is reasonably large:

Example:

```
> (check-within (roots-table (list 1.0 2.0 3.0))
                (list
                  (make-roots 1.0 1.0)
                  (make-roots 2  1.414)
                  (make-roots 3  1.713))
                0.1)
The test passed!
```

In contrast, when *delta* is small, the test fails:

Example:

```
> (check-within (roots-table (list 2.0))
                (list
                  (make-roots 2  1.414))
                #i1e-5)
Ran 1 test.
0 tests passed.
Check failures:

        Actual value | '((make-roots 2.0 1.4142135623730951)) | is
not within 1e-5 of expected value | '((make-roots 2 1.414)) |.


at line 5, column 0
```

It is an error for `expressions` or *expected-expression* to produce a function value; see note on `check-expect` for details.

If *delta* is not a number, `check-within` reports an error.

```
(check-random-within expression expected-expression delta)
```

Combines `check-random` and `check-within`. Like `check-random`, *expression* and *expected-expression* are evaluated with the same random-number sequence, so calls to `random` in both expressions draw the same numbers. Like `check-within`, the test then succeeds if every number in the value of *expression* is within *delta* of the corresponding number in the value of *expected-expression*.

Use `check-random-within` in place of `check-random` when the random computation produces inexact numbers, which `check-random` refuses to compare.

If *delta* is not a number, `check-random-within` reports an error.

Added in version 1.9 of package `htdp-lib`.

```
(check-error expression expected-error-message)
(check-error expression)
```

Checks that the *expression* reports an error, where the error messages matches the value of *expected-error-message*, if it is present.

Here is a typical beginner example that calls for a use of `check-error`:

```
(define sample-table
  '(("matthias" 10)
    ("matthew"  20)
    ("robby"    -1)
    ("shriram"  18)))

; [List-of [list String Number]] String -> Number
; determine the number associated with s in table

(define (lookup table s)
  (cond
    [(empty? table) (error (string-append s " not found"))]
    [else (if (string=? (first (first table)) s)
              (second (first table))
              (lookup (rest table)))]))
```

Consider the following two examples in this context:

Example:

```
> (check-expect (lookup sample-table "matthew") 20)
The test passed!
```

Example:

```
> (check-error (lookup sample-table "kathi") "kathi not found")
The test passed!
```

(check-member-of *expression expression expression ...*)

Checks that the value of the first *expression* is that of one of the following *expression*s.

```
; [List-of X] -> X
; pick a random element from the given list l
(define (pick-one l)
  (list-ref l (random (length l))))
```

Example:

```
> (check-member-of (pick-one '("a" "b" "c")) "a" "b" "c")
The test passed!
```

It is an error for any of expressions to produce a function value; see note on check-expect for details.

```
(check-range expression low-expression high-expression)
```

Checks that the value of the first *expression* is a number in between the value of the *low-expression* and the *high-expression*, inclusive.

A `check-range` form is best used to delimit the possible results of functions that compute inexact numbers:

```
(define EPSILON 0.001)

; [Real -> Real] Real -> Real
; what is the slope of f at x?
(define (differentiate f x)
  (slope f (- x EPSILON) (+ x EPSILON)))

; [Real -> Real] Real Real -> Real
(define (slope f left right)
  (/ (- (f right) (f left))
     2 EPSILON))

(check-range (differentiate sin 0) 0.99 1.0)
```

It is an error for *expression*, *low-expression*, or *high-expression* to produce a function value; see note on `check-expect` for details.

```
(require string)
```

Makes the definitions of the module specified by *string* available in the current module (i.e., the current file), where *string* refers to a file relative to the current file.

The *string* is constrained in several ways to avoid problems with different path conventions on different platforms: a `/` is a directory separator, `.` always means the current directory, `..` always means the parent directory, path elements can use only `a` through `z` (uppercase or lowercase), `0` through `9`, `-`, `_`, and `.`, and the string cannot be empty or contain a leading or trailing `/`.

```
(require module-name)
```

Accesses a file in an installed library. The library name is an identifier with the same constraints as for a relative-path string (though without the quotes), with the additional constraint that it must not contain a `..`.

```
(require (lib string string ...))
```

Accesses a file in an installed library, making its definitions available in the current module (i.e., the current file). The first *string* names the library file, and the remaining *string*s name the collection (and sub-collection, and so on) where the file is installed. Each string is constrained in the same way as for the `(require string)` form.

```
(require (planet string (string string number number)))
(require (planet id))
(require (planet string))
```

Accesses a library that is distributed on the internet via the PLaneT server, making it definitions available in the current module (i.e., current file).

The full grammar for planet requires is given in §3.2 "Importing and Exporting: `require` and `provide`", but the best place to find examples of the syntax is on the the PLaneT server, in the description of a specific package.

3.5 Signatures

Signatures do not have to be comment: They can also be part of the code. When a signature is attached to a function, DrRacket will check that program uses the function in accordance with the signature and display signature violations along with the test results.

A signature is a regular value, and is specified as a *signature form*, a special syntax that only works with `:` signature declarations and inside `signature` expressions.

```
(: name signature-form)
```

This attaches the signature specified by *signature-form* to the definition of *name*. There must be a definition of *name* somewhere in the program.

```
(: age Integer)
(define age 42)

(: area-of-square (Number -> Number))
(define (area-of-square len)
  (sqr len))
```

On running the program, Racket checks whether the signatures attached with `:` actually match the value of the variable. If they don't, Racket reports *signature violation* along with test failures.

For example, this piece of code:

```
(: age Integer)
(define age "fortytwo")
```

Yields this output:

```
1 signature violation.

Signature violations:
        got "fortytwo" at line 2, column 12, signature at line 1,
column 7
```

Note that a signature violation does not stop the running program.

`(signature` *`signature-form`*`)`

This returns the signature described by *`signature-form`* as a value.

3.5.1 Signature Forms

Any expression can be a signature form, in which case the signature is the value returned by that expression. There are a few special signature forms, however:

In a signature form, any name that starts with a `%` is a *signature variable* that stands for any signature depending on how the signature is used.

Example:

```
(: same (%a -> %a))

(define (same x) x)
```

`(`*`input-signature-form ...`* `->` *`output-signature-form`*`)`

This signature form describes a function with inputs described by the *`input-signature-form`*s and output described by *`output-signature-form`*.

`(enum` *`expr ...`*`)`

This signature describes an enumeration of the values returned by the *`expr`*s.

Example:

```
(: cute? ((enum "cat" "snake") -> Boolean))

(define (cute? pet)
  (cond
    [(string=? pet "cat") #t]
    [(string=? pet "snake") #f]))
```

```
(mixed signature-form ...)
```

This signature describes mixed data, i.e. an itemization where each of the cases has a signature described by a *signature-form*.

Example:

```
(define SIGS (signature (mixed Aim Fired)))
```

```
(ListOf signature-form)
```

This signature describes a list where the elements are described by *signature-form*.

```
(predicate expression)
```

This signature describes values through a predicate: *expression* must evaluate to a function of one argument that returns a boolean. The signature matches all values for which the predicate returns `#true`.

3.5.2 Struct Signatures

A `define-struct` form defines two additional names that can be used in signatures. For a struct called `struct`, these are `Struct` and `StructOf`. Note that these names are capitalized. In particular, a struct called `Struct`, will also define `Struct` and `StructOf`. Moreover, when forming the additional names, hyphens are removed, and each letter following a hyphen is capitalized - so a struct called `foo-bar` will define `FooBar` and `FooBarOf`.

`Struct` is a signature that describes struct values from this structure type. `StructOf` is a function that takes as input a signature for each field. It returns a signature describing values of this structure type, additionally describing the values of the fields of the value.

```
(define-struct pair [fst snd])

(: add-pair ((PairOf Number Number) -> Number))
(define (add-pair p)
  (+ (pair-fst p) (pair-snd p)))
```

3.6 Pre-defined Functions

The remaining subsections list those functions that are built into the programming language. All other functions are imported from a teachpack or must be defined in the program.

3.7 Numbers: Integers, Rationals, Reals, Complex, Exacts, Inexacts

```
(- x y ...) → number
  x : number
  y : number
```

Subtracts the second (and following) number(s) from the first ; negates the number if there is only one argument.

```
> (- 5)
-5
> (- 5 3)
2
> (- 5 3 1)
1
```

```
(< x y z ...) → boolean?
  x : real
  y : real
  z : real
```

Compares two or more (real) numbers for less-than.

```
> (< 42 2/5)
#false
```

```
(<= x y z ...) → boolean?
  x : real
  y : real
  z : real
```

Compares two or more (real) numbers for less-than or equality.

```
> (<= 42 2/5)
#false
```

```
(> x y z ...) → boolean?
  x : real
  y : real
  z : real
```

Compares two or more (real) numbers for greater-than.

```
> (> 42 2/5)
#true
```

```
(>= x y z ...) → boolean?
  x : real
  y : real
  z : real
```

Compares two or more (real) numbers for greater-than or equality.

```
> (>= 42 42)
#true
```

```
(abs x) → real
  x : real
```

Determines the absolute value of a real number.

```
> (abs -12)
12
```

```
(acos x) → number
  x : number
```

Computes the arccosine (inverse of cos) of a number.

```
> (acos 0)
#i1.5707963267948966
```

```
(add1 x) → number
  x : number
```

Increments the given number.

```
> (add1 2)
3
```

```
(angle x) → real
  x : number
```

Extracts the angle from a complex number.

```
> (angle (make-polar 3 4))
#i-2.2831853071795867
```

```
(asin x) → number
  x : number
```

Computes the arcsine (inverse of sin) of a number.

```
> (asin 0)
0
```

```
(atan x) → number
  x : number
```

Computes the arctangent of the given number:

```
> (atan 0)
0
> (atan 0.5)
#i0.46364760900080615
```

Also comes in a two-argument version where `(atan y x)` computes `(atan (/ y x))` but the signs of `y` and `x` determine the quadrant of the result and the result tends to be more accurate than that of the 1-argument version in borderline cases:

```
> (atan 3 4)
#i0.6435011087932844
> (atan -2 -1)
#i-2.0344439357957027
```

```
(ceiling x) → integer
  x : real
```

Determines the closest integer (exact or inexact) above a real number. See round.

```
> (ceiling 12.3)
#i13.0
```

```
(complex? x) → boolean?
  x : any/c
```

Determines whether some value is complex.

```
> (complex? 1-2i)
#true
```

```
(conjugate x) → number
  x : number
```

Flips the sign of the imaginary part of a complex number.

```
> (conjugate 3+4i)
3-4i
> (conjugate -2-5i)
-2+5i
> (conjugate (make-polar 3 4))
#i-1.960930862590836+2.270407485923785i
```

```
(cos x) → number
  x : number
```

Computes the cosine of a number (radians).

```
> (cos pi)
#i-1.0
```

```
(cosh x) → number
  x : number
```

Computes the hyperbolic cosine of a number.

```
> (cosh 10)
#i11013.232920103324
```

```
(current-seconds) → integer
```

Determines the current time in seconds elapsed (since a platform-specific starting date).

```
> (current-seconds)
1793426329
```

```
(denominator x) → integer
  x : rational?
```

Computes the denominator of a rational.

```
> (denominator 2/3)
3
```

```
e : real
```

Euler's number.

```
> e
#i2.718281828459045
```

```
(even? x) → boolean?
  x : integer
```

Determines if some integer (exact or inexact) is even or not.

```
> (even? 2)
#true
```

```
(exact->inexact x) → number
  x : number
```

Converts an exact number to an inexact one.

```
> (exact->inexact 12)
#i12.0
```

```
(exact? x) → boolean?
  x : number
```

Determines whether some number is exact.

```
> (exact? (sqrt 2))
#false
```

```
(exp x) → number
  x : number
```

Determines e raised to a number.

```
> (exp -2)
#i0.1353352832366127
```

```
(expt x y) → number
  x : number
  y : number
```

Computes the power of the first to the second number, which is to say, exponentiation.

```
> (expt 16 1/2)
4
> (expt 3 -4)
1/81
```

```
(floor x) → integer
  x : real
```

Determines the closest integer (exact or inexact) below a real number. See `round`.

```
> (floor 12.3)
#i12.0
```

```
(gcd x y ...) → integer
  x : integer
  y : integer
```

Determines the greatest common divisor of two integers (exact or inexact).

```
> (gcd 6 12 8)
2
```

```
(imag-part x) → real
  x : number
```

Extracts the imaginary part from a complex number.

```
> (imag-part 3+4i)
4
```

```
(inexact->exact x) → number
  x : number
```

Approximates an inexact number by an exact one.

```
> (inexact->exact 12.0)
12
```

```
(inexact? x) → boolean?
  x : number
```

Determines whether some number is inexact.

```
> (inexact? 1-2i)
#false
```

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

Looks up the character that corresponds to the given exact integer in the ASCII table (if any).

```
> (integer->char 42)
#\*
```

```
(integer-sqrt x) → complex
  x : integer
```

Computes the integer or imaginary-integer square root of an integer.

```
> (integer-sqrt 11)
3
> (integer-sqrt -11)
0+3i
```

```
(integer? x) → boolean?
  x : any/c
```

Determines whether some value is an integer (exact or inexact).

```
> (integer? (sqrt 2))
#false
```

```
(lcm x y ...) → integer
  x : integer
  y : integer
```

Determines the least common multiple of two integers (exact or inexact).

```
> (lcm 6 12 8)
24
```

```
(log x) → number
  x : number
```

Determines the base-e logarithm of a number.

```
> (log 12)
#i2.4849066497880004
```

```
(magnitude x) → real
  x : number
```

Determines the magnitude of a complex number.

```
> (magnitude (make-polar 3 4))
#i3.0000000000000004
```

```
(make-polar x y) → number
  x : real
  y : real
```

Creates a complex from a magnitude and angle.

```
> (make-polar 3 4)
#i-1.960930862590836-2.270407485923785i
```

```
(make-rectangular x y) → number
  x : real
  y : real
```

Creates a complex from a real and an imaginary part.

```
> (make-rectangular 3 4)
3+4i
```

```
(max x y ...) → real
  x : real
  y : real
```

Determines the largest number—aka, the maximum.

```
> (max 3 2 8 7 2 9 0)
9
```

```
(min x y ...) → real
  x : real
  y : real
```

Determines the smallest number—aka, the minimum.

```
> (min 3 2 8 7 2 9 0)
0
```

```
(modulo x y) → integer
  x : integer
  y : integer
```

Finds the remainder of the division of the first number by the second:

```
> (modulo 9 2)
1
> (modulo 3 -4)
-1
```

```
(negative? x) → boolean?
  x : real
```

Determines if some real number is strictly smaller than zero.

```
> (negative? -2)
#true
```

```
(number->string x) → string
  x : number
```

Converts a number to a string.

```
> (number->string 42)
"42"
```

```
(number->string-digits x p) → string
  x : number
  p : posint
```

Converts a number x to a string with the specified number of digits.

```
> (number->string-digits 0.9 2)
"0.9"
> (number->string-digits pi 4)
"3.1416"
```

```
(number? n) → boolean?
  n : any/c
```

Determines whether some value is a number:

```
> (number? "hello world")
#false
> (number? 42)
#true
```

```
(numerator x) → integer
  x : rational?
```

Computes the numerator of a rational.

```
> (numerator 2/3)
2
```

```
(odd? x) → boolean?
  x : integer
```

Determines if some integer (exact or inexact) is odd or not.

```
> (odd? 2)
#false
```

```
pi : real
```

The ratio of a circle's circumference to its diameter.

```
> pi
#i3.141592653589793
```

```
(positive? x) → boolean?
  x : real
```

Determines if some real number is strictly larger than zero.

```
> (positive? -2)
#false
```

```
(quotient x y) → integer
  x : integer
  y : integer
```

Divides the first integer—also called dividend—by the second—known as divisor—to obtain the quotient.

```
> (quotient 9 2)
4
> (quotient 3 4)
0
```

```
(random x) → natural?
  x : (and/c natural? positive?)
```

Generates a random natural number less than some given exact natural.

```
> (random 42)
37
```

```
(rational? x) → boolean?
  x : any/c
```

Determines whether some value is a rational number.

```
> (rational? 1)
#true
> (rational? -2.349)
#true
> (rational? #i1.23456789)
#true
> (rational? (sqrt -1))
#false
> (rational? pi)
#true
> (rational? e)
#true
> (rational? 1-2i)
#false
```

As the interactions show, the teaching languages considers many more numbers as rationals than expected. In particular, `pi` is a rational number because it is only a finite approximation

to the mathematical π. Think of `rational?` as a suggestion to think of these numbers as fractions.

```
(real-part x) → real
  x : number
```

Extracts the real part from a complex number.

```
> (real-part 3+4i)
3
```

```
(real? x) → boolean?
  x : any/c
```

Determines whether some value is a real number.

```
> (real? 1-2i)
#false
```

```
(remainder x y) → integer
  x : integer
  y : integer
```

Determines the remainder of dividing the first by the second integer (exact or inexact).

```
> (remainder 9 2)
1
> (remainder 3 4)
3
```

```
(round x) → integer
  x : real
```

Rounds a real number to an integer (rounds to even to break ties). See `floor` and `ceiling`.

```
> (round 12.3)
#i12.0
```

```
(sgn x) → (union 1 #i1.0 0 #i0.0 -1 #i-1.0)
  x : real
```

Determines the sign of a real number.

```
> (sgn -12)
-1
```

```
(sin x) → number
  x : number
```

Computes the sine of a number (radians).

```
> (sin pi)
#i1.2246467991473532e-16
```

```
(sinh x) → number
  x : number
```

Computes the hyperbolic sine of a number.

```
> (sinh 10)
#i11013.232874703393
```

```
(sqr x) → number
  x : number
```

Computes the square of a number.

```
> (sqr 8)
64
```

```
(sqrt x) → number
  x : number
```

Computes the square root of a number.

```
> (sqrt 9)
3
> (sqrt 2)
#i1.4142135623730951
```

```
(sub1 x) → number
  x : number
```

Decrements the given number.

```
> (sub1 2)
1
```

```
(tan x) → number
  x : number
```

Computes the tangent of a number (radians).

```
> (tan pi)
#i-1.2246467991473532e-16
```

```
(zero? x) → boolean?
  x : number
```

Determines if some number is zero or not.

```
> (zero? 2)
#false
```

3.8 Booleans

```
(boolean->string x) → string
  x : boolean?
```

Produces a string for the given boolean

```
> (boolean->string #false)
"#false"
> (boolean->string #true)
"#true"
```

```
(boolean=? x y) → boolean?
  x : boolean?
  y : boolean?
```

Determines whether two booleans are equal.

```
> (boolean=? #true #false)
#false
```

```
(boolean? x) → boolean?
  x : any/c
```

Determines whether some value is a boolean.

```
> (boolean? 42)
#false
> (boolean? #false)
#true
```

```
(false? x) → boolean?
  x : any/c
```

Determines whether a value is false.

```
> (false? #false)
#true
```

```
(not x) → boolean?
  x : boolean?
```

Negates a boolean value.

```
> (not #false)
#true
```

3.9 Symbols

```
(symbol->string x) → string
  x : symbol
```

Converts a symbol to a string.

```
> (symbol->string 'c)
"c"
```

```
(symbol=? x y) → boolean?
  x : symbol
  y : symbol
```

Determines whether two symbols are equal.

```
> (symbol=? 'a 'b)
#false
```

```
(symbol? x) → boolean?
  x : any/c
```

Determines whether some value is a symbol.

```
> (symbol? 'a)
#true
```

3.10 Lists

```
(append l ...) → (listof any)
  l : (listof any)
```

Creates a single list from several, by concatenation of the items. In ISL and up: `append` also works when applied to one list or none.

```
> (append (cons 1 (cons 2 '())) (cons "a" (cons "b" '())))
(list 1 2 "a" "b")
> (append)
'()
```

```
(assoc x l) → (union (listof any) #false)
  x : any/c
  l : (listof any)
```

Produces the first pair on `l` whose `first` is `equal?` to `x`; otherwise it produces `#false`.

```
> (assoc "hello" '(("world" 2) ("hello" 3) ("good" 0)))
(list "hello" 3)
```

```
(assq x l) → (union #false cons?)
  x : any/c
  l : list?
```

Determines whether some item is the first item of a pair in a list of pairs. (It compares the items with `eq?`.)

```
> a
(list (list 'a 22) (list 'b 8) (list 'c 70))
> (assq 'b a)
(list 'b 8)
```

```
(caaar x) → any/c
  x : list?
```

LISP-style selector: `(car (car (car x)))`.

```
> w
(list (list (list (list "bye") 3) #true) 42)
> (caaar w)
(list "bye")
```

```
(caadr x) → any/c
  x : list?
```

LISP-style selector: `(car (car (cdr x)))`.

```
> (caadr (cons 1 (cons (cons 'a '()) (cons (cons 'd '()) '()))))
'a
```

```
(caar x) → any/c
  x : list?
```

LISP-style selector: `(car (car x))`.

```
> y
(list (list (list 1 2 3) #false "world"))
> (caar y)
(list 1 2 3)
```

```
(cadar x) → any/c
  x : list?
```

LISP-style selector: `(car (cdr (car x)))`.

```
> w
(list (list (list (list "bye") 3) #true) 42)
> (cadar w)
#true
```

```
(cadddr x) → any/c
  x : list?
```

LISP-style selector: `(car (cdr (cdr (cdr x))))`.

```
> v
(list 1 2 3 4 5 6 7 8 9 'A)
> (cadddr v)
4
```

```
(caddr x) → any/c
  x : list?
```

LISP-style selector: `(car (cdr (cdr x)))`.

```
> x
(list 2 "hello" #true)
> (caddr x)
#true
```

```
(cadr x) → any/c
  x : list?
```

LISP-style selector: `(car (cdr x))`.

```
> x
(list 2 "hello" #true)
> (cadr x)
"hello"
```

```
(car x) → any/c
  x : cons?
```

Selects the first item of a non-empty list.

```
> x
(list 2 "hello" #true)
> (car x)
2
```

```
(cdaar x) → any/c
  x : list?
```

LISP-style selector: `(cdr (car (car x)))`.

```
> w
(list (list (list (list "bye") 3) #true) 42)
> (cdaar w)
(list 3)
```

```
(cdadr x) → any/c
  x : list?
```

LISP-style selector: `(cdr (car (cdr x)))`.

```
> (cdadr (list 1 (list 2 "a") 3))
(list "a")
```

```
(cdar x) → list?
  x : list?
```

LISP-style selector: `(cdr (car x))`.

```
> y
(list (list (list 1 2 3) #false "world"))
> (cdar y)
(list #false "world")
```

```
(cddar x) → any/c
  x : list?
```

LISP-style selector: `(cdr (cdr (car x)))`

```
> w
(list (list (list (list "bye") 3) #true) 42)
> (cddar w)
'()
```

```
(cdddr x) → any/c
  x : list?
```

LISP-style selector: `(cdr (cdr (cdr x)))`.

```
> v
(list 1 2 3 4 5 6 7 8 9 'A)
> (cdddr v)
(list 4 5 6 7 8 9 'A)
```

```
(cddr x) → list?
  x : list?
```

LISP-style selector: `(cdr (cdr x))`.

```
> x
(list 2 "hello" #true)
> (cddr x)
(list #true)
```

```
(cdr x) → any/c
  x : cons?
```

Selects the rest of a non-empty list.

```
> x
(list 2 "hello" #true)
> (cdr x)
(list "hello" #true)
```

```
(cons x y) → list?
  x : any/c
  y : list?
```

Constructs a list.

```
> (cons 1 '())
(cons 1 '())
```

```
(cons? x) → boolean?
  x : any/c
```

Determines whether some value is a constructed list.

```
> (cons? (cons 1 '()))
#true
> (cons? 42)
#false
```

```
(eighth x) → any/c
  x : list?
```

Selects the eighth item of a non-empty list.

```
> v
(list 1 2 3 4 5 6 7 8 9 'A)
> (eighth v)
8
```

```
(empty? x) → boolean?
  x : any/c
```

Determines whether some value is the empty list.

```
> (empty? '())
#true
> (empty? 42)
#false
```

```
(fifth x) → any/c
  x : list?
```

Selects the fifth item of a non-empty list.

```
> v
(list 1 2 3 4 5 6 7 8 9 'A)
> (fifth v)
5
```

(first x) → any/c
  x : cons?

Selects the first item of a non-empty list.

```
> x
(list 2 "hello" #true)
> (first x)
2
```

(fourth x) → any/c
  x : list?

Selects the fourth item of a non-empty list.

```
> v
(list 1 2 3 4 5 6 7 8 9 'A)
> (fourth v)
4
```

(length l) → natural?
  l : list?

Evaluates the number of items on a list.

```
> x
(list 2 "hello" #true)
> (length x)
3
```

(list x ...) → list?
  x : any/c

Constructs a list of its arguments.

```
> (list 1 2 3 4 5 6 7 8 9 0)
(cons 1 (cons 2 (cons 3 (cons 4 (cons 5 (cons 6 (cons 7 (cons 8
(cons 9 (cons 0 '()))))))))))
```

```
(list* x ... l) → list?
  x : any/c
  l : list?
```

Constructs a list by adding multiple items to a list.

```
> x
(list 2 "hello" #true)
> (list* 4 3 x)
(list 4 3 2 "hello" #true)
```

```
(list-ref x i) → any/c
  x : list?
  i : natural?
```

Extracts the indexed item from the list.

```
> v
(list 1 2 3 4 5 6 7 8 9 'A)
> (list-ref v 9)
'A
```

```
(list? x) → boolean?
  x : any/c
```

Checks whether the given value is a list.

```
> (list? 42)
#false
> (list? '())
#true
> (list? (cons 1 (cons 2 '())))
#true
```

```
(make-list i x) → list?
  i : natural?
  x : any/c
```

Constructs a list of *i* copies of *x*.

```
> (make-list 3 "hello")
(cons "hello" (cons "hello" (cons "hello" '())))
```

```
(member x l) → boolean?
  x : any/c
  l : list?
```

Determines whether some value is on the list (comparing values with equal?).

```
> x
(list 2 "hello" #true)
> (member "hello" x)
#true
```

```
(member? x l) → boolean?
  x : any/c
  l : list?
```

Determines whether some value is on the list (comparing values with equal?).

```
> x
(list 2 "hello" #true)
> (member? "hello" x)
#true
```

```
(memq x l) → boolean?
  x : any/c
  l : list?
```

Determines whether some value *x* is on some list *l*, using `eq?` to compare *x* with items on *l*.

```
> x
(list 2 "hello" #true)
> (memq (list (list 1 2 3)) x)
#false
```

```
(memq? x l) → boolean?
  x : any/c
  l : list?
```

Determines whether some value *x* is on some list *l*, using `eq?` to compare *x* with items on *l*.

```
> x
(list 2 "hello" #true)
> (memq? (list (list 1 2 3)) x)
#false
```

```
(memv x l) → (or/c #false list)
  x : any/c
  l : list?
```

Determines whether some value is on the list if so, it produces the suffix of the list that starts with x if not, it produces false. (It compares values with the eqv? predicate.)

```
> x
(list 2 "hello" #true)
> (memv (list (list 1 2 3)) x)
#false
```

```
null : list
```

Another name for the empty list

```
> null
'()
```

```
(null? x) → boolean?
  x : any/c
```

Determines whether some value is the empty list.

```
> (null? '())
#true
> (null? 42)
#false
```

```
(range start end step) → list?
  start : number
  end : number
  step : number
```

Constructs a list of numbers by *step*ping from *start* to *end*.

```
> (range 0 10 2)
(cons 0 (cons 2 (cons 4 (cons 6 (cons 8 '()))))
```

```
(remove x l) → list?
  x : any/c
  l : list?
```

Constructs a list like the given one, with the first occurrence of the given item removed (comparing values with equal?).

```
> x
(list 2 "hello" #true)
> (remove "hello" x)
(list 2 #true)
> hello-2
(list 2 "hello" #true "hello")
> (remove "hello" hello-2)
(list 2 #true "hello")
```

```
(remove-all x l) → list?
  x : any/c
  l : list?
```

Constructs a list like the given one, with all occurrences of the given item removed (comparing values with equal?).

```
> x
(list 2 "hello" #true)
> (remove-all "hello" x)
(list 2 #true)
> hello-2
(list 2 "hello" #true "hello")
> (remove-all "hello" hello-2)
(list 2 #true)
```

```
(rest x) → any/c
  x : cons?
```

Selects the rest of a non-empty list.

```
> x
(list 2 "hello" #true)
> (rest x)
(list "hello" #true)
```

```
(reverse l) → list
  l : list?
```

Creates a reversed version of a list.

```
> x
(list 2 "hello" #true)
> (reverse x)
(list #true "hello" 2)
```

```
(second x) → any/c
  x : list?
```

Selects the second item of a non-empty list.

```
> x
(list 2 "hello" #true)
> (second x)
"hello"
```

```
(seventh x) → any/c
  x : list?
```

Selects the seventh item of a non-empty list.

```
> v
(list 1 2 3 4 5 6 7 8 9 'A)
> (seventh v)
7
```

```
(sixth x) → any/c
  x : list?
```

Selects the sixth item of a non-empty list.

```
> v
(list 1 2 3 4 5 6 7 8 9 'A)
> (sixth v)
6
```

(third x) → any/c
  x : list?

Selects the third item of a non-empty list.

```
> x
(list 2 "hello" #true)
> (third x)
#true
```

3.11 Posns

(make-posn x y) → posn
  x : any/c
  y : any/c

Constructs a posn from two arbitrary values.

```
> (make-posn 3 3)
(make-posn 3 3)
> (make-posn "hello" #true)
(make-posn "hello" #true)
```

(posn-x p) → any/c
  p : posn

Extracts the x component of a posn.

```
> p
(make-posn 2 -3)
> (posn-x p)
2
```

(posn-y p) → any/c
  p : posn

Extracts the y component of a posn.

```
> p
(make-posn 2 -3)
> (posn-y p)
-3
```

```
(posn? x) → boolean?
  x : any/c
```

Determines if its input is a posn.

```
> q
(make-posn "bye" 2)
> (posn? q)
#true
> (posn? 42)
#false
```

3.12 Characters

```
(char->integer c) → integer
  c : char
```

Looks up the number that corresponds to the given character in the ASCII table (if any).

```
> (char->integer #\a)
97
> (char->integer #\z)
122
```

```
(char-alphabetic? c) → boolean?
  c : char
```

Determines whether a character represents an alphabetic character.

```
> (char-alphabetic? #\Q)
#true
```

```
(char-ci<=? c d e ...) → boolean?
  c : char
  d : char
  e : char
```

Determines whether the characters are ordered in an increasing and case-insensitive manner.

```
> (char-ci<=? #\b #\B)
#true
> (char<=? #\b #\B)
#false
```

```
(char-ci<? c d e ...) → boolean?
  c : char
  d : char
  e : char
```

Determines whether the characters are ordered in a strictly increasing and case-insensitive manner.

```
> (char-ci<? #\B #\c)
#true
> (char<? #\b #\B)
#false
```

```
(char-ci=? c d e ...) → boolean?
  c : char
  d : char
  e : char
```

Determines whether two characters are equal in a case-insensitive manner.

```
> (char-ci=? #\b #\B)
#true
```

```
(char-ci>=? c d e ...) → boolean?
  c : char
  d : char
  e : char
```

Determines whether the characters are sorted in a decreasing and case-insensitive manner.

```
> (char-ci>=? #\b #\C)
#false
> (char>=? #\b #\C)
#true
```

```
(char-ci>? c d e ...) → boolean?
  c : char
  d : char
  e : char
```

Determines whether the characters are sorted in a strictly decreasing and case-insensitive manner.

```
> (char-ci>? #\b #\B)
#false
> (char>? #\b #\B)
#true
```

```
(char-downcase c) → char
  c : char
```

Produces the equivalent lower-case character.

```
> (char-downcase #\T)
#\t
```

```
(char-lower-case? c) → boolean?
  c : char
```

Determines whether a character is a lower-case character.

```
> (char-lower-case? #\T)
#false
```

```
(char-numeric? c) → boolean?
  c : char
```

Determines whether a character represents a digit.

```
> (char-numeric? #\9)
#true
```

```
(char-upcase c) → char
  c : char
```

Produces the equivalent upper-case character.

```
> (char-upcase #\t)
#\T
```

```
(char-upper-case? c) → boolean?
  c : char
```

Determines whether a character is an upper-case character.

```
> (char-upper-case? #\T)
#true
```

```
(char-whitespace? c) → boolean?
  c : char
```

Determines whether a character represents space.

```
> (char-whitespace? #\tab)
#true
```

```
(char<=? c d e ...) → boolean?
  c : char
  d : char
  e : char
```

Determines whether the characters are ordered in an increasing manner.

```
> (char<=? #\a #\a #\b)
#true
```

```
(char<? x d e ...) → boolean?
  x : char
  d : char
  e : char
```

Determines whether the characters are ordered in a strictly increasing manner.

```
> (char<? #\a #\b #\c)
#true
```

```
(char=? c d e ...) → boolean?
  c : char
  d : char
  e : char
```

Determines whether the characters are equal.

```
> (char=? #\b #\a)
#false
```

```
(char>=? c d e ...) → boolean?
  c : char
  d : char
  e : char
```

Determines whether the characters are sorted in a decreasing manner.

```
> (char>=? #\b #\b #\a)
#true
```

```
(char>? c d e ...) → boolean?
  c : char
  d : char
  e : char
```

Determines whether the characters are sorted in a strictly decreasing manner.

```
> (char>? #\A #\z #\a)
#false
```

```
(char? x) → boolean?
  x : any/c
```

Determines whether a value is a character.

```
> (char? "a")
#false
> (char? #\a)
#true
```

3.13 Strings

```
(explode s) → (listof string)
  s : string
```

Translates a string into a list of 1-letter strings.

```
> (explode "cat")
(list "c" "a" "t")
```

```
(format f x ...) → string
  f : string
  x : any/c
```

Formats a string, possibly embedding values.

```
> (format "Dear Dr. ~a:" "Flatt")
"Dear Dr. Flatt:"
> (format "Dear Dr. ~s:" "Flatt")
"Dear Dr. \"Flatt\":"
```

```
(implode l) → string
  l : list?
```

Concatenates the list of 1-letter strings into one string.

```
> (implode (cons "c" (cons "a" (cons "t" '()))))
"cat"
```

```
(int->string i) → string
  i : integer
```

Converts an integer in [0,55295] or [57344 1114111] to a 1-letter string.

```
> (int->string 65)
"A"
```

```
(list->string l) → string
  l : list?
```

Converts a s list of characters into a string.

```
> (list->string (cons #\c (cons #\a (cons #\t '()))))
"cat"
```

(make-string *i* *c*) → string
  i : natural?
  c : char

Produces a string of length *i* from *c*.

```
> (make-string 3 #\d)
"ddd"
```

(replicate *i* *s*) → string
  i : natural?
  s : string

Replicates *s* *i* times.

```
> (replicate 3 "h")
"hhh"
```

(string *c* ...) → string?
  c : char

Builds a string of the given characters.

```
> (string #\d #\o #\g)
"dog"
```

(string->int *s*) → integer
  s : string

Converts a 1-letter string to an integer in [0,55295] or [57344, 1114111].

```
> (string->int "a")
97
```

```
(string->list s) → (listof char)
  s : string
```

Converts a string into a list of characters.

```
> (string->list "hello")
(list #\h #\e #\l #\l #\o)
```

```
(string->number s) → (union number #false)
  s : string
```

Converts a string into a number, produce false if impossible.

```
> (string->number "-2.03")
-2.03
> (string->number "1-2i")
1-2i
```

```
(string->symbol s) → symbol
  s : string
```

Converts a string into a symbol.

```
> (string->symbol "hello")
'hello
```

```
(string-alphabetic? s) → boolean?
  s : string
```

Determines whether all 'letters' in the string are alphabetic.

```
> (string-alphabetic? "123")
#false
> (string-alphabetic? "cat")
#true
```

```
(string-contains-ci? s t) → boolean?
  s : string
  t : string
```

Determines whether the first string appears in the second one without regard to the case of the letters.

```
> (string-contains-ci? "At" "caT")
#true
```

```
(string-contains? s t) → boolean?
  s : string
  t : string
```

Determines whether the first string appears literally in the second one.

```
> (string-contains? "at" "cat")
#true
```

```
(string-copy s) → string
  s : string
```

Copies a string.

```
> (string-copy "hello")
"hello"
```

```
(string-downcase s) → string
  s : string
```

Produces a string like the given one with all 'letters' as lower case.

```
> (string-downcase "CAT")
"cat"
> (string-downcase "cAt")
"cat"
```

```
(string-ith s i) → 1string?
  s : string
  i : natural?
```

Extracts the *i* th 1-letter substring from *s*.

```
> (string-ith "hello world" 1)
"e"
```

```
(string-length s) → nat
  s : string
```

Determines the length of a string.

```
> (string-length "hello world")
11
```

```
(string-lower-case? s) → boolean?
  s : string
```

Determines whether all 'letters' in the string are lower case.

```
> (string-lower-case? "CAT")
#false
```

```
(string-numeric? s) → boolean?
  s : string
```

Determines whether all 'letters' in the string are numeric.

```
> (string-numeric? "123")
#true
> (string-numeric? "1-2i")
#false
```

```
(string-ref s i) → char
  s : string
  i : natural?
```

Extracts the *i* th character from *s*.

```
> (string-ref "cat" 2)
#\t
```

```
(string-upcase s) → string
  s : string
```

Produces a string like the given one with all 'letters' as upper case.

```
> (string-upcase "cat")
"CAT"
> (string-upcase "cAt")
"CAT"
```

```
(string-upper-case? s) → boolean?
  s : string
```

Determines whether all 'letters' in the string are upper case.

```
> (string-upper-case? "CAT")
#true
```

```
(string-whitespace? s) → boolean?
  s : string
```

Determines whether all 'letters' in the string are white space.

```
> (string-whitespace? (string-append " " (string #\tab #\newline #\return)))
#true
```

```
(string? x) → boolean?
  x : any/c
```

Determines whether a value is a string.

```
> (string? "hello world")
#true
> (string? 42)
#false
```

```
(substring s i j) → string
  s : string
  i : natural?
  j : natural?
```

Extracts the substring starting at *i* up to *j* (or the end if *j* is not provided).

```
> (substring "hello world" 1 5)
"ello"
> (substring "hello world" 1 8)
"ello wo"
> (substring "hello world" 4)
"o world"
```

3.14 Images

```
(image=? i j) → boolean?
  i : image
  j : image
```

Determines whether two images are equal.

```
> c1
```

```
> (image=? (circle 5 "solid" "green") c1)
#false
> (image=? (circle 10 "solid" "green") c1)
#true
```

```
(image? x) → boolean?
  x : any/c
```

Determines whether a value is an image.

```
> c1
```

```
> (image? c1)
#true
```

3.15 Misc

```
(=~ x y eps) → boolean?
  x : number
  y : number
  eps : non-negative-real
```

Checks whether *x* and *y* are within *eps* of either other.

```
> (=~ 1.01 1.0 0.1)
#true
> (=~ 1.01 1.5 0.1)
#false
```

```
eof : eof-object?
```

A value that represents the end of a file:

```
> eof
#<eof>
```

```
(eof-object? x) → boolean?
  x : any/c
```

Determines whether some value is the end-of-file value.

```
> (eof-object? eof)
#true
> (eof-object? 42)
#false
```

```
(eq? x y) → boolean?
  x : any/c
  y : any/c
```

Determines whether two values are equivalent from the computer's perspective (intensional).

```
> (eq? (cons 1 '()) (cons 1 '()))
#false
> one
(list 1)
> (eq? one one)
#true
```

```
(equal? x y) → boolean?
  x : any/c
  y : any/c
```

Determines whether two values are structurally equal where basic values are compared with the eqv? predicate.

```
> (equal? (make-posn 1 2) (make-posn (- 2 1) (+ 1 1)))
#true
```

```
(equal~? x y z) → boolean?
  x : any/c
  y : any/c
  z : non-negative-real
```

Compares *x* and *y* like `equal?` but uses =~ in the case of numbers.

```
> (equal~? (make-posn 1.01 1.0) (make-posn 1.01 0.99) 0.2)
#true
```

```
(eqv? x y) → boolean?
  x : any/c
  y : any/c
```

Determines whether two values are equivalent from the perspective of all functions that can be applied to it (extensional).

```
> (eqv? (cons 1 '()) (cons 1 '()))
#false
> one
(list 1)
> (eqv? one one)
#true
```

```
(error x ...) → void?
  x : any/c
```

Signals an error, combining the given values into an error message. If any of the values' printed representations is too long, it is truncated and "..." is put into the string. If the first value is a symbol, it is suffixed with a colon and the result pre-pended on to the error message.

```
> zero
0
> (if (= zero 0) (error "can't divide by 0") (/ 1 zero))
can't divide by 0
```

```
(exit) → void
```

Evaluating `(exit)` terminates the running program.

```
(identity x) → any/c
  x : any/c
```

Returns *x*.

```
> (identity 42)
42
> (identity c1)
```

```
> (identity "hello")
"hello"
```

```
(struct? x) → boolean?
  x : any/c
```

Determines whether some value is a structure.

```
> (struct? (make-posn 1 2))
#true
> (struct? 43)
#false
```

3.16 Signatures

```
Any : signature?
```

Signature for any value.

```
Boolean : signature?
```

Signature for booleans.

```
Char : signature?
```

Signature for chararacters.

```
(ConsOf first-sig rest-sig) → signature?
  first-sig : signature?
  rest-sig : signature?
```

Signature for a cons pair.

```
EmptyList : signature?
```

Signature for the empty list.

```
False : signature?
```

Signature for just false.

```
Integer : signature?
```

Signature for integers.

```
Natural : signature?
```

Signature for natural numbers.

```
Number : signature?
```

Signature for arbitrary numbers.

```
Rational : signature?
```

Signature for rational numbers.

```
Real : signature?
```

Signature for real numbers.

```
String : signature?
```

Signature for strings.

```
Symbol : signature?
```

Signature for symbols.

```
True : signature?
```

Signature for just true.

3.17 Numbers (relaxed conditions)

```
(* x ...) → number
  x : number
```

Multiplies all given numbers. In ISL and up: `*` works when applied to only one number or none.

```
> (* 5 3)
15
> (* 5 3 2)
30
> (* 2)
2
> (*)
1
```

```
(+ x ...) → number
  x : number
```

Adds all given numbers. In ISL and up: `+` works when applied to only one number or none.

```
> (+ 2/3 1/16)
35/48
> (+ 3 2 5 8)
18
> (+ 1)
1
> (+)
0
```

```
(/ x y ...) → number
  x : number
  y : number
```

Divides the first by all remaining numbers. In ISL and up: `/` computes the inverse when applied to one number.

```
> (/ 12 2)
6
> (/ 12 2 3)
2
> (/ 3)
1/3
```

```
(= x ...) → number
  x : number
```

Compares numbers for equality. In ISL and up: = works when applied to only one number.

```
> (= 10 10)
#true
> (= 11)
#true
> (= 0)
#true
```

3.18 String (relaxed conditions)

```
(string-append s ...) → string
  s : string
```

Concatenates the characters of several strings.

```
> (string-append "hello" " " "world" " " "good bye")
"hello world good bye"
```

```
(string-ci<=? s t x ...) → boolean?
  s : string
  t : string
  x : string
```

Determines whether the strings are ordered in a lexicographically increasing and case-insensitive manner.

```
> (string-ci<=? "hello" "WORLD" "zoo")
#true
```

```
(string-ci<? s t x ...) → boolean?
  s : string
  t : string
  x : string
```

Determines whether the strings are ordered in a lexicographically strictly increasing and case-insensitive manner.

```
> (string-ci<? "hello" "WORLD" "zoo")
#true
```

```
(string-ci=? s t x ...) → boolean?
  s : string
  t : string
  x : string
```

Determines whether all strings are equal, character for character, regardless of case.

```
> (string-ci=?  "hello" "HellO")
#true
```

```
(string-ci>=? s t x ...) → boolean?
  s : string
  t : string
  x : string
```

Determines whether the strings are ordered in a lexicographically decreasing and case-insensitive manner.

```
> (string-ci>?  "zoo" "WORLD" "hello")
#true
```

```
(string-ci>? s t x ...) → boolean?
  s : string
  t : string
  x : string
```

Determines whether the strings are ordered in a lexicographically strictly decreasing and case-insensitive manner.

```
> (string-ci>?  "zoo" "WORLD" "hello")
#true
```

```
(string<=? s t x ...) → boolean?
  s : string
  t : string
  x : string
```

Determines whether the strings are ordered in a lexicographically increasing manner.

```
> (string<=? "hello" "hello" "world" "zoo")
#true
```

```
(string<? s t x ...) → boolean?
  s : string
  t : string
  x : string
```

Determines whether the strings are ordered in a lexicographically strictly increasing manner.

```
> (string<? "hello" "world" "zoo")
#true
```

```
(string=? s t x ...) → boolean?
  s : string
  t : string
  x : string
```

Determines whether all strings are equal, character for character.

```
> (string=? "hello" "world")
#false
> (string=? "bye" "bye")
#true
```

```
(string>=? s t x ...) → boolean?
  s : string
  t : string
  x : string
```

Determines whether the strings are ordered in a lexicographically decreasing manner.

```
> (string>=?  "zoo" "zoo" "world" "hello")
#true
```

```
(string>? s t x ...) → boolean?
  s : string
  t : string
  x : string
```

Determines whether the strings are ordered in a lexicographically strictly decreasing manner.

```
> (string>?  "zoo" "world" "hello")
#true
```

3.19 Posn

```
(posn) → signature
```

Signature for posns.

3.20 Higher-Order Functions

```
(andmap p? l ...) → boolean
  p? : (X ... -> boolean)
  l : (listof X)
```

Determines whether *p?* holds for all items of *l* ...:

```
(andmap p (list x-1 ... x-n)) = (and (p x-1) ... (p x-n))
```

```
(andmap p (list x-1 ... x-n) (list y-1 ... y-n)) = (and (p x-1 y-
1) ... (p x-n y-n))
```

```
> (andmap odd? '(1 3 5 7 9))
#true
```

```
> (andmap even? '())
#true
```

Making sure all numbers are below some threshold:

```
> (define (small-enough? x)
    (< x 3))
```

```
> (andmap small-enough? '(0 1 2))
#true
```

Checking that all items in the first list satisfy the corresponding predictate in the 2nd:

```
> (define (and-satisfies? x f)
    (f x))
```

```
> (andmap and-satisfies? (list 0 1 2) (list odd? even? positive?))
#false
```

```
(apply f x-1 ... l) → Y
  f : (X-1 ... X-N -> Y)
  x-1 : X-1
  l : (list X-i+1 ... X-N)
```

Applies a function using items from a list as the arguments:

```
(apply f (list x-1 ... x-n)) = (f x-1 ... x-n)
```

```
> a-list
(list 0 1 2 3 4 5 6 7 8 9)
> (apply max a-list)
9
```

```
(argmax f l) → X
  f : (X -> real)
  l : (listof X)
```

Finds the (first) element of the list that maximizes the output of the function.

```
> (argmax second '((sam 98) (carl 78) (vincent 93) (asumu 99)))
(list 'asumu 99)
```

```
(argmin f l) → X
  f : (X -> real)
  l : (listof X)
```

Finds the (first) element of the list that minimizes the output of the function.

```
> (argmin second '((sam 98) (carl 78) (vincent 93) (asumu 99)))
(list 'carl 78)
```

```
(build-list n f) → (listof X)
  n : nat
  f : (nat -> X)
```

Constructs a list by applying `f` to the numbers between `0` and `(- n 1)`:

```
(build-list n f) = (list (f 0) ... (f (- n 1)))
```

```
> (build-list 22 add1)
(list 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22)
```

Creating a diagonal matrix:

```
> (define (diagonalize i)
    (local ((define (off j)
              (if (= i j) 1 0)))
      (build-list 3 off)))


> (build-list 3 diagonalize)
(list (list 1 0 0) (list 0 1 0) (list 0 0 1))
```

(build-string *n* *f*) → string
  n : nat
  f : (nat -> char)

Constructs a string by applying *f* to the numbers between 0 and (- *n* 1):

```
(build-string n f) = (string (f 0) ... (f (- n 1)))
```

```
> (build-string 10 integer->char)
"\u0000\u0001\u0002\u0003\u0004\u0005\u0006\a\b\t"
```

Making the alphabet:

```
> (define (starting-at-a x)
    (integer->char (+ 65 x)))


> (build-string 26 starting-at-a)
"ABCDEFGHIJKLMNOPQRSTUVWXYZ"
```

(compose *f* *g*) → (X -> Z)
  f : (Y -> Z)
  g : (X -> Y)

Composes a sequence of procedures into a single procedure:

```
(compose f g)
```

is equivalent to

```
(define (f-after-g x)
  (f (g x)))
```

```
> (map (compose add1 second) '((add 3) (sub 2) (mul 4)))
(list 4 3 5)
```

```
(filter p? l) → (listof X)
  p? : (X -> boolean)
  l : (listof X)
```

Constructs a list from all those items on a list for which the predicate holds.

```
> (filter odd? '(0 1 2 3 4 5 6 7 8 9))
(list 1 3 5 7 9)
```

Keep only numbers that are large enough:

```
> (define (large-enough? x)
    (>= x 3))
```

```
> (filter large-enough? '(0 1 2 3 4 5 6 7 8 9))
(list 3 4 5 6 7 8 9)
```

```
(foldl f base l ...) → Y
  f : (X ... Y -> Y)
  base : Y
  l : (listof X)
```

```
(foldl f base (list x-1 ... x-n)) = (f x-n ... (f x-1 base))
```

```
(foldl f base (list x-1 ... x-n) (list y-1 ... y-n))
= (f x-n y-n ... (f x-1 y-1 base))
```

```
> (foldl + 0 '(0 1 2 3 4 5 6 7 8 9))
45
```

```
> (foldl cons '() '(a b c))
(list 'c 'b 'a)
```

```
(foldr f base l ...) → Y
  f : (X ... Y -> Y)
  base : Y
  l : (listof X)
```

```
(foldr f base (list x-1 ... x-n)) = (f x-1 ... (f x-n base))
```

```
(foldr f base (list x-1 ... x-n) (list y-1 ... y-n))
= (f x-1 y-1 ... (f x-n y-n base))
```

```
> (foldr + 0 '(0 1 2 3 4 5 6 7 8 9))
45
```

Append all rests of all lists:

```
> (define (append-rests f r)
    (append (rest f) r))
```

```
> (foldr append-rests '() '((1 a) (2 b c) (3 d e f)))
(list 'a 'b 'c 'd 'e 'f)
```

Add two lists of numbers:

```
> (define (add-two-lists x y r)
    (+ x y r))
```

```
> (foldr add-two-lists 0 '(1 2 3) '(10 11 12))
39
```

```
(map f l ...) → (listof Z)
  f : (X ... -> Z)
  l : (listof X)
```

Constructs a new list by applying a function to each item on one or more existing lists:

```
(map f (list x-1 ... x-n)) = (list (f x-1) ... (f x-n))
```

```
(map f (list x-1 ... x-n) (list y-1 ... y-n)) = (list (f x-1 y-
1) ... (f x-n y-n))
```

```
> (map add1 (list 3 -4.01 2/5))
(list 4 #i-3.01 1.4)
```

Mapping a user-defined function:

```
> (define (tag-with-a x)
    (list "a" (+ x 1)))


> (map tag-with-a (list 3 -4.01 2/5))
(list (list "a" 4) (list "a" #i-3.01) (list "a" 1.4))
```

Mapping over two lists:

```
> (define (add-and-multiply x y)
    (+ x (* x y)))


> (map add-and-multiply (list 3 -4 2/5) '(1 2 3))
(list 6 -12 1.6)
```

```
(memf p? l) → (union #false (listof X))
  p? : (X -> any)
  l : (listof X)
```

Produces `#false` if *p?* produces `false` for all items on *l*. If *p?* produces `#true` for any of the items on *l*, `memf` returns the sub-list starting from that item.

```
> (memf odd? '(2 4 6 3 8 0))
(list 3 8 0)
```

```
(ormap p? l ...) → boolean
  p? : (X ... -> boolean)
  l : (listof X)
```

Determines whether *p?* holds for at least one items of *l*:

```
(ormap p (list x-1 ... x-n)) = (or (p x-1) ... (p x-n))


(ormap p (list x-1 ... x-n) (list y-1 ... y-n)) = (or (p x-1 y-
1) ... (p x-n y-n))
```

```
> (ormap odd? '(1 3 5 7 9))
#true


> (ormap even? '())
#false
```

Making sure at least one number is below some threshold:

```
> (define (a-small-one? x)
    (< x 3))


> (ormap a-small-one? '(6 7 8 1 5))
#true
```

Checking that one item in the first list satisfy the corresponding predictate in the 2nd:

```
> (define (or-satisfies? x f)
    (f x))


> (ormap or-satisfies? (list 0 1 2) (list odd? even? positive?))
#true
```

```
(procedure? x) → boolean?
  x : any
```

Produces true if the value is a procedure.

```
> (procedure? cons)
#true


> (procedure? add1)
#true
```

Checking a programmer-defined function:

```
> (define (my-function x)
    x)


> (procedure? my-function)
#true
```

```
(quicksort l comp) → (listof X)
  l : (listof X)
  comp : (X X -> boolean)
```

Sorts the items on *l*, in an order according to *comp* (using the quicksort algorithm).

```
> (quicksort '(6 7 2 1 3 4 0 5 9 8) <)
(list 0 1 2 3 4 5 6 7 8 9)
```

```
(sort l comp) → (listof X)
  l : (listof X)
  comp : (X X -> boolean)
```

Sorts the items on *l*, in an order according to *comp*.

```
> (sort '(6 7 2 1 3 4 0 5 9 8) <)
(list 0 1 2 3 4 5 6 7 8 9)
```

4 Intermediate Student with Lambda

The grammar notation uses the notation `X` **...** (bold dots) to indicate that `X` may occur an arbitrary number of times (zero, one, or more). Separately, the grammar also defines `...` as an identifier to be used in templates.

```
                program = def-or-expr ...

          def-or-expr = definition
                      | expr
                      | test-case
                      | library-require
                      | signature-declaration

           definition = (define (name variable variable ...) expr)
                      | (define name expr)
                      | (define-struct name (name ...))

                expr = (lambda (variable variable ...) expr)
                     | (λ (variable variable ...) expr)
                     | (local [definition ...] expr)
                     | (letrec ([name expr] ...) expr)
                     | (let ([name expr] ...) expr)
                     | (let* ([name expr] ...) expr)
                     | (expr expr expr ...)
                     | (cond [expr expr] ... [expr expr])
                     | (cond [expr expr] ... [else expr])
                     | (if expr expr expr)
                     | (and expr expr expr ...)
                     | (or expr expr expr ...)
                     | (time expr)
                     | name
                     | prim-op
                     | 'quoted
                     | `quasiquoted
                     | '()
                     | number
                     | boolean
                     | string
                     | character
                     | (signature signature-form)

signature-declaration = (: name signature-form)

       signature-form = (enum expr ...)
```

```
                    | (mixed signature-form ...)
                    | (signature-form ... -> signature-form)
                    | (ListOf signature-form)
                    | signature-variable
                    | expr

signature-variable = %name

            quoted = name
                    | number
                    | string
                    | character
                    | (quoted ...)
                    | 'quoted
                    | `quoted
                    | ,quoted
                    | ,@quoted

        quasiquoted = name
                    | number
                    | string
                    | character
                    | (quasiquoted ...)
                    | 'quasiquoted
                    | `quasiquoted
                    | ,expr
                    | ,@expr

          test-case = (check-expect expr expr)
                    | (check-random expr expr)
                    | (check-within expr expr expr)
                    | (check-random-within expr expr expr)
                    | (check-member-of expr expr ...)
                    | (check-range expr expr expr)
                    | (check-satisfied expr expr)
                    | (check-error expr expr)
                    | (check-error expr)

     library-require = (require string)
                    | (require (lib string string ...))
                    | (require (planet string package))

            package = (string string number number)
```

A *name* or a *variable* is a sequence of characters not including a space or one of the following:

" , ' ` ( ) [ ] { } | ; #

A *number* is a number such as 123, 3/2, or 5.5.

A *boolean* is one of: #true or #false.

Alternative spellings for the #true constant are #t, true, and #T. Similarly, #f, false, or #F are also recognized as #false.

A *symbol* is a quote character followed by a name. A symbol is a value, just like 42, '(), or #false.

A *string* is a sequence of characters enclosed by a pair of ". Unlike symbols, strings may be split into characters and manipulated by a variety of functions. For example, "abcdef", "This is a string", and "This is a string with \" inside" are all strings.

A *character* begins with #\ and has the name of the character. For example, #\a, #\b, and #\space are characters.

In function calls, the function appearing immediately after the open parenthesis can be any functions defined with define or define-struct, or any one of the pre-defined functions.

4.1 Pre-defined Variables

empty : empty?

The empty list.

true : boolean?

The #true value.

false : boolean?

The #false value.

4.2 Template Variables

. .

A placeholder for indicating that a function definition is a template.

```
...
```

A placeholder for indicating that a function definition is a template.

```
....
```

A placeholder for indicating that a function definition is a template.

```
.....
```

A placeholder for indicating that a function definition is a template.

```
......
```

A placeholder for indicating that a function definition is a template.

4.3 Syntax for Intermediate with Lambda

```
(lambda (variable variable ...) expression)
```

Creates a function that takes as many arguments as given *variable*s, and whose body is *expression*.

```
(λ (variable variable ...) expression)
```

The Greek letter λ is a synonym for `lambda`.

```
(expression expression expression ...)
```

Calls the function that results from evaluating the first *expression*. The value of the call is the value of function's body when every instance of `name`'s variables are replaced by the values of the corresponding *expression*s.

The function being called must come from either a definition appearing before the function call, or from a `lambda` expression. The number of argument *expression*s must be the same as the number of arguments expected by the function.

```
(local [definition ...] expression)
```

Groups related definitions for use in *expression*. Each *definition* can be either a `define` or a `define-struct`.

When evaluating `local`, each *definition* is evaluated in order, and finally the body *expression* is evaluated. Only the expressions within the `local` (including the right-hand-sides of the *definition*s and the *expression*) may refer to the names defined by the *definition*s. If a name defined in the `local` is the same as a top-level binding, the inner one "shadows" the outer one. That is, inside the `local`, any references to that name refer to the inner one.

```
(letrec ([name expr-for-let] ...) expression)
```

Like `local`, but with a simpler syntax. Each *name* defines a variable (or a function) with the value of the corresponding *expr-for-let*. If *expr-for-let* is a `lambda`, `letrec` defines a function, otherwise it defines a variable.

```
(let* ([name expr-for-let] ...) expression)
```

Like `letrec`, but each *name* can only be used in *expression*, and in *expr-for-let*s occuring after that *name*.

```
(let ([name expr-for-let] ...) expression)
```

Like `letrec`, but the defined *name*s can be used only in the last *expression*, not the *expr-for-let*s next to the *name*s.

```
(time expression)
```

Measures the time taken to evaluate *expression*. After evaluating *expression*, `time` prints out the time taken by the evaluation (including real time, time taken by the CPU, and the time spent collecting free memory). The value of `time` is the same as that of *expression*.

4.4 Common Syntaxes

The following syntaxes behave the same in the *Intermediate with Lambda* level as they did in the §3 "Intermediate Student" level.

```
(define (name variable variable ...) expression)
```

Defines a function named *name*. The *expression* is the body of the function. When the function is called, the values of the arguments are inserted into the body in place of the *variable*s. The function returns the value of that new expression.

The function name's cannot be the same as that of another function or variable.

```
(define name expression)
```

Defines a variable called *name* with the the value of *expression*. The variable name's cannot be the same as that of another function or variable, and *name* itself must not appear in *expression*.

```
'name
'part
(quote name)
(quote part)
```

A quoted name is a symbol. A quoted part is an abbreviation for a nested lists.

Normally, this quotation is written with a `'`, like `'(apple banana)`, but it can also be written with `quote`, like `(quote (apple banana))`.

```
`name
`part
(quasiquote name)
(quasiquote part)
```

Like `quote`, but also allows escaping to expression "unquotes."

Normally, quasi-quotations are written with a backquote, `` ` ``, like `` `(apple ,(+ 1 2)) ``, but they can also be written with `quasiquote`, like `(quasiquote (apple ,(+ 1 2)))`.

```
,expression
(unquote expression)
```

Under a single quasiquote, `,`*expression* escapes from the quote to include an evaluated expression whose result is inserted into the abbreviated list.

Under multiple quasiquotes, `,`*expression* is really the literal `,`*expression*, decrementing the quasiquote count by one for *expression*.

Normally, an unquote is written with `,`, but it can also be written with `unquote`.

```
,@expression
(unquote-splicing expression)
```

Under a single quasiquote, `,@`*expression* escapes from the quote to include an evaluated expression whose result is a list to splice into the abbreviated list.

Under multiple quasiquotes, a splicing unquote is like an unquote; that is, it decrements the quasiquote count by one.

Normally, a splicing unquote is written with `,@`, but it can also be written with `unquote-splicing`.

```
(define-struct structure-name (field-name ...))
```

Defines a new structure called *structure-name*. The structure's fields are named by the *field-name*s. After the `define-struct`, the following new functions are available:

- `make-`*structure-name* : takes a number of arguments equal to the number of fields in the structure, and creates a new instance of that structure.
- *structure-name-field-name* : takes an instance of the structure and returns the value in the field named by *field-name*.
- *structure-name*`?` : takes any value, and returns `#true` if the value is an instance of the structure.

The name of the new functions introduced by `define-struct` must not be the same as that of other functions or variables, otherwise `define-struct` reports an error.

```
(cond [question-expression answer-expression] ...)
(cond [question-expression answer-expression]
      ...
      [else answer-expression])
```

Chooses a clause based on some condition. `cond` finds the first *question-expression* that evaluates to `#true`, then evaluates the corresponding *answer-expression*.

If none of the *question-expression*s evaluates to `#true`, `cond`'s value is the *answer-expression* of the `else` clause. If there is no `else`, `cond` reports an error. If the result of a *question-expression* is neither `#true` nor `#false`, `cond` also reports an error.

`else` cannot be used outside of `cond`.

```
(if question-expression
    then-answer-expression
    else-answer-expression)
```

When the value of the *question-expression* is #true, if evaluates the *then-answer-expression*. When the test is #false, if evaluates the *else-answer-expression*.

If the *question-expression* is neither #true nor #false, if reports an error.

```
(and expression expression expression ...)
```

Evaluates to #true if all the *expression*s are #true. If any *expression* is #false, the and expression evaluates to #false (and the expressions to the right of that expression are not evaluated.)

If any of the expressions evaluate to a value other than #true or #false, and reports an error.

```
(or expression expression expression ...)
```

Evaluates to #true as soon as one of the *expression*s is #true (and the expressions to the right of that expression are not evaluated.) If all of the *expression*s are #false, the or expression evaluates to #false.

If any of the expressions evaluate to a value other than #true or #false, or reports an error.

```
(check-expect expression expected-expression)
```

Checks that the first *expression* evaluates to the same value as the *expected-expression*.

```
(check-expect (fahrenheit->celsius 212) 100)
(check-expect (fahrenheit->celsius -40) -40)

(define (fahrenheit->celsius f)
  (* 5/9 (- f 32)))
```

A check-expect expression must be placed at the top-level of a student program. Also it may show up anywhere in the program, including ahead of the tested function definition. By placing check-expects there, a programmer conveys to a future reader the intention behind the program with working examples, thus making it often superfluous to read the function

definition proper. Syntax errors in `check-expect` (and all check forms) are intentionally delayed to run time so that students can write tests *without* necessarily writing complete function headers.

It is an error for `expr` or `expected-expr` to produce an inexact number or a function value. As for inexact numbers, it is *morally* wrong to compare them for plain equality. Instead one tests whether they are both within a small interval; see `check-within`. As for functions (see Intermediate and up), it is provably impossible to compare functions.

```
(check-random expression expected-expression)
```

Checks that the first *expression* evaluates to the same value as the *expected-expression*.

The form supplies the same random-number generator to both parts. If both parts request `random` numbers from the same interval in the same order, they receive the same random numbers.

Here is a simple example of where `check-random` is useful:

```
(define WIDTH 100)
(define HEIGHT (* 2 WIDTH))

(define-struct player (name x y))
; A Player is (make-player String Nat Nat)

; String -> Player

(check-random (create-randomly-placed-player "David Van Horn")
              (make-player "David Van Horn" (random WIDTH) (random HEIGHT)))

(define (create-randomly-placed-player name)
  (make-player name (random WIDTH) (random HEIGHT)))
```

Note how `random` is called on the same numbers in the same order in both parts of `check-random`. If the two parts call `random` for different intervals, they are likely to fail:

```
; String -> Player

(check-random (create-randomly-placed-player "David Van Horn")
              (make-player "David Van Horn" (random WIDTH) (random HEIGHT)))

(define (create-randomly-placed-player name)
  (a-helper-function name (random HEIGHT)))
```

```
; String Number -> Player
(define (a-helper-function name height)
   (make-player name (random WIDTH) height))
```

Because the argument to `a-helper-function` is evaluated first, `random` is first called for the interval [0,*HEIGHT*) and then for [0,*WIDTH*), that is, in a different order than in the preceding `check-random`.

It is an error for `expr` or `expected-expr` to produce a function value or an inexact number; see note on `check-expect` for details.

```
(check-satisfied expression predicate)
```

Checks that the first *expression* satisfies the named *predicate* (function of one argument). Recall that "satisfies" means "the function produces `#true` for the given value."

Here are simple examples for `check-satisfied`:

```
> (check-satisfied 1 odd?)
The test passed!


> (check-satisfied 1 even?)
Ran 1 test.
0 tests passed.
Check failures:

        Actual value | 1 | does not satisfy even?.

at line 3, column 0
```

In general `check-satisfied` empowers program designers to use defined functions to formulate test suites:

```
; [cons Number [List-of Number]] -> Boolean
; a function for testing htdp-sort

(check-expect (sorted? (list 1 2 3)) #true)
(check-expect (sorted? (list 2 1 3)) #false)

(define (sorted? l)
  (cond
    [(empty? (rest l)) #true]
    [else (and (<= (first l) (second l)) (sorted? (rest l)))]))
```

```
; [List-of Number] -> [List-of Number]
; create a sorted version of the given list of numbers

(check-satisfied (htdp-sort (list 1 2 0 3)) sorted?)

(define (htdp-sort l)
  (cond
    [(empty? l) l]
    [else (insert (first l) (htdp-sort (rest l)))]))

; Number [List-of Number] -> [List-of Number]
; insert x into l at proper place
; assume l is arranged in ascending order
; the result is sorted in the same way
(define (insert x l)
  (cond
    [(empty? l) (list x)]
    [else (if (<= x (first l)) (cons x l) (cons (first l) (insert x (rest l))))]))
```

And yes, the results of `htdp-sort` satisfy the `sorted?` predicate:

```
> (check-satisfied (htdp-sort (list 1 2 0 3)) sorted?)
```

`(check-within` *`expression expected-expression delta`*`)`

Checks whether the value of the *`expression`* expression is structurally equal to the value produced by the *`expected-expression`* expression; every number in the first expression must be within *`delta`* of the corresponding number in the second expression.

```
(define-struct roots (x sqrt))
; RT is [List-of (make-roots Number Number)]

(define (root-of a)
  (make-roots a (sqrt a)))

(define (roots-table xs)
  (cond
    [(empty? xs) '()]
    [else (cons (root-of (first xs)) (roots-table (rest xs)))]))
```

Due to the presence of inexact numbers in nested data, `check-within` is the correct choice for testing, and the test succeeds if *`delta`* is reasonably large:

Example:

```
> (check-within (roots-table (list 1.0 2.0 3.0))
                (list
                  (make-roots 1.0 1.0)
                  (make-roots 2  1.414)
                  (make-roots 3  1.713))
                0.1)
The test passed!
```

In contrast, when *delta* is small, the test fails:

Example:

```
> (check-within (roots-table (list 2.0))
                (list
                  (make-roots 2  1.414))
                #i1e-5)
Ran 1 test.
0 tests passed.
Check failures:
                     ┌──────────────────────────────────────┐

        Actual value │ '((make-roots 2.0 1.4142135623730951)) │ is
not within 1e-5 of expected value │ '((make-roots 2 1.414)) │.
                     └──────────────────────────────────────┘

at line 5, column 0
```

It is an error for `expressions` or *expected-expression* to produce a function value; see note on `check-expect` for details.

If *delta* is not a number, `check-within` reports an error.

```
(check-random-within expression expected-expression delta)
```

Combines `check-random` and `check-within`. Like `check-random`, *expression* and *expected-expression* are evaluated with the same random-number sequence, so calls to `random` in both expressions draw the same numbers. Like `check-within`, the test then succeeds if every number in the value of *expression* is within *delta* of the corresponding number in the value of *expected-expression*.

Use `check-random-within` in place of `check-random` when the random computation produces inexact numbers, which `check-random` refuses to compare.

If *delta* is not a number, `check-random-within` reports an error.

Added in version 1.9 of package `htdp-lib`.

```
(check-error expression expected-error-message)
(check-error expression)
```

Checks that the *expression* reports an error, where the error messages matches the value of *expected-error-message*, if it is present.

Here is a typical beginner example that calls for a use of `check-error`:

```
(define sample-table
  '(("matthias" 10)
    ("matthew"  20)
    ("robby"    -1)
    ("shriram"  18)))

; [List-of [list String Number]] String -> Number
; determine the number associated with s in table

(define (lookup table s)
  (cond
    [(empty? table) (error (string-append s " not found"))]
    [else (if (string=? (first (first table)) s)
              (second (first table))
              (lookup (rest table)))]))
```

Consider the following two examples in this context:

Example:

```
> (check-expect (lookup sample-table "matthew") 20)
The test passed!
```

Example:

```
> (check-error (lookup sample-table "kathi") "kathi not found")
The test passed!
```

```
(check-member-of expression expression expression ...)
```

Checks that the value of the first *expression* is that of one of the following *expression*s.

```
; [List-of X] -> X
; pick a random element from the given list l
(define (pick-one l)
  (list-ref l (random (length l))))
```

Example:

```
> (check-member-of (pick-one '("a" "b" "c")) "a" "b" "c")
The test passed!
```

It is an error for any of `expressions` to produce a function value; see note on `check-expect` for details.

`(check-range` *`expression low-expression high-expression`*`)`

Checks that the value of the first *`expression`* is a number in between the value of the *`low-expression`* and the *`high-expression`*, inclusive.

A `check-range` form is best used to delimit the possible results of functions that compute inexact numbers:

```
(define EPSILON 0.001)

; [Real -> Real] Real -> Real
; what is the slope of f at x?
(define (differentiate f x)
  (slope f (- x EPSILON) (+ x EPSILON)))

; [Real -> Real] Real Real -> Real
(define (slope f left right)
  (/ (- (f right) (f left))
     2 EPSILON))

(check-range (differentiate sin 0) 0.99 1.0)
```

It is an error for *`expression`*, *`low-expression`*, or *`high-expression`* to produce a function value; see note on `check-expect` for details.

`(require` *`string`*`)`

Makes the definitions of the module specified by *`string`* available in the current module (i.e., the current file), where *`string`* refers to a file relative to the current file.

The *string* is constrained in several ways to avoid problems with different path conventions on different platforms: a `/` is a directory separator, `.` always means the current directory, `..` always means the parent directory, path elements can use only `a` through `z` (uppercase or lowercase), `0` through `9`, `-`, `_`, and `.`, and the string cannot be empty or contain a leading or trailing `/`.

```
(require module-name)
```

Accesses a file in an installed library. The library name is an identifier with the same constraints as for a relative-path string (though without the quotes), with the additional constraint that it must not contain a `..`.

```
(require (lib string string ...))
```

Accesses a file in an installed library, making its definitions available in the current module (i.e., the current file). The first *string* names the library file, and the remaining *string*s name the collection (and sub-collection, and so on) where the file is installed. Each string is constrained in the same way as for the `(require string)` form.

```
(require (planet string (string string number number)))
(require (planet id))
(require (planet string))
```

Accesses a library that is distributed on the internet via the PLaneT server, making it definitions available in the current module (i.e., current file).

The full grammar for planet requires is given in §3.2 “Importing and Exporting: `require` and `provide`”, but the best place to find examples of the syntax is on the the PLaneT server, in the description of a specific package.

4.5 Pre-defined Functions

4.6 Signatures

Signatures do not have to be comment: They can also be part of the code. When a signature is attached to a function, DrRacket will check that program uses the function in accordance with the signature and display signature violations along with the test results.

A signature is a regular value, and is specified as a *signature form*, a special syntax that only works with `:` signature declarations and inside `signature` expressions.

```
(: name signature-form)
```

This attaches the signature specified by *signature-form* to the definition of *name*. There must be a definition of *name* somewhere in the program.

```
(: age Integer)
(define age 42)

(: area-of-square (Number -> Number))
(define (area-of-square len)
  (sqr len))
```

On running the program, Racket checks whether the signatures attached with : actually match the value of the variable. If they don't, Racket reports *signature violation* along with test failures.

For example, this piece of code:

```
(: age Integer)
(define age "fortytwo")
```

Yields this output:

```
1 signature violation.

Signature violations:
        got "fortytwo" at line 2, column 12, signature at line 1,
column 7
```

Note that a signature violation does not stop the running program.

```
(signature signature-form)
```

This returns the signature described by *signature-form* as a value.

4.6.1 Signature Forms

Any expression can be a signature form, in which case the signature is the value returned by that expression. There are a few special signature forms, however:

In a signature form, any name that starts with a % is a *signature variable* that stands for any signature depending on how the signature is used.

Example:

```
(: same (%a -> %a))

(define (same x) x)
```

`(input-signature-form ... -> output-signature-form)`

This signature form describes a function with inputs described by the *input-signature-form*s and output described by *output-signature-form*.

`(enum expr ...)`

This signature describes an enumeration of the values returned by the *expr*s.

Example:

```
(: cute? ((enum "cat" "snake") -> Boolean))

(define (cute? pet)
  (cond
    [(string=? pet "cat") #t]
    [(string=? pet "snake") #f]))
```

`(mixed signature-form ...)`

This signature describes mixed data, i.e. an itemization where each of the cases has a signature described by a *signature-form*.

Example:

```
(define SIGS (signature (mixed Aim Fired)))
```

`(ListOf signature-form)`

This signature describes a list where the elements are described by *signature-form*.

`(predicate expression)`

This signature describes values through a predicate: *expression* must evaluate to a function of one argument that returns a boolean. The signature matches all values for which the predicate returns `#true`.

4.6.2 Struct Signatures

A `define-struct` form defines two additional names that can be used in signatures. For a struct called `struct`, these are `Struct` and `StructOf`. Note that these names are capitalized. In particular, a struct called `Struct`, will also define `Struct` and `StructOf`. Moreover, when forming the additional names, hyphens are removed, and each letter following a hyphen is capitalized - so a struct called `foo-bar` will define `FooBar` and `FooBarOf`.

`Struct` is a signature that describes struct values from this structure type. `StructOf` is a function that takes as input a signature for each field. It returns a signature describing values of this structure type, additionally describing the values of the fields of the value.

```
(define-struct pair [fst snd])

(: add-pair ((PairOf Number Number) -> Number))
(define (add-pair p)
  (+ (pair-fst p) (pair-snd p)))
```

The remaining subsections list those functions that are built into the programming language. All other functions are imported from a teachpack or must be defined in the program.

4.7 Numbers: Integers, Rationals, Reals, Complex, Exacts, Inexacts

```
(- x y ...) → number
  x : number
  y : number
```

Subtracts the second (and following) number(s) from the first ; negates the number if there is only one argument.

```
> (- 5)
-5
> (- 5 3)
2
> (- 5 3 1)
1
```

```
(< x y z ...) → boolean?
  x : real
  y : real
  z : real
```

Compares two or more (real) numbers for less-than.

```
> (< 42 2/5)
#false
```

```
(<= x y z ...) → boolean?
  x : real
  y : real
  z : real
```

Compares two or more (real) numbers for less-than or equality.

```
> (<= 42 2/5)
#false
```

```
(> x y z ...) → boolean?
  x : real
  y : real
  z : real
```

Compares two or more (real) numbers for greater-than.

```
> (> 42 2/5)
#true
```

```
(>= x y z ...) → boolean?
  x : real
  y : real
  z : real
```

Compares two or more (real) numbers for greater-than or equality.

```
> (>= 42 42)
#true
```

```
(abs x) → real
  x : real
```

Determines the absolute value of a real number.

```
> (abs -12)
12
```

```
(acos x) → number
  x : number
```

Computes the arccosine (inverse of cos) of a number.

```
> (acos 0)
#i1.5707963267948966
```

```
(add1 x) → number
  x : number
```

Increments the given number.

```
> (add1 2)
3
```

```
(angle x) → real
  x : number
```

Extracts the angle from a complex number.

```
> (angle (make-polar 3 4))
#i-2.2831853071795867
```

```
(asin x) → number
  x : number
```

Computes the arcsine (inverse of sin) of a number.

```
> (asin 0)
0
```

```
(atan x) → number
  x : number
```

Computes the arctangent of the given number:

```
> (atan 0)
0
> (atan 0.5)
#i0.46364760900080615
```

Also comes in a two-argument version where `(atan y x)` computes `(atan (/ y x))` but the signs of `y` and `x` determine the quadrant of the result and the result tends to be more accurate than that of the 1-argument version in borderline cases:

```
> (atan 3 4)
#i0.6435011087932844
> (atan -2 -1)
#i-2.0344439357957027
```

```
(ceiling x) → integer
  x : real
```

Determines the closest integer (exact or inexact) above a real number. See `round`.

```
> (ceiling 12.3)
#i13.0
```

```
(complex? x) → boolean?
  x : any/c
```

Determines whether some value is complex.

```
> (complex? 1-2i)
#true
```

```
(conjugate x) → number
  x : number
```

Flips the sign of the imaginary part of a complex number.

```
> (conjugate 3+4i)
3-4i
> (conjugate -2-5i)
-2+5i
> (conjugate (make-polar 3 4))
#i-1.960930862590836+2.270407485923785i
```

```
(cos x) → number
  x : number
```

Computes the cosine of a number (radians).

```
> (cos pi)
#i-1.0
```

(cosh x) → number
  x : number

Computes the hyperbolic cosine of a number.

```
> (cosh 10)
#i11013.232920103324
```

(current-seconds) → integer

Determines the current time in seconds elapsed (since a platform-specific starting date).

```
> (current-seconds)
1793426334
```

(denominator x) → integer
  x : rational?

Computes the denominator of a rational.

```
> (denominator 2/3)
3
```

e : real

Euler's number.

```
> e
#i2.718281828459045
```

(even? x) → boolean?
  x : integer

Determines if some integer (exact or inexact) is even or not.

```
> (even? 2)
#true
```

```
(exact->inexact x) → number
  x : number
```

Converts an exact number to an inexact one.

```
> (exact->inexact 12)
#i12.0
```

```
(exact? x) → boolean?
  x : number
```

Determines whether some number is exact.

```
> (exact? (sqrt 2))
#false
```

```
(exp x) → number
  x : number
```

Determines e raised to a number.

```
> (exp -2)
#i0.1353352832366127
```

```
(expt x y) → number
  x : number
  y : number
```

Computes the power of the first to the second number, which is to say, exponentiation.

```
> (expt 16 1/2)
4
> (expt 3 -4)
1/81
```

```
(floor x) → integer
  x : real
```

Determines the closest integer (exact or inexact) below a real number. See `round`.

```
> (floor 12.3)
#i12.0
```

```
(gcd x y ...) → integer
  x : integer
  y : integer
```

Determines the greatest common divisor of two integers (exact or inexact).

```
> (gcd 6 12 8)
2
```

```
(imag-part x) → real
  x : number
```

Extracts the imaginary part from a complex number.

```
> (imag-part 3+4i)
4
```

```
(inexact->exact x) → number
  x : number
```

Approximates an inexact number by an exact one.

```
> (inexact->exact 12.0)
12
```

```
(inexact? x) → boolean?
  x : number
```

Determines whether some number is inexact.

```
> (inexact? 1-2i)
#false
```

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

Looks up the character that corresponds to the given exact integer in the ASCII table (if any).

```
> (integer->char 42)
#\*
```

```
(integer-sqrt x) → complex
  x : integer
```

Computes the integer or imaginary-integer square root of an integer.

```
> (integer-sqrt 11)
3
> (integer-sqrt -11)
0+3i
```

```
(integer? x) → boolean?
  x : any/c
```

Determines whether some value is an integer (exact or inexact).

```
> (integer? (sqrt 2))
#false
```

```
(lcm x y ...) → integer
  x : integer
  y : integer
```

Determines the least common multiple of two integers (exact or inexact).

```
> (lcm 6 12 8)
24
```

```
(log x) → number
  x : number
```

Determines the base-e logarithm of a number.

```
> (log 12)
#i2.4849066497880004
```

```
(magnitude x) → real
  x : number
```

Determines the magnitude of a complex number.

```
> (magnitude (make-polar 3 4))
#i3.0000000000000004
```

```
(make-polar x y) → number
  x : real
  y : real
```

Creates a complex from a magnitude and angle.

```
> (make-polar 3 4)
#i-1.960930862590836-2.270407485923785i
```

```
(make-rectangular x y) → number
  x : real
  y : real
```

Creates a complex from a real and an imaginary part.

```
> (make-rectangular 3 4)
3+4i
```

```
(max x y ...) → real
  x : real
  y : real
```

Determines the largest number—aka, the maximum.

```
> (max 3 2 8 7 2 9 0)
9
```

```
(min x y ...) → real
  x : real
  y : real
```

Determines the smallest number—aka, the minimum.

```
> (min 3 2 8 7 2 9 0)
0
```

```
(modulo x y) → integer
  x : integer
  y : integer
```

Finds the remainder of the division of the first number by the second:

```
> (modulo 9 2)
1
> (modulo 3 -4)
-1
```

```
(negative? x) → boolean?
  x : real
```

Determines if some real number is strictly smaller than zero.

```
> (negative? -2)
#true
```

```
(number->string x) → string
  x : number
```

Converts a number to a string.

```
> (number->string 42)
"42"
```

```
(number->string-digits x p) → string
  x : number
  p : posint
```

Converts a number x to a string with the specified number of digits.

```
> (number->string-digits 0.9 2)
"0.9"
> (number->string-digits pi 4)
"3.1416"
```

```
(number? n) → boolean?
  n : any/c
```

Determines whether some value is a number:

```
> (number? "hello world")
#false
> (number? 42)
#true
```

```
(numerator x) → integer
  x : rational?
```

Computes the numerator of a rational.

```
> (numerator 2/3)
2
```

```
(odd? x) → boolean?
  x : integer
```

Determines if some integer (exact or inexact) is odd or not.

```
> (odd? 2)
#false
```

```
pi : real
```

The ratio of a circle's circumference to its diameter.

```
> pi
#i3.141592653589793
```

```
(positive? x) → boolean?
  x : real
```

Determines if some real number is strictly larger than zero.

```
> (positive? -2)
#false
```

```
(quotient x y) → integer
  x : integer
  y : integer
```

Divides the first integer—also called dividend—by the second—known as divisor—to obtain the quotient.

```
> (quotient 9 2)
4
> (quotient 3 4)
0
```

```
(random x) → natural?
  x : (and/c natural? positive?)
```

Generates a random natural number less than some given exact natural.

```
> (random 42)
27
```

```
(rational? x) → boolean?
  x : any/c
```

Determines whether some value is a rational number.

```
> (rational? 1)
#true
> (rational? -2.349)
#true
> (rational? #i1.23456789)
#true
> (rational? (sqrt -1))
```

```
#false
> (rational? pi)
#true
> (rational? e)
#true
> (rational? 1-2i)
#false
```

As the interactions show, the teaching languages considers many more numbers as rationals than expected. In particular, `pi` is a rational number because it is only a finite approximation to the mathematical π. Think of `rational?` as a suggestion to think of these numbers as fractions.

```
(real-part x) → real
  x : number
```

Extracts the real part from a complex number.

```
> (real-part 3+4i)
3
```

```
(real? x) → boolean?
  x : any/c
```

Determines whether some value is a real number.

```
> (real? 1-2i)
#false
```

```
(remainder x y) → integer
  x : integer
  y : integer
```

Determines the remainder of dividing the first by the second integer (exact or inexact).

```
> (remainder 9 2)
1
> (remainder 3 4)
3
```

```
(round x) → integer
  x : real
```

Rounds a real number to an integer (rounds to even to break ties). See `floor` and `ceiling`.

```
> (round 12.3)
#i12.0
```

```
(sgn x) → (union 1 #i1.0 0 #i0.0 -1 #i-1.0)
  x : real
```

Determines the sign of a real number.

```
> (sgn -12)
-1
```

```
(sin x) → number
  x : number
```

Computes the sine of a number (radians).

```
> (sin pi)
#i1.2246467991473532e-16
```

```
(sinh x) → number
  x : number
```

Computes the hyperbolic sine of a number.

```
> (sinh 10)
#i11013.232874703393
```

```
(sqr x) → number
  x : number
```

Computes the square of a number.

```
> (sqr 8)
64
```

```
(sqrt x) → number
  x : number
```

Computes the square root of a number.

```
> (sqrt 9)
3
> (sqrt 2)
#i1.4142135623730951
```

```
(sub1 x) → number
  x : number
```

Decrements the given number.

```
> (sub1 2)
1
```

```
(tan x) → number
  x : number
```

Computes the tangent of a number (radians).

```
> (tan pi)
#i-1.2246467991473532e-16
```

```
(zero? x) → boolean?
  x : number
```

Determines if some number is zero or not.

```
> (zero? 2)
#false
```

4.8 Booleans

```
(boolean->string x) → string
  x : boolean?
```

Produces a string for the given boolean

```
> (boolean->string #false)
"#false"
> (boolean->string #true)
"#true"
```

```
(boolean=? x y) → boolean?
  x : boolean?
  y : boolean?
```

Determines whether two booleans are equal.

```
> (boolean=? #true #false)
#false
```

```
(boolean? x) → boolean?
  x : any/c
```

Determines whether some value is a boolean.

```
> (boolean? 42)
#false
> (boolean? #false)
#true
```

```
(false? x) → boolean?
  x : any/c
```

Determines whether a value is false.

```
> (false? #false)
#true
```

```
(not x) → boolean?
  x : boolean?
```

Negates a boolean value.

```
> (not #false)
#true
```

4.9 Symbols

```
(symbol->string x) → string
  x : symbol
```

Converts a symbol to a string.

```
> (symbol->string 'c)
"c"
```

```
(symbol=? x y) → boolean?
  x : symbol
  y : symbol
```

Determines whether two symbols are equal.

```
> (symbol=? 'a 'b)
#false
```

```
(symbol? x) → boolean?
  x : any/c
```

Determines whether some value is a symbol.

```
> (symbol? 'a)
#true
```

4.10 Lists

```
(append l ...) → (listof any)
  l : (listof any)
```

Creates a single list from several, by concatenation of the items. In ISL and up: `append` also works when applied to one list or none.

```
> (append (cons 1 (cons 2 '())) (cons "a" (cons "b" '())))
(list 1 2 "a" "b")
> (append)
'()
```

```
(assoc x l) → (union (listof any) #false)
  x : any/c
  l : (listof any)
```

Produces the first pair on `l` whose `first` is `equal?` to `x`; otherwise it produces `#false`.

```
> (assoc "hello" '(("world" 2) ("hello" 3) ("good" 0)))
(list "hello" 3)
```

```
(assq x l) → (union #false cons?)
  x : any/c
  l : list?
```

Determines whether some item is the first item of a pair in a list of pairs. (It compares the items with `eq?`.)

```
> a
(list (list 'a 22) (list 'b 8) (list 'c 70))
> (assq 'b a)
(list 'b 8)
```

```
(caaar x) → any/c
  x : list?
```

LISP-style selector: `(car (car (car x)))`.

```
> w
(list (list (list (list "bye") 3) #true) 42)
> (caaar w)
(list "bye")
```

```
(caadr x) → any/c
  x : list?
```

LISP-style selector: `(car (car (cdr x)))`.

```
> (caadr (cons 1 (cons (cons 'a '()) (cons (cons 'd '()) '()))))
'a
```

```
(caar x) → any/c
  x : list?
```

LISP-style selector: `(car (car x))`.

```
> y
(list (list (list 1 2 3) #false "world"))
> (caar y)
(list 1 2 3)
```

```
(cadar x) → any/c
  x : list?
```

LISP-style selector: `(car (cdr (car x)))`.

```
> w
(list (list (list (list "bye") 3) #true) 42)
> (cadar w)
#true
```

```
(cadddr x) → any/c
  x : list?
```

LISP-style selector: `(car (cdr (cdr (cdr x))))`.

```
> v
(list 1 2 3 4 5 6 7 8 9 'A)
> (cadddr v)
4
```

```
(caddr x) → any/c
  x : list?
```

LISP-style selector: `(car (cdr (cdr x)))`.

```
> x
(list 2 "hello" #true)
> (caddr x)
#true
```

```
(cadr x) → any/c
  x : list?
```

LISP-style selector: `(car (cdr x))`.

```
> x
(list 2 "hello" #true)
> (cadr x)
"hello"
```

```
(car x) → any/c
  x : cons?
```

Selects the first item of a non-empty list.

```
> x
(list 2 "hello" #true)
> (car x)
2
```

```
(cdaar x) → any/c
  x : list?
```

LISP-style selector: `(cdr (car (car x)))`.

```
> w
(list (list (list (list "bye") 3) #true) 42)
> (cdaar w)
(list 3)
```

```
(cdadr x) → any/c
  x : list?
```

LISP-style selector: `(cdr (car (cdr x)))`.

```
> (cdadr (list 1 (list 2 "a") 3))
(list "a")
```

```
(cdar x) → list?
  x : list?
```

LISP-style selector: `(cdr (car x))`.

```
> y
(list (list (list 1 2 3) #false "world"))
> (cdar y)
(list #false "world")
```

```
(cddar x) → any/c
  x : list?
```

LISP-style selector: `(cdr (cdr (car x)))`

```
> w
(list (list (list (list "bye") 3) #true) 42)
> (cddar w)
'()
```

```
(cdddr x) → any/c
  x : list?
```

LISP-style selector: `(cdr (cdr (cdr x)))`.

```
> v
(list 1 2 3 4 5 6 7 8 9 'A)
> (cdddr v)
(list 4 5 6 7 8 9 'A)
```

```
(cddr x) → list?
  x : list?
```

LISP-style selector: `(cdr (cdr x))`.

```
> x
(list 2 "hello" #true)
> (cddr x)
(list #true)
```

```
(cdr x) → any/c
  x : cons?
```

Selects the rest of a non-empty list.

```
> x
(list 2 "hello" #true)
> (cdr x)
(list "hello" #true)
```

```
(cons x y) → list?
  x : any/c
  y : list?
```

Constructs a list.

```
> (cons 1 '())
(cons 1 '())
```

```
(cons? x) → boolean?
  x : any/c
```

Determines whether some value is a constructed list.

```
> (cons? (cons 1 '()))
#true
> (cons? 42)
#false
```

```
(eighth x) → any/c
  x : list?
```

Selects the eighth item of a non-empty list.

```
> v
(list 1 2 3 4 5 6 7 8 9 'A)
> (eighth v)
8
```

```
(empty? x) → boolean?
  x : any/c
```

Determines whether some value is the empty list.

```
> (empty? '())
#true
> (empty? 42)
#false
```

```
(fifth x) → any/c
  x : list?
```

Selects the fifth item of a non-empty list.

```
> v
(list 1 2 3 4 5 6 7 8 9 'A)
> (fifth v)
5
```

```
(first x) → any/c
  x : cons?
```

Selects the first item of a non-empty list.

```
> x
(list 2 "hello" #true)
> (first x)
2
```

```
(fourth x) → any/c
  x : list?
```

Selects the fourth item of a non-empty list.

```
> v
(list 1 2 3 4 5 6 7 8 9 'A)
> (fourth v)
4
```

```
(length l) → natural?
  l : list?
```

Evaluates the number of items on a list.

```
> x
(list 2 "hello" #true)
> (length x)
3
```

```
(list x ...) → list?
  x : any/c
```

Constructs a list of its arguments.

```
> (list 1 2 3 4 5 6 7 8 9 0)
(cons 1 (cons 2 (cons 3 (cons 4 (cons 5 (cons 6 (cons 7 (cons 8
(cons 9 (cons 0 '()))))))))))
```

```
(list* x ... l) → list?
  x : any/c
  l : list?
```

Constructs a list by adding multiple items to a list.

```
> x
(list 2 "hello" #true)
> (list* 4 3 x)
(list 4 3 2 "hello" #true)
```

```
(list-ref x i) → any/c
  x : list?
  i : natural?
```

Extracts the indexed item from the list.

```
> v
(list 1 2 3 4 5 6 7 8 9 'A)
> (list-ref v 9)
'A
```

```
(list? x) → boolean?
  x : any/c
```

Checks whether the given value is a list.

```
> (list? 42)
#false
> (list? '())
#true
> (list? (cons 1 (cons 2 '())))
#true
```

```
(make-list i x) → list?
  i : natural?
  x : any/c
```

Constructs a list of `i` copies of `x`.

```
> (make-list 3 "hello")
(cons "hello" (cons "hello" (cons "hello" '())))
```

```
(member x l) → boolean?
  x : any/c
  l : list?
```

Determines whether some value is on the list (comparing values with equal?).

```
> x
(list 2 "hello" #true)
> (member "hello" x)
#true
```

```
(member? x l) → boolean?
  x : any/c
  l : list?
```

Determines whether some value is on the list (comparing values with equal?).

```
> x
(list 2 "hello" #true)
> (member? "hello" x)
#true
```

```
(memq x l) → boolean?
  x : any/c
  l : list?
```

Determines whether some value *x* is on some list *l*, using `eq?` to compare *x* with items on *l*.

```
> x
(list 2 "hello" #true)
> (memq (list (list 1 2 3)) x)
#false
```

```
(memq? x l) → boolean?
  x : any/c
  l : list?
```

Determines whether some value *x* is on some list *l*, using `eq?` to compare *x* with items on *l*.

```
> x
(list 2 "hello" #true)
> (memq? (list (list 1 2 3)) x)
#false
```

```
(memv x l) → (or/c #false list)
  x : any/c
  l : list?
```

Determines whether some value is on the list if so, it produces the suffix of the list that starts with x if not, it produces false. (It compares values with the eqv? predicate.)

```
> x
(list 2 "hello" #true)
> (memv (list (list 1 2 3)) x)
#false
```

```
null : list
```

Another name for the empty list

```
> null
'()
```

```
(null? x) → boolean?
  x : any/c
```

Determines whether some value is the empty list.

```
> (null? '())
#true
> (null? 42)
#false
```

```
(range start end step) → list?
  start : number
  end : number
  step : number
```

Constructs a list of numbers by *step*ping from *start* to *end*.

```
> (range 0 10 2)
(cons 0 (cons 2 (cons 4 (cons 6 (cons 8 '())))))
```

```
(remove x l) → list?
  x : any/c
  l : list?
```

Constructs a list like the given one, with the first occurrence of the given item removed (comparing values with equal?).

```
> x
(list 2 "hello" #true)
> (remove "hello" x)
(list 2 #true)
> hello-2
(list 2 "hello" #true "hello")
> (remove "hello" hello-2)
(list 2 #true "hello")
```

```
(remove-all x l) → list?
  x : any/c
  l : list?
```

Constructs a list like the given one, with all occurrences of the given item removed (comparing values with equal?).

```
> x
```

```
(list 2 "hello" #true)
> (remove-all "hello" x)
(list 2 #true)
> hello-2
(list 2 "hello" #true "hello")
> (remove-all "hello" hello-2)
(list 2 #true)
```

(rest x) → any/c
  x : cons?

Selects the rest of a non-empty list.

```
> x
(list 2 "hello" #true)
> (rest x)
(list "hello" #true)
```

(reverse l) → list
  l : list?

Creates a reversed version of a list.

```
> x
(list 2 "hello" #true)
> (reverse x)
(list #true "hello" 2)
```

(second x) → any/c
  x : list?

Selects the second item of a non-empty list.

```
> x
(list 2 "hello" #true)
> (second x)
"hello"
```

(seventh x) → any/c
  x : list?

Selects the seventh item of a non-empty list.

```
> v
(list 1 2 3 4 5 6 7 8 9 'A)
> (seventh v)
7
```

```
(sixth x) → any/c
  x : list?
```

Selects the sixth item of a non-empty list.

```
> v
(list 1 2 3 4 5 6 7 8 9 'A)
> (sixth v)
6
```

```
(third x) → any/c
  x : list?
```

Selects the third item of a non-empty list.

```
> x
(list 2 "hello" #true)
> (third x)
#true
```

4.11 Posns

```
(make-posn x y) → posn
  x : any/c
  y : any/c
```

Constructs a posn from two arbitrary values.

```
> (make-posn 3 3)
(make-posn 3 3)
> (make-posn "hello" #true)
(make-posn "hello" #true)
```

```
(posn-x p) → any/c
  p : posn
```

Extracts the x component of a posn.

```
> p
(make-posn 2 -3)
> (posn-x p)
2
```

```
(posn-y p) → any/c
  p : posn
```

Extracts the y component of a posn.

```
> p
(make-posn 2 -3)
> (posn-y p)
-3
```

```
(posn? x) → boolean?
  x : any/c
```

Determines if its input is a posn.

```
> q
(make-posn "bye" 2)
> (posn? q)
#true
> (posn? 42)
#false
```

4.12 Characters

```
(char->integer c) → integer
  c : char
```

Looks up the number that corresponds to the given character in the ASCII table (if any).

```
> (char->integer #\a)
97
> (char->integer #\z)
122
```

```
(char-alphabetic? c) → boolean?
  c : char
```

Determines whether a character represents an alphabetic character.

```
> (char-alphabetic? #\Q)
#true
```

```
(char-ci<=? c d e ...) → boolean?
  c : char
  d : char
  e : char
```

Determines whether the characters are ordered in an increasing and case-insensitive manner.

```
> (char-ci<=? #\b #\B)
#true
> (char<=? #\b #\B)
#false
```

```
(char-ci<? c d e ...) → boolean?
  c : char
  d : char
  e : char
```

Determines whether the characters are ordered in a strictly increasing and case-insensitive manner.

```
> (char-ci<? #\B #\c)
#true
> (char<? #\b #\B)
#false
```

```
(char-ci=? c d e ...) → boolean?
  c : char
  d : char
  e : char
```

Determines whether two characters are equal in a case-insensitive manner.

```
> (char-ci=? #\b #\B)
#true
```

```
(char-ci>=? c d e ...) → boolean?
  c : char
  d : char
  e : char
```

Determines whether the characters are sorted in a decreasing and case-insensitive manner.

```
> (char-ci>=? #\b #\C)
#false
> (char>=? #\b #\C)
#true
```

```
(char-ci>? c d e ...) → boolean?
  c : char
  d : char
  e : char
```

Determines whether the characters are sorted in a strictly decreasing and case-insensitive manner.

```
> (char-ci>? #\b #\B)
#false
> (char>? #\b #\B)
#true
```

```
(char-downcase c) → char
  c : char
```

Produces the equivalent lower-case character.

```
> (char-downcase #\T)
#\t
```

```
(char-lower-case? c) → boolean?
  c : char
```

Determines whether a character is a lower-case character.

```
> (char-lower-case? #\T)
#false
```

```
(char-numeric? c) → boolean?
  c : char
```

Determines whether a character represents a digit.

```
> (char-numeric? #\9)
#true
```

```
(char-upcase c) → char
  c : char
```

Produces the equivalent upper-case character.

```
> (char-upcase #\t)
#\T
```

```
(char-upper-case? c) → boolean?
  c : char
```

Determines whether a character is an upper-case character.

```
> (char-upper-case? #\T)
#true
```

```
(char-whitespace? c) → boolean?
  c : char
```

Determines whether a character represents space.

```
> (char-whitespace? #\tab)
#true
```

```
(char<=? c d e ...) → boolean?
  c : char
  d : char
  e : char
```

Determines whether the characters are ordered in an increasing manner.

```
> (char<=? #\a #\a #\b)
#true
```

```
(char<? x d e ...) → boolean?
  x : char
  d : char
  e : char
```

Determines whether the characters are ordered in a strictly increasing manner.

```
> (char<? #\a #\b #\c)
#true
```

```
(char=? c d e ...) → boolean?
  c : char
  d : char
  e : char
```

Determines whether the characters are equal.

```
> (char=? #\b #\a)
#false
```

```
(char>=? c d e ...) → boolean?
  c : char
  d : char
  e : char
```

Determines whether the characters are sorted in a decreasing manner.

```
> (char>=? #\b #\b #\a)
#true
```

```
(char>? c d e ...) → boolean?
  c : char
  d : char
  e : char
```

Determines whether the characters are sorted in a strictly decreasing manner.

```
> (char>? #\A #\z #\a)
#false
```

```
(char? x) → boolean?
  x : any/c
```

Determines whether a value is a character.

```
> (char? "a")
#false
> (char? #\a)
#true
```

4.13 Strings

```
(explode s) → (listof string)
  s : string
```

Translates a string into a list of 1-letter strings.

```
> (explode "cat")
(list "c" "a" "t")
```

```
(format f x ...) → string
  f : string
  x : any/c
```

Formats a string, possibly embedding values.

```
> (format "Dear Dr. ~a:" "Flatt")
"Dear Dr. Flatt:"
> (format "Dear Dr. ~s:" "Flatt")
"Dear Dr. \"Flatt\":"
```

```
(implode l) → string
  l : list?
```

Concatenates the list of 1-letter strings into one string.

```
> (implode (cons "c" (cons "a" (cons "t" '()))))
"cat"
```

```
(int->string i) → string
  i : integer
```

Converts an integer in [0,55295] or [57344 1114111] to a 1-letter string.

```
> (int->string 65)
"A"
```

```
(list->string l) → string
  l : list?
```

Converts a s list of characters into a string.

```
> (list->string (cons #\c (cons #\a (cons #\t '()))))
"cat"
```

```
(make-string i c) → string
  i : natural?
  c : char
```

Produces a string of length *i* from *c*.

```
> (make-string 3 #\d)
"ddd"
```

```
(replicate i s) → string
  i : natural?
  s : string
```

Replicates *s* *i* times.

```
> (replicate 3 "h")
"hhh"
```

```
(string c ...) → string?
  c : char
```

Builds a string of the given characters.

```
> (string #\d #\o #\g)
"dog"
```

```
(string->int s) → integer
  s : string
```

Converts a 1-letter string to an integer in [0,55295] or [57344, 1114111].

```
> (string->int "a")
97
```

```
(string->list s) → (listof char)
  s : string
```

Converts a string into a list of characters.

```
> (string->list "hello")
(list #\h #\e #\l #\l #\o)
```

```
(string->number s) → (union number #false)
  s : string
```

Converts a string into a number, produce false if impossible.

```
> (string->number "-2.03")
-2.03
> (string->number "1-2i")
1-2i
```

```
(string->symbol s) → symbol
  s : string
```

Converts a string into a symbol.

```
> (string->symbol "hello")
'hello
```

```
(string-alphabetic? s) → boolean?
  s : string
```

Determines whether all 'letters' in the string are alphabetic.

```
> (string-alphabetic? "123")
#false
> (string-alphabetic? "cat")
#true
```

```
(string-contains-ci? s t) → boolean?
  s : string
  t : string
```

Determines whether the first string appears in the second one without regard to the case of the letters.

```
> (string-contains-ci? "At" "caT")
#true
```

```
(string-contains? s t) → boolean?
  s : string
  t : string
```

Determines whether the first string appears literally in the second one.

```
> (string-contains? "at" "cat")
#true
```

```
(string-copy s) → string
  s : string
```

Copies a string.

```
> (string-copy "hello")
"hello"
```

```
(string-downcase s) → string
  s : string
```

Produces a string like the given one with all 'letters' as lower case.

```
> (string-downcase "CAT")
"cat"
> (string-downcase "cAt")
"cat"
```

```
(string-ith s i) → 1string?
  s : string
  i : natural?
```

Extracts the ith 1-letter substring from s.

```
> (string-ith "hello world" 1)
"e"
```

```
(string-length s) → nat
  s : string
```

Determines the length of a string.

```
> (string-length "hello world")
11
```

```
(string-lower-case? s) → boolean?
  s : string
```

Determines whether all 'letters' in the string are lower case.

```
> (string-lower-case? "CAT")
#false
```

```
(string-numeric? s) → boolean?
  s : string
```

Determines whether all 'letters' in the string are numeric.

```
> (string-numeric? "123")
#true
> (string-numeric? "1-2i")
#false
```

```
(string-ref s i) → char
  s : string
  i : natural?
```

Extracts the *i*th character from *s*.

```
> (string-ref "cat" 2)
#\t
```

```
(string-upcase s) → string
  s : string
```

Produces a string like the given one with all 'letters' as upper case.

```
> (string-upcase "cat")
"CAT"
> (string-upcase "cAt")
"CAT"
```

```
(string-upper-case? s) → boolean?
  s : string
```

Determines whether all 'letters' in the string are upper case.

```
> (string-upper-case? "CAT")
#true
```

```
(string-whitespace? s) → boolean?
  s : string
```

Determines whether all 'letters' in the string are white space.

```
> (string-whitespace? (string-append " " (string #\tab #\newline #\return)))
#true
```

```
(string? x) → boolean?
  x : any/c
```

Determines whether a value is a string.

```
> (string? "hello world")
#true
> (string? 42)
#false
```

```
(substring s i j) → string
  s : string
  i : natural?
  j : natural?
```

Extracts the substring starting at *i* up to *j* (or the end if *j* is not provided).

```
> (substring "hello world" 1 5)
"ello"
> (substring "hello world" 1 8)
"ello wo"
> (substring "hello world" 4)
"o world"
```

4.14 Images

```
(image=? i j) → boolean?
  i : image
  j : image
```

Determines whether two images are equal.

```
> c1
```

```
> (image=? (circle 5 "solid" "green") c1)
#false
> (image=? (circle 10 "solid" "green") c1)
#true
```

```
(image? x) → boolean?
  x : any/c
```

Determines whether a value is an image.

```
> c1
```

```
> (image? c1)
#true
```

4.15 Misc

```
(=~ x y eps) → boolean?
  x : number
  y : number
  eps : non-negative-real
```

Checks whether *x* and *y* are within *eps* of either other.

```
> (=~ 1.01 1.0 0.1)
#true
> (=~ 1.01 1.5 0.1)
#false
```

```
eof : eof-object?
```

A value that represents the end of a file:

```
> eof
#<eof>
```

```
(eof-object? x) → boolean?
  x : any/c
```

Determines whether some value is the end-of-file value.

```
> (eof-object? eof)
#true
> (eof-object? 42)
#false
```

```
(eq? x y) → boolean?
  x : any/c
  y : any/c
```

Determines whether two values are equivalent from the computer's perspective (intensional).

```
> (eq? (cons 1 '()) (cons 1 '()))
#false
> one
(list 1)
> (eq? one one)
#true
```

```
(equal? x y) → boolean?
  x : any/c
  y : any/c
```

Determines whether two values are structurally equal where basic values are compared with the eqv? predicate.

```
> (equal? (make-posn 1 2) (make-posn (- 2 1) (+ 1 1)))
#true
```

```
(equal~? x y z) → boolean?
  x : any/c
  y : any/c
  z : non-negative-real
```

Compares *x* and *y* like `equal?` but uses =~ in the case of numbers.

```
> (equal~? (make-posn 1.01 1.0) (make-posn 1.01 0.99) 0.2)
#true
```

```
(eqv? x y) → boolean?
  x : any/c
  y : any/c
```

Determines whether two values are equivalent from the perspective of all functions that can be applied to it (extensional).

```
> (eqv? (cons 1 '()) (cons 1 '()))
#false
> one
(list 1)
> (eqv? one one)
#true
```

```
(error x ...) → void?
  x : any/c
```

Signals an error, combining the given values into an error message. If any of the values' printed representations is too long, it is truncated and "..." is put into the string. If the first value is a symbol, it is suffixed with a colon and the result pre-pended on to the error message.

```
> zero
0
> (if (= zero 0) (error "can't divide by 0") (/ 1 zero))
can't divide by 0
```

```
(exit) → void
```

Evaluating `(exit)` terminates the running program.

```
(identity x) → any/c
  x : any/c
```

Returns *x*.

```
> (identity 42)
42
> (identity c1)
```

```
> (identity "hello")
"hello"
```

```
(struct? x) → boolean?
  x : any/c
```

Determines whether some value is a structure.

```
> (struct? (make-posn 1 2))
#true
> (struct? 43)
#false
```

4.16 Signatures

```
Any : signature?
```

Signature for any value.

```
Boolean : signature?
```

Signature for booleans.

```
Char : signature?
```

Signature for chararacters.

```
(ConsOf first-sig rest-sig) → signature?
  first-sig : signature?
  rest-sig : signature?
```

Signature for a cons pair.

```
EmptyList : signature?
```

Signature for the empty list.

```
False : signature?
```

Signature for just false.

```
Integer : signature?
```

Signature for integers.

```
Natural : signature?
```

Signature for natural numbers.

```
Number : signature?
```

Signature for arbitrary numbers.

```
Rational : signature?
```

Signature for rational numbers.

```
Real : signature?
```

Signature for real numbers.

```
String : signature?
```

Signature for strings.

```
Symbol : signature?
```

Signature for symbols.

```
True : signature?
```

Signature for just true.

4.17 Numbers (relaxed conditions)

4.18 String (relaxed conditions)

```
(string-append s ...) → string
  s : string
```

Concatenates the characters of several strings.

```
> (string-append "hello" " " "world" " " "good bye")
"hello world good bye"
```

```
(string-ci<=? s t x ...) → boolean?
  s : string
  t : string
  x : string
```

Determines whether the strings are ordered in a lexicographically increasing and case-insensitive manner.

```
> (string-ci<=? "hello" "WORLD" "zoo")
#true
```

```
(string-ci<? s t x ...) → boolean?
  s : string
  t : string
  x : string
```

Determines whether the strings are ordered in a lexicographically strictly increasing and case-insensitive manner.

```
> (string-ci<? "hello" "WORLD" "zoo")
#true
```

```
(string-ci=? s t x ...) → boolean?
  s : string
  t : string
  x : string
```

Determines whether all strings are equal, character for character, regardless of case.

```
> (string-ci=?  "hello" "HellO")
#true
```

```
(string-ci>=? s t x ...) → boolean?
  s : string
  t : string
  x : string
```

Determines whether the strings are ordered in a lexicographically decreasing and case-insensitive manner.

```
> (string-ci>?  "zoo" "WORLD" "hello")
#true
```

```
(string-ci>? s t x ...) → boolean?
  s : string
  t : string
  x : string
```

Determines whether the strings are ordered in a lexicographically strictly decreasing and case-insensitive manner.

```
> (string-ci>?  "zoo" "WORLD" "hello")
#true
```

```
(string<=? s t x ...) → boolean?
  s : string
  t : string
  x : string
```

Determines whether the strings are ordered in a lexicographically increasing manner.

```
> (string<=? "hello" "hello" "world" "zoo")
#true
```

```
(string<? s t x ...) → boolean?
  s : string
  t : string
  x : string
```

Determines whether the strings are ordered in a lexicographically strictly increasing manner.

```
> (string<? "hello" "world" "zoo")
#true
```

```
(string=? s t x ...) → boolean?
  s : string
  t : string
  x : string
```

Determines whether all strings are equal, character for character.

```
> (string=? "hello" "world")
#false
> (string=? "bye" "bye")
#true
```

```
(string>=? s t x ...) → boolean?
  s : string
  t : string
  x : string
```

Determines whether the strings are ordered in a lexicographically decreasing manner.

```
> (string>=?  "zoo" "zoo" "world" "hello")
#true
```

```
(string>? s t x ...) → boolean?
  s : string
  t : string
  x : string
```

Determines whether the strings are ordered in a lexicographically strictly decreasing manner.

```
> (string>?  "zoo" "world" "hello")
#true
```

4.19 Posn

```
Posn : signature
```

Signature for posns.

```
(PosnOf x-sig y-sig) → signature
  x-sig : signature
  y-sig : signature
```

Creates a parametric signature for posns from signatures for its fields.

```
posn : struct
```

Name for using `match` with posns.

4.20 Higher-Order Functions

4.21 Numbers (relaxed conditions plus)

```
(* x ...) → number
  x : number
```

Multiplies all given numbers. In ISL and up: `*` works when applied to only one number or none.

```
> (* 5 3)
15
> (* 5 3 2)
30
> (* 2)
2
> (*)
1
```

```
(+ x ...) → number
  x : number
```

Adds all given numbers. In ISL and up: + works when applied to only one number or none.

```
> (+ 2/3 1/16)
35/48
> (+ 3 2 5 8)
18
> (+ 1)
1
> (+)
0
```

```
(/ x y ...) → number
  x : number
  y : number
```

Divides the first by all remaining numbers. In ISL and up: / computes the inverse when applied to one number.

```
> (/ 12 2)
6
> (/ 12 2 3)
2
> (/ 3)
1/3
```

```
(= x ...) → number
  x : number
```

Compares numbers for equality. In ISL and up: = works when applied to only one number.

```
> (= 10 10)
#true
> (= 11)
#true
> (= 0)
#true
```

4.22 Higher-Order Functions (with Lambda)

```
(andmap p? [l]) → boolean
  p? : (X ... -> boolean)
  l : (listof X) = ...
```

Determines whether *p?* holds for all items of *l* ...:

```
(andmap p (list x-1 ... x-n)) = (and (p x-1) ... (p x-n))
```

```
(andmap p (list x-1 ... x-n) (list y-1 ... y-n)) = (and (p x-1 y-
1) ... (p x-n y-n))
```

```
> (andmap odd? '(1 3 5 7 9))
#true
> threshold
3
> (andmap (lambda (x) (< x threshold)) '(0 1 2))
#true
> (andmap even? '())
#true
> (andmap (lambda (x f) (f x)) (list 0 1 2) (list odd? even? positive?))
#false
```

```
(apply f x-1 ... l) → Y
  f : (X-1 ... X-N -> Y)
  x-1 : X-1
  l : (list X-i+1 ... X-N)
```

Applies a function using items from a list as the arguments:

```
(apply f (list x-1 ... x-n)) = (f x-1 ... x-n)
```

```
> a-list
(list 0 1 2 3 4 5 6 7 8 9)
> (apply max a-list)
9
```

```
(argmax f l) → X
  f : (X -> real)
  l : (listof X)
```

Finds the (first) element of the list that maximizes the output of the function.

```
> (argmax second '((sam 98) (carl 78) (vincent 93) (asumu 99)))
(list 'asumu 99)
```

```
(argmin f l) → X
  f : (X -> real)
  l : (listof X)
```

Finds the (first) element of the list that minimizes the output of the function.

```
> (argmin second '((sam 98) (carl 78) (vincent 93) (asumu 99)))
(list 'carl 78)
```

```
(build-list n f) → (listof X)
  n : nat
  f : (nat -> X)
```

Constructs a list by applying *f* to the numbers between 0 and (- *n* 1):

```
(build-list n f) = (list (f 0) ... (f (- n 1)))
```

```
> (build-list 22 add1)
(list 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22)
> i
3
> (build-list 3 (lambda (j) (+ j i)))
(list 3 4 5)
> (build-list 5
              (lambda (i)
                (build-list 5
                            (lambda (j)
                              (if (= i j) 1 0)))))
```

```
(list (list 1 0 0 0 0) (list 0 1 0 0 0) (list 0 0 1 0 0) (list 0 0
0 1 0) (list 0 0 0 0 1))
```

```
(build-string n f) → string
  n : nat
  f : (nat -> char)
```

Constructs a string by applying *f* to the numbers between 0 and (- *n* 1):

```
(build-string n f) = (string (f 0) ... (f (- n 1)))
```

```
> (build-string 10 integer->char)
"\u0000\u0001\u0002\u0003\u0004\u0005\u0006\a\b\t"
> (build-string 26 (lambda (x) (integer->char (+ 65 x))))
"ABCDEFGHIJKLMNOPQRSTUVWXYZ"
```

```
(compose f g) → (X -> Z)
  f : (Y -> Z)
  g : (X -> Y)
```

Composes a sequence of procedures into a single procedure:

```
(compose f g) = (lambda (x) (f (g x)))
```

```
> ((compose add1 second) '(add 3))
4
> (map (compose add1 second) '((add 3) (sub 2) (mul 4)))
(list 4 3 5)
```

```
(filter p? l) → (listof X)
  p? : (X -> boolean)
  l : (listof X)
```

Constructs a list from all those items on a list for which the predicate holds.

```
> (filter odd? '(0 1 2 3 4 5 6 7 8 9))
(list 1 3 5 7 9)
> threshold
3
> (filter (lambda (x) (>= x threshold)) '(0 1 2 3 4 5 6 7 8 9))
(list 3 4 5 6 7 8 9)
```

```
(foldl f base l ...) → Y
  f : (X ... Y -> Y)
  base : Y
  l : (listof X)
```

```
(foldl f base (list x-1 ... x-n)) = (f x-n ... (f x-1 base))
```

```
(foldl f base (list x-1 ... x-n) (list x-1 ... x-n))
= (f x-n y-n ... (f x-1 y-1 base))
```

```
> (foldl + 0 '(0 1 2 3 4 5 6 7 8 9))
45
> a-list
(list 0 1 2 3 4 5 6 7 8 9)
> (foldl (lambda (x r) (if (> x threshold) (cons (* 2 x) r) r)) '() a-
list)
(list 18 16 14 12 10 8)
> (foldl (lambda (x y r) (+ x y r)) 0 '(1 2 3) '(10 11 12))
39
```

```
(foldr f base l ...) → Y
  f : (X ... Y -> Y)
  base : Y
  l : (listof X)
```

```
(foldr f base (list x-1 ... x-n)) = (f x-1 ... (f x-n base))
```

```
(foldr f base (list x-1 ... x-n) (list y-1 ... y-n))
= (f x-1 y-1 ... (f x-n y-n base))
```

```
> (foldr + 0 '(0 1 2 3 4 5 6 7 8 9))
45
> a-list
(list 0 1 2 3 4 5 6 7 8 9)
> (foldr (lambda (x r) (if (> x threshold) (cons (* 2 x) r) r)) '() a-
list)
(list 8 10 12 14 16 18)
> (foldr (lambda (x y r) (+ x y r)) 0 '(1 2 3) '(10 11 12))
39
```

```
(for-each f l ...) → void?
  f : (any ... -> any)
  l : (listof any)
```

Applies a function to each item on one or more lists for effect only.

Although the *Intermediate Student with Lambda* provides the `for-each` function, it is intended to be used in the §5 "Advanced Student" level.

```
(map f l ...) → (listof Z)
  f : (X ... -> Z)
  l : (listof X)
```

Constructs a new list by applying a function to each item on one or more existing lists:

```
(map f (list x-1 ... x-n)) = (list (f x-1) ... (f x-n))
```

```
(map f (list x-1 ... x-n) (list y-1 ... y-n)) = (list (f x-1 y-
1) ... (f x-n y-n))
```

```
> (map add1 (list 3 -4.01 2/5))
(list 4 #i-3.01 1.4)
```

```
> (define (tag-with-a x)
    (list "a" (+ x 1)))
```

```
> (map tag-with-a (list 3 -4.01 2/5))
(list (list "a" 4) (list "a" #i-3.01) (list "a" 1.4))
```

```
> (define (add-and-multiply x y)
    (+ x (* x y)))
```

```
> (map add-and-multiply (list 3 -4 2/5) '(1 2 3))
(list 6 -12 1.6)
```

```
(memf p? l) → (union #false (listof X))
  p? : (X -> any)
  l : (listof X)
```

Produces `#false` if *p?* produces `false` for all items on *l*. If *p?* produces `#true` for any of the items on *l*, `memf` returns the sub-list starting from that item.

```
> (memf odd? '(2 4 6 3 8 0))
(list 3 8 0)
```

```
(ormap p? l) → boolean
  p? : (X -> boolean)
  l : (listof X)
```

Determines whether *p?* holds for at least one items of *l*:

```
(ormap p (list x-1 ... x-n)) = (or (p x-1) ... (p x-n))
```

```
(ormap p (list x-1 ... x-n) (list y-1 ... y-n)) = (or (p x-1 y-
1) ... (p x-n y-n))
```

```
> (ormap odd? '(1 3 5 7 9))
#true
> threshold
3
> (ormap (lambda (x) (< x threshold)) '(6 7 8 1 5))
#true
> (ormap even? '())
#false
> (ormap (lambda (x f) (f x)) (list 0 1 2) (list odd? even? positive?))
#true
```

```
(procedure? x) → boolean?
  x : any
```

Produces true if the value is a procedure.

```
> (procedure? cons)
#true
> (procedure? add1)
#true
> (procedure? (lambda (x) (> x 22)))
#true
```

```
(quicksort l comp) → (listof X)
  l : (listof X)
  comp : (X X -> boolean)
```

Sorts the items on *l*, in an order according to *comp* (using the quicksort algorithm).

```
> (quicksort '(6 7 2 1 3 4 0 5 9 8) <)
(list 0 1 2 3 4 5 6 7 8 9)
```

```
(sort l comp) → (listof X)
  l : (listof X)
  comp : (X X -> boolean)
```

Sorts the items on *l*, in an order according to *comp*.

```
> (sort '(6 7 2 1 3 4 0 5 9 8) <)
(list 0 1 2 3 4 5 6 7 8 9)
```

5 Advanced Student

The grammar notation uses the notation `X ...` (bold dots) to indicate that `X` may occur an arbitrary number of times (zero, one, or more). Separately, the grammar also defines `...` as an identifier to be used in templates.

```
    program = def-or-expr ...

def-or-expr = definition
            | expr
            | test-case
            | library-require
            | signature-declaration

 definition = (define (name variable ...) expr)
            | (define name expr)
            | (define-struct name (name ...))
            | (define-datatype name (name name ...) ...)

       expr = (begin expr expr ...)
            | (begin0 expr expr ...)
            | (set! variable expr)
            | (delay expr)
            | (lambda (variable ...) expr)
            | (λ (variable ...) expr)
            | (local [definition ...] expr)
            | (letrec ([name expr] ...) expr)
            | (shared ([name expr] ...) expr)
            | (let ([name expr] ...) expr)
            | (let name ([name expr] ...) expr)
            | (let* ([name expr] ...) expr)
            | (recur name ([name expr] ...) expr)
            | (expr expr ...)
            | (cond [expr expr] ... [expr expr])
            | (cond [expr expr] ... [else expr])
            | (case expr [(choice choice ...) expr] ...
                         [(choice choice ...) expr])
            | (case expr [(choice choice ...) expr] ...
                         [else expr])
            | (match expr [pattern expr] ...)
            | (if expr expr expr)
            | (when expr expr)
            | (unless expr expr)
            | (and expr expr expr ...)
            | (or expr expr expr ...)
```

```
                    | (time expr)
                    | name
                    | 'quoted
                    | `quasiquoted
                    | '()
                    | number
                    | boolean
                    | string
                    | character
                    | (signature signature-form)

              choice = name
                     | number

             pattern = _
                     | name
                     | number
                     | true
                     | false
                     | string
                     | character
                     | 'quoted
                     | `quasiquoted-pattern
                     | (cons pattern pattern)
                     | (list pattern ...)
                     | (list* pattern ...)
                     | (struct id (pattern ...))
                     | (vector pattern ...)
                     | (box pattern)

quasiquoted-pattern = name
                     | number
                     | string
                     | character
                     | (quasiquoted-pattern ...)
                     | 'quasiquoted-pattern
                     | `quasiquoted-pattern
                     | ,pattern
                     | ,@pattern

signature-declaration = (: name signature-form)

      signature-form = (enum expr ...)
                     | (mixed signature-form ...)
                     | (signature-form ... -> signature-form)
                     | (ListOf signature-form)
```

```
                    | signature-variable
                    | expr

signature-variable = %name

          quoted = name
                 | number
                 | string
                 | character
                 | (quoted ...)
                 | 'quoted
                 | `quoted
                 | ,quoted
                 | ,@quoted

      quasiquoted = name
                  | number
                  | string
                  | character
                  | (quasiquoted ...)
                  | 'quasiquoted
                  | `quasiquoted
                  | ,expr
                  | ,@expr

        test-case = (check-expect expr expr)
                  | (check-random expr expr)
                  | (check-within expr expr expr)
                  | (check-random-within expr expr expr)
                  | (check-error expr expr ...)
                  | (check-member-of expr expr expr)
                  | (check-satisfied expr expr)
                  | (check-range expr expr)
                  | (check-range expr)

  library-require = (require string)
                  | (require (lib string string ...))
                  | (require (planet string package))

          package = (string string number number)
```

A *name* or a *variable* is a sequence of characters not including a space or one of the following:

```
" , ' ` ( ) [ ] { } | ; #
```

A *number* is a number such as 123, 3/2, or 5.5.

A *boolean* is one of: #true or #false.

Alternative spellings for the #true constant are #t, true, and #T. Similarly, #f, false, or #F are also recognized as #false.

A *symbol* is a quote character followed by a name. A symbol is a value, just like 42, '(), or #false.

A *string* is a sequence of characters enclosed by a pair of ". Unlike symbols, strings may be split into characters and manipulated by a variety of functions. For example, "abcdef", "This is a string", and "This is a string with \" inside" are all strings.

A *character* begins with #\ and has the name of the character. For example, #\a, #\b, and #\space are characters.

In function calls, the function appearing immediately after the open parenthesis can be any functions defined with define or define-struct, or any one of the pre-defined functions.

5.1 Pre-defined Variables

empty : empty?

The empty list.

true : boolean?

The #true value.

false : boolean?

The #false value.

5.2 Template Variables

. .

A placeholder for indicating that a function definition is a template.

. . .

A placeholder for indicating that a function definition is a template.

```
....
```

A placeholder for indicating that a function definition is a template.

```
.....
```

A placeholder for indicating that a function definition is a template.

```
......
```

A placeholder for indicating that a function definition is a template.

5.3 Syntax for Advanced

In Advanced, `set!` can be used to mutate variables, and `define-struct`'s structures are mutatable. `define` and `lambda` can define functions of zero arguments, and function calls can invoke functions of zero arguments.

```
(lambda (variable ...) expression)
```

Creates a function that takes as many arguments as given *variable*s, and whose body is *expression*.

```
(λ (variable ...) expression)
```

The Greek letter λ is a synonym for `lambda`.

```
(expression expression ...)
```

Calls the function that results from evaluating the first *expression*. The value of the call is the value of function's body when every instance of `name`'s variables are replaced by the values of the corresponding *expression*s.

The function being called must come from either a definition appearing before the function call, or from a `lambda` expression. The number of argument *expression*s must be the same as the number of arguments expected by the function.

```
(define-datatype dataype-name [variant-name field-name ...] ...)
```

A short-hand for defining a group of related structures. The following `define-datatype`:

```
(define-datatype datatype-name
  [variant-name field-name #, ...]
  ...)
```

is equivalent to:

```
(define (datatype-name? x)
  (or (variant-name? x) ...))
(define-struct variant-name (field-name ...))
...
```

(begin *expression expression* **...**)

Evaluates the *expression*s in order from left to right. The value of the `begin` expression is the value of the last *expression*.

(begin0 *expression expression* **...**)

Evaluates the *expression*s in order from left to right. The value of the `begin` expression is the value of the first *expression*.

(set! *variable expression*)

Evaluates *expression*, and then changes the value of the *variable* to have *expression*'s value. The *variable* must be defined by `define`, `letrec`, `let*`, `let`, or `local`.

(delay *expression*)

Produces a "promise" to evaluate *expression*. The *expression* is not evaluated until the promise is forced with `force`; when the promise is forced, the result is recorded, so that any further `force` of the promise immediately produces the remembered value.

(shared ([*name expression*] **...**) *expression*)

Like `letrec`, but when an *expression* next to an `id` is a `cons`, `list`, `vector`, quasiquoted expression, or `make-`*struct-name* from a `define-struct`, the *expression* can refer directly to any *name*, not just *name*s defined earlier. Thus, `shared` can be used to create cyclic data structures.

(recur *name* ([*name expression*] **...**) *expression*)

A short-hand syntax for recursive loops. The first *name* corresponds to the name of the recursive function. The *name*s in the parenthesis are the function's arguments, and each corresponding *expression* is a value supplied for that argument in an initial starting call of the function. The last *expression* is the body of the function.

More precisely, the following `recur`:

```
(recur func-name ([arg-name arg-expression] ...)
  body-expression)
```

is equivalent to:

```
(local [(define (func-name arg-name ...) body-expression)]
  (func-name arg-expression ...))
```

(*let name* ([*name expression*] **...**) *expression*)

An alternate syntax for `recur`.

(`case` *expression* [(*choice* **...**) *expression*] **...** [(*choice* **...**) *expression*])

A `case` form contains one or more clauses. Each clause contains a choices (in parentheses)—either numbers or names—and an answer *expression*. The initial *expression* is evaluated, and its value is compared to the choices in each clause, where the lines are considered in order. The first line that contains a matching choice provides an answer *expression* whose value is the result of the whole `case` expression. Numbers match with the numbers in the choices, and symbols match with the names. If none of the lines contains a matching choice, it is an error.

(`case` *expression* [(*choice* **...**) *expression*] **...** [`else` *expression*])

This form of `case` is similar to the prior one, except that the final `else` clause is taken if no clause contains a choice matching the value of the initial *expression*.

(`match` *expression* [*pattern expression*] **...**)

A `match` form contains one or more clauses that are surrounded by square brackets. Each clause contains a pattern—a description of a value—and an answer *expression*. The initial *expression* is evaluated, and its value is matched against the pattern in each clause, where the clauses are considered in order. The first clause that contains a matching pattern provides an answer *expression* whose value is the result of the whole `match` expression. This *expression* may reference identifiers defined in the matching pattern. If none of the clauses contains a matching pattern, it is an error.

(`when` *question-expression body-expression*)

If *question-expression* evaluates to `true`, the result of the `when` expression is the result of evaluating the *body-expression*, otherwise the result is `(void)` and the *body-expression* is not evaluated. If the result of evaluating the *question-expression* is neither `true` nor `false`, it is an error.

```
(unless question-expression body-expression)
```

Like `when`, but the *body-expression* is evaluated when the *question-expression* produces `false` instead of `true`.

5.4 Common Syntaxes

The following syntaxes behave the same in the *Advanced* level as they did in the §4 "Intermediate Student with Lambda" level.

```
(local [definition ...] expression)
```

Groups related definitions for use in *expression*. Each *definition* can be either a `define` or a `define-struct`.

When evaluating `local`, each *definition* is evaluated in order, and finally the body *expression* is evaluated. Only the expressions within the `local` (including the right-hand-sides of the *definition*s and the *expression*) may refer to the names defined by the *definition*s. If a name defined in the `local` is the same as a top-level binding, the inner one "shadows" the outer one. That is, inside the `local`, any references to that name refer to the inner one.

```
(letrec ([name expr-for-let] ...) expression)
```

Like `local`, but with a simpler syntax. Each *name* defines a variable (or a function) with the value of the corresponding *expr-for-let*. If *expr-for-let* is a `lambda`, `letrec` defines a function, otherwise it defines a variable.

```
(let* ([name expr-for-let] ...) expression)
```

Like `letrec`, but each *name* can only be used in *expression*, and in *expr-for-let*s occuring after that *name*.

```
(let ([name expr-for-let] ...) expression)
```

Like `letrec`, but the defined *name*s can be used only in the last *expression*, not the *expr-for-let*s next to the *name*s.

```
(time expression)
```

Measures the time taken to evaluate *expression*. After evaluating *expression*, `time` prints out the time taken by the evaluation (including real time, time taken by the CPU, and the time spent collecting free memory). The value of `time` is the same as that of *expression*.

```
(define (name variable variable ...) expression)
```

Defines a function named *name*. The *expression* is the body of the function. When the function is called, the values of the arguments are inserted into the body in place of the *variable*s. The function returns the value of that new expression.

The function name's cannot be the same as that of another function or variable.

```
(define name expression)
```

Defines a variable called *name* with the the value of *expression*. The variable name's cannot be the same as that of another function or variable, and *name* itself must not appear in *expression*.

```
(define-struct structure-name (field-name ...))
```

Defines a new structure called *structure-name*. The structure's fields are named by the *field-name*s. After the `define-struct`, the following new functions are available:

- `make-`*structure-name* : takes a number of arguments equal to the number of fields in the structure, and creates a new instance of that structure.
- *structure-name*`-`*field-name* : takes an instance of the structure and returns the value in the field named by *field-name*.
- *structure-name*`?` : takes any value, and returns `#true` if the value is an instance of the structure.

The name of the new functions introduced by `define-struct` must not be the same as that of other functions or variables, otherwise `define-struct` reports an error.

In Advanced, `define-struct` introduces one additional function:

- `set-`*structure-name*`-`*field-name*`!` : takes an instance of the structure and a value, and mutates the instance's field to the given value.

```
(cond [question-expression answer-expression] ...)
(cond [question-expression answer-expression]
      ...
      [else answer-expression])
```

Chooses a clause based on some condition. `cond` finds the first *question-expression* that evaluates to `#true`, then evaluates the corresponding *answer-expression*.

If none of the *question-expression*s evaluates to `#true`, `cond`'s value is the *answer-expression* of the `else` clause. If there is no `else`, `cond` reports an error. If the result of a *question-expression* is neither `#true` nor `#false`, `cond` also reports an error.

`else` cannot be used outside of `cond`.

```
(if question-expression
    then-answer-expression
    else-answer-expression)
```

When the value of the *question-expression* is `#true`, `if` evaluates the *then-answer-expression*. When the test is `#false`, `if` evaluates the *else-answer-expression*.

If the *question-expression* is neither `#true` nor `#false`, `if` reports an error.

```
(and expression expression expression ...)
```

Evaluates to `#true` if all the *expression*s are `#true`. If any *expression* is `#false`, the `and` expression evaluates to `#false` (and the expressions to the right of that expression are not evaluated.)

If any of the expressions evaluate to a value other than `#true` or `#false`, `and` reports an error.

```
(or expression expression expression ...)
```

Evaluates to `#true` as soon as one of the *expression*s is `#true` (and the expressions to the right of that expression are not evaluated.) If all of the *expression*s are `#false`, the `or` expression evaluates to `#false`.

If any of the expressions evaluate to a value other than `#true` or `#false`, `or` reports an error.

```
(check-expect expression expected-expression)
```

Checks that the first *expression* evaluates to the same value as the *expected-expression*.

```
(check-expect (fahrenheit->celsius 212) 100)
(check-expect (fahrenheit->celsius -40) -40)

(define (fahrenheit->celsius f)
  (* 5/9 (- f 32)))
```

A `check-expect` expression must be placed at the top-level of a student program. Also it may show up anywhere in the program, including ahead of the tested function definition. By placing `check-expect`s there, a programmer conveys to a future reader the intention behind the program with working examples, thus making it often superfluous to read the function definition proper. Syntax errors in `check-expect` (and all check forms) are intentionally delayed to run time so that students can write tests *without* necessarily writing complete function headers.

It is an error for `expr` or `expected-expr` to produce an inexact number or a function value. As for inexact numbers, it is *morally* wrong to compare them for plain equality. Instead one tests whether they are both within a small interval; see `check-within`. As for functions (see Intermediate and up), it is provably impossible to compare functions.

```
(check-random expression expected-expression)
```

Checks that the first *expression* evaluates to the same value as the *expected-expression*.

The form supplies the same random-number generator to both parts. If both parts request `random` numbers from the same interval in the same order, they receive the same random numbers.

Here is a simple example of where `check-random` is useful:

```
(define WIDTH 100)
(define HEIGHT (* 2 WIDTH))

(define-struct player (name x y))
; A Player is (make-player String Nat Nat)

; String -> Player

(check-random (create-randomly-placed-player "David Van Horn")
              (make-player "David Van Horn" (random WIDTH) (random HEIGHT)))

(define (create-randomly-placed-player name)
  (make-player name (random WIDTH) (random HEIGHT)))
```

Note how `random` is called on the same numbers in the same order in both parts of `check-random`. If the two parts call `random` for different intervals, they are likely to fail:

```
; String -> Player

(check-random (create-randomly-placed-player "David Van Horn")
              (make-player "David Van Horn" (random WIDTH) (random HEIGHT)))

(define (create-randomly-placed-player name)
  (a-helper-function name (random HEIGHT)))

; String Number -> Player
(define (a-helper-function name height)
   (make-player name (random WIDTH) height))
```

Because the argument to `a-helper-function` is evaluated first, `random` is first called for the interval [0,*HEIGHT*) and then for [0,*WIDTH*), that is, in a different order than in the preceding `check-random`.

It is an error for `expr` or `expected-expr` to produce a function value or an inexact number; see note on `check-expect` for details.

```
(check-satisfied expression predicate)
```

Checks that the first *`expression`* satisfies the named *`predicate`* (function of one argument). Recall that "satisfies" means "the function produces `#true` for the given value."

Here are simple examples for `check-satisfied`:

```
> (check-satisfied 1 odd?)
The test passed!


> (check-satisfied 1 even?)
Ran 1 test.
0 tests passed.
Check failures:
        Actual value | 1 | does not satisfy even?.

at line 3, column 0
```

In general `check-satisfied` empowers program designers to use defined functions to formulate test suites:

```
; [cons Number [List-of Number]] -> Boolean
; a function for testing htdp-sort

(check-expect (sorted? (list 1 2 3)) #true)
(check-expect (sorted? (list 2 1 3)) #false)

(define (sorted? l)
  (cond
    [(empty? (rest l)) #true]
    [else (and (<= (first l) (second l)) (sorted? (rest l)))]))

; [List-of Number] -> [List-of Number]
; create a sorted version of the given list of numbers

(check-satisfied (htdp-sort (list 1 2 0 3)) sorted?)

(define (htdp-sort l)
  (cond
    [(empty? l) l]
    [else (insert (first l) (htdp-sort (rest l)))]))

; Number [List-of Number] -> [List-of Number]
; insert x into l at proper place
; assume l is arranged in ascending order
; the result is sorted in the same way
(define (insert x l)
  (cond
    [(empty? l) (list x)]
    [else (if (<= x (first l)) (cons x l) (cons (first l) (insert x (rest l))))]))
```

And yes, the results of `htdp-sort` satisfy the `sorted?` predicate:

```
> (check-satisfied (htdp-sort (list 1 2 0 3)) sorted?)
```

`(check-within` *`expression expected-expression delta`*`)`

Checks whether the value of the *`expression`* expression is structurally equal to the value produced by the *`expected-expression`* expression; every number in the first expression must be within *`delta`* of the corresponding number in the second expression.

```
(define-struct roots (x sqrt))
; RT is [List-of (make-roots Number Number)]

(define (root-of a)
```

```
  (make-roots a (sqrt a)))

(define (roots-table xs)
  (cond
    [(empty? xs) '()]
    [else (cons (root-of (first xs)) (roots-table (rest xs)))]))
```

Due to the presence of inexact numbers in nested data, `check-within` is the correct choice for testing, and the test succeeds if *`delta`* is reasonably large:

Example:

```
> (check-within (roots-table (list 1.0 2.0 3.0))
                (list
                  (make-roots 1.0 1.0)
                  (make-roots 2  1.414)
                  (make-roots 3  1.713))
                0.1)
The test passed!
```

In contrast, when *`delta`* is small, the test fails:

Example:

```
> (check-within (roots-table (list 2.0))
                (list
                  (make-roots 2  1.414))
                #i1e-5)
Ran 1 test.
0 tests passed.
Check failures:

        Actual value | '((make-roots 2.0 1.4142135623730951)) | is
not within 1e-5 of expected value | '((make-roots 2 1.414)) |.


at line 5, column 0
```

It is an error for `expressions` or *`expected-expression`* to produce a function value; see note on `check-expect` for details.

If *`delta`* is not a number, `check-within` reports an error.

```
(check-random-within expression expected-expression delta)
```

Combines `check-random` and `check-within`. Like `check-random`, *`expression`* and *`expected-expression`* are evaluated with the same random-number sequence, so calls to `random` in both expressions draw the same numbers. Like `check-within`, the test then succeeds if every number in the value of *`expression`* is within *`delta`* of the corresponding number in the value of *`expected-expression`*.

Use `check-random-within` in place of `check-random` when the random computation produces inexact numbers, which `check-random` refuses to compare.

If *`delta`* is not a number, `check-random-within` reports an error.

Added in version 1.9 of package `htdp-lib`.

```
(check-error expression expected-error-message)
(check-error expression)
```

Checks that the *`expression`* reports an error, where the error messages matches the value of *`expected-error-message`*, if it is present.

Here is a typical beginner example that calls for a use of `check-error`:

```
(define sample-table
  '(("matthias" 10)
    ("matthew"  20)
    ("robby"    -1)
    ("shriram"  18)))

; [List-of [list String Number]] String -> Number
; determine the number associated with s in table

(define (lookup table s)
  (cond
    [(empty? table) (error (string-append s " not found"))]
    [else (if (string=? (first (first table)) s)
              (second (first table))
              (lookup (rest table)))]))
```

Consider the following two examples in this context:

Example:

```
> (check-expect (lookup sample-table "matthew") 20)
The test passed!
```

Example:

```
> (check-error (lookup sample-table "kathi") "kathi not found")
The test passed!
```

(check-member-of *expression expression expression ...*)

Checks that the value of the first *expression* is that of one of the following *expression*s.

```
; [List-of X] -> X
; pick a random element from the given list l
(define (pick-one l)
  (list-ref l (random (length l))))
```

Example:

```
> (check-member-of (pick-one '("a" "b" "c")) "a" "b" "c")
The test passed!
```

It is an error for any of `expressions` to produce a function value; see note on `check-expect` for details.

(check-range *expression low-expression high-expression*)

Checks that the value of the first *expression* is a number in between the value of the *low-expression* and the *high-expression*, inclusive.

A `check-range` form is best used to delimit the possible results of functions that compute inexact numbers:

```
(define EPSILON 0.001)

; [Real -> Real] Real -> Real
; what is the slope of f at x?
(define (differentiate f x)
  (slope f (- x EPSILON) (+ x EPSILON)))

; [Real -> Real] Real Real -> Real
(define (slope f left right)
  (/ (- (f right) (f left))
     2 EPSILON))

(check-range (differentiate sin 0) 0.99 1.0)
```

It is an error for *expression*, *low-expression*, or *high-expression* to produce a function value; see note on `check-expect` for details.

```
(require string)
```

Makes the definitions of the module specified by *string* available in the current module (i.e., the current file), where *string* refers to a file relative to the current file.

The *string* is constrained in several ways to avoid problems with different path conventions on different platforms: a `/` is a directory separator, `.` always means the current directory, `..` always means the parent directory, path elements can use only `a` through `z` (uppercase or lowercase), `0` through `9`, `-`, `_`, and `.`, and the string cannot be empty or contain a leading or trailing `/`.

```
(require module-name)
```

Accesses a file in an installed library. The library name is an identifier with the same constraints as for a relative-path string (though without the quotes), with the additional constraint that it must not contain a `..`

```
(require (lib string string ...))
```

Accesses a file in an installed library, making its definitions available in the current module (i.e., the current file). The first *string* names the library file, and the remaining *string*s name the collection (and sub-collection, and so on) where the file is installed. Each string is constrained in the same way as for the `(require` *string*`)` form.

```
(require (planet string (string string number number)))
(require (planet id))
(require (planet string))
```

Accesses a library that is distributed on the internet via the PLaneT server, making it definitions available in the current module (i.e., current file).

The full grammar for planet requires is given in §3.2 “Importing and Exporting: `require` and `provide`”, but the best place to find examples of the syntax is on the the PLaneT server, in the description of a specific package.

5.5 Pre-Defined Functions

5.6 Signatures

Signatures do not have to be comment: They can also be part of the code. When a signature is attached to a function, DrRacket will check that program uses the function in accordance with the signature and display signature violations along with the test results.

A signature is a regular value, and is specified as a *signature form*, a special syntax that only works with `:` signature declarations and inside `signature` expressions.

```
(: name signature-form)
```

This attaches the signature specified by *signature-form* to the definition of *name*. There must be a definition of *name* somewhere in the program.

```
(: age Integer)
(define age 42)

(: area-of-square (Number -> Number))
(define (area-of-square len)
  (sqr len))
```

On running the program, Racket checks whether the signatures attached with `:` actually match the value of the variable. If they don't, Racket reports *signature violation* along with test failures.

For example, this piece of code:

```
(: age Integer)
(define age "fortytwo")
```

Yields this output:

```
1 signature violation.

Signature violations:
        got "fortytwo" at line 2, column 12, signature at line 1,
column 7
```

Note that a signature violation does not stop the running program.

```
(signature signature-form)
```

This returns the signature described by *signature-form* as a value.

5.6.1 Signature Forms

Any expression can be a signature form, in which case the signature is the value returned by that expression. There are a few special signature forms, however:

In a signature form, any name that starts with a % is a *signature variable* that stands for any signature depending on how the signature is used.

Example:

```
(: same (%a -> %a))

(define (same x) x)
```

`(`*input-signature-form* `...` `->` *output-signature-form*`)`

This signature form describes a function with inputs described by the *input-signature-form*s and output described by *output-signature-form*.

`(enum` *expr* `...)`

This signature describes an enumeration of the values returned by the *expr*s.

Example:

```
(: cute? ((enum "cat" "snake") -> Boolean))

(define (cute? pet)
  (cond
    [(string=? pet "cat") #t]
    [(string=? pet "snake") #f]))
```

`(mixed` *signature-form* `...)`

This signature describes mixed data, i.e. an itemization where each of the cases has a signature described by a *signature-form*.

Example:

```
(define SIGS (signature (mixed Aim Fired)))
```

`(ListOf` *signature-form*`)`

This signature describes a list where the elements are described by *signature-form*.

```
(predicate expression)
```

This signature describes values through a predicate: *expression* must evaluate to a function of one argument that returns a boolean. The signature matches all values for which the predicate returns #true.

5.6.2 Struct Signatures

A advanced form defines two additional names that can be used in signatures. For a struct called struct, these are Struct and StructOf. Note that these names are capitalized. In particular, a struct called Struct, will also define Struct and StructOf. Moreover, when forming the additional names, hyphens are removed, and each letter following a hyphen is capitalized - so a struct called foo-bar will define FooBar and FooBarOf.

Struct is a signature that describes struct values from this structure type. StructOf is a function that takes as input a signature for each field. It returns a signature describing values of this structure type, additionally describing the values of the fields of the value.

```
(define-struct pair [fst snd])

(: add-pair ((PairOf Number Number) -> Number))
(define (add-pair p)
  (+ (pair-fst p) (pair-snd p)))
```

The remaining subsections list those functions that are built into the programming language. All other functions are imported from a teachpack or must be defined in the program.

5.7 Numbers: Integers, Rationals, Reals, Complex, Exacts, Inexacts

```
(- x y ...) → number
  x : number
  y : number
```

Subtracts the second (and following) number(s) from the first ; negates the number if there is only one argument.

```
> (- 5)
-5
> (- 5 3)
2
> (- 5 3 1)
1
```

```
(< x y z ...) → boolean?
  x : real
  y : real
  z : real
```

Compares two or more (real) numbers for less-than.

```
> (< 42 2/5)
#false
```

```
(<= x y z ...) → boolean?
  x : real
  y : real
  z : real
```

Compares two or more (real) numbers for less-than or equality.

```
> (<= 42 2/5)
#false
```

```
(> x y z ...) → boolean?
  x : real
  y : real
  z : real
```

Compares two or more (real) numbers for greater-than.

```
> (> 42 2/5)
#true
```

```
(>= x y z ...) → boolean?
  x : real
  y : real
  z : real
```

Compares two or more (real) numbers for greater-than or equality.

```
> (>= 42 42)
#true
```

```
(abs x) → real
  x : real
```

Determines the absolute value of a real number.

```
> (abs -12)
12
```

```
(acos x) → number
  x : number
```

Computes the arccosine (inverse of cos) of a number.

```
> (acos 0)
#i1.5707963267948966
```

```
(add1 x) → number
  x : number
```

Increments the given number.

```
> (add1 2)
3
```

```
(angle x) → real
  x : number
```

Extracts the angle from a complex number.

```
> (angle (make-polar 3 4))
#i-2.2831853071795867
```

```
(asin x) → number
  x : number
```

Computes the arcsine (inverse of sin) of a number.

```
> (asin 0)
0
```

```
(atan x) → number
  x : number
```

Computes the arctangent of the given number:

```
> (atan 0)
0
> (atan 0.5)
#i0.46364760900080615
```

Also comes in a two-argument version where `(atan y x)` computes `(atan (/ y x))` but the signs of `y` and `x` determine the quadrant of the result and the result tends to be more accurate than that of the 1-argument version in borderline cases:

```
> (atan 3 4)
#i0.6435011087932844
> (atan -2 -1)
#i-2.0344439357957027
```

```
(ceiling x) → integer
  x : real
```

Determines the closest integer (exact or inexact) above a real number. See `round`.

```
> (ceiling 12.3)
#i13.0
```

```
(complex? x) → boolean?
  x : any/c
```

Determines whether some value is complex.

```
> (complex? 1-2i)
#true
```

```
(conjugate x) → number
  x : number
```

Flips the sign of the imaginary part of a complex number.

```
> (conjugate 3+4i)
3-4i
> (conjugate -2-5i)
-2+5i
> (conjugate (make-polar 3 4))
#i-1.960930862590836+2.270407485923785i
```

```
(cos x) → number
  x : number
```

Computes the cosine of a number (radians).

```
> (cos pi)
#i-1.0
```

```
(cosh x) → number
  x : number
```

Computes the hyperbolic cosine of a number.

```
> (cosh 10)
#i11013.232920103324
```

```
(current-seconds) → integer
```

Determines the current time in seconds elapsed (since a platform-specific starting date).

```
> (current-seconds)
1793426339
```

```
(denominator x) → integer
  x : rational?
```

Computes the denominator of a rational.

```
> (denominator 2/3)
3
```

```
e : real
```

Euler's number.

```
> e
#i2.718281828459045
```

```
(even? x) → boolean?
  x : integer
```

Determines if some integer (exact or inexact) is even or not.

```
> (even? 2)
#true
```

```
(exact->inexact x) → number
  x : number
```

Converts an exact number to an inexact one.

```
> (exact->inexact 12)
#i12.0
```

```
(exact? x) → boolean?
  x : number
```

Determines whether some number is exact.

```
> (exact? (sqrt 2))
#false
```

```
(exp x) → number
  x : number
```

Determines e raised to a number.

```
> (exp -2)
#i0.1353352832366127
```

```
(expt x y) → number
  x : number
  y : number
```

Computes the power of the first to the second number, which is to say, exponentiation.

```
> (expt 16 1/2)
4
> (expt 3 -4)
1/81
```

```
(floor x) → integer
  x : real
```

Determines the closest integer (exact or inexact) below a real number. See round.

```
> (floor 12.3)
#i12.0
```

```
(gcd x y ...) → integer
  x : integer
  y : integer
```

Determines the greatest common divisor of two integers (exact or inexact).

```
> (gcd 6 12 8)
2
```

```
(imag-part x) → real
  x : number
```

Extracts the imaginary part from a complex number.

```
> (imag-part 3+4i)
4
```

```
(inexact->exact x) → number
  x : number
```

Approximates an inexact number by an exact one.

```
> (inexact->exact 12.0)
12
```

```
(inexact? x) → boolean?
  x : number
```

Determines whether some number is inexact.

```
> (inexact? 1-2i)
#false
```

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

Looks up the character that corresponds to the given exact integer in the ASCII table (if any).

```
> (integer->char 42)
#\*
```

```
(integer-sqrt x) → complex
  x : integer
```

Computes the integer or imaginary-integer square root of an integer.

```
> (integer-sqrt 11)
3
> (integer-sqrt -11)
0+3i
```

```
(integer? x) → boolean?
  x : any/c
```

Determines whether some value is an integer (exact or inexact).

```
> (integer? (sqrt 2))
#false
```

```
(lcm x y ...) → integer
  x : integer
  y : integer
```

Determines the least common multiple of two integers (exact or inexact).

```
> (lcm 6 12 8)
24
```

```
(log x) → number
  x : number
```

Determines the base-e logarithm of a number.

```
> (log 12)
#i2.4849066497880004
```

```
(magnitude x) → real
  x : number
```

Determines the magnitude of a complex number.

```
> (magnitude (make-polar 3 4))
#i3.0000000000000004
```

```
(make-polar x y) → number
  x : real
  y : real
```

Creates a complex from a magnitude and angle.

```
> (make-polar 3 4)
#i-1.960930862590836-2.270407485923785i
```

```
(make-rectangular x y) → number
  x : real
  y : real
```

Creates a complex from a real and an imaginary part.

```
> (make-rectangular 3 4)
3+4i
```

```
(max x y ...) → real
  x : real
  y : real
```

Determines the largest number—aka, the maximum.

```
> (max 3 2 8 7 2 9 0)
9
```

```
(min x y ...) → real
  x : real
  y : real
```

Determines the smallest number—aka, the minimum.

```
> (min 3 2 8 7 2 9 0)
0
```

```
(modulo x y) → integer
  x : integer
  y : integer
```

Finds the remainder of the division of the first number by the second:

```
> (modulo 9 2)
1
> (modulo 3 -4)
-1
```

```
(negative? x) → boolean?
  x : real
```

Determines if some real number is strictly smaller than zero.

```
> (negative? -2)
#true
```

```
(number->string x) → string
  x : number
```

Converts a number to a string.

```
> (number->string 42)
"42"
```

```
(number->string-digits x p) → string
  x : number
  p : posint
```

Converts a number x to a string with the specified number of digits.

```
> (number->string-digits 0.9 2)
"0.9"
> (number->string-digits pi 4)
"3.1416"
```

```
(number? n) → boolean?
  n : any/c
```

Determines whether some value is a number:

```
> (number? "hello world")
#false
> (number? 42)
#true
```

```
(numerator x) → integer
  x : rational?
```

Computes the numerator of a rational.

```
> (numerator 2/3)
2
```

```
(odd? x) → boolean?
  x : integer
```

Determines if some integer (exact or inexact) is odd or not.

```
> (odd? 2)
#false
```

```
pi : real
```

The ratio of a circle's circumference to its diameter.

```
> pi
#i3.141592653589793
```

```
(positive? x) → boolean?
  x : real
```

Determines if some real number is strictly larger than zero.

```
> (positive? -2)
#false
```

```
(quotient x y) → integer
  x : integer
  y : integer
```

Divides the first integer—also called dividend—by the second—known as divisor—to obtain the quotient.

```
> (quotient 9 2)
4
> (quotient 3 4)
0
```

```
(random x) → natural
  x : natural
```

Generates a random number. If given one argument `random` returns a natural number less than the given natural. In ASL, if given no arguments, `random` generates a random inexact number between 0.0 and 1.0 exclusive.

```
> (random)
#i0.7130929665461503

> (random)
#i0.7854571289790522

> (random 42)
25
```

```
> (random 42)
26
```

```
(rational? x) → boolean?
  x : any/c
```

Determines whether some value is a rational number.

```
> (rational? 1)
#true
> (rational? -2.349)
#true
> (rational? #i1.23456789)
#true
> (rational? (sqrt -1))
#false
> (rational? pi)
#true
> (rational? e)
#true
> (rational? 1-2i)
#false
```

As the interactions show, the teaching languages considers many more numbers as rationals than expected. In particular, `pi` is a rational number because it is only a finite approximation to the mathematical π. Think of `rational?` as a suggestion to think of these numbers as fractions.

```
(real-part x) → real
  x : number
```

Extracts the real part from a complex number.

```
> (real-part 3+4i)
3
```

```
(real? x) → boolean?
  x : any/c
```

Determines whether some value is a real number.

```
> (real? 1-2i)
#false
```

```
(remainder x y) → integer
  x : integer
  y : integer
```

Determines the remainder of dividing the first by the second integer (exact or inexact).

```
> (remainder 9 2)
1
> (remainder 3 4)
3
```

```
(round x) → integer
  x : real
```

Rounds a real number to an integer (rounds to even to break ties). See `floor` and `ceiling`.

```
> (round 12.3)
#i12.0
```

```
(sgn x) → (union 1 #i1.0 0 #i0.0 -1 #i-1.0)
  x : real
```

Determines the sign of a real number.

```
> (sgn -12)
-1
```

```
(sin x) → number
  x : number
```

Computes the sine of a number (radians).

```
> (sin pi)
#i1.2246467991473532e-16
```

```
(sinh x) → number
  x : number
```

Computes the hyperbolic sine of a number.

```
> (sinh 10)
#i11013.232874703393
```

```
(sqr x) → number
  x : number
```

Computes the square of a number.

```
> (sqr 8)
64
```

```
(sqrt x) → number
  x : number
```

Computes the square root of a number.

```
> (sqrt 9)
3
> (sqrt 2)
#i1.4142135623730951
```

```
(sub1 x) → number
  x : number
```

Decrements the given number.

```
> (sub1 2)
1
```

```
(tan x) → number
  x : number
```

Computes the tangent of a number (radians).

```
> (tan pi)
#i-1.2246467991473532e-16
```

```
(zero? x) → boolean?
  x : number
```

Determines if some number is zero or not.

```
> (zero? 2)
#false
```

5.8 Booleans

```
(boolean->string x) → string
  x : boolean?
```

Produces a string for the given boolean

```
> (boolean->string #false)
"#false"
> (boolean->string #true)
"#true"
```

```
(boolean=? x y) → boolean?
  x : boolean?
  y : boolean?
```

Determines whether two booleans are equal.

```
> (boolean=? #true #false)
#false
```

```
(boolean? x) → boolean?
  x : any/c
```

Determines whether some value is a boolean.

```
> (boolean? 42)
#false
> (boolean? #false)
#true
```

```
(false? x) → boolean?
  x : any/c
```

Determines whether a value is false.

```
> (false? #false)
#true
```

```
(not x) → boolean?
  x : boolean?
```

Negates a boolean value.

```
> (not #false)
#true
```

5.9 Symbols

```
(symbol->string x) → string
  x : symbol
```

Converts a symbol to a string.

```
> (symbol->string 'c)
"c"
```

```
(symbol=? x y) → boolean?
  x : symbol
  y : symbol
```

Determines whether two symbols are equal.

```
> (symbol=? 'a 'b)
#false
```

```
(symbol? x) → boolean?
  x : any/c
```

Determines whether some value is a symbol.

```
> (symbol? 'a)
#true
```

5.10 Lists

```
(append l ...) → (listof any)
  l : (listof any)
```

Creates a single list from several. In ASL, `list*` also deals with cyclic lists.

```
(assoc x l) → (union (listof any) #false)
  x : any/c
  l : (listof any)
```

Produces the first pair on `l` whose `first` is `equal?` to `x`; otherwise it produces `#false`.

```
> (assoc "hello" '(("world" 2) ("hello" 3) ("good" 0)))
(list "hello" 3)
```

```
(assq x l) → (union #false cons?)
  x : any/c
  l : list?
```

Determines whether some item is the first item of a pair in a list of pairs. (It compares the items with eq?.)

```
> a
(list (list 'a 22) (list 'b 8) (list 'c 70))
> (assq 'b a)
(list 'b 8)
```

```
(caaar x) → any/c
  x : list?
```

LISP-style selector: (car (car (car x))).

```
> w
(list (list (list (list "bye") 3) #true) 42)
> (caaar w)
(list "bye")
```

```
(caadr x) → any/c
  x : list?
```

LISP-style selector: (car (car (cdr x))).

```
> (caadr (cons 1 (cons (cons 'a '()) (cons (cons 'd '()) '()))))
'a
```

```
(caar x) → any/c
  x : list?
```

LISP-style selector: (car (car x)).

```
> y
(list (list (list 1 2 3) #false "world"))
> (caar y)
(list 1 2 3)
```

```
(cadar x) → any/c
  x : list?
```

LISP-style selector: `(car (cdr (car x)))`.

```
> w
(list (list (list (list "bye") 3) #true) 42)
> (cadar w)
#true
```

```
(cadddr x) → any/c
  x : list?
```

LISP-style selector: `(car (cdr (cdr (cdr x))))`.

```
> v
(list 1 2 3 4 5 6 7 8 9 'A)
> (cadddr v)
4
```

```
(caddr x) → any/c
  x : list?
```

LISP-style selector: `(car (cdr (cdr x)))`.

```
> x
(list 2 "hello" #true)
> (caddr x)
#true
```

```
(cadr x) → any/c
  x : list?
```

LISP-style selector: `(car (cdr x))`.

```
> x
(list 2 "hello" #true)
> (cadr x)
"hello"
```

```
(car x) → any/c
  x : cons?
```

Selects the first item of a non-empty list.

```
> x
(list 2 "hello" #true)
> (car x)
2
```

```
(cdaar x) → any/c
  x : list?
```

LISP-style selector: `(cdr (car (car x)))`.

```
> w
(list (list (list (list "bye") 3) #true) 42)
> (cdaar w)
(list 3)
```

```
(cdadr x) → any/c
  x : list?
```

LISP-style selector: `(cdr (car (cdr x)))`.

```
> (cdadr (list 1 (list 2 "a") 3))
(list "a")
```

```
(cdar x) → list?
  x : list?
```

LISP-style selector: `(cdr (car x))`.

```
> y
(list (list (list 1 2 3) #false "world"))
> (cdar y)
(list #false "world")
```

```
(cddar x) → any/c
  x : list?
```

LISP-style selector: `(cdr (cdr (car x)))`

```
> w
(list (list (list (list "bye") 3) #true) 42)
> (cddar w)
'()
```

```
(cdddr x) → any/c
  x : list?
```

LISP-style selector: `(cdr (cdr (cdr x)))`.

```
> v
(list 1 2 3 4 5 6 7 8 9 'A)
> (cdddr v)
(list 4 5 6 7 8 9 'A)
```

```
(cddr x) → list?
  x : list?
```

LISP-style selector: `(cdr (cdr x))`.

```
> x
(list 2 "hello" #true)
> (cddr x)
(list #true)
```

```
(cdr x) → any/c
  x : cons?
```

Selects the rest of a non-empty list.

```
> x
(list 2 "hello" #true)
> (cdr x)
(list "hello" #true)
```

```
(cons x l) → (listof X)
  x : X
  l : (listof X)
```

Constructs a list. In ASL, `cons` creates a mutable list.

```
(cons? x) → boolean?
  x : any/c
```

Determines whether some value is a constructed list.

```
> (cons? (cons 1 '()))
#true
> (cons? 42)
#false
```

```
(eighth x) → any/c
  x : list?
```

Selects the eighth item of a non-empty list.

```
> v
(list 1 2 3 4 5 6 7 8 9 'A)
> (eighth v)
8
```

```
(empty? x) → boolean?
  x : any/c
```

Determines whether some value is the empty list.

```
> (empty? '())
#true
> (empty? 42)
#false
```

```
(fifth x) → any/c
  x : list?
```

Selects the fifth item of a non-empty list.

```
> v
(list 1 2 3 4 5 6 7 8 9 'A)
> (fifth v)
5
```

```
(first x) → any/c
  x : cons?
```

Selects the first item of a non-empty list.

```
> x
(list 2 "hello" #true)
> (first x)
2
```

```
(for-each f l ...) → void?
  f : (any ... -> any)
  l : (listof any)
```

Applies a function to each item on one or more lists for effect only:

```
(for-each f (list x-1 ... x-n)) = (begin (f x-1) ... (f x-n))
```

```
> (for-each (lambda (x) (begin (display x) (newline))) '(1 2 3))
1
2
3
```

```
(fourth x) → any/c
  x : list?
```

Selects the fourth item of a non-empty list.

```
> v
(list 1 2 3 4 5 6 7 8 9 'A)
> (fourth v)
4
```

```
(length l) → natural?
  l : list?
```

Evaluates the number of items on a list.

```
> x
(list 2 "hello" #true)
> (length x)
3
```

```
(list x ...) → list?
  x : any/c
```

Constructs a list of its arguments.

```
> (list 1 2 3 4 5 6 7 8 9 0)
(cons 1 (cons 2 (cons 3 (cons 4 (cons 5 (cons 6 (cons 7 (cons 8
(cons 9 (cons 0 '()))))))))))
```

```
(list* x ... l) → (listof any)
  x : any
  l : (listof any)
```

Constructs a list by adding multiple items to a list. In ASL, `list*` also deals with cyclic lists.

```
(list-ref x i) → any/c
  x : list?
  i : natural?
```

Extracts the indexed item from the list.

```
> v
(list 1 2 3 4 5 6 7 8 9 'A)
> (list-ref v 9)
'A
```

```
(list? x) → boolean?
  x : any/c
```

Checks whether the given value is a list.

```
> (list? 42)
#false
> (list? '())
#true
> (list? (cons 1 (cons 2 '())))
#true
```

```
(make-list i x) → list?
  i : natural?
  x : any/c
```

Constructs a list of `i` copies of `x`.

```
> (make-list 3 "hello")
(cons "hello" (cons "hello" (cons "hello" '())))
```

```
(member x l) → boolean?
  x : any/c
  l : list?
```

Determines whether some value is on the list (comparing values with equal?).

```
> x
(list 2 "hello" #true)
> (member "hello" x)
#true
```

```
(member? x l) → boolean?
  x : any/c
  l : list?
```

Determines whether some value is on the list (comparing values with equal?).

```
> x
(list 2 "hello" #true)
> (member? "hello" x)
#true
```

```
(memq x l) → boolean?
  x : any/c
  l : list?
```

Determines whether some value `x` is on some list `l`, using `eq?` to compare `x` with items on `l`.

```
> x
(list 2 "hello" #true)
> (memq (list (list 1 2 3)) x)
#false
```

```
(memq? x l) → boolean?
  x : any/c
  l : list?
```

Determines whether some value *x* is on some list *l*, using `eq?` to compare *x* with items on *l*.

```
> x
(list 2 "hello" #true)
> (memq? (list (list 1 2 3)) x)
#false
```

```
(memv x l) → (or/c #false list)
  x : any/c
  l : list?
```

Determines whether some value is on the list if so, it produces the suffix of the list that starts with x if not, it produces false. (It compares values with the eqv? predicate.)

```
> x
(list 2 "hello" #true)
> (memv (list (list 1 2 3)) x)
#false
```

```
null : list
```

Another name for the empty list

```
> null
'()
```

```
(null? x) → boolean?
  x : any/c
```

Determines whether some value is the empty list.

```
> (null? '())
#true
> (null? 42)
#false
```

```
(range start end step) → list?
  start : number
  end : number
  step : number
```

Constructs a list of numbers by *step*ping from *start* to *end*.

```
> (range 0 10 2)
(cons 0 (cons 2 (cons 4 (cons 6 (cons 8 '()))))
```

```
(remove x l) → list?
  x : any/c
  l : list?
```

Constructs a list like the given one, with the first occurrence of the given item removed (comparing values with equal?).

```
> x
(list 2 "hello" #true)
> (remove "hello" x)
(list 2 #true)
> hello-2
(list 2 "hello" #true "hello")
> (remove "hello" hello-2)
(list 2 #true "hello")
```

```
(remove-all x l) → list?
  x : any/c
  l : list?
```

Constructs a list like the given one, with all occurrences of the given item removed (comparing values with equal?).

```
> x
(list 2 "hello" #true)
> (remove-all "hello" x)
(list 2 #true)
> hello-2
(list 2 "hello" #true "hello")
> (remove-all "hello" hello-2)
(list 2 #true)
```

```
(rest x) → any/c
  x : cons?
```

Selects the rest of a non-empty list.

```
> x
(list 2 "hello" #true)
> (rest x)
(list "hello" #true)
```

```
(reverse l) → list
  l : list?
```

Creates a reversed version of a list.

```
> x
(list 2 "hello" #true)
> (reverse x)
(list #true "hello" 2)
```

```
(second x) → any/c
  x : list?
```

Selects the second item of a non-empty list.

```
> x
(list 2 "hello" #true)
> (second x)
"hello"
```

```
(seventh x) → any/c
  x : list?
```

Selects the seventh item of a non-empty list.

```
> v
(list 1 2 3 4 5 6 7 8 9 'A)
> (seventh v)
7
```

```
(sixth x) → any/c
  x : list?
```

Selects the sixth item of a non-empty list.

```
> v
(list 1 2 3 4 5 6 7 8 9 'A)
> (sixth v)
6
```

```
(third x) → any/c
  x : list?
```

Selects the third item of a non-empty list.

```
> x
(list 2 "hello" #true)
> (third x)
#true
```

5.11 Posns

```
(make-posn x y) → posn
  x : any/c
  y : any/c
```

Constructs a posn from two arbitrary values.

```
> (make-posn 3 3)
(make-posn 3 3)
> (make-posn "hello" #true)
(make-posn "hello" #true)
```

```
(posn-x p) → any/c
  p : posn
```

Extracts the x component of a posn.

```
> p
(make-posn 2 -3)
> (posn-x p)
2
```

```
(posn-y p) → any/c
  p : posn
```

Extracts the y component of a posn.

```
> p
(make-posn 2 -3)
> (posn-y p)
-3
```

```
(posn? x) → boolean?
  x : any/c
```

Determines if its input is a posn.

```
> q
(make-posn "bye" 2)
> (posn? q)
#true
> (posn? 42)
#false
```

```
(set-posn-x! p x) → void?
  p : posn
  x : any
```

Updates the x component of a posn.

```
> p
(make-posn 2 -3)
> (set-posn-x! p 678)
> p
(make-posn 678 -3)
```

```
(set-posn-y! p x) → void
  p : posn
  x : any
```

Updates the y component of a posn.

```
> q
(make-posn "bye" 2)
> (set-posn-y! q 678)
> q
(make-posn "bye" 678)
```

5.12 Characters

```
(char->integer c) → integer
  c : char
```

Looks up the number that corresponds to the given character in the ASCII table (if any).

```
> (char->integer #\a)
97
> (char->integer #\z)
122
```

```
(char-alphabetic? c) → boolean?
  c : char
```

Determines whether a character represents an alphabetic character.

```
> (char-alphabetic? #\Q)
#true
```

```
(char-ci<=? c d e ...) → boolean?
  c : char
  d : char
  e : char
```

Determines whether the characters are ordered in an increasing and case-insensitive manner.

```
> (char-ci<=? #\b #\B)
#true
> (char<=? #\b #\B)
#false
```

```
(char-ci<? c d e ...) → boolean?
  c : char
  d : char
  e : char
```

Determines whether the characters are ordered in a strictly increasing and case-insensitive manner.

```
> (char-ci<? #\B #\c)
#true
> (char<? #\b #\B)
#false
```

```
(char-ci=? c d e ...) → boolean?
  c : char
  d : char
  e : char
```

Determines whether two characters are equal in a case-insensitive manner.

```
> (char-ci=? #\b #\B)
#true
```

```
(char-ci>=? c d e ...) → boolean?
  c : char
  d : char
  e : char
```

Determines whether the characters are sorted in a decreasing and case-insensitive manner.

```
> (char-ci>=? #\b #\C)
#false
> (char>=? #\b #\C)
#true
```

```
(char-ci>? c d e ...) → boolean?
  c : char
  d : char
  e : char
```

Determines whether the characters are sorted in a strictly decreasing and case-insensitive manner.

```
> (char-ci>? #\b #\B)
#false
> (char>? #\b #\B)
#true
```

```
(char-downcase c) → char
  c : char
```

Produces the equivalent lower-case character.

```
> (char-downcase #\T)
#\t
```

```
(char-lower-case? c) → boolean?
  c : char
```

Determines whether a character is a lower-case character.

```
> (char-lower-case? #\T)
#false
```

```
(char-numeric? c) → boolean?
  c : char
```

Determines whether a character represents a digit.

```
> (char-numeric? #\9)
#true
```

```
(char-upcase c) → char
  c : char
```

Produces the equivalent upper-case character.

```
> (char-upcase #\t)
#\T
```

```
(char-upper-case? c) → boolean?
  c : char
```

Determines whether a character is an upper-case character.

```
> (char-upper-case? #\T)
#true
```

```
(char-whitespace? c) → boolean?
  c : char
```

Determines whether a character represents space.

```
> (char-whitespace? #\tab)
#true
```

```
(char<=? c d e ...) → boolean?
  c : char
  d : char
  e : char
```

Determines whether the characters are ordered in an increasing manner.

```
> (char<=? #\a #\a #\b)
#true
```

```
(char<? x d e ...) → boolean?
  x : char
  d : char
  e : char
```

Determines whether the characters are ordered in a strictly increasing manner.

```
> (char<? #\a #\b #\c)
#true
```

```
(char=? c d e ...) → boolean?
  c : char
  d : char
  e : char
```

Determines whether the characters are equal.

```
> (char=? #\b #\a)
#false
```

```
(char>=? c d e ...) → boolean?
  c : char
  d : char
  e : char
```

Determines whether the characters are sorted in a decreasing manner.

```
> (char>=? #\b #\b #\a)
#true
```

```
(char>? c d e ...) → boolean?
  c : char
  d : char
  e : char
```

Determines whether the characters are sorted in a strictly decreasing manner.

```
> (char>? #\A #\z #\a)
#false
```

```
(char? x) → boolean?
  x : any/c
```

Determines whether a value is a character.

```
> (char? "a")
#false
> (char? #\a)
#true
```

5.13 Strings

```
(explode s) → (listof string)
  s : string
```

Translates a string into a list of 1-letter strings.

```
> (explode "cat")
(list "c" "a" "t")
```

```
(format f x ...) → string
  f : string
  x : any/c
```

Formats a string, possibly embedding values.

```
> (format "Dear Dr. ~a:" "Flatt")
"Dear Dr. Flatt:"
> (format "Dear Dr. ~s:" "Flatt")
"Dear Dr. \"Flatt\":"
```

```
(implode l) → string
  l : list?
```

Concatenates the list of 1-letter strings into one string.

```
> (implode (cons "c" (cons "a" (cons "t" '()))))
"cat"
```

```
(int->string i) → string
  i : integer
```

Converts an integer in [0,55295] or [57344 1114111] to a 1-letter string.

```
> (int->string 65)
"A"
```

```
(list->string l) → string
  l : list?
```

Converts a s list of characters into a string.

```
> (list->string (cons #\c (cons #\a (cons #\t '()))))
"cat"
```

```
(make-string i c) → string
  i : natural?
  c : char
```

Produces a string of length *i* from *c*.

```
> (make-string 3 #\d)
"ddd"
```

```
(replicate i s) → string
  i : natural?
  s : string
```

Replicates *s* *i* times.

```
> (replicate 3 "h")
"hhh"
```

```
(string c ...) → string?
  c : char
```

Builds a string of the given characters.

```
> (string #\d #\o #\g)
"dog"
```

```
(string->int s) → integer
  s : string
```

Converts a 1-letter string to an integer in [0,55295] or [57344, 1114111].

```
> (string->int "a")
97
```

```
(string->list s) → (listof char)
  s : string
```

Converts a string into a list of characters.

```
> (string->list "hello")
(list #\h #\e #\l #\l #\o)
```

```
(string->number s) → (union number #false)
  s : string
```

Converts a string into a number, produce false if impossible.

```
> (string->number "-2.03")
-2.03
> (string->number "1-2i")
1-2i
```

```
(string->symbol s) → symbol
  s : string
```

Converts a string into a symbol.

```
> (string->symbol "hello")
'hello
```

```
(string-alphabetic? s) → boolean?
  s : string
```

Determines whether all 'letters' in the string are alphabetic.

```
> (string-alphabetic? "123")
#false
> (string-alphabetic? "cat")
#true
```

```
(string-contains-ci? s t) → boolean?
  s : string
  t : string
```

Determines whether the first string appears in the second one without regard to the case of the letters.

```
> (string-contains-ci? "At" "caT")
#true
```

```
(string-contains? s t) → boolean?
  s : string
  t : string
```

Determines whether the first string appears literally in the second one.

```
> (string-contains? "at" "cat")
#true
```

```
(string-copy s) → string
  s : string
```

Copies a string.

```
> (string-copy "hello")
"hello"
```

```
(string-downcase s) → string
  s : string
```

Produces a string like the given one with all 'letters' as lower case.

```
> (string-downcase "CAT")
"cat"
> (string-downcase "cAt")
"cat"
```

```
(string-ith s i) → 1string?
  s : string
  i : natural?
```

Extracts the ith 1-letter substring from s.

```
> (string-ith "hello world" 1)
"e"
```

```
(string-length s) → nat
  s : string
```

Determines the length of a string.

```
> (string-length "hello world")
11
```

```
(string-lower-case? s) → boolean?
  s : string
```

Determines whether all 'letters' in the string are lower case.

```
> (string-lower-case? "CAT")
#false
```

```
(string-numeric? s) → boolean?
  s : string
```

Determines whether all 'letters' in the string are numeric.

```
> (string-numeric? "123")
#true
> (string-numeric? "1-2i")
#false
```

```
(string-ref s i) → char
  s : string
  i : natural?
```

Extracts the *i*th character from *s*.

```
> (string-ref "cat" 2)
#\t
```

```
(string-upcase s) → string
  s : string
```

Produces a string like the given one with all 'letters' as upper case.

```
> (string-upcase "cat")
"CAT"
> (string-upcase "cAt")
"CAT"
```

```
(string-upper-case? s) → boolean?
  s : string
```

Determines whether all 'letters' in the string are upper case.

```
> (string-upper-case? "CAT")
#true
```

```
(string-whitespace? s) → boolean?
  s : string
```

Determines whether all 'letters' in the string are white space.

```
> (string-whitespace? (string-append " " (string #\tab #\newline #\return)))
#true
```

```
(string? x) → boolean?
  x : any/c
```

Determines whether a value is a string.

```
> (string? "hello world")
#true
> (string? 42)
#false
```

```
(substring s i j) → string
  s : string
  i : natural?
  j : natural?
```

Extracts the substring starting at *i* up to *j* (or the end if *j* is not provided).

```
> (substring "hello world" 1 5)
"ello"
> (substring "hello world" 1 8)
"ello wo"
> (substring "hello world" 4)
"o world"
```

5.14 Images

```
(image=? i j) → boolean?
  i : image
  j : image
```

Determines whether two images are equal.

```
> c1
```

```
> (image=? (circle 5 "solid" "green") c1)
#false
> (image=? (circle 10 "solid" "green") c1)
#true
```

(image? x) → boolean?
  x : any/c

Determines whether a value is an image.

```
> c1
```

```
> (image? c1)
#true
```

5.15 Misc

(=~ x y eps) → boolean?
  x : number
  y : number
  eps : non-negative-real

Checks whether *x* and *y* are within *eps* of either other.

```
> (=~ 1.01 1.0 0.1)
#true
> (=~ 1.01 1.5 0.1)
#false
```

(current-milliseconds) → exact-integer

Returns the current "time" in fixnum milliseconds (possibly negative).

```
> (current-milliseconds)
1793426336930
```

```
eof : eof-object?
```

A value that represents the end of a file:

```
> eof
#<eof>
```

```
(eof-object? x) → boolean?
  x : any/c
```

Determines whether some value is the end-of-file value.

```
> (eof-object? eof)
#true
> (eof-object? 42)
#false
```

```
(eq? x y) → boolean?
  x : any/c
  y : any/c
```

Determines whether two values are equivalent from the computer's perspective (intensional).

```
> (eq? (cons 1 '()) (cons 1 '()))
#false
> one
(list 1)
> (eq? one one)
#true
```

```
(equal? x y) → boolean?
  x : any/c
  y : any/c
```

Determines whether two values are structurally equal where basic values are compared with the eqv? predicate.

```
> (equal? (make-posn 1 2) (make-posn (- 2 1) (+ 1 1)))
#true
```

```
(equal~? x y z) → boolean?
  x : any/c
  y : any/c
  z : non-negative-real
```

Compares *x* and *y* like `equal?` but uses =~ in the case of numbers.

```
> (equal~? (make-posn 1.01 1.0) (make-posn 1.01 0.99) 0.2)
#true
```

```
(eqv? x y) → boolean?
  x : any/c
  y : any/c
```

Determines whether two values are equivalent from the perspective of all functions that can be applied to it (extensional).

```
> (eqv? (cons 1 '()) (cons 1 '()))
#false
> one
(list 1)
> (eqv? one one)
#true
```

```
(error x ...) → void?
  x : any/c
```

Signals an error, combining the given values into an error message. If any of the values' printed representations is too long, it is truncated and "..." is put into the string. If the first value is a symbol, it is suffixed with a colon and the result pre-pended on to the error message.

```
> zero
0
> (if (= zero 0) (error "can't divide by 0") (/ 1 zero))
can't divide by 0
```

```
(exit) → void
```

Evaluating `(exit)` terminates the running program.

```
(force v) → any
  v : any
```

Finds the delayed value; see also delay.

```
(gensym) → symbol?
```

Generates a new symbol, different from all symbols in the program.

```
> (gensym)
'g1328366
```

```
(identity x) → any/c
  x : any/c
```

Returns x.

```
> (identity 42)
42
> (identity c1)
```

```
> (identity "hello")
"hello"
```

```
(promise? x) → boolean?
  x : any
```

Determines if a value is delayed.

```
(sleep sec) → void
  sec : positive-num
```

Causes the program to sleep for the given number of seconds.

```
(struct? x) → boolean?
  x : any/c
```

Determines whether some value is a structure.

```
> (struct? (make-posn 1 2))
#true
> (struct? 43)
#false
```

```
(void) → void?
```

Produces a void value.

```
> (void)
```

```
(void? x) → boolean?
  x : any
```

Determines if a value is void.

```
> (void? (void))
#true
> (void? 42)
#false
```

5.16 Signatures

```
Any : signature?
```

Signature for any value.

```
Boolean : signature?
```

Signature for booleans.

```
Char : signature?
```

Signature for chararacters.

```
(ConsOf first-sig rest-sig) → signature?
  first-sig : signature?
  rest-sig : signature?
```

Signature for a cons pair.

```
EmptyList : signature?
```

Signature for the empty list.

```
False : signature?
```

Signature for just false.

```
Integer : signature?
```

Signature for integers.

```
Natural : signature?
```

Signature for natural numbers.

```
Number : signature?
```

Signature for arbitrary numbers.

```
Rational : signature?
```

Signature for rational numbers.

```
Real : signature?
```

Signature for real numbers.

```
String : signature?
```

Signature for strings.

```
Symbol : signature?
```

Signature for symbols.

```
True : signature?
```

Signature for just true.

5.17 Numbers (relaxed conditions)

5.18 String (relaxed conditions)

```
(string-append s ...) → string
  s : string
```

Concatenates the characters of several strings.

```
> (string-append "hello" " " "world" " " "good bye")
"hello world good bye"
```

```
(string-ci<=? s t x ...) → boolean?
  s : string
  t : string
  x : string
```

Determines whether the strings are ordered in a lexicographically increasing and case-insensitive manner.

```
> (string-ci<=? "hello" "WORLD" "zoo")
#true
```

```
(string-ci<? s t x ...) → boolean?
  s : string
  t : string
  x : string
```

Determines whether the strings are ordered in a lexicographically strictly increasing and case-insensitive manner.

```
> (string-ci<? "hello" "WORLD" "zoo")
#true
```

```
(string-ci=? s t x ...) → boolean?
  s : string
  t : string
  x : string
```

Determines whether all strings are equal, character for character, regardless of case.

```
> (string-ci=?  "hello" "HellO")
#true
```

```
(string-ci>=? s t x ...) → boolean?
  s : string
  t : string
  x : string
```

Determines whether the strings are ordered in a lexicographically decreasing and case-insensitive manner.

```
> (string-ci>?  "zoo" "WORLD" "hello")
#true
```

```
(string-ci>? s t x ...) → boolean?
  s : string
  t : string
  x : string
```

Determines whether the strings are ordered in a lexicographically strictly decreasing and case-insensitive manner.

```
> (string-ci>?  "zoo" "WORLD" "hello")
#true
```

```
(string<=? s t x ...) → boolean?
  s : string
  t : string
  x : string
```

Determines whether the strings are ordered in a lexicographically increasing manner.

```
> (string<=? "hello" "hello" "world" "zoo")
#true
```

```
(string<? s t x ...) → boolean?
  s : string
  t : string
  x : string
```

Determines whether the strings are ordered in a lexicographically strictly increasing manner.

```
> (string<? "hello" "world" "zoo")
#true
```

```
(string=? s t x ...) → boolean?
  s : string
  t : string
  x : string
```

Determines whether all strings are equal, character for character.

```
> (string=? "hello" "world")
#false
> (string=? "bye" "bye")
#true
```

```
(string>=? s t x ...) → boolean?
  s : string
  t : string
  x : string
```

Determines whether the strings are ordered in a lexicographically decreasing manner.

```
> (string>=?  "zoo" "zoo" "world" "hello")
#true
```

```
(string>? s t x ...) → boolean?
  s : string
  t : string
  x : string
```

Determines whether the strings are ordered in a lexicographically strictly decreasing manner.

```
> (string>?  "zoo" "world" "hello")
#true
```

5.19 Posn

```
Posn : signature
```

Signature for posns.

```
(PosnOf x-sig y-sig) → signature
  x-sig : signature
  y-sig : signature
```

Creates a parametric signature for posns from signatures for its fields.

```
posn : struct
```

Name for using `match` with posns.

5.20 Higher-Order Functions

5.21 Numbers (relaxed conditions plus)

```
(* x ...) → number
  x : number
```

Multiplies all given numbers. In ISL and up: `*` works when applied to only one number or none.

```
> (* 5 3)
15
> (* 5 3 2)
30
> (* 2)
2
> (*)
1
```

```
(+ x ...) → number
  x : number
```

Adds all given numbers. In ISL and up: `+` works when applied to only one number or none.

```
> (+ 2/3 1/16)
35/48
> (+ 3 2 5 8)
18
> (+ 1)
1
> (+)
0
```

```
(/ x y ...) → number
  x : number
  y : number
```

Divides the first by all remaining numbers. In ISL and up: / computes the inverse when applied to one number.

```
> (/ 12 2)
6
> (/ 12 2 3)
2
> (/ 3)
1/3
```

```
(= x ...) → number
  x : number
```

Compares numbers for equality. In ISL and up: = works when applied to only one number.

```
> (= 10 10)
#true
> (= 11)
#true
> (= 0)
#true
```

5.22 Higher-Order Functions (with Lambda)

```
(andmap p? [l]) → boolean
  p? : (X ... -> boolean)
  l : (listof X) = ...
```

Determines whether *p?* holds for all items of *l* ...:

```
(andmap p (list x-1 ... x-n)) = (and (p x-1) ... (p x-n))
```

```
(andmap p (list x-1 ... x-n) (list y-1 ... y-n)) = (and (p x-1 y-
1) ... (p x-n y-n))
```

```
> (andmap odd? '(1 3 5 7 9))
#true
> threshold
3
> (andmap (lambda (x) (< x threshold)) '(0 1 2))
#true
> (andmap even? '())
#true
> (andmap (lambda (x f) (f x)) (list 0 1 2) (list odd? even? positive?))
#false
```

```
(apply f x-1 ... l) → Y
  f : (X-1 ... X-N -> Y)
  x-1 : X-1
  l : (list X-i+1 ... X-N)
```

Applies a function using items from a list as the arguments:

```
(apply f (list x-1 ... x-n)) = (f x-1 ... x-n)
```

```
> a-list
(list 0 1 2 3 4 5 6 7 8 9)
> (apply max a-list)
9
```

```
(argmax f l) → X
  f : (X -> real)
  l : (listof X)
```

Finds the (first) element of the list that maximizes the output of the function.

```
> (argmax second '((sam 98) (carl 78) (vincent 93) (asumu 99)))
(list 'asumu 99)
```

```
(argmin f l) → X
  f : (X -> real)
  l : (listof X)
```

Finds the (first) element of the list that minimizes the output of the function.

```
> (argmin second '((sam 98) (carl 78) (vincent 93) (asumu 99)))
(list 'carl 78)
```

```
(build-list n f) → (listof X)
  n : nat
  f : (nat -> X)
```

Constructs a list by applying *f* to the numbers between 0 and (- *n* 1):

```
(build-list n f) = (list (f 0) ... (f (- n 1)))
```

```
> (build-list 22 add1)
(list 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22)
> i
3
> (build-list 3 (lambda (j) (+ j i)))
(list 3 4 5)
> (build-list 5
              (lambda (i)
                (build-list 5
                            (lambda (j)
                              (if (= i j) 1 0)))))
(list (list 1 0 0 0 0) (list 0 1 0 0 0) (list 0 0 1 0 0) (list 0 0
0 1 0) (list 0 0 0 0 1))
```

```
(build-string n f) → string
  n : nat
  f : (nat -> char)
```

Constructs a string by applying *f* to the numbers between 0 and (- *n* 1):

```
(build-string n f) = (string (f 0) ... (f (- n 1)))
```

```
> (build-string 10 integer->char)
"\u0000\u0001\u0002\u0003\u0004\u0005\u0006\a\b\t"
> (build-string 26 (lambda (x) (integer->char (+ 65 x))))
"ABCDEFGHIJKLMNOPQRSTUVWXYZ"
```

```
(compose f g) → (X -> Z)
  f : (Y -> Z)
  g : (X -> Y)
```

Composes a sequence of procedures into a single procedure:

```
(compose f g) = (lambda (x) (f (g x)))
```

```
> ((compose add1 second) '(add 3))
4
> (map (compose add1 second) '((add 3) (sub 2) (mul 4)))
(list 4 3 5)
```

```
(filter p? l) → (listof X)
  p? : (X -> boolean)
  l : (listof X)
```

Constructs a list from all those items on a list for which the predicate holds.

```
> (filter odd? '(0 1 2 3 4 5 6 7 8 9))
(list 1 3 5 7 9)
> threshold
3
> (filter (lambda (x) (>= x threshold)) '(0 1 2 3 4 5 6 7 8 9))
(list 3 4 5 6 7 8 9)
```

```
(foldl f base l ...) → Y
  f : (X ... Y -> Y)
  base : Y
  l : (listof X)
```

```
(foldl f base (list x-1 ... x-n)) = (f x-n ... (f x-1 base))
```

```
(foldl f base (list x-1 ... x-n) (list x-1 ... x-n))
= (f x-n y-n ... (f x-1 y-1 base))
```

```
> (foldl + 0 '(0 1 2 3 4 5 6 7 8 9))
45
> a-list
(list 0 1 2 3 4 5 6 7 8 9)
> (foldl (lambda (x r) (if (> x threshold) (cons (* 2 x) r) r)) '() a-
list)
(list 18 16 14 12 10 8)
> (foldl (lambda (x y r) (+ x y r)) 0 '(1 2 3) '(10 11 12))
39
```

```
(foldr f base l ...) → Y
  f : (X ... Y -> Y)
  base : Y
  l : (listof X)
```

```
(foldr f base (list x-1 ... x-n)) = (f x-1 ... (f x-n base))
```

```
(foldr f base (list x-1 ... x-n) (list y-1 ... y-n))
= (f x-1 y-1 ... (f x-n y-n base))
```

```
> (foldr + 0 '(0 1 2 3 4 5 6 7 8 9))
45
> a-list
(list 0 1 2 3 4 5 6 7 8 9)
> (foldr (lambda (x r) (if (> x threshold) (cons (* 2 x) r) r)) '() a-
list)
(list 8 10 12 14 16 18)
> (foldr (lambda (x y r) (+ x y r)) 0 '(1 2 3) '(10 11 12))
39
```

```
(map f l ...) → (listof Z)
  f : (X ... -> Z)
  l : (listof X)
```

Constructs a new list by applying a function to each item on one or more existing lists:

```
(map f (list x-1 ... x-n)) = (list (f x-1) ... (f x-n))
```

```
(map f (list x-1 ... x-n) (list y-1 ... y-n)) = (list (f x-1 y-
1) ... (f x-n y-n))
```

```
> (map add1 (list 3 -4.01 2/5))
(list 4 #i-3.01 1.4)
```

```
> (define (tag-with-a x)
    (list "a" (+ x 1)))
```
tag-with-a:this name was defined previously and cannot be re-defined

```
> (map tag-with-a (list 3 -4.01 2/5))
(list (list "a" 4) (list "a" #i-3.01) (list "a" 1.4))
```

```
> (define (add-and-multiply x y)
    (+ x (* x y)))
```
add-and-multiply:this name was defined previously and cannot be re-defined

```
> (map add-and-multiply (list 3 -4 2/5) '(1 2 3))
(list 6 -12 1.6)
```

```
(memf p? l) → (union #false (listof X))
  p? : (X -> any)
  l : (listof X)
```

Produces `#false` if *p?* produces `false` for all items on *l*. If *p?* produces `#true` for any of the items on *l*, `memf` returns the sub-list starting from that item.

```
> (memf odd? '(2 4 6 3 8 0))
(list 3 8 0)
```

```
(ormap p? l) → boolean
  p? : (X -> boolean)
  l : (listof X)
```

Determines whether *p?* holds for at least one items of *l*:

```
(ormap p (list x-1 ... x-n)) = (or (p x-1) ... (p x-n))
```

```
(ormap p (list x-1 ... x-n) (list y-1 ... y-n)) = (or (p x-1 y-
1) ... (p x-n y-n))
```

```
> (ormap odd? '(1 3 5 7 9))
#true
> threshold
3
> (ormap (lambda (x) (< x threshold)) '(6 7 8 1 5))
#true
> (ormap even? '())
#false
> (ormap (lambda (x f) (f x)) (list 0 1 2) (list odd? even? positive?))
#true
```

```
(procedure? x) → boolean?
  x : any
```

Produces true if the value is a procedure.

```
> (procedure? cons)
#true
> (procedure? add1)
#true
> (procedure? (lambda (x) (> x 22)))
#true
```

```
(quicksort l comp) → (listof X)
  l : (listof X)
  comp : (X X -> boolean)
```

Sorts the items on *l*, in an order according to *comp* (using the quicksort algorithm).

```
> (quicksort '(6 7 2 1 3 4 0 5 9 8) <)
(list 0 1 2 3 4 5 6 7 8 9)
```

```
(sort l comp) → (listof X)
  l : (listof X)
  comp : (X X -> boolean)
```

Sorts the items on *l*, in an order according to *comp*.

```
> (sort '(6 7 2 1 3 4 0 5 9 8) <)
(list 0 1 2 3 4 5 6 7 8 9)
```

5.23 Reading and Printing

```
(display x) → void
  x : any
```

Prints the argument to stdout (without quotes on symbols and strings, etc.).

```
> (display 10)
10
> (display "hello")
hello
> (display 'hello)
hello
```

```
(newline) → void
```

Prints a newline.

```
(pretty-print x) → void
  x : any
```

Pretty prints S-expressions (like `write`).

```
> (pretty-print '((1 2 3) ((a) ("hello world" #true) (((false "good
bye"))))))
((1 2 3) ((a) ("hello world" #true) (((false "good bye")))))
> (pretty-print (build-list 10 (lambda (i) (build-
list 10 (lambda (j) (= i j))))))
((#true #false #false #false #false #false #false #false #false
#false)
 (#false #true #false #false #false #false #false #false #false
#false)
 (#false #false #true #false #false #false #false #false #false
#false)
 (#false #false #false #true #false #false #false #false #false
#false)
 (#false #false #false #false #true #false #false #false #false
#false)
 (#false #false #false #false #false #true #false #false #false
#false)
 (#false #false #false #false #false #false #true #false #false
#false)
 (#false #false #false #false #false #false #false #true #false
#false)
 (#false #false #false #false #false #false #false #false #true
#false)
 (#false #false #false #false #false #false #false #false #false
#true))
```

```
(print x) → void
  x : any
```

Prints the argument as a value.

```
> (print 10)
10
```

```
> (print "hello")
"hello"
> (print 'hello)
'hello
```

```
(printf f x ...) → void
  f : string
  x : any
```

Formats the rest of the arguments according to the first argument and print it.

```
(read) → sexp
```

Reads input from the user.

```
(with-input-from-file f p) → any
  f : string
  p : (-> any)
```

Opens the named input file *f* and allows *p* to read from it.

```
(with-input-from-string s p) → any
  s : string
  p : (-> any)
```

Turns *s* into input for `read` operations in *p*.

```
> (with-input-from-string "hello" read)
'hello
> (string-length (symbol->string (with-input-from-
string "hello" read)))
5
```

```
(with-output-to-file f p) → any
  f : string
  p : (-> any)
```

Opens the named output file *f* and allows *p* to write to it.

```
(with-output-to-string p) → any
  p : (-> any)
```

Produces a string from all write/display/print operations in *p*.

```
> (with-output-to-string (lambda () (display 10)))
"10"
```

```
(write x) → void
  x : any
```

Prints the argument to stdout (in a traditional style that is somewhere between `print` and `display`).

```
> (write 10)
10
> (write "hello")
"hello"
> (write 'hello)
hello
```

5.24 Vectors

```
(build-vector n f) → (vectorof X)
  n : nat
  f : (nat -> X)
```

Constructs a vector by applying *f* to the numbers `0` through `(- n 1)`.

```
> (build-vector 5 add1)
(vector 1 2 3 4 5)
```

```
(list->vector l) → (vectorof X)
  l : (listof X)
```

Transforms *l* into a vector.

```
> (list->vector (list "hello" "world" "good" "bye"))
(vector "hello" "world" "good" "bye")
```

```
(make-vector n x) → (vectorof X)
  n : number
  x : X
```

Constructs a vector of n copies of x.

```
> (make-vector 5 0)
(vector 0 0 0 0 0)
```

```
(vector x ...) → (vector X ...)
  x : X
```

Constructs a vector from the given values.

```
> (vector 1 2 3 -1 -2 -3)
(vector 1 2 3 -1 -2 -3)
```

```
(vector->list v) → (listof X)
  v : (vectorof X)
```

Transforms v into a list.

```
> (vector->list (vector 'a 'b 'c))
(list 'a 'b 'c)
```

```
(vector-length v) → nat
  v : (vector X)
```

Determines the length of v.

```
> v
(vector "a" "b" "c" "d" "e")
> (vector-length v)
5
```

```
(vector-ref v n) → X
  v : (vector X)
  n : nat
```

Extracts the nth element from v.

```
> v
(vector "a" "b" "c" "d" "e")
> (vector-ref v 3)
"d"
```

```
(vector-set! v n x) → void
  v : (vectorof X)
  n : nat
  x : X
```

Updates v at position n to be x.

```
> v
(vector "a" "b" "c" "d" "e")
> (vector-set! v 3 77)
> v
(vector "a" "b" "c" 77 "e")
```

```
(vector? x) → boolean
  x : any
```

Determines if a value is a vector.

```
> v
(vector "a" "b" "c" 77 "e")
> (vector? v)
#true
> (vector? 42)
#false
```

5.25 Boxes

```
(box x) → box?
  x : any/c
```

Constructs a box.

```
> (box 42)
(box 42)
```

```
(box? x) → boolean?
  x : any/c
```

Determines if a value is a box.

```
> b
(box 33)
> (box? b)
#true
> (box? 42)
#false
```

```
(set-box! b x) → void
  b : box?
  x : any/c
```

Updates a box.

```
> b
(box 33)
> (set-box! b 31)
> b
(box 31)
```

```
(unbox b) → any
  b : box?
```

Extracts the boxed value.

```
> b
(box 31)
> (unbox b)
31
```

5.26 Hash Tables

```
(hash-copy h) → hash
  h : hash
```

Copies a hash table.

```
(hash-count h) → integer
  h : hash
```

Determines the number of keys mapped by a hash table.

```
> ish
(make-immutable-hash (list (list 'b 69) (list 'r 999) (list 'c 42)
(list 'e 61)))
> (hash-count ish)
4
```

```
(hash-eq? h) → boolean
  h : hash
```

Determines if a hash table uses eq? for comparisons.

```
> hsh
(make-hash (list (list 'c 42) (list 'r 999) (list 'b 69) (list 'e
61)))
> (hash-eq? hsh)
#false
> heq
(make-hasheq (list (list 'r 999) (list 'e 61) (list 'c 42) (list
'b 69)))
> (hash-eq? heq)
#true
```

```
(hash-equal? h) → boolean
  h : hash?
```

Determines if a hash table uses equal? for comparisons.

```
> ish
(make-immutable-hash (list (list 'b 69) (list 'r 999) (list 'c 42)
(list 'e 61)))
> (hash-equal? ish)
#true
> ieq
(make-immutable-hasheq (list (list 'b 69) (list 'r 999) (list 'c
42) (list 'e 61)))
> (hash-equal? ieq)
#false
```

```
(hash-eqv? h) → boolean
  h : hash
```

Determines if a hash table uses eqv? for comparisons.

```
> heq
(make-hasheq (list (list 'r 999) (list 'e 61) (list 'c 42) (list
'b 69)))
> (hash-eqv? heq)
#false
> heqv
(make-hasheqv (list (list 'r 999) (list 'e 61) (list 'c 42) (list
'b 69)))
> (hash-eqv? heqv)
#true
```

```
(hash-for-each h f) → void?
  h : (hash X Y)
  f : (X Y -> any)
```

Applies a function to each mapping of a hash table for effect only.

```
> hsh
(make-hash (list (list 'c 42) (list 'r 999) (list 'b 69) (list 'e
61)))
> (hash-for-each hsh (lambda (ky vl) (hash-set! hsh ky (+ vl 1))))
> hsh
(make-hash (list (list 'c 43) (list 'r 1000) (list 'b 70) (list 'e
62)))
```

```
(hash-has-key? h x) → boolean
  h : (hash X Y)
  x : X
```

Determines if a key is associated with a value in a hash table.

```
> ish
(make-immutable-hash (list (list 'b 69) (list 'r 999) (list 'c 42)
(list 'e 61)))
> (hash-has-key? ish 'b)
#true
> hsh
```

```
(make-hash (list (list 'c 43) (list 'r 1000) (list 'b 70) (list 'e
62)))
> (hash-has-key? hsh 'd)
#false
```

```
(hash-map h f) → (listof Z)
  h : (hash X Y)
  f : (X Y -> Z)
```

Constructs a new list by applying a function to each mapping of a hash table.

```
> ish
(make-immutable-hash (list (list 'b 69) (list 'r 999) (list 'c 42)
(list 'e 61)))
> (hash-map ish list)
(list (list 'b 69) (list 'r 999) (list 'c 42) (list 'e 61))
```

```
(hash-ref h k) → Y
  h : (hash X Y)
  k : X
```

Extracts the value associated with a key from a hash table; the three argument case allows a default value or default value computation.

```
> hsh
(make-hash (list (list 'c 43) (list 'r 1000) (list 'b 70) (list 'e
62)))
> (hash-ref hsh 'b)
70
```

```
(hash-ref! h k v) → Y
  h : (hash X Y)
  k : X
  v : Y
```

Extracts the value associated with a key from a mutable hash table; if the key does not have an mapping, the third argument is used as the value (or used to compute the value) and is added to the hash table associated with the key.

```
> hsh
(make-hash (list (list 'c 43) (list 'r 1000) (list 'b 70) (list 'e
62)))
```

```
> (hash-ref! hsh 'd 99)
99
> hsh
(make-hash (list (list 'c 43) (list 'd 99) (list 'r 1000) (list 'b
70) (list 'e 62)))
```

(hash-remove *h* *k*) → (hash X Y)
  h : (hash X Y)
  k : X

Constructs an immutable hash table with one less mapping than an existing immutable hash table.

```
> ish
(make-immutable-hash (list (list 'b 69) (list 'r 999) (list 'c 42)
(list 'e 61)))
> (hash-remove ish 'b)
(make-immutable-hash (list (list 'c 42) (list 'r 999) (list 'e
61)))
```

(hash-remove! *h* *x*) → void
  h : (hash X Y)
  x : X

Removes an mapping from a mutable hash table.

```
> hsh
(make-hash (list (list 'c 43) (list 'd 99) (list 'r 1000) (list 'b
70) (list 'e 62)))
> (hash-remove! hsh 'r)
> hsh
(make-hash (list (list 'c 43) (list 'd 99) (list 'b 70) (list 'e
62)))
```

(hash-set *h* *k* *v*) → (hash X Y)
  h : (hash X Y)
  k : X
  v : Y

Constructs an immutable hash table with one new mapping from an existing immutable hash table.

```
> (hash-set ish 'a 23)
(make-immutable-hash (list (list 'b 69) (list 'r 999) (list 'c 42)
(list 'a 23) (list 'e 61)))
```

```
(hash-set! h k v) → void?
  h : (hash X Y)
  k : X
  v : Y
```

Updates a mutable hash table with a new mapping.

```
> hsh
(make-hash (list (list 'c 43) (list 'd 99) (list 'b 70) (list 'e
62)))
> (hash-set! hsh 'a 23)
> hsh
(make-hash (list (list 'c 43) (list 'a 23) (list 'd 99) (list 'b
70) (list 'e 62)))
```

```
(hash-update h k f) → (hash X Y)
  h : (hash X Y)
  k : X
  f : (Y -> Y)
```

Composes hash-ref and hash-set to update an existing mapping; the third argument is used to compute the new mapping value; the fourth argument is used as the third argument to hash-ref.

```
> (hash-update ish 'b (lambda (old-b) (+ old-b 1)))
(make-immutable-hash (list (list 'b 70) (list 'r 999) (list 'c 42)
(list 'e 61)))
```

```
(hash-update! h k f) → void?
  h : (hash X Y)
  k : X
  f : (Y -> Y)
```

Composes hash-ref and hash-set! to update an existing mapping; the third argument is used to compute the new mapping value; the fourth argument is used as the third argument to hash-ref.

```
> hsh
(make-hash (list (list 'c 43) (list 'a 23) (list 'd 99) (list 'b
70) (list 'e 62)))
> (hash-update! hsh 'b (lambda (old-b) (+ old-b 1)))
> hsh
(make-hash (list (list 'c 43) (list 'a 23) (list 'd 99) (list 'b
71) (list 'e 62)))
```

(hash? *x*) → boolean
  x : any

Determines if a value is a hash table.

```
> ish
(make-immutable-hash (list (list 'b 69) (list 'r 999) (list 'c 42)
(list 'e 61)))
> (hash? ish)
#true
> (hash? 42)
#false
```

(make-hash) → (hash X Y)

Constructs a mutable hash table from an optional list of mappings that uses equal? for comparisons.

```
> (make-hash)
(make-hash)
> (make-hash '((b 69) (e 61) (i 999)))
(make-hash (list (list 'i 999) (list 'b 69) (list 'e 61)))
```

(make-hasheq) → (hash X Y)

Constructs a mutable hash table from an optional list of mappings that uses eq? for comparisons.

```
> (make-hasheq)
(make-hasheq)
> (make-hasheq '((b 69) (e 61) (i 999)))
(make-hasheq (list (list 'e 61) (list 'i 999) (list 'b 69)))
```

(make-hasheqv) → (hash X Y)

Constructs a mutable hash table from an optional list of mappings that uses eqv? for comparisons.

```
> (make-hasheqv)
(make-hasheqv)
> (make-hasheqv '((b 69) (e 61) (i 999)))
(make-hasheqv (list (list 'e 61) (list 'i 999) (list 'b 69)))
```

(make-immutable-hash) → (hash X Y)

Constructs an immutable hash table from an optional list of mappings that uses equal? for comparisons.

```
> (make-immutable-hash)
(make-immutable-hash)
> (make-immutable-hash '((b 69) (e 61) (i 999)))
(make-immutable-hash (list (list 'b 69) (list 'e 61) (list 'i
999)))
```

(make-immutable-hasheq) → (hash X Y)

Constructs an immutable hash table from an optional list of mappings that uses eq? for comparisons.

```
> (make-immutable-hasheq)
(make-immutable-hasheq)
> (make-immutable-hasheq '((b 69) (e 61) (i 999)))
(make-immutable-hasheq (list (list 'b 69) (list 'e 61) (list 'i
999)))
```

(make-immutable-hasheqv) → (hash X Y)

Constructs an immutable hash table from an optional list of mappings that uses eqv? for comparisons.

```
> (make-immutable-hasheqv)
(make-immutable-hasheqv)
> (make-immutable-hasheqv '((b 69) (e 61) (i 999)))
(make-immutable-hasheqv (list (list 'b 69) (list 'e 61) (list 'i
999)))
```