How to Design Programs Languages

Version 4.0

June 11, 2008

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

Contents

1 Beginning Student

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

def-or-expr = definition
            | expr
            | library-require

 definition = (define (id id id ...) expr)
            | (define id expr)
            | (define id (lambda (id id ...) expr))
            | (define-struct id (id ...))

       expr = (id expr expr ...) ; function call
            | (prim-op expr ...) ; primitive operation call
            | (cond [expr expr] ... [expr expr])
            | (cond [expr expr] ... [else expr])
            | (if expr expr expr)
            | (and expr expr expr ...)
            | (or expr expr expr ...)
            | test-case
            | empty
            | id
            | id ; identifier
            | 'id ; symbol
            | number
            | true
            | false
            | string
            | character

  test-case = (check-expect expr expr)
            | (check-within expr expr expr)
            | (check-error expr expr)

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

    package = (string string number number)
```

An *id* 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 *string* is enclosed by a pair of ". Unlike symbols, strings may be split into characters and manipulated by a variety of primitive 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.

A *prim-op* is one of:

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

```
* : (num num num ... -> num)
+ : (num num num ... -> num)
- : (num num ... -> num)
/ : (num num num ... -> num)
< : (real real real ... -> boolean)
<= : (real real real ... -> boolean)
= : (num num num ... -> boolean)
> : (real real real ... -> boolean)
>= : (real real ... -> boolean)
abs : (real -> real)
acos : (num -> num)
add1 : (number -> number)
angle : (num -> real)
asin : (num -> num)
atan : (num -> num)
ceiling : (real -> int)
complex? : (any -> boolean)
conjugate : (num -> num)
cos : (num -> num)
cosh : (num -> num)
current-seconds : (-> int)
denominator : (rat -> int)
e : real
even? : (integer -> boolean)
exact->inexact : (num -> num)
exact? : (num -> boolean)
exp : (num -> num)
expt : (num num -> num)
floor : (real -> int)
gcd : (int int ... -> int)
imag-part : (num -> real)
inexact->exact : (num -> num)
inexact? : (num -> boolean)
integer->char : (int -> char)
integer? : (any -> boolean)
lcm : (int int ... -> int)
log : (num -> num)
```

```
magnitude : (num -> real)
make-polar : (real real -> num)
max : (real real ... -> real)
min : (real real ... -> real)
modulo : (int int -> int)
negative? : (number -> boolean)
number->string : (num -> string)
number? : (any -> boolean)
numerator : (rat -> int)
odd? : (integer -> boolean)
pi : real
positive? : (number -> boolean)
quotient : (int int -> int)
random : (int -> int)
rational? : (any -> boolean)
real-part : (num -> real)
real? : (any -> boolean)
remainder : (int int -> int)
round : (real -> int)
sgn : (real -> (union 1 1.0 0 0.0 -1 -1.0))
sin : (num -> num)
sinh : (num -> num)
sqr : (num -> num)
sqrt : (num -> num)
sub1 : (number -> number)
tan : (num -> num)
zero? : (number -> boolean)
```

Booleans

```
boolean=? : (boolean boolean -> boolean)
boolean? : (any -> boolean)
false? : (any -> boolean)
not : (boolean -> boolean)
```

Symbols

```
symbol->string : (symbol -> string)
symbol=? : (symbol symbol -> boolean)
symbol? : (any -> boolean)
```

Lists

```
append : ((listof any)
          (listof any)
          (listof any)
          ...
          ->
          (listof any))
```

```
assq : (X
        (listof (cons X Y))
        ->
        (union false (cons X Y)))
caaar : ((cons
          (cons (cons W (listof Z)) (listof Y))
          (listof X))
         ->
         W)
caadr : ((cons
          (cons (cons W (listof Z)) (listof Y))
          (listof X))
         ->
         (listof Z))
caar : ((cons (cons Z (listof Y)) (listof X)) -> Z)
cadar : ((cons (cons W (cons Z (listof Y))) (listof X))
         ->
         Z)
cadddr : ((listof Y) -> Y)
caddr : ((cons W (cons Z (cons Y (listof X)))) -> Y)
cadr : ((cons Z (cons Y (listof X))) -> Y)
car : ((cons Y (listof X)) -> Y)
cdaar : ((cons
          (cons (cons W (listof Z)) (listof Y))
          (listof X))
         ->
         (listof Z))
cdadr : ((cons W (cons (cons Z (listof Y)) (listof X)))
         ->
         (listof Y))
cdar : ((cons (cons Z (listof Y)) (listof X))
        ->
        (listof Y))
cddar : ((cons (cons W (cons Z (listof Y))) (listof X))
         ->
         (listof Y))
cdddr : ((cons W (cons Z (cons Y (listof X))))
         ->
         (listof X))
cddr : ((cons Z (cons Y (listof X))) -> (listof X))
cdr : ((cons Y (listof X)) -> (listof X))
cons : (X (listof X) -> (listof X))
cons? : (any -> boolean)
eighth : ((listof Y) -> Y)
empty? : (any -> boolean)
fifth : ((listof Y) -> Y)
```

```
first : ((cons Y (listof X)) -> Y)
fourth : ((listof Y) -> Y)
length : (list -> number)
list : (any ... -> (listof any))
list* : (any ... (listof any) -> (listof any))
list-ref : ((listof X) natural-number -> X)
member : (any list -> (union false list))
memq : (any list -> (union false list))
memv : (any list -> (union false list))
null : empty
null? : (any -> boolean)
pair? : (any -> boolean)
rest : ((cons Y (listof X)) -> (listof X))
reverse : (list -> list)
second : ((cons Z (cons Y (listof X))) -> Y)
seventh : ((listof Y) -> Y)
sixth : ((listof Y) -> Y)
third : ((cons W (cons Z (cons Y (listof X)))) -> Y)
```

Posns

```
make-posn : (number number -> posn)
posn-x : (posn -> number)
posn-y : (posn -> number)
posn? : (anything -> boolean)
```

Characters

```
char->integer : (char -> integer)
char-alphabetic? : (char -> boolean)
char-ci<=? : (char char ... -> boolean)
char-ci<? : (char char ... -> boolean)
char-ci=? : (char char ... -> boolean)
char-ci>=? : (char char ... -> boolean)
char-ci>? : (char char ... -> boolean)
char-downcase : (char -> char)
char-lower-case? : (char -> boolean)
char-numeric? : (char -> boolean)
char-upcase : (char -> char)
char-upper-case? : (char -> boolean)
char-whitespace? : (char -> boolean)
char<=? : (char char ... -> boolean)
char<? : (char char ... -> boolean)
char=? : (char char ... -> boolean)
char>=? : (char char ... -> boolean)
char>? : (char char ... -> boolean)
char? : (any -> boolean)
```

Strings

```
format : (string any ... -> string)
list->string : ((listof char) -> string)
```

```
make-string : (nat char -> string)
string : (char ... -> string)
string->list : (string -> (listof char))
string->number : (string -> (union number false))
string->symbol : (string -> symbol)
string-append : (string ... -> string)
string-ci<=? : (string string ... -> boolean)
string-ci<? : (string string ... -> boolean)
string-ci=? : (string string ... -> boolean)
string-ci>=? : (string string ... -> boolean)
string-ci>? : (string string ... -> boolean)
string-copy : (string -> string)
string-length : (string -> nat)
string-ref : (string nat -> char)
string<=? : (string string ... -> boolean)
string<? : (string string ... -> boolean)
string=? : (string string ... -> boolean)
string>=? : (string string ... -> boolean)
string>? : (string string ... -> boolean)
string? : (any -> boolean)
substring : (string nat nat -> string)
```

Images

```
image=? : (image image -> boolean)
image? : (any -> boolean)
```

Misc

```
=~ : (real real non-negative-real -> boolean)
eof : eof
eof-object? : (any -> boolean)
eq? : (any any -> boolean)
equal? : (any any -> boolean)
equal~? : (any any non-negative-real -> boolean)
eqv? : (any any -> boolean)
error : (symbol string -> void)
exit : (-> void)
identity : (any -> any)
struct? : (any -> boolean)
```

1.1 `define`

`(define (id id id ...) expr)`

Defines a function. The first *id* inside the parentheses is the name of the function. All remaining *id*s are the names of the function's arguments. The *expr* is the body of the

function, evaluated whenever the function is called. The name of the function cannot be that of a primitive or another definition.

`(define id expr)`

Defines a constant *id* as a synonym for the value produced by *expr*. The defined name cannot be that of a primitive or another definition, and *id* itself must not appear in *expr*.

`(define id (lambda (id id ...) expr))`

An alternate form for defining functions. The first *id* is the name of the function. The *id*s in parentheses are the names of the function's arguments, and the *expr* is the body of the function, which evaluated whenever the function is called. The name of the function cannot be that of a primitive or another definition.

`lambda`

The `lambda` keyword can only be used with `define` in the alternative function-definition syntax.

1.2 `define-struct`

`(define-struct structid (fieldid ...))`

Define a new type of structure. The structure's fields are named by the *fieldid*s in parentheses. After evaluation of a define-struct form, a set of new primitives is available for creation, extraction, and type-like queries:

- `make-`*structid* : takes a number of arguments equal to the number of fields in the structure type, and creates a new instance of the structure type.
- *structid*`-`*fieldid* : takes an instance of the structure and returns the field named by *structid*.
- *structid*`?` : takes any value, and returns `true` if the value is an instance of the structure type.
- *structid* : an identifier representing the structure type, but never used directly.

The created names must not be the same as a primitive or another defined name.

1.3 Function Calls

```
(id expr expr ...)
```

Calls a function. The *id* must refer to a defined function, and the *expr*s are evaluated from left to right to produce the values that are passed as arguments to the function. The result of the function call is the result of evaluating the function's body with every instance of an argument name replaced by the value passed for that argument. The number of argument *expr*s must be the same as the number of arguments expected by the function.

```
(#%app id expr expr ...)
```

A function call can be written with `#%app`, though it's practically never written that way.

1.4 Primitive Calls

```
(prim-op expr ...)
```

Like a function call, but for a primitive operation. The *expr*s are evaluated from left to right, and passed as arguments to the primitive operation named by *prim-op*. A `define-struct` form creates new primitives.

1.5 `cond`

```
(cond [expr expr] ... [expr expr])
```

A `cond` form contains one or more "lines" that are surrounded by parentheses or square brackets. Each line contains two *expr*s: a question *expr* and an answer *expr*.

The lines are considered in order. To evaluate a line, first evaluate the question *expr*. If the result is `true`, then the result of the whole `cond` expression is the result of evaluating the answer *expr* of the same line. If the result of evaluating the question *expr* is `false`, the line is discarded and evaluation proceeds with the next line.

If the result of a question *expr* is neither `true` nor `false`, it is an error. If none of the question *expr*s evaluates to `true`, it is also an error.

```
(cond [expr expr] ... [else expr])
```

This form of `cond` is similar to the prior one, except that the final `else` clause is always taken if no prior line's test expression evaluates to `true`. In other words, `else` acts like `true`, so there is no possibility to "fall off them end" of the `cond` form.

`else`

The `else` keyword can be used only with `cond`.

1.6 `if`

`(if` *`expr expr expr`*`)`

The first *`expr`* (known as the "test" *`expr`*) is evaluated. If it evaluates to `true`, the result of the `if` expression is the result of evaluating the second *`expr`* (often called the "then" *`expr`*). If the text *`expr`* evaluates to `false`, the result of the `if` expression is the result of evaluating the third *`expr`* (known as the "else" *`expr`*). If the result of evaluating the test *`expr`* is neither `true` nor `false`, it is an error.

1.7 `and`

`(and` *`expr expr expr ...`*`)`

The *`expr`*s are evaluated from left to right. If the first *`expr`* evaluates to `false`, the `and` expression immediately evaluates to `false`. If the first *`expr`* evaluates to `true`, the next expression is considered. If all *`expr`*s evaluate to `true`, the `and` expression evaluates to `true`. If any of the expressions evaluate to a value other than `true` or `false`, it is an error.

1.8 `or`

`(or` *`expr expr expr ...`*`)`

The *`expr`*s are evaluated from left to right. If the first *`expr`* evaluates to `true`, the `or` expression immediately evaluates to `true`. If the first *`expr`* evaluates to `false`, the next expression is considered. If all *`expr`*s evaluate to `false`, the `or` expression evaluates to `false`. If any of the expressions evaluate to a value other than `true` or `false`, it is an error.

1.9 Test Cases

`(check-expect` *expr* *expr*`)`

A test case to check that the first *expr* produces the same value as the second *expr*, where the latter is normally an immediate value.

`(check-within` *expr* *expr* *expr*`)`

Like `check-expect`, but with an extra expression that produces a number *delta*. The test case checks that each number in the result of the first *expr* is within *delta* of each corresponding number from the second *expr*.

`(check-error` *expr* *expr*`)`

A test case to check that the first *expr* signals an error, where the error messages matches the string produced by the second *expr*.

1.10 empty

`empty` : `empty?`

The empty list.

1.11 Identifiers

---

id

An *id* refers to a defined constant or argument within a function body. If no definition or argument matches the *id* name, an error is reported. Similarly, if *id* matches the name of a defined function or primitive operation, an error is reported.

1.12 Symbols

`'`*id*
`(quote` *id*`)`

A quoted *id* is a symbol. A symbol is a constant, like `0` and `empty`.

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

The *id* for a symbol is a sequence of characters not including a space or one of the following:

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

1.13 `true` and `false`

`true` : `boolean?`

The true value.

`false` : `boolean?`

The false value.

1.14 `require`

`(require` *string*`)`

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

Accesses a file in an installed library, making its definitions available in the current module (i.e., 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)))
```

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).

1.15 Primitive Operations

```
* : (num num num ... -> num)
```

Purpose: to compute the product of all of the input numbers

```
+ : (num num num ... -> num)
```

Purpose: to compute the sum of the input numbers

```
- : (num num ... -> num)
```

Purpose: to subtract the second (and following) number(s) from the first; negate the number if there is only one argument

---

```
/ : (num num num ... -> num)
```

Purpose: to divide the first by the second (and all following) number(s); only the first number can be zero.

```
< : (real real real ... -> boolean)
```

Purpose: to compare real numbers for less-than

```
<= : (real real real ... -> boolean)
```

Purpose: to compare real numbers for less-than or equality

```
= : (num num num ... -> boolean)
```

Purpose: to compare numbers for equality

```
> : (real real real ... -> boolean)
```

Purpose: to compare real numbers for greater-than

```
>= : (real real ... -> boolean)
```

Purpose: to compare real numbers for greater-than or equality

---

```
abs : (real -> real)
```

Purpose: to compute the absolute value of a real number

```
acos : (num -> num)
```

Purpose: to compute the arccosine (inverse of cos) of a number

---

```
add1 : (number -> number)
```

Purpose: to compute a number one larger than a given number

```
angle : (num -> real)
```

Purpose: to extract the angle from a complex number

```
asin : (num -> num)
```

Purpose: to compute the arcsine (inverse of sin) of a number

```
atan : (num -> num)
```

Purpose: to compute the arctan (inverse of tan) of a number

```
ceiling : (real -> int)
```

Purpose: to determine the closest integer above a real number

---

```
complex? : (any -> boolean)
```

Purpose: to determine whether some value is complex

`conjugate : (num -> num)`

Purpose: to compute the conjugate of a complex number

`cos : (num -> num)`

Purpose: to compute the cosine of a number (radians)

`cosh : (num -> num)`

Purpose: to compute the hyperbolic cosine of a number

`current-seconds : (-> int)`

Purpose: to compute the current time in seconds elapsed (since a platform-specific starting date)

`denominator : (rat -> int)`

Purpose: to compute the denominator of a rational

`e : real`

Purpose: Euler's number

`even? : (integer -> boolean)`

Purpose: to determine if some value is even or not

`exact->inexact : (num -> num)`

Purpose: to convert an exact number to an inexact one

`exact? : (num -> boolean)`

Purpose: to determine whether some number is exact

`exp : (num -> num)`

Purpose: to compute e raised to a number

```
expt : (num num -> num)
```

Purpose: to compute the power of the first to the second number

---

```
floor : (real -> int)
```

Purpose: to determine the closest integer below a real number

---

```
gcd : (int int ... -> int)
```

Purpose: to compute the greatest common divisior

```
imag-part : (num -> real)
```

Purpose: to extract the imaginary part from a complex number

```
inexact->exact : (num -> num)
```

Purpose: to approximate an inexact number by an exact one

```
inexact? : (num -> boolean)
```

Purpose: to determine whether some number is inexact

```
integer->char : (int -> char)
```

Purpose: to lookup the character that corresponds to the given integer in the ASCII table (if any)

```
integer? : (any -> boolean)
```

Purpose: to determine whether some value is an integer (exact or inexact)

```
lcm : (int int ... -> int)
```

Purpose: to compute the least common multiple of two integers

---

`log : (num -> num)`

Purpose: to compute the base-e logarithm of a number

`magnitude : (num -> real)`

Purpose: to determine the magnitude of a complex number

`make-polar : (real real -> num)`

Purpose: to create a complex from a magnitude and angle

`max : (real real ... -> real)`

Purpose: to determine the largest number

`min : (real real ... -> real)`

Purpose: to determine the smallest number

`modulo : (int int -> int)`

Purpose: to compute first number modulo second number

`negative? : (number -> boolean)`

Purpose: to determine if some value is strictly smaller than zero

---

`number->string : (num -> string)`

Purpose: to convert a number to a string

`number? : (any -> boolean)`

Purpose: to determine whether some value is a number

`numerator : (rat -> int)`

Purpose: to compute the numerator of a rational

```
odd? : (integer -> boolean)
```

Purpose: to determine if some value is odd or not

```
pi : real
```

Purpose: the ratio of a circle's circumference to its diameter

---

```
positive? : (number -> boolean)
```

Purpose: to determine if some value is strictly larger than zero

---

```
quotient : (int int -> int)
```

Purpose: to compute the quotient of two integers

```
random : (int -> int)
```

Purpose: to generate a random natural number less than some given integer

---

```
rational? : (any -> boolean)
```

Purpose: to determine whether some value is a rational number

---

```
real-part : (num -> real)
```

Purpose: to extract the real part from a complex number

```
real? : (any -> boolean)
```

Purpose: to determine whether some value is a real number

```
remainder : (int int -> int)
```

Purpose: to compute the remainder of dividing the first by the second integer

---

```
round : (real -> int)
```

Purpose: to round a real number to an integer (rounds to even to break ties)

```
sgn : (real -> (union 1 1.0 0 0.0 -1 -1.0))
```

Purpose: to compute the sign of a real number

```
sin : (num -> num)
```

Purpose: to compute the sine of a number (radians)

```
sinh : (num -> num)
```

Purpose: to compute the hyperbolic sine of a number

```
sqr : (num -> num)
```

Purpose: to compute the square of a number

```
sqrt : (num -> num)
```

Purpose: to compute the square root of a number

```
sub1 : (number -> number)
```

Purpose: to compute a number one smaller than a given number

```
tan : (num -> num)
```

Purpose: to compute the tangent of a number (radians)

```
zero? : (number -> boolean)
```

Purpose: to determine if some value is zero or not

```
boolean=? : (boolean boolean -> boolean)
```

Purpose: to determine whether two booleans are equal

```
boolean? : (any -> boolean)
```

Purpose: to determine whether some value is a boolean

```
false? : (any -> boolean)
```

Purpose: to determine whether a value is false

```
not : (boolean -> boolean)
```

Purpose: to compute the negation of a boolean value

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

Purpose: to convert a symbol to a string

```
symbol=? : (symbol symbol -> boolean)
```

Purpose: to determine whether two symbols are equal

```
symbol? : (any -> boolean)
```

Purpose: to determine whether some value is a symbol

```
append : ((listof any)
          (listof any)
          (listof any)
          ...
          ->
          (listof any))
```

Purpose: to create a single list from several, by juxtaposition of the items

---

```
assq : (X
        (listof (cons X Y))
        ->
        (union false (cons X Y)))
```

Purpose: to determine whether some item is the first item of a pair in a list of pairs

---

```
caaar : ((cons
          (cons (cons W (listof Z)) (listof Y))
          (listof X))
         ->
         W)
```

Purpose: to select the first item of the first list in the first list of a list

---

```
caadr : ((cons
          (cons (cons W (listof Z)) (listof Y))
          (listof X))
         ->
         (listof Z))
```

Purpose: to select the rest of the first list in the first list of a list

---

```
caar : ((cons (cons Z (listof Y)) (listof X)) -> Z)
```

Purpose: to select the first item of the first list in a list

---

```
cadar : ((cons (cons W (cons Z (listof Y))) (listof X))
         ->
         Z)
```

Purpose: to select the second item of the first list of a list

---

```
cadddr : ((listof Y) -> Y)
```

Purpose: to select the fourth item of a non-empty list

```
caddr : ((cons W (cons Z (cons Y (listof X)))) -> Y)
```

Purpose: to select the third item of a non-empty list

```
cadr : ((cons Z (cons Y (listof X))) -> Y)
```

Purpose: to select the second item of a non-empty list

```
car : ((cons Y (listof X)) -> Y)
```

Purpose: to select the first item of a non-empty list

```
cdaar : ((cons
          (cons (cons W (listof Z)) (listof Y))
          (listof X))
         ->
         (listof Z))
```

Purpose: to select the rest of the first list in the first list of a list

---

```
cdadr : ((cons W (cons (cons Z (listof Y)) (listof X)))
         ->
         (listof Y))
```

Purpose: to select the rest of the first list in the rest of a list

---

```
cdar : ((cons (cons Z (listof Y)) (listof X))
        ->
        (listof Y))
```

Purpose: to select the rest of a non-empty list in a list

```
cddar : ((cons (cons W (cons Z (listof Y))) (listof X))
         ->
         (listof Y))
```

Purpose: to select the rest of the rest of the first list of a list

---

```
cdddr : ((cons W (cons Z (cons Y (listof X))))
         ->
         (listof X))
```

Purpose: to select the rest of the rest of the rest of a list

---

```
cddr : ((cons Z (cons Y (listof X))) -> (listof X))
```

Purpose: to select the rest of the rest of a list

```
cdr : ((cons Y (listof X)) -> (listof X))
```

Purpose: to select the rest of a non-empty list

```
cons : (X (listof X) -> (listof X))
```

Purpose: to construct a list

```
cons? : (any -> boolean)
```

Purpose: to determine whether some value is a constructed list

---

```
eighth : ((listof Y) -> Y)
```

Purpose: to select the eighth item of a non-empty list

```
empty? : (any -> boolean)
```

Purpose: to determine whether some value is the empty list

```
fifth : ((listof Y) -> Y)
```

Purpose: to select the fifth item of a non-empty list

```
first : ((cons Y (listof X)) -> Y)
```

Purpose: to select the first item of a non-empty list

```
fourth : ((listof Y) -> Y)
```

Purpose: to select the fourth item of a non-empty list

```
length : (list -> number)
```

Purpose: to compute the number of items on a list

```
list : (any ... -> (listof any))
```

Purpose: to construct a list of its arguments

```
list* : (any ... (listof any) -> (listof any))
```

Purpose: to construct a list by adding multiple items to a list

---

```
list-ref : ((listof X) natural-number -> X)
```

Purpose: to extract the indexed item from the list

```
member : (any list -> (union false list))
```

Purpose: to determine whether some value is on the list (comparing values with equal?)

```
memq : (any list -> (union false list))
```

Purpose: to determine whether some value is on some list (comparing values with eq?)

```
memv : (any list -> (union false list))
```

Purpose: to determine whether some value is on the list (comparing values with eqv?)

```
null : empty
```

Purpose: the empty list

```
null? : (any -> boolean)
```

Purpose: to determine whether some value is the empty list

```
pair? : (any -> boolean)
```

Purpose: to determine whether some value is a constructed list

---

```
rest : ((cons Y (listof X)) -> (listof X))
```

Purpose: to select the rest of a non-empty list

```
reverse : (list -> list)
```

Purpose: to create a reversed version of a list

```
second : ((cons Z (cons Y (listof X))) -> Y)
```

Purpose: to select the second item of a non-empty list

```
seventh : ((listof Y) -> Y)
```

Purpose: to select the seventh item of a non-empty list

```
sixth : ((listof Y) -> Y)
```

Purpose: to select the sixth item of a non-empty list

```
third : ((cons W (cons Z (cons Y (listof X)))) -> Y)
```

Purpose: to select the third item of a non-empty list

```
make-posn : (number number -> posn)
```

Purpose: to construct a posn

```
posn-x : (posn -> number)
```

Purpose: to extract the x component of a posn

```
posn-y : (posn -> number)
```

Purpose: to extract the y component of a posn

```
posn? : (anything -> boolean)
```

Purpose: to determine if its input is a posn

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

Purpose: to lookup the number that corresponds to the given character in the ASCII table (if any)

```
char-alphabetic? : (char -> boolean)
```

Purpose: to determine whether a character represents an alphabetic character

---

`char-ci<=?` : `(char char ... -> boolean)`

Purpose: to determine whether a character precedes another (or is equal to it) in a case-insensitive manner

---

`char-ci<?` : `(char char ... -> boolean)`

Purpose: to determine whether a character precedes another in a case-insensitive manner

---

`char-ci=?` : `(char char ... -> boolean)`

Purpose: to determine whether two characters are equal in a case-insensitive manner

---

`char-ci>=?` : `(char char ... -> boolean)`

Purpose: to determine whether a character succeeds another (or is equal to it) in a case-insensitive manner

---

`char-ci>?` : `(char char ... -> boolean)`

Purpose: to determine whether a character succeeds another in a case-insensitive manner

---

`char-downcase` : `(char -> char)`

Purpose: to determine the equivalent lower-case character

`char-lower-case?` : `(char -> boolean)`

Purpose: to determine whether a character is a lower-case character

---

`char-numeric?` : `(char -> boolean)`

Purpose: to determine whether a character represents a digit

`char-upcase` : `(char -> char)`

Purpose: to determine the equivalent upper-case character

`char-upper-case?` : `(char -> boolean)`

Purpose: to determine whether a character is an upper-case character

---

`char-whitespace?` : `(char -> boolean)`

Purpose: to determine whether a character represents space

`char<=?` : `(char char ... -> boolean)`

Purpose: to determine whether a character precedes another (or is equal to it)

`char<?` : `(char char ... -> boolean)`

Purpose: to determine whether a character precedes another

`char=?` : `(char char ... -> boolean)`

Purpose: to determine whether two characters are equal

`char>=?` : `(char char ... -> boolean)`

Purpose: to determine whether a character succeeds another (or is equal to it)

`char>?` : `(char char ... -> boolean)`

Purpose: to determine whether a character succeeds another

`char?` : `(any -> boolean)`

Purpose: to determine whether a value is a character

`format` : `(string any ... -> string)`

Purpose: to format a string, possibly embedding values

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

Purpose: to convert a s list of characters into a string

`make-string : (nat char -> string)`

Purpose: to produce a string of given length from a single given character

---

`string : (char ... -> string)`

Purpose: (string c1 c2 ...) builds a string

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

Purpose: to convert a string into a list of characters

`string->number : (string -> (union number false))`

Purpose: to convert a string into a number, produce false if impossible

---

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

Purpose: to convert a string into a symbol

`string-append : (string ... -> string)`

Purpose: to juxtapose the characters of several strings

`string-ci<=? : (string string ... -> boolean)`

Purpose: to determine whether one string alphabetically precedes another (or is equal to it) in a case-insensitive manner

---

`string-ci<? : (string string ... -> boolean)`

Purpose: to determine whether one string alphabetically precedes another in a case-insensitive manner

---

`string-ci=? : (string string ... -> boolean)`

Purpose: to compare two strings character-wise in a case-insensitive manner

---

`string-ci>=?` : `(string string ... -> boolean)`

Purpose: to determine whether one string alphabetically succeeds another (or is equal to it) in a case-insensitive manner

---

`string-ci>?` : `(string string ... -> boolean)`

Purpose: to determine whether one string alphabetically succeeds another in a case-insensitive manner

---

`string-copy` : `(string -> string)`

Purpose: to copy a string

`string-length` : `(string -> nat)`

Purpose: to determine the length of a string

`string-ref` : `(string nat -> char)`

Purpose: to extract the i-the character from a string

`string<=?` : `(string string ... -> boolean)`

Purpose: to determine whether one string alphabetically precedes another (or is equal to it)

`string<?` : `(string string ... -> boolean)`

Purpose: to determine whether one string alphabetically precedes another

---

`string=?` : `(string string ... -> boolean)`

Purpose: to compare two strings character-wise

`string>=?` : `(string string ... -> boolean)`

Purpose: to determine whether one string alphabetically succeeds another (or is equal to it)

`string>?` : `(string string ... -> boolean)`

Purpose: to determine whether one string alphabetically succeeds another

---

`string?` : `(any -> boolean)`

Purpose: to determine whether a value is a string

`substring` : `(string nat nat -> string)`

Purpose: to extract the substring starting at a 0-based index up to the second 0-based index (exclusive)

`image=?` : `(image image -> boolean)`

Purpose: to determine whether two images are equal

`image?` : `(any -> boolean)`

Purpose: to determine whether a value is an image

`=~` : `(real real non-negative-real -> boolean)`

Purpose: to check whether two real numbers are within some amount (the third argument) of either other

---

`eof` : `eof`

Purpose: the end-of-file value

`eof-object?` : `(any -> boolean)`

Purpose: to determine whether some value is the end-of-file value

---

`eq?` : `(any any -> boolean)`

Purpose: to compare two values

`equal?` : `(any any -> boolean)`

Purpose: to determine whether two values are structurally equal

---

`equal~?` : `(any any non-negative-real -> boolean)`

Purpose: to compare like equal? on the first two arguments, except using =~ in the case of real numbers

---

`eqv?` : `(any any -> boolean)`

Purpose: to compare two values

`error` : `(symbol string -> void)`

Purpose: to signal an error

`exit` : `(-> void)`

Purpose: to exit the running program

`identity` : `(any -> any)`

Purpose: to return the argument unchanged

`struct?` : `(any -> boolean)`

Purpose: to determine whether some value is a structure

2 Beginning Student with List Abbreviations

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

def-or-expr = definition
            | expr
            | library-require

 definition = (define (id id id ...) expr)
            | (define id expr)
            | (define id (lambda (id id ...) expr))
            | (define-struct id (id ...))

       expr = (id expr expr ...) ; function call
            | (prim-op expr ...) ; primitive operation call
            | (cond [expr expr] ... [expr expr])
            | (cond [expr expr] ... [else expr])
            | (if expr expr expr)
            | (and expr expr expr ...)
            | (or expr expr expr ...)
            | test-case
            | empty
            | id
            | 'quoted ; quoted value
            | `quasiquoted ; quasiquote
            | number
            | true
            | false
            | string
            | character

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

quasiquoted = id
            | number
            | string
            | character
```

```
                | (quasiquoted ...)
                | 'quasiquoted
                | `quasiquoted
                | ,expr
                | ,@expr

      test-case = (check-expect expr expr)
                | (check-within expr expr expr)
                | (check-error expr expr)

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

        package = (string string number number)
```

An *id* 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 *string* is enclosed by a pair of ". Unlike symbols, strings may be split into characters and manipulated by a variety of primitive 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.

A *prim-op* is one of:

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

```
* : (num num num ... -> num)
+ : (num num num ... -> num)
- : (num num ... -> num)
/ : (num num num ... -> num)
< : (real real real ... -> boolean)
<= : (real real real ... -> boolean)
= : (num num num ... -> boolean)
> : (real real real ... -> boolean)
>= : (real real ... -> boolean)
abs : (real -> real)
acos : (num -> num)
add1 : (number -> number)
angle : (num -> real)
asin : (num -> num)
atan : (num -> num)
ceiling : (real -> int)
```

```
complex? : (any -> boolean)
conjugate : (num -> num)
cos : (num -> num)
cosh : (num -> num)
current-seconds : (-> int)
denominator : (rat -> int)
e : real
even? : (integer -> boolean)
exact->inexact : (num -> num)
exact? : (num -> boolean)
exp : (num -> num)
expt : (num num -> num)
floor : (real -> int)
gcd : (int int ... -> int)
imag-part : (num -> real)
inexact->exact : (num -> num)
inexact? : (num -> boolean)
integer->char : (int -> char)
integer? : (any -> boolean)
lcm : (int int ... -> int)
log : (num -> num)
magnitude : (num -> real)
make-polar : (real real -> num)
max : (real real ... -> real)
min : (real real ... -> real)
modulo : (int int -> int)
negative? : (number -> boolean)
number->string : (num -> string)
number? : (any -> boolean)
numerator : (rat -> int)
odd? : (integer -> boolean)
pi : real
positive? : (number -> boolean)
quotient : (int int -> int)
random : (int -> int)
rational? : (any -> boolean)
real-part : (num -> real)
real? : (any -> boolean)
remainder : (int int -> int)
round : (real -> int)
sgn : (real -> (union 1 1.0 0 0.0 -1 -1.0))
sin : (num -> num)
sinh : (num -> num)
sqr : (num -> num)
sqrt : (num -> num)
sub1 : (number -> number)
```

```
tan : (num -> num)
zero? : (number -> boolean)
```
Booleans
```
boolean=? : (boolean boolean -> boolean)
boolean? : (any -> boolean)
false? : (any -> boolean)
not : (boolean -> boolean)
```
Symbols
```
symbol->string : (symbol -> string)
symbol=? : (symbol symbol -> boolean)
symbol? : (any -> boolean)
```
Lists
```
append : ((listof any)
          (listof any)
          (listof any)
          ...
          ->
          (listof any))
assq : (X
        (listof (cons X Y))
        ->
        (union false (cons X Y)))
caaar : ((cons
          (cons (cons W (listof Z)) (listof Y))
          (listof X))
         ->
         W)
caadr : ((cons
          (cons (cons W (listof Z)) (listof Y))
          (listof X))
         ->
         (listof Z))
caar : ((cons (cons Z (listof Y)) (listof X)) -> Z)
cadar : ((cons (cons W (cons Z (listof Y))) (listof X))
         ->
         Z)
cadddr : ((listof Y) -> Y)
caddr : ((cons W (cons Z (cons Y (listof X)))) -> Y)
cadr : ((cons Z (cons Y (listof X))) -> Y)
car : ((cons Y (listof X)) -> Y)
cdaar : ((cons
          (cons (cons W (listof Z)) (listof Y))
          (listof X))
         ->
         (listof Z))
```

```
cdadr : ((cons W (cons (cons Z (listof Y)) (listof X)))
         ->
         (listof Y))
cdar : ((cons (cons Z (listof Y)) (listof X))
        ->
        (listof Y))
cddar : ((cons (cons W (cons Z (listof Y))) (listof X))
         ->
         (listof Y))
cdddr : ((cons W (cons Z (cons Y (listof X))))
         ->
         (listof X))
cddr : ((cons Z (cons Y (listof X))) -> (listof X))
cdr : ((cons Y (listof X)) -> (listof X))
cons : (X (listof X) -> (listof X))
cons? : (any -> boolean)
eighth : ((listof Y) -> Y)
empty? : (any -> boolean)
fifth : ((listof Y) -> Y)
first : ((cons Y (listof X)) -> Y)
fourth : ((listof Y) -> Y)
length : (list -> number)
list : (any ... -> (listof any))
list* : (any ... (listof any) -> (listof any))
list-ref : ((listof X) natural-number -> X)
member : (any list -> (union false list))
memq : (any list -> (union false list))
memv : (any list -> (union false list))
null : empty
null? : (any -> boolean)
pair? : (any -> boolean)
rest : ((cons Y (listof X)) -> (listof X))
reverse : (list -> list)
second : ((cons Z (cons Y (listof X))) -> Y)
seventh : ((listof Y) -> Y)
sixth : ((listof Y) -> Y)
third : ((cons W (cons Z (cons Y (listof X)))) -> Y)
```

Posns

```
make-posn : (number number -> posn)
posn-x : (posn -> number)
posn-y : (posn -> number)
posn? : (anything -> boolean)
```

Characters

```
char->integer : (char -> integer)
char-alphabetic? : (char -> boolean)
char-ci<=? : (char char ... -> boolean)
```

```
char-ci<? : (char char ... -> boolean)
char-ci=? : (char char ... -> boolean)
char-ci>=? : (char char ... -> boolean)
char-ci>? : (char char ... -> boolean)
char-downcase : (char -> char)
char-lower-case? : (char -> boolean)
char-numeric? : (char -> boolean)
char-upcase : (char -> char)
char-upper-case? : (char -> boolean)
char-whitespace? : (char -> boolean)
char<=? : (char char ... -> boolean)
char<? : (char char ... -> boolean)
char=? : (char char ... -> boolean)
char>=? : (char char ... -> boolean)
char>? : (char char ... -> boolean)
char? : (any -> boolean)
```

Strings

```
format : (string any ... -> string)
list->string : ((listof char) -> string)
make-string : (nat char -> string)
string : (char ... -> string)
string->list : (string -> (listof char))
string->number : (string -> (union number false))
string->symbol : (string -> symbol)
string-append : (string ... -> string)
string-ci<=? : (string string ... -> boolean)
string-ci<? : (string string ... -> boolean)
string-ci=? : (string string ... -> boolean)
string-ci>=? : (string string ... -> boolean)
string-ci>? : (string string ... -> boolean)
string-copy : (string -> string)
string-length : (string -> nat)
string-ref : (string nat -> char)
string<=? : (string string ... -> boolean)
string<? : (string string ... -> boolean)
string=? : (string string ... -> boolean)
string>=? : (string string ... -> boolean)
string>? : (string string ... -> boolean)
string? : (any -> boolean)
substring : (string nat nat -> string)
```

Images

```
image=? : (image image -> boolean)
image? : (any -> boolean)
```

Misc

```
=~ : (real real non-negative-real -> boolean)
eof : eof
```

```
eof-object? : (any -> boolean)
eq? : (any any -> boolean)
equal? : (any any -> boolean)
equal~? : (any any non-negative-real -> boolean)
eqv? : (any any -> boolean)
error : (symbol string -> void)
exit : (-> void)
identity : (any -> any)
struct? : (any -> boolean)
```

2.1 Quote

```
'quoted
(quote quoted)
```

Creates symbols and abbreviates nested lists.

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

2.2 Quasiquote

```
`quasiquoted
(quasiquote quasiquoted)
```

Creates symbols and abbreviates nested lists, but also allows escaping to expression "unquotes."

Normally, this form is written with a backquote, `` ` ``, like `` `(apple ,(+ 1 2)) ``, but it can also be written with `quasiquote`, like `(quasiquote (apple ,(+ 1 2)))`.

```
,quasiquoted
(unquote expr)
```

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

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

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

```
,@quasiquoted
(unquote-splicing expr)
```

Under a single quasiquote, `,@expr` 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`.

2.3 Primitive Operations

```
* : (num num num ... -> num)
```

Purpose: to compute the product of all of the input numbers

```
+ : (num num num ... -> num)
```

Purpose: to compute the sum of the input numbers

```
- : (num num ... -> num)
```

Purpose: to subtract the second (and following) number(s) from the first; negate the number if there is only one argument

---

```
/ : (num num num ... -> num)
```

Purpose: to divide the first by the second (and all following) number(s); only the first number can be zero.

```
< : (real real real ... -> boolean)
```

Purpose: to compare real numbers for less-than

```
<= : (real real real ... -> boolean)
```

Purpose: to compare real numbers for less-than or equality

`= : (num num num ... -> boolean)`

Purpose: to compare numbers for equality

`> : (real real real ... -> boolean)`

Purpose: to compare real numbers for greater-than

`>= : (real real ... -> boolean)`

Purpose: to compare real numbers for greater-than or equality

---

`abs : (real -> real)`

Purpose: to compute the absolute value of a real number

`acos : (num -> num)`

Purpose: to compute the arccosine (inverse of cos) of a number

---

`add1 : (number -> number)`

Purpose: to compute a number one larger than a given number

`angle : (num -> real)`

Purpose: to extract the angle from a complex number

`asin : (num -> num)`

Purpose: to compute the arcsine (inverse of sin) of a number

`atan : (num -> num)`

Purpose: to compute the arctan (inverse of tan) of a number

`ceiling : (real -> int)`

Purpose: to determine the closest integer above a real number

---

`complex?` : `(any -> boolean)`

Purpose: to determine whether some value is complex

`conjugate` : `(num -> num)`

Purpose: to compute the conjugate of a complex number

`cos` : `(num -> num)`

Purpose: to compute the cosine of a number (radians)

`cosh` : `(num -> num)`

Purpose: to compute the hyperbolic cosine of a number

`current-seconds` : `(-> int)`

Purpose: to compute the current time in seconds elapsed (since a platform-specific starting date)

`denominator` : `(rat -> int)`

Purpose: to compute the denominator of a rational

`e` : `real`

Purpose: Euler's number

`even?` : `(integer -> boolean)`

Purpose: to determine if some value is even or not

`exact->inexact` : `(num -> num)`

Purpose: to convert an exact number to an inexact one

`exact?` : `(num -> boolean)`

Purpose: to determine whether some number is exact

`exp` : `(num -> num)`

Purpose: to compute e raised to a number

`expt` : `(num num -> num)`

Purpose: to compute the power of the first to the second number

---

`floor` : `(real -> int)`

Purpose: to determine the closest integer below a real number

---

`gcd` : `(int int ... -> int)`

Purpose: to compute the greatest common divisior

`imag-part` : `(num -> real)`

Purpose: to extract the imaginary part from a complex number

`inexact->exact` : `(num -> num)`

Purpose: to approximate an inexact number by an exact one

`inexact?` : `(num -> boolean)`

Purpose: to determine whether some number is inexact

`integer->char` : `(int -> char)`

Purpose: to lookup the character that corresponds to the given integer in the ASCII table (if any)

`integer?` : `(any -> boolean)`

Purpose: to determine whether some value is an integer (exact or inexact)

`lcm : (int int ... -> int)`

Purpose: to compute the least common multiple of two integers

---

`log : (num -> num)`

Purpose: to compute the base-e logarithm of a number

`magnitude : (num -> real)`

Purpose: to determine the magnitude of a complex number

`make-polar : (real real -> num)`

Purpose: to create a complex from a magnitude and angle

`max : (real real ... -> real)`

Purpose: to determine the largest number

`min : (real real ... -> real)`

Purpose: to determine the smallest number

`modulo : (int int -> int)`

Purpose: to compute first number modulo second number

`negative? : (number -> boolean)`

Purpose: to determine if some value is strictly smaller than zero

---

`number->string : (num -> string)`

Purpose: to convert a number to a string

`number?` : `(any -> boolean)`

Purpose: to determine whether some value is a number

`numerator` : `(rat -> int)`

Purpose: to compute the numerator of a rational

`odd?` : `(integer -> boolean)`

Purpose: to determine if some value is odd or not

`pi` : `real`

Purpose: the ratio of a circle's circumference to its diameter

---

`positive?` : `(number -> boolean)`

Purpose: to determine if some value is strictly larger than zero

---

`quotient` : `(int int -> int)`

Purpose: to compute the quotient of two integers

`random` : `(int -> int)`

Purpose: to generate a random natural number less than some given integer

---

`rational?` : `(any -> boolean)`

Purpose: to determine whether some value is a rational number

---

`real-part` : `(num -> real)`

Purpose: to extract the real part from a complex number

`real?` : `(any -> boolean)`

Purpose: to determine whether some value is a real number

```
remainder : (int int -> int)
```

Purpose: to compute the remainder of dividing the first by the second integer

---

```
round : (real -> int)
```

Purpose: to round a real number to an integer (rounds to even to break ties)

```
sgn : (real -> (union 1 1.0 0 0.0 -1 -1.0))
```

Purpose: to compute the sign of a real number

```
sin : (num -> num)
```

Purpose: to compute the sine of a number (radians)

```
sinh : (num -> num)
```

Purpose: to compute the hyperbolic sine of a number

```
sqr : (num -> num)
```

Purpose: to compute the square of a number

```
sqrt : (num -> num)
```

Purpose: to compute the square root of a number

```
sub1 : (number -> number)
```

Purpose: to compute a number one smaller than a given number

```
tan : (num -> num)
```

Purpose: to compute the tangent of a number (radians)

```
zero? : (number -> boolean)
```

Purpose: to determine if some value is zero or not

```
boolean=? : (boolean boolean -> boolean)
```

Purpose: to determine whether two booleans are equal

```
boolean? : (any -> boolean)
```

Purpose: to determine whether some value is a boolean

```
false? : (any -> boolean)
```

Purpose: to determine whether a value is false

```
not : (boolean -> boolean)
```

Purpose: to compute the negation of a boolean value

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

Purpose: to convert a symbol to a string

```
symbol=? : (symbol symbol -> boolean)
```

Purpose: to determine whether two symbols are equal

```
symbol? : (any -> boolean)
```

Purpose: to determine whether some value is a symbol

```
append : ((listof any)
          (listof any)
          (listof any)
          ...
          ->
          (listof any))
```

Purpose: to create a single list from several, by juxtaposition of the items

---

```
assq : (X
        (listof (cons X Y))
        ->
        (union false (cons X Y)))
```

Purpose: to determine whether some item is the first item of a pair in a list of pairs

---

```
caaar : ((cons
          (cons (cons W (listof Z)) (listof Y))
          (listof X))
         ->
         W)
```

Purpose: to select the first item of the first list in the first list of a list

---

```
caadr : ((cons
          (cons (cons W (listof Z)) (listof Y))
          (listof X))
         ->
         (listof Z))
```

Purpose: to select the rest of the first list in the first list of a list

---

```
caar : ((cons (cons Z (listof Y)) (listof X)) -> Z)
```

Purpose: to select the first item of the first list in a list

---

```
cadar : ((cons (cons W (cons Z (listof Y))) (listof X))
         ->
         Z)
```

Purpose: to select the second item of the first list of a list

---

```
cadddr : ((listof Y) -> Y)
```

Purpose: to select the fourth item of a non-empty list

```
caddr : ((cons W (cons Z (cons Y (listof X)))) -> Y)
```

Purpose: to select the third item of a non-empty list

```
cadr : ((cons Z (cons Y (listof X))) -> Y)
```

Purpose: to select the second item of a non-empty list

```
car : ((cons Y (listof X)) -> Y)
```

Purpose: to select the first item of a non-empty list

```
cdaar : ((cons
          (cons (cons W (listof Z)) (listof Y))
          (listof X))
         ->
         (listof Z))
```

Purpose: to select the rest of the first list in the first list of a list

---

```
cdadr : ((cons W (cons (cons Z (listof Y)) (listof X)))
         ->
         (listof Y))
```

Purpose: to select the rest of the first list in the rest of a list

---

```
cdar : ((cons (cons Z (listof Y)) (listof X))
        ->
        (listof Y))
```

Purpose: to select the rest of a non-empty list in a list

```
cddar : ((cons (cons W (cons Z (listof Y))) (listof X))
         ->
         (listof Y))
```

Purpose: to select the rest of the rest of the first list of a list

---

```
cdddr : ((cons W (cons Z (cons Y (listof X))))
         ->
         (listof X))
```

Purpose: to select the rest of the rest of the rest of a list

---

```
cddr : ((cons Z (cons Y (listof X))) -> (listof X))
```

Purpose: to select the rest of the rest of a list

```
cdr : ((cons Y (listof X)) -> (listof X))
```

Purpose: to select the rest of a non-empty list

```
cons : (X (listof X) -> (listof X))
```

Purpose: to construct a list

```
cons? : (any -> boolean)
```

Purpose: to determine whether some value is a constructed list

---

```
eighth : ((listof Y) -> Y)
```

Purpose: to select the eighth item of a non-empty list

```
empty? : (any -> boolean)
```

Purpose: to determine whether some value is the empty list

```
fifth : ((listof Y) -> Y)
```

Purpose: to select the fifth item of a non-empty list

```
first : ((cons Y (listof X)) -> Y)
```

Purpose: to select the first item of a non-empty list

```
fourth : ((listof Y) -> Y)
```

Purpose: to select the fourth item of a non-empty list

```
length : (list -> number)
```

Purpose: to compute the number of items on a list

```
list : (any ... -> (listof any))
```

Purpose: to construct a list of its arguments

```
list* : (any ... (listof any) -> (listof any))
```

Purpose: to construct a list by adding multiple items to a list

---

```
list-ref : ((listof X) natural-number -> X)
```

Purpose: to extract the indexed item from the list

```
member : (any list -> (union false list))
```

Purpose: to determine whether some value is on the list (comparing values with equal?)

```
memq : (any list -> (union false list))
```

Purpose: to determine whether some value is on some list (comparing values with eq?)

```
memv : (any list -> (union false list))
```

Purpose: to determine whether some value is on the list (comparing values with eqv?)

```
null : empty
```

Purpose: the empty list

```
null? : (any -> boolean)
```

Purpose: to determine whether some value is the empty list

```
pair? : (any -> boolean)
```

Purpose: to determine whether some value is a constructed list

---

```
rest : ((cons Y (listof X)) -> (listof X))
```

Purpose: to select the rest of a non-empty list

```
reverse : (list -> list)
```

Purpose: to create a reversed version of a list

```
second : ((cons Z (cons Y (listof X))) -> Y)
```

Purpose: to select the second item of a non-empty list

```
seventh : ((listof Y) -> Y)
```

Purpose: to select the seventh item of a non-empty list

```
sixth : ((listof Y) -> Y)
```

Purpose: to select the sixth item of a non-empty list

```
third : ((cons W (cons Z (cons Y (listof X)))) -> Y)
```

Purpose: to select the third item of a non-empty list

```
make-posn : (number number -> posn)
```

Purpose: to construct a posn

```
posn-x : (posn -> number)
```

Purpose: to extract the x component of a posn

```
posn-y : (posn -> number)
```

Purpose: to extract the y component of a posn

```
posn? : (anything -> boolean)
```

Purpose: to determine if its input is a posn

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

Purpose: to lookup the number that corresponds to the given character in the ASCII table (if any)

`char-alphabetic? : (char -> boolean)`

Purpose: to determine whether a character represents an alphabetic character

---

`char-ci<=? : (char char ... -> boolean)`

Purpose: to determine whether a character precedes another (or is equal to it) in a case-insensitive manner

---

`char-ci<? : (char char ... -> boolean)`

Purpose: to determine whether a character precedes another in a case-insensitive manner

---

`char-ci=? : (char char ... -> boolean)`

Purpose: to determine whether two characters are equal in a case-insensitive manner

---

`char-ci>=? : (char char ... -> boolean)`

Purpose: to determine whether a character succeeds another (or is equal to it) in a case-insensitive manner

---

`char-ci>? : (char char ... -> boolean)`

Purpose: to determine whether a character succeeds another in a case-insensitive manner

---

`char-downcase : (char -> char)`

Purpose: to determine the equivalent lower-case character

`char-lower-case? : (char -> boolean)`

Purpose: to determine whether a character is a lower-case character

---

`char-numeric?` : `(char -> boolean)`

Purpose: to determine whether a character represents a digit

`char-upcase` : `(char -> char)`

Purpose: to determine the equivalent upper-case character

`char-upper-case?` : `(char -> boolean)`

Purpose: to determine whether a character is an upper-case character

---

`char-whitespace?` : `(char -> boolean)`

Purpose: to determine whether a character represents space

`char<=?` : `(char char ... -> boolean)`

Purpose: to determine whether a character precedes another (or is equal to it)

`char<?` : `(char char ... -> boolean)`

Purpose: to determine whether a character precedes another

`char=?` : `(char char ... -> boolean)`

Purpose: to determine whether two characters are equal

`char>=?` : `(char char ... -> boolean)`

Purpose: to determine whether a character succeeds another (or is equal to it)

`char>?` : `(char char ... -> boolean)`

Purpose: to determine whether a character succeeds another

`char?` : `(any -> boolean)`

Purpose: to determine whether a value is a character

`format : (string any ... -> string)`

Purpose: to format a string, possibly embedding values

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

Purpose: to convert a s list of characters into a string

`make-string : (nat char -> string)`

Purpose: to produce a string of given length from a single given character

---

`string : (char ... -> string)`

Purpose: (string c1 c2 ...) builds a string

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

Purpose: to convert a string into a list of characters

`string->number : (string -> (union number false))`

Purpose: to convert a string into a number, produce false if impossible

---

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

Purpose: to convert a string into a symbol

`string-append : (string ... -> string)`

Purpose: to juxtapose the characters of several strings

`string-ci<=? : (string string ... -> boolean)`

Purpose: to determine whether one string alphabetically precedes another (or is equal to it) in a case-insensitive manner

---

`string-ci<?` : `(string string ... -> boolean)`

Purpose: to determine whether one string alphabetically precedes another in a case-insensitive manner

---

`string-ci=?` : `(string string ... -> boolean)`

Purpose: to compare two strings character-wise in a case-insensitive manner

---

`string-ci>=?` : `(string string ... -> boolean)`

Purpose: to determine whether one string alphabetically succeeds another (or is equal to it) in a case-insensitive manner

---

`string-ci>?` : `(string string ... -> boolean)`

Purpose: to determine whether one string alphabetically succeeds another in a case-insensitive manner

---

`string-copy` : `(string -> string)`

Purpose: to copy a string

`string-length` : `(string -> nat)`

Purpose: to determine the length of a string

`string-ref` : `(string nat -> char)`

Purpose: to extract the i-the character from a string

`string<=?` : `(string string ... -> boolean)`

Purpose: to determine whether one string alphabetically precedes another (or is equal to it)

`string<?` : `(string string ... -> boolean)`

Purpose: to determine whether one string alphabetically precedes another

---

`string=?` : `(string string ... -> boolean)`

Purpose: to compare two strings character-wise

`string>=?` : `(string string ... -> boolean)`

Purpose: to determine whether one string alphabetically succeeds another (or is equal to it)

`string>?` : `(string string ... -> boolean)`

Purpose: to determine whether one string alphabetically succeeds another

---

`string?` : `(any -> boolean)`

Purpose: to determine whether a value is a string

`substring` : `(string nat nat -> string)`

Purpose: to extract the substring starting at a 0-based index up to the second 0-based index (exclusive)

`image=?` : `(image image -> boolean)`

Purpose: to determine whether two images are equal

`image?` : `(any -> boolean)`

Purpose: to determine whether a value is an image

`=~` : `(real real non-negative-real -> boolean)`

Purpose: to check whether two real numbers are within some amount (the third argument) of either other

---

`eof` : `eof`

Purpose: the end-of-file value

`eof-object?` : `(any -> boolean)`

Purpose: to determine whether some value is the end-of-file value

---

`eq?` : `(any any -> boolean)`

Purpose: to compare two values

`equal?` : `(any any -> boolean)`

Purpose: to determine whether two values are structurally equal

---

`equal~?` : `(any any non-negative-real -> boolean)`

Purpose: to compare like equal? on the first two arguments, except using $=\sim$ in the case of real numbers

---

`eqv?` : `(any any -> boolean)`

Purpose: to compare two values

`error` : `(symbol string -> void)`

Purpose: to signal an error

`exit` : `(-> void)`

Purpose: to exit the running program

`identity` : `(any -> any)`

Purpose: to return the argument unchanged

`struct?` : `(any -> boolean)`

Purpose: to determine whether some value is a structure

2.4 Unchanged Forms

```
(define (id id id ...) expr)
(define id expr)
(define id (lambda (id id ...) expr))
lambda
```

The same as Beginning's `define`.

```
(define-struct structid (fieldid ...))
```

The same as Beginning's `define-struct`.

```
(cond [expr expr] ... [expr expr])
else
```

The same as Beginning's `cond`.

```
(if expr expr expr)
```

The same as Beginning's `if`.

```
(and expr expr expr ...)
(or expr expr expr ...)
```

The same as Beginning's `and` and `or`.

```
(check-expect expr expr)
(check-within expr expr expr)
(check-error expr expr)
```

The same as Beginning's `check-expect`, etc.

```
empty : empty?
true : boolean?
false : boolean?
```

Constants for the empty list, true, and false.

```
(require string)
```

The same as Beginning's `require`.

3 Intermediate Student

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

def-or-expr = definition
            | expr
            | library-require

 definition = (define (id id id ...) expr)
            | (define id expr)
            | (define id (lambda (id id ...) expr))
            | (define-struct id (id ...))

       expr = (local [definition ...] expr)
            | (letrec ([id expr-for-let] ...) expr)
            | (let ([id expr-for-let] ...) expr)
            | (let* ([id expr-for-let] ...) expr)
            | (id expr expr ...) ; function call
            | (prim-op expr ...) ; primitive operation call
            | (cond [expr expr] ... [expr expr])
            | (cond [expr expr] ... [else expr])
            | (if expr expr expr)
            | (and expr expr expr ...)
            | (or expr expr expr ...)
            | (time expr)
            | test-case
            | empty
            | id ; identifier
            | prim-op ; primitive operation
            | 'id
            | 'quoted ; quoted value
            | `quasiquoted ; quasiquote
            | number
            | true
            | false
            | string
            | character

expr-for-let = (lambda (id id ...) expr)
            | expr

     quoted = id
            | number
            | string
            | character
```

```
                | (quoted ...)
                | 'quoted
                | `quoted
                | ,quoted
                | ,@quoted

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

     test-case = (check-expect expr expr)
                | (check-within expr expr expr)
                | (check-error expr expr)

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

       package = (string string number number)
```

An *id* 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 *string* is enclosed by a pair of `"`. Unlike symbols, strings may be split into characters and manipulated by a variety of primitive 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.

A *prim-op* is one of:

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

```
* : (num num num ... -> num)
+ : (num num num ... -> num)
- : (num num ... -> num)
/ : (num num num ... -> num)
< : (real real real ... -> boolean)
<= : (real real real ... -> boolean)
```

```
= : (num num num ... -> boolean)
> : (real real real ... -> boolean)
>= : (real real ... -> boolean)
abs : (real -> real)
acos : (num -> num)
add1 : (number -> number)
angle : (num -> real)
asin : (num -> num)
atan : (num -> num)
ceiling : (real -> int)
complex? : (any -> boolean)
conjugate : (num -> num)
cos : (num -> num)
cosh : (num -> num)
current-seconds : (-> int)
denominator : (rat -> int)
e : real
even? : (integer -> boolean)
exact->inexact : (num -> num)
exact? : (num -> boolean)
exp : (num -> num)
expt : (num num -> num)
floor : (real -> int)
gcd : (int int ... -> int)
imag-part : (num -> real)
inexact->exact : (num -> num)
inexact? : (num -> boolean)
integer->char : (int -> char)
integer? : (any -> boolean)
lcm : (int int ... -> int)
log : (num -> num)
magnitude : (num -> real)
make-polar : (real real -> num)
max : (real real ... -> real)
min : (real real ... -> real)
modulo : (int int -> int)
negative? : (number -> boolean)
number->string : (num -> string)
number? : (any -> boolean)
numerator : (rat -> int)
odd? : (integer -> boolean)
pi : real
positive? : (number -> boolean)
quotient : (int int -> int)
random : (int -> int)
rational? : (any -> boolean)
```

```
real-part : (num -> real)
real? : (any -> boolean)
remainder : (int int -> int)
round : (real -> int)
sgn : (real -> (union 1 1.0 0 0.0 -1 -1.0))
sin : (num -> num)
sinh : (num -> num)
sqr : (num -> num)
sqrt : (num -> num)
sub1 : (number -> number)
tan : (num -> num)
zero? : (number -> boolean)
```

Booleans

```
boolean=? : (boolean boolean -> boolean)
boolean? : (any -> boolean)
false? : (any -> boolean)
not : (boolean -> boolean)
```

Symbols

```
symbol->string : (symbol -> string)
symbol=? : (symbol symbol -> boolean)
symbol? : (any -> boolean)
```

Lists

```
append : ((listof any)
          (listof any)
          (listof any)
          ...
          ->
          (listof any))
assq : (X
        (listof (cons X Y))
        ->
        (union false (cons X Y)))
caaar : ((cons
          (cons (cons W (listof Z)) (listof Y))
          (listof X))
         ->
         W)
caadr : ((cons
          (cons (cons W (listof Z)) (listof Y))
          (listof X))
         ->
         (listof Z))
caar : ((cons (cons Z (listof Y)) (listof X)) -> Z)
cadar : ((cons (cons W (cons Z (listof Y))) (listof X))
         ->
         Z)
```

```
cadddr : ((listof Y) -> Y)
caddr : ((cons W (cons Z (cons Y (listof X)))) -> Y)
cadr : ((cons Z (cons Y (listof X))) -> Y)
car : ((cons Y (listof X)) -> Y)
cdaar : ((cons
          (cons (cons W (listof Z)) (listof Y))
          (listof X))
         ->
         (listof Z))
cdadr : ((cons W (cons (cons Z (listof Y)) (listof X)))
         ->
         (listof Y))
cdar : ((cons (cons Z (listof Y)) (listof X))
        ->
        (listof Y))
cddar : ((cons (cons W (cons Z (listof Y))) (listof X))
         ->
         (listof Y))
cdddr : ((cons W (cons Z (cons Y (listof X))))
         ->
         (listof X))
cddr : ((cons Z (cons Y (listof X))) -> (listof X))
cdr : ((cons Y (listof X)) -> (listof X))
cons : (X (listof X) -> (listof X))
cons? : (any -> boolean)
eighth : ((listof Y) -> Y)
empty? : (any -> boolean)
fifth : ((listof Y) -> Y)
first : ((cons Y (listof X)) -> Y)
fourth : ((listof Y) -> Y)
length : (list -> number)
list : (any ... -> (listof any))
list* : (any ... (listof any) -> (listof any))
list-ref : ((listof X) natural-number -> X)
member : (any list -> (union false list))
memq : (any list -> (union false list))
memv : (any list -> (union false list))
null : empty
null? : (any -> boolean)
pair? : (any -> boolean)
rest : ((cons Y (listof X)) -> (listof X))
reverse : (list -> list)
second : ((cons Z (cons Y (listof X))) -> Y)
seventh : ((listof Y) -> Y)
sixth : ((listof Y) -> Y)
third : ((cons W (cons Z (cons Y (listof X)))) -> Y)
```

Posns

```
make-posn : (number number -> posn)
posn-x : (posn -> number)
posn-y : (posn -> number)
posn? : (anything -> boolean)
```

Characters

```
char->integer : (char -> integer)
char-alphabetic? : (char -> boolean)
char-ci<=? : (char char ... -> boolean)
char-ci<? : (char char ... -> boolean)
char-ci=? : (char char ... -> boolean)
char-ci>=? : (char char ... -> boolean)
char-ci>? : (char char ... -> boolean)
char-downcase : (char -> char)
char-lower-case? : (char -> boolean)
char-numeric? : (char -> boolean)
char-upcase : (char -> char)
char-upper-case? : (char -> boolean)
char-whitespace? : (char -> boolean)
char<=? : (char char ... -> boolean)
char<? : (char char ... -> boolean)
char=? : (char char ... -> boolean)
char>=? : (char char ... -> boolean)
char>? : (char char ... -> boolean)
char? : (any -> boolean)
```

Strings

```
format : (string any ... -> string)
list->string : ((listof char) -> string)
make-string : (nat char -> string)
string : (char ... -> string)
string->list : (string -> (listof char))
string->number : (string -> (union number false))
string->symbol : (string -> symbol)
string-append : (string ... -> string)
string-ci<=? : (string string ... -> boolean)
string-ci<? : (string string ... -> boolean)
string-ci=? : (string string ... -> boolean)
string-ci>=? : (string string ... -> boolean)
string-ci>? : (string string ... -> boolean)
string-copy : (string -> string)
string-length : (string -> nat)
string-ref : (string nat -> char)
string<=? : (string string ... -> boolean)
string<? : (string string ... -> boolean)
string=? : (string string ... -> boolean)
string>=? : (string string ... -> boolean)
```

```
string>? : (string string ... -> boolean)
string? : (any -> boolean)
substring : (string nat nat -> string)
```

Images

```
image=? : (image image -> boolean)
image? : (any -> boolean)
```

Misc

```
=~ : (real real non-negative-real -> boolean)
eof : eof
eof-object? : (any -> boolean)
eq? : (any any -> boolean)
equal? : (any any -> boolean)
equal~? : (any any non-negative-real -> boolean)
eqv? : (any any -> boolean)
error : (symbol string -> void)
exit : (-> void)
identity : (any -> any)
struct? : (any -> boolean)
```

Higher-Order Functions

```
andmap : ((X -> boolean) (listof X) -> boolean)
apply : ((X-1 ... X-N -> Y)
         X-1
         ...
         X-i
         (list X-i+1 ... X-N)
         ->
         Y)
build-list : (nat (nat -> X) -> (listof X))
build-string : (nat (nat -> char) -> string)
compose : ((Y-1 -> Z)
           ...
           (Y-N -> Y-N-1)
           (X-1 ... X-N -> Y-N)
           ->
           (X-1 ... X-N -> Z))
filter : ((X -> boolean) (listof X) -> (listof X))
foldl : ((X Y -> Y) Y (listof X) -> Y)
foldr : ((X Y -> Y) Y (listof X) -> Y)
for-each : ((any ... -> any) (listof any) ... -> void)
map : ((X ... -> Z) (listof X) ... -> (listof Z))
memf : ((X -> boolean)
        (listof X)
        ->
        (union false (listof X)))
ormap : ((X -> boolean) (listof X) -> boolean)
procedure? : (any -> boolean)
```

```
quicksort : ((listof X) (X X -> boolean) -> (listof X))
```

3.1 define

```
(define (id id id ...) expr)
(define id expr)
(define id (lambda (id id ...) expr))
```

Besides working in `local`, definition forms are the same as Beginning's `define`.

`lambda`

As in Beginning, `lambda` keyword can only be used with `define` in the alternative function-definition syntax.

3.2 define-struct

```
(define-struct structid (fieldid ...))
```

Besides working in `local`, this form is the same as Beginning's `define-struct`.

3.3 local

```
(local [definition ...] expr)
```

Groups related definitions for use in *expr*. Each *definition* is evaluated in order, and finally the body *expr* is evaluated. Only the expressions within the `local` form (including the right-hand-sides of the *definition*s and the *expr*) may refer to the names defined by the *definition*s. If a name defined in the `local` form is the same as a top-level binding, the inner one "shadows" the outer one. That is, inside the `local` form, any references to that name refer to the inner one.

Since `local` is an expression and may occur anywhere an expression may occur, it introduces the notion of lexical scope. Expressions within the local may "escape" the scope of the local, but these expressions may still refer to the bindings established by the local.

3.4 `letrec`, `let`, and `let*`

`(letrec ([id expr-for-let] ...) expr)`

Similar to `local`, but essentially omitting the `define` for each definition.

A *`expr-for-let`* can be either an expression for a constant definition or a `lambda` form for a function definition.

`(let ([id expr-for-let] ...) expr)`

Like `letrec`, but the defined *`id`*s can be used only in the last *`expr`*, not the *`expr-for-let`*s next to the *`id`*s.

`(let* ([id expr-for-let] ...) expr)`

Like `let`, but each *`id`* can be used in any subsequent *`expr-for-let`*, in addition to *`expr`*.

3.5 Function Calls

`(id expr expr ...)`

A function call in Intermediate is the same as a Beginning function call, except that it can also call `local`ly defined functions or functions passed as arguments. That is, *`id`* can be a function defined in `local` or an argument name while in a function.

`(#%app id expr expr ...)`

A function call can be written with `#%app`, though it's practically never written that way.

3.6 `time`

`(time expr)`

This form is used to measure the time taken to evaluate *`expr`*. After evaluating *`expr`*, Scheme prints out the time taken by the evaluation (including real time, time taken by the cpu, and the time spent collecting free memory) and returns the result of the expression.

3.7 Identifiers

---

id

An *id* refers to a defined constant (possibly local), defined function (possibly local), or argument within a function body. If no definition or argument matches the *id* name, an error is reported.

3.8 Primitive Operations

---

prim-op

The name of a primitive operation can be used as an expression. If it is passed to a function, then it can be used in a function call within the function's body.

`* : (num num num ... -> num)`

Purpose: to compute the product of all of the input numbers

`+ : (num num num ... -> num)`

Purpose: to compute the sum of the input numbers

`- : (num num ... -> num)`

Purpose: to subtract the second (and following) number(s) from the first; negate the number if there is only one argument

---

`/ : (num num num ... -> num)`

Purpose: to divide the first by the second (and all following) number(s); only the first number can be zero.

`< : (real real real ... -> boolean)`

Purpose: to compare real numbers for less-than

<= : (real real real ... -> boolean)

Purpose: to compare real numbers for less-than or equality

= : (num num num ... -> boolean)

Purpose: to compare numbers for equality

> : (real real real ... -> boolean)

Purpose: to compare real numbers for greater-than

>= : (real real ... -> boolean)

Purpose: to compare real numbers for greater-than or equality

---

abs : (real -> real)

Purpose: to compute the absolute value of a real number

---

acos : (num -> num)

Purpose: to compute the arccosine (inverse of cos) of a number

---

add1 : (number -> number)

Purpose: to compute a number one larger than a given number

---

angle : (num -> real)

Purpose: to extract the angle from a complex number

---

asin : (num -> num)

Purpose: to compute the arcsine (inverse of sin) of a number

---

atan : (num -> num)

Purpose: to compute the arctan (inverse of tan) of a number

```
ceiling : (real -> int)
```

Purpose: to determine the closest integer above a real number

---

```
complex? : (any -> boolean)
```

Purpose: to determine whether some value is complex

```
conjugate : (num -> num)
```

Purpose: to compute the conjugate of a complex number

```
cos : (num -> num)
```

Purpose: to compute the cosine of a number (radians)

```
cosh : (num -> num)
```

Purpose: to compute the hyperbolic cosine of a number

```
current-seconds : (-> int)
```

Purpose: to compute the current time in seconds elapsed (since a platform-specific starting date)

```
denominator : (rat -> int)
```

Purpose: to compute the denominator of a rational

```
e : real
```

Purpose: Euler's number

```
even? : (integer -> boolean)
```

Purpose: to determine if some value is even or not

```
exact->inexact : (num -> num)
```

Purpose: to convert an exact number to an inexact one

```
exact? : (num -> boolean)
```

Purpose: to determine whether some number is exact

```
exp : (num -> num)
```

Purpose: to compute e raised to a number

```
expt : (num num -> num)
```

Purpose: to compute the power of the first to the second number

---

```
floor : (real -> int)
```

Purpose: to determine the closest integer below a real number

---

```
gcd : (int int ... -> int)
```

Purpose: to compute the greatest common divisior

```
imag-part : (num -> real)
```

Purpose: to extract the imaginary part from a complex number

```
inexact->exact : (num -> num)
```

Purpose: to approximate an inexact number by an exact one

```
inexact? : (num -> boolean)
```

Purpose: to determine whether some number is inexact

```
integer->char : (int -> char)
```

Purpose: to lookup the character that corresponds to the given integer in the ASCII table (if any)

`integer?` : `(any -> boolean)`

Purpose: to determine whether some value is an integer (exact or inexact)

`lcm` : `(int int ... -> int)`

Purpose: to compute the least common multiple of two integers

---

`log` : `(num -> num)`

Purpose: to compute the base-e logarithm of a number

`magnitude` : `(num -> real)`

Purpose: to determine the magnitude of a complex number

`make-polar` : `(real real -> num)`

Purpose: to create a complex from a magnitude and angle

`max` : `(real real ... -> real)`

Purpose: to determine the largest number

`min` : `(real real ... -> real)`

Purpose: to determine the smallest number

`modulo` : `(int int -> int)`

Purpose: to compute first number modulo second number

`negative?` : `(number -> boolean)`

Purpose: to determine if some value is strictly smaller than zero

---

`number->string` : `(num -> string)`

Purpose: to convert a number to a string

`number?` : `(any -> boolean)`

Purpose: to determine whether some value is a number

`numerator` : `(rat -> int)`

Purpose: to compute the numerator of a rational

`odd?` : `(integer -> boolean)`

Purpose: to determine if some value is odd or not

`pi` : `real`

Purpose: the ratio of a circle's circumference to its diameter

---

`positive?` : `(number -> boolean)`

Purpose: to determine if some value is strictly larger than zero

---

`quotient` : `(int int -> int)`

Purpose: to compute the quotient of two integers

`random` : `(int -> int)`

Purpose: to generate a random natural number less than some given integer

---

`rational?` : `(any -> boolean)`

Purpose: to determine whether some value is a rational number

---

`real-part` : `(num -> real)`

Purpose: to extract the real part from a complex number

```
real? : (any -> boolean)
```

Purpose: to determine whether some value is a real number

```
remainder : (int int -> int)
```

Purpose: to compute the remainder of dividing the first by the second integer

---

```
round : (real -> int)
```

Purpose: to round a real number to an integer (rounds to even to break ties)

```
sgn : (real -> (union 1 1.0 0 0.0 -1 -1.0))
```

Purpose: to compute the sign of a real number

```
sin : (num -> num)
```

Purpose: to compute the sine of a number (radians)

```
sinh : (num -> num)
```

Purpose: to compute the hyperbolic sine of a number

```
sqr : (num -> num)
```

Purpose: to compute the square of a number

```
sqrt : (num -> num)
```

Purpose: to compute the square root of a number

```
sub1 : (number -> number)
```

Purpose: to compute a number one smaller than a given number

```
tan : (num -> num)
```

Purpose: to compute the tangent of a number (radians)

```
zero? : (number -> boolean)
```

Purpose: to determine if some value is zero or not

```
boolean=? : (boolean boolean -> boolean)
```

Purpose: to determine whether two booleans are equal

```
boolean? : (any -> boolean)
```

Purpose: to determine whether some value is a boolean

```
false? : (any -> boolean)
```

Purpose: to determine whether a value is false

```
not : (boolean -> boolean)
```

Purpose: to compute the negation of a boolean value

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

Purpose: to convert a symbol to a string

```
symbol=? : (symbol symbol -> boolean)
```

Purpose: to determine whether two symbols are equal

```
symbol? : (any -> boolean)
```

Purpose: to determine whether some value is a symbol

```
append : ((listof any)
          (listof any)
          (listof any)
          ...
          ->
          (listof any))
```

Purpose: to create a single list from several, by juxtaposition of the items

---

```
assq : (X
        (listof (cons X Y))
        ->
        (union false (cons X Y)))
```

Purpose: to determine whether some item is the first item of a pair in a list of pairs

---

```
caaar : ((cons
          (cons (cons W (listof Z)) (listof Y))
          (listof X))
         ->
         W)
```

Purpose: to select the first item of the first list in the first list of a list

---

```
caadr : ((cons
          (cons (cons W (listof Z)) (listof Y))
          (listof X))
         ->
         (listof Z))
```

Purpose: to select the rest of the first list in the first list of a list

---

```
caar : ((cons (cons Z (listof Y)) (listof X)) -> Z)
```

Purpose: to select the first item of the first list in a list

---

```
cadar : ((cons (cons W (cons Z (listof Y))) (listof X))
         ->
         Z)
```

Purpose: to select the second item of the first list of a list

---

```
cadddr : ((listof Y) -> Y)
```

Purpose: to select the fourth item of a non-empty list

```
caddr : ((cons W (cons Z (cons Y (listof X)))) -> Y)
```

Purpose: to select the third item of a non-empty list

```
cadr : ((cons Z (cons Y (listof X))) -> Y)
```

Purpose: to select the second item of a non-empty list

```
car : ((cons Y (listof X)) -> Y)
```

Purpose: to select the first item of a non-empty list

```
cdaar : ((cons
          (cons (cons W (listof Z)) (listof Y))
          (listof X))
         ->
         (listof Z))
```

Purpose: to select the rest of the first list in the first list of a list

---

```
cdadr : ((cons W (cons (cons Z (listof Y)) (listof X)))
         ->
         (listof Y))
```

Purpose: to select the rest of the first list in the rest of a list

---

```
cdar : ((cons (cons Z (listof Y)) (listof X))
        ->
        (listof Y))
```

Purpose: to select the rest of a non-empty list in a list

```
cddar : ((cons (cons W (cons Z (listof Y))) (listof X))
         ->
         (listof Y))
```

Purpose: to select the rest of the rest of the first list of a list

---

```
cdddr : ((cons W (cons Z (cons Y (listof X))))
         ->
         (listof X))
```

Purpose: to select the rest of the rest of the rest of a list

---

```
cddr : ((cons Z (cons Y (listof X))) -> (listof X))
```

Purpose: to select the rest of the rest of a list

```
cdr : ((cons Y (listof X)) -> (listof X))
```

Purpose: to select the rest of a non-empty list

```
cons : (X (listof X) -> (listof X))
```

Purpose: to construct a list

```
cons? : (any -> boolean)
```

Purpose: to determine whether some value is a constructed list

---

```
eighth : ((listof Y) -> Y)
```

Purpose: to select the eighth item of a non-empty list

```
empty? : (any -> boolean)
```

Purpose: to determine whether some value is the empty list

```
fifth : ((listof Y) -> Y)
```

Purpose: to select the fifth item of a non-empty list

```
first : ((cons Y (listof X)) -> Y)
```

Purpose: to select the first item of a non-empty list

```
fourth : ((listof Y) -> Y)
```

Purpose: to select the fourth item of a non-empty list

```
length : (list -> number)
```

Purpose: to compute the number of items on a list

```
list : (any ... -> (listof any))
```

Purpose: to construct a list of its arguments

```
list* : (any ... (listof any) -> (listof any))
```

Purpose: to construct a list by adding multiple items to a list

---

```
list-ref : ((listof X) natural-number -> X)
```

Purpose: to extract the indexed item from the list

```
member : (any list -> (union false list))
```

Purpose: to determine whether some value is on the list (comparing values with equal?)

```
memq : (any list -> (union false list))
```

Purpose: to determine whether some value is on some list (comparing values with eq?)

```
memv : (any list -> (union false list))
```

Purpose: to determine whether some value is on the list (comparing values with eqv?)

```
null : empty
```

Purpose: the empty list

```
null? : (any -> boolean)
```

Purpose: to determine whether some value is the empty list

```
pair? : (any -> boolean)
```

Purpose: to determine whether some value is a constructed list

---

```
rest : ((cons Y (listof X)) -> (listof X))
```

Purpose: to select the rest of a non-empty list

```
reverse : (list -> list)
```

Purpose: to create a reversed version of a list

```
second : ((cons Z (cons Y (listof X))) -> Y)
```

Purpose: to select the second item of a non-empty list

```
seventh : ((listof Y) -> Y)
```

Purpose: to select the seventh item of a non-empty list

```
sixth : ((listof Y) -> Y)
```

Purpose: to select the sixth item of a non-empty list

```
third : ((cons W (cons Z (cons Y (listof X)))) -> Y)
```

Purpose: to select the third item of a non-empty list

```
make-posn : (number number -> posn)
```

Purpose: to construct a posn

```
posn-x : (posn -> number)
```

Purpose: to extract the x component of a posn

```
posn-y : (posn -> number)
```

Purpose: to extract the y component of a posn

```
posn? : (anything -> boolean)
```

Purpose: to determine if its input is a posn

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

Purpose: to lookup the number that corresponds to the given character in the ASCII table (if any)

`char-alphabetic? : (char -> boolean)`

Purpose: to determine whether a character represents an alphabetic character

---

`char-ci<=? : (char char ... -> boolean)`

Purpose: to determine whether a character precedes another (or is equal to it) in a case-insensitive manner

---

`char-ci<? : (char char ... -> boolean)`

Purpose: to determine whether a character precedes another in a case-insensitive manner

---

`char-ci=? : (char char ... -> boolean)`

Purpose: to determine whether two characters are equal in a case-insensitive manner

---

`char-ci>=? : (char char ... -> boolean)`

Purpose: to determine whether a character succeeds another (or is equal to it) in a case-insensitive manner

---

`char-ci>? : (char char ... -> boolean)`

Purpose: to determine whether a character succeeds another in a case-insensitive manner

---

`char-downcase : (char -> char)`

Purpose: to determine the equivalent lower-case character

`char-lower-case? : (char -> boolean)`

Purpose: to determine whether a character is a lower-case character

---

`char-numeric? : (char -> boolean)`

Purpose: to determine whether a character represents a digit

`char-upcase : (char -> char)`

Purpose: to determine the equivalent upper-case character

`char-upper-case? : (char -> boolean)`

Purpose: to determine whether a character is an upper-case character

---

`char-whitespace? : (char -> boolean)`

Purpose: to determine whether a character represents space

`char<=? : (char char ... -> boolean)`

Purpose: to determine whether a character precedes another (or is equal to it)

`char<? : (char char ... -> boolean)`

Purpose: to determine whether a character precedes another

`char=? : (char char ... -> boolean)`

Purpose: to determine whether two characters are equal

`char>=? : (char char ... -> boolean)`

Purpose: to determine whether a character succeeds another (or is equal to it)

`char>? : (char char ... -> boolean)`

Purpose: to determine whether a character succeeds another

`char?` : `(any -> boolean)`

Purpose: to determine whether a value is a character

`format` : `(string any ... -> string)`

Purpose: to format a string, possibly embedding values

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

Purpose: to convert a s list of characters into a string

`make-string` : `(nat char -> string)`

Purpose: to produce a string of given length from a single given character

---

`string` : `(char ... -> string)`

Purpose: (string c1 c2 ...) builds a string

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

Purpose: to convert a string into a list of characters

`string->number` : `(string -> (union number false))`

Purpose: to convert a string into a number, produce false if impossible

---

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

Purpose: to convert a string into a symbol

`string-append` : `(string ... -> string)`

Purpose: to juxtapose the characters of several strings

`string-ci<=?` : `(string string ... -> boolean)`

Purpose: to determine whether one string alphabetically precedes another (or is equal to it)

in a case-insensitive manner

`string-ci<?` : `(string string ... -> boolean)`

Purpose: to determine whether one string alphabetically precedes another in a case-insensitive manner

---

`string-ci=?` : `(string string ... -> boolean)`

Purpose: to compare two strings character-wise in a case-insensitive manner

---

`string-ci>=?` : `(string string ... -> boolean)`

Purpose: to determine whether one string alphabetically succeeds another (or is equal to it) in a case-insensitive manner

---

`string-ci>?` : `(string string ... -> boolean)`

Purpose: to determine whether one string alphabetically succeeds another in a case-insensitive manner

---

`string-copy` : `(string -> string)`

Purpose: to copy a string

`string-length` : `(string -> nat)`

Purpose: to determine the length of a string

`string-ref` : `(string nat -> char)`

Purpose: to extract the i-the character from a string

`string<=?` : `(string string ... -> boolean)`

Purpose: to determine whether one string alphabetically precedes another (or is equal to it)

`string<?` : `(string string ... -> boolean)`

Purpose: to determine whether one string alphabetically precedes another

---

`string=?` : `(string string ... -> boolean)`

Purpose: to compare two strings character-wise

`string>=?` : `(string string ... -> boolean)`

Purpose: to determine whether one string alphabetically succeeds another (or is equal to it)

`string>?` : `(string string ... -> boolean)`

Purpose: to determine whether one string alphabetically succeeds another

---

`string?` : `(any -> boolean)`

Purpose: to determine whether a value is a string

`substring` : `(string nat nat -> string)`

Purpose: to extract the substring starting at a 0-based index up to the second 0-based index (exclusive)

`image=?` : `(image image -> boolean)`

Purpose: to determine whether two images are equal

`image?` : `(any -> boolean)`

Purpose: to determine whether a value is an image

`=~` : `(real real non-negative-real -> boolean)`

Purpose: to check whether two real numbers are within some amount (the third argument) of either other

---

`eof` : `eof`

Purpose: the end-of-file value

`eof-object?` : `(any -> boolean)`

Purpose: to determine whether some value is the end-of-file value

---

`eq?` : `(any any -> boolean)`

Purpose: to compare two values

`equal?` : `(any any -> boolean)`

Purpose: to determine whether two values are structurally equal

---

`equal~?` : `(any any non-negative-real -> boolean)`

Purpose: to compare like equal? on the first two arguments, except using $=\sim$ in the case of real numbers

---

`eqv?` : `(any any -> boolean)`

Purpose: to compare two values

`error` : `(symbol string -> void)`

Purpose: to signal an error

`exit` : `(-> void)`

Purpose: to exit the running program

`identity` : `(any -> any)`

Purpose: to return the argument unchanged

`struct?` : `(any -> boolean)`

Purpose: to determine whether some value is a structure

`andmap` : `((X -> boolean) (listof X) -> boolean)`

Purpose: (andmap p (list x-1 ... x-n)) = (and (p x-1) (and ... (p x-n)))

```
apply : ((X-1 ... X-N -> Y)
         X-1
         ...
         X-i
         (list X-i+1 ... X-N)
         ->
         Y)
```

Purpose: to apply a function using items from a list as the arguments

---

```
build-list : (nat (nat -> X) -> (listof X))
```

Purpose: (build-list n f) = (list (f 0) ... (f (- n 1)))

```
build-string : (nat (nat -> char) -> string)
```

Purpose: (build-string n f) = (string (f 0) ... (f (- n 1)))

```
compose : ((Y-1 -> Z)
           ...
           (Y-N -> Y-N-1)
           (X-1 ... X-N -> Y-N)
           ->
           (X-1 ... X-N -> Z))
```

Purpose: to compose a sequence of procedures into a single procedure

---

```
filter : ((X -> boolean) (listof X) -> (listof X))
```

Purpose: to construct a list from all those items on a list for which the predicate holds

---

```
foldl : ((X Y -> Y) Y (listof X) -> Y)
```

Purpose: (foldl f base (list x-1 ... x-n)) = (f x-n ... (f x-1 base))

```
foldr : ((X Y -> Y) Y (listof X) -> Y)
```

Purpose: (foldr f base (list x-1 ... x-n)) = (f x-1 ... (f x-n base))

```
for-each : ((any ... -> any) (listof any) ... -> void)
```

Purpose: to apply a function to each item on one or more lists for effect only

```
map : ((X ... -> Z) (listof X) ... -> (listof Z))
```

Purpose: to construct a new list by applying a function to each item on one or more existing lists

```
memf : ((X -> boolean)
        (listof X)
        ->
        (union false (listof X)))
```

Purpose: to determine whether the first argument produces true for some value in the second argument

```
ormap : ((X -> boolean) (listof X) -> boolean)
```

Purpose: (ormap p (list x-1 ... x-n)) = (or (p x-1) (or ... (p x-n)))

```
procedure? : (any -> boolean)
```

Purpose: to determine if a value is a procedure

```
quicksort : ((listof X) (X X -> boolean) -> (listof X))
```

Purpose: to construct a list from all items on a list in an order according to a predicate

3.9 Unchanged Forms

```
(cond [expr expr] ... [expr expr])
else
```

The same as Beginning's `cond`.

```
(if expr expr expr)
```

The same as Beginning's `if`.

```
(and expr expr expr ...)
(or expr expr expr ...)
```

The same as Beginning's `and` and `or`.

```
(check-expect expr expr)
(check-within expr expr expr)
(check-error expr expr)
```

The same as Beginning's `check-expect`, etc.

```
empty : empty?
true : boolean?
false : boolean?
```

Constants for the empty list, true, and false.

```
(require string)
```

The same as Beginning's `require`.

4 Intermediate Student with Lambda

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

def-or-expr = definition
            | expr
            | library-require

 definition = (define (id id id ...) expr)
            | (define id expr)
            | (define-struct id (id ...))

       expr = (lambda (id id ...) expr)
            | (local [definition ...] expr)
            | (letrec ([id expr] ...) expr)
            | (let ([id expr] ...) expr)
            | (let* ([id expr] ...) expr)
            | (expr expr expr ...) ; function call
            | (cond [expr expr] ... [expr expr])
            | (cond [expr expr] ... [else expr])
            | (if expr expr expr)
            | (and expr expr expr ...)
            | (or expr expr expr ...)
            | (time expr)
            | test-case
            | empty
            | id ; identifier
            | prim-op ; primitive operation
            | 'id
            | 'quoted ; quoted value
            | `quasiquoted ; quasiquote
            | number
            | true
            | false
            | string
            | character

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

```
                | ,@quoted

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

     test-case = (check-expect expr expr)
                | (check-within expr expr expr)
                | (check-error expr expr)

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

       package = (string string number number)
```

An *id* 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 *string* is enclosed by a pair of `"`. Unlike symbols, strings may be split into characters and manipulated by a variety of primitive 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.

A *prim-op* is one of:

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

```
* : (num num num ... -> num)
+ : (num num num ... -> num)
- : (num num ... -> num)
/ : (num num num ... -> num)
< : (real real real ... -> boolean)
<= : (real real real ... -> boolean)
= : (num num num ... -> boolean)
> : (real real real ... -> boolean)
>= : (real real ... -> boolean)
abs : (real -> real)
```

```
acos : (num -> num)
add1 : (number -> number)
angle : (num -> real)
asin : (num -> num)
atan : (num -> num)
ceiling : (real -> int)
complex? : (any -> boolean)
conjugate : (num -> num)
cos : (num -> num)
cosh : (num -> num)
current-seconds : (-> int)
denominator : (rat -> int)
e : real
even? : (integer -> boolean)
exact->inexact : (num -> num)
exact? : (num -> boolean)
exp : (num -> num)
expt : (num num -> num)
floor : (real -> int)
gcd : (int int ... -> int)
imag-part : (num -> real)
inexact->exact : (num -> num)
inexact? : (num -> boolean)
integer->char : (int -> char)
integer? : (any -> boolean)
lcm : (int int ... -> int)
log : (num -> num)
magnitude : (num -> real)
make-polar : (real real -> num)
max : (real real ... -> real)
min : (real real ... -> real)
modulo : (int int -> int)
negative? : (number -> boolean)
number->string : (num -> string)
number? : (any -> boolean)
numerator : (rat -> int)
odd? : (integer -> boolean)
pi : real
positive? : (number -> boolean)
quotient : (int int -> int)
random : (int -> int)
rational? : (any -> boolean)
real-part : (num -> real)
real? : (any -> boolean)
remainder : (int int -> int)
round : (real -> int)
```

```
sgn : (real -> (union 1 1.0 0 0.0 -1 -1.0))
sin : (num -> num)
sinh : (num -> num)
sqr : (num -> num)
sqrt : (num -> num)
sub1 : (number -> number)
tan : (num -> num)
zero? : (number -> boolean)
```

Booleans

```
boolean=? : (boolean boolean -> boolean)
boolean? : (any -> boolean)
false? : (any -> boolean)
not : (boolean -> boolean)
```

Symbols

```
symbol->string : (symbol -> string)
symbol=? : (symbol symbol -> boolean)
symbol? : (any -> boolean)
```

Lists

```
append : ((listof any)
          (listof any)
          (listof any)
          ...
          ->
          (listof any))
assq : (X
        (listof (cons X Y))
        ->
        (union false (cons X Y)))
caaar : ((cons
          (cons (cons W (listof Z)) (listof Y))
          (listof X))
         ->
         W)
caadr : ((cons
          (cons (cons W (listof Z)) (listof Y))
          (listof X))
         ->
         (listof Z))
caar : ((cons (cons Z (listof Y)) (listof X)) -> Z)
cadar : ((cons (cons W (cons Z (listof Y))) (listof X))
         ->
         Z)
cadddr : ((listof Y) -> Y)
caddr : ((cons W (cons Z (cons Y (listof X)))) -> Y)
cadr : ((cons Z (cons Y (listof X))) -> Y)
car : ((cons Y (listof X)) -> Y)
```

```
cdaar : ((cons
          (cons (cons W (listof Z)) (listof Y))
          (listof X))
         ->
         (listof Z))
cdadr : ((cons W (cons (cons Z (listof Y)) (listof X)))
         ->
         (listof Y))
cdar : ((cons (cons Z (listof Y)) (listof X))
        ->
        (listof Y))
cddar : ((cons (cons W (cons Z (listof Y))) (listof X))
         ->
         (listof Y))
cdddr : ((cons W (cons Z (cons Y (listof X))))
         ->
         (listof X))
cddr : ((cons Z (cons Y (listof X))) -> (listof X))
cdr : ((cons Y (listof X)) -> (listof X))
cons : (X (listof X) -> (listof X))
cons? : (any -> boolean)
eighth : ((listof Y) -> Y)
empty? : (any -> boolean)
fifth : ((listof Y) -> Y)
first : ((cons Y (listof X)) -> Y)
fourth : ((listof Y) -> Y)
length : (list -> number)
list : (any ... -> (listof any))
list* : (any ... (listof any) -> (listof any))
list-ref : ((listof X) natural-number -> X)
member : (any list -> (union false list))
memq : (any list -> (union false list))
memv : (any list -> (union false list))
null : empty
null? : (any -> boolean)
pair? : (any -> boolean)
rest : ((cons Y (listof X)) -> (listof X))
reverse : (list -> list)
second : ((cons Z (cons Y (listof X))) -> Y)
seventh : ((listof Y) -> Y)
sixth : ((listof Y) -> Y)
third : ((cons W (cons Z (cons Y (listof X)))) -> Y)
```

Posns

```
make-posn : (number number -> posn)
posn-x : (posn -> number)
posn-y : (posn -> number)
```

```
posn? : (anything -> boolean)
```

Characters

```
char->integer : (char -> integer)
char-alphabetic? : (char -> boolean)
char-ci<=? : (char char ... -> boolean)
char-ci<? : (char char ... -> boolean)
char-ci=? : (char char ... -> boolean)
char-ci>=? : (char char ... -> boolean)
char-ci>? : (char char ... -> boolean)
char-downcase : (char -> char)
char-lower-case? : (char -> boolean)
char-numeric? : (char -> boolean)
char-upcase : (char -> char)
char-upper-case? : (char -> boolean)
char-whitespace? : (char -> boolean)
char<=? : (char char ... -> boolean)
char<? : (char char ... -> boolean)
char=? : (char char ... -> boolean)
char>=? : (char char ... -> boolean)
char>? : (char char ... -> boolean)
char? : (any -> boolean)
```

Strings

```
format : (string any ... -> string)
list->string : ((listof char) -> string)
make-string : (nat char -> string)
string : (char ... -> string)
string->list : (string -> (listof char))
string->number : (string -> (union number false))
string->symbol : (string -> symbol)
string-append : (string ... -> string)
string-ci<=? : (string string ... -> boolean)
string-ci<? : (string string ... -> boolean)
string-ci=? : (string string ... -> boolean)
string-ci>=? : (string string ... -> boolean)
string-ci>? : (string string ... -> boolean)
string-copy : (string -> string)
string-length : (string -> nat)
string-ref : (string nat -> char)
string<=? : (string string ... -> boolean)
string<? : (string string ... -> boolean)
string=? : (string string ... -> boolean)
string>=? : (string string ... -> boolean)
string>? : (string string ... -> boolean)
string? : (any -> boolean)
substring : (string nat nat -> string)
```

Images

```
image=? : (image image -> boolean)
image? : (any -> boolean)
```
Misc
```
=~ : (real real non-negative-real -> boolean)
eof : eof
eof-object? : (any -> boolean)
eq? : (any any -> boolean)
equal? : (any any -> boolean)
equal~? : (any any non-negative-real -> boolean)
eqv? : (any any -> boolean)
error : (symbol string -> void)
exit : (-> void)
identity : (any -> any)
struct? : (any -> boolean)
```
Higher-Order Functions
```
andmap : ((X -> boolean) (listof X) -> boolean)
apply : ((X-1 ... X-N -> Y)
         X-1
         ...
         X-i
         (list X-i+1 ... X-N)
         ->
         Y)
build-list : (nat (nat -> X) -> (listof X))
build-string : (nat (nat -> char) -> string)
compose : ((Y-1 -> Z)
           ...
           (Y-N -> Y-N-1)
           (X-1 ... X-N -> Y-N)
           ->
           (X-1 ... X-N -> Z))
filter : ((X -> boolean) (listof X) -> (listof X))
foldl : ((X Y -> Y) Y (listof X) -> Y)
foldr : ((X Y -> Y) Y (listof X) -> Y)
for-each : ((any ... -> any) (listof any) ... -> void)
map : ((X ... -> Z) (listof X) ... -> (listof Z))
memf : ((X -> boolean)
        (listof X)
        ->
        (union false (listof X)))
ormap : ((X -> boolean) (listof X) -> boolean)
procedure? : (any -> boolean)
quicksort : ((listof X) (X X -> boolean) -> (listof X))
```

4.1 `define`

```
(define (id id id ...) expr)
(define id expr)
```

The same as Intermediate's `define`. No special case is needed for `lambda`, since a `lambda` form is an expression.

4.2 `lambda`

```
(lambda (id id ...) expr)
```

Creates a function that takes as many arguments as given *id*s, and whose body is *expr*.

4.3 Function Calls

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

Like a Beginning function call, except that the function position can be an arbitrary expression—perhaps a `lambda` expression or a *prim-op*.

```
(#%app id expr expr ...)
```

A function call can be written with `#%app`, though it's practically never written that way.

4.4 Primitive Operation Names

---

prim-op

The name of a primitive operation can be used as an expression. It produces a function version of the operation.

```
* : (num num num ... -> num)
```

Purpose: to compute the product of all of the input numbers

`+ : (num num num ... -> num)`

Purpose: to compute the sum of the input numbers

`- : (num num ... -> num)`

Purpose: to subtract the second (and following) number(s) from the first; negate the number if there is only one argument

---

`/ : (num num num ... -> num)`

Purpose: to divide the first by the second (and all following) number(s); only the first number can be zero.

`< : (real real real ... -> boolean)`

Purpose: to compare real numbers for less-than

`<= : (real real real ... -> boolean)`

Purpose: to compare real numbers for less-than or equality

`= : (num num num ... -> boolean)`

Purpose: to compare numbers for equality

`> : (real real real ... -> boolean)`

Purpose: to compare real numbers for greater-than

`>= : (real real ... -> boolean)`

Purpose: to compare real numbers for greater-than or equality

---

`abs : (real -> real)`

Purpose: to compute the absolute value of a real number

`acos : (num -> num)`

Purpose: to compute the arccosine (inverse of cos) of a number

---

`add1 : (number -> number)`

Purpose: to compute a number one larger than a given number

`angle : (num -> real)`

Purpose: to extract the angle from a complex number

`asin : (num -> num)`

Purpose: to compute the arcsine (inverse of sin) of a number

`atan : (num -> num)`

Purpose: to compute the arctan (inverse of tan) of a number

`ceiling : (real -> int)`

Purpose: to determine the closest integer above a real number

---

`complex? : (any -> boolean)`

Purpose: to determine whether some value is complex

`conjugate : (num -> num)`

Purpose: to compute the conjugate of a complex number

`cos : (num -> num)`

Purpose: to compute the cosine of a number (radians)

`cosh : (num -> num)`

Purpose: to compute the hyperbolic cosine of a number

`current-seconds : (-> int)`

Purpose: to compute the current time in seconds elapsed (since a platform-specific starting date)

`denominator : (rat -> int)`

Purpose: to compute the denominator of a rational

`e : real`

Purpose: Euler's number

`even? : (integer -> boolean)`

Purpose: to determine if some value is even or not

`exact->inexact : (num -> num)`

Purpose: to convert an exact number to an inexact one

`exact? : (num -> boolean)`

Purpose: to determine whether some number is exact

`exp : (num -> num)`

Purpose: to compute e raised to a number

`expt : (num num -> num)`

Purpose: to compute the power of the first to the second number

---

`floor : (real -> int)`

Purpose: to determine the closest integer below a real number

---

`gcd : (int int ... -> int)`

Purpose: to compute the greatest common divisior

`imag-part : (num -> real)`

Purpose: to extract the imaginary part from a complex number

`inexact->exact : (num -> num)`

Purpose: to approximate an inexact number by an exact one

`inexact? : (num -> boolean)`

Purpose: to determine whether some number is inexact

`integer->char : (int -> char)`

Purpose: to lookup the character that corresponds to the given integer in the ASCII table (if any)

`integer? : (any -> boolean)`

Purpose: to determine whether some value is an integer (exact or inexact)

`lcm : (int int ... -> int)`

Purpose: to compute the least common multiple of two integers

---

`log : (num -> num)`

Purpose: to compute the base-e logarithm of a number

`magnitude : (num -> real)`

Purpose: to determine the magnitude of a complex number

`make-polar : (real real -> num)`

Purpose: to create a complex from a magnitude and angle

`max : (real real ... -> real)`

Purpose: to determine the largest number

`min : (real real ... -> real)`

Purpose: to determine the smallest number

`modulo : (int int -> int)`

Purpose: to compute first number modulo second number

`negative? : (number -> boolean)`

Purpose: to determine if some value is strictly smaller than zero

---

`number->string : (num -> string)`

Purpose: to convert a number to a string

`number? : (any -> boolean)`

Purpose: to determine whether some value is a number

`numerator : (rat -> int)`

Purpose: to compute the numerator of a rational

`odd? : (integer -> boolean)`

Purpose: to determine if some value is odd or not

`pi : real`

Purpose: the ratio of a circle's circumference to its diameter

---

`positive? : (number -> boolean)`

Purpose: to determine if some value is strictly larger than zero

---

```
quotient : (int int -> int)
```

Purpose: to compute the quotient of two integers

```
random : (int -> int)
```

Purpose: to generate a random natural number less than some given integer

---

```
rational? : (any -> boolean)
```

Purpose: to determine whether some value is a rational number

---

```
real-part : (num -> real)
```

Purpose: to extract the real part from a complex number

```
real? : (any -> boolean)
```

Purpose: to determine whether some value is a real number

```
remainder : (int int -> int)
```

Purpose: to compute the remainder of dividing the first by the second integer

---

```
round : (real -> int)
```

Purpose: to round a real number to an integer (rounds to even to break ties)

```
sgn : (real -> (union 1 1.0 0 0.0 -1 -1.0))
```

Purpose: to compute the sign of a real number

```
sin : (num -> num)
```

Purpose: to compute the sine of a number (radians)

`sinh : (num -> num)`

Purpose: to compute the hyperbolic sine of a number

`sqr : (num -> num)`

Purpose: to compute the square of a number

`sqrt : (num -> num)`

Purpose: to compute the square root of a number

`sub1 : (number -> number)`

Purpose: to compute a number one smaller than a given number

`tan : (num -> num)`

Purpose: to compute the tangent of a number (radians)

`zero? : (number -> boolean)`

Purpose: to determine if some value is zero or not

`boolean=? : (boolean boolean -> boolean)`

Purpose: to determine whether two booleans are equal

`boolean? : (any -> boolean)`

Purpose: to determine whether some value is a boolean

`false? : (any -> boolean)`

Purpose: to determine whether a value is false

`not : (boolean -> boolean)`

Purpose: to compute the negation of a boolean value

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

Purpose: to convert a symbol to a string

```
symbol=? : (symbol symbol -> boolean)
```

Purpose: to determine whether two symbols are equal

```
symbol? : (any -> boolean)
```

Purpose: to determine whether some value is a symbol

```
append : ((listof any)
          (listof any)
          (listof any)
          ...
          ->
          (listof any))
```

Purpose: to create a single list from several, by juxtaposition of the items

---

```
assq : (X
        (listof (cons X Y))
        ->
        (union false (cons X Y)))
```

Purpose: to determine whether some item is the first item of a pair in a list of pairs

---

```
caaar : ((cons
          (cons (cons W (listof Z)) (listof Y))
          (listof X))
         ->
         W)
```

Purpose: to select the first item of the first list in the first list of a list

---

```
caadr : ((cons
          (cons (cons W (listof Z)) (listof Y))
          (listof X))
         ->
         (listof Z))
```

Purpose: to select the rest of the first list in the first list of a list

---

```
caar : ((cons (cons Z (listof Y)) (listof X)) -> Z)
```

Purpose: to select the first item of the first list in a list

---

```
cadar : ((cons (cons W (cons Z (listof Y))) (listof X))
         ->
         Z)
```

Purpose: to select the second item of the first list of a list

---

```
cadddr : ((listof Y) -> Y)
```

Purpose: to select the fourth item of a non-empty list

```
caddr : ((cons W (cons Z (cons Y (listof X)))) -> Y)
```

Purpose: to select the third item of a non-empty list

```
cadr : ((cons Z (cons Y (listof X))) -> Y)
```

Purpose: to select the second item of a non-empty list

```
car : ((cons Y (listof X)) -> Y)
```

Purpose: to select the first item of a non-empty list

```
cdaar : ((cons
          (cons (cons W (listof Z)) (listof Y))
          (listof X))
         ->
         (listof Z))
```

Purpose: to select the rest of the first list in the first list of a list

---

```
cdadr : ((cons W (cons (cons Z (listof Y)) (listof X)))
         ->
         (listof Y))
```

Purpose: to select the rest of the first list in the rest of a list

---

```
cdar : ((cons (cons Z (listof Y)) (listof X))
        ->
        (listof Y))
```

Purpose: to select the rest of a non-empty list in a list

```
cddar : ((cons (cons W (cons Z (listof Y))) (listof X))
         ->
         (listof Y))
```

Purpose: to select the rest of the rest of the first list of a list

---

```
cdddr : ((cons W (cons Z (cons Y (listof X))))
         ->
         (listof X))
```

Purpose: to select the rest of the rest of the rest of a list

---

```
cddr : ((cons Z (cons Y (listof X))) -> (listof X))
```

Purpose: to select the rest of the rest of a list

```
cdr : ((cons Y (listof X)) -> (listof X))
```

Purpose: to select the rest of a non-empty list

```
cons : (X (listof X) -> (listof X))
```

Purpose: to construct a list

```
cons? : (any -> boolean)
```

Purpose: to determine whether some value is a constructed list

---

```
eighth : ((listof Y) -> Y)
```

Purpose: to select the eighth item of a non-empty list

```
empty? : (any -> boolean)
```

Purpose: to determine whether some value is the empty list

```
fifth : ((listof Y) -> Y)
```

Purpose: to select the fifth item of a non-empty list

```
first : ((cons Y (listof X)) -> Y)
```

Purpose: to select the first item of a non-empty list

```
fourth : ((listof Y) -> Y)
```

Purpose: to select the fourth item of a non-empty list

```
length : (list -> number)
```

Purpose: to compute the number of items on a list

```
list : (any ... -> (listof any))
```

Purpose: to construct a list of its arguments

```
list* : (any ... (listof any) -> (listof any))
```

Purpose: to construct a list by adding multiple items to a list

---

```
list-ref : ((listof X) natural-number -> X)
```

Purpose: to extract the indexed item from the list

```
member : (any list -> (union false list))
```

Purpose: to determine whether some value is on the list (comparing values with equal?)

```
memq : (any list -> (union false list))
```

Purpose: to determine whether some value is on some list (comparing values with eq?)

```
memv : (any list -> (union false list))
```

Purpose: to determine whether some value is on the list (comparing values with eqv?)

```
null : empty
```

Purpose: the empty list

```
null? : (any -> boolean)
```

Purpose: to determine whether some value is the empty list

```
pair? : (any -> boolean)
```

Purpose: to determine whether some value is a constructed list

---

```
rest : ((cons Y (listof X)) -> (listof X))
```

Purpose: to select the rest of a non-empty list

```
reverse : (list -> list)
```

Purpose: to create a reversed version of a list

```
second : ((cons Z (cons Y (listof X))) -> Y)
```

Purpose: to select the second item of a non-empty list

```
seventh : ((listof Y) -> Y)
```

Purpose: to select the seventh item of a non-empty list

`sixth : ((listof Y) -> Y)`

Purpose: to select the sixth item of a non-empty list

`third : ((cons W (cons Z (cons Y (listof X)))) -> Y)`

Purpose: to select the third item of a non-empty list

`make-posn : (number number -> posn)`

Purpose: to construct a posn

`posn-x : (posn -> number)`

Purpose: to extract the x component of a posn

`posn-y : (posn -> number)`

Purpose: to extract the y component of a posn

`posn? : (anything -> boolean)`

Purpose: to determine if its input is a posn

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

Purpose: to lookup the number that corresponds to the given character in the ASCII table (if any)

`char-alphabetic? : (char -> boolean)`

Purpose: to determine whether a character represents an alphabetic character

---

`char-ci<=? : (char char ... -> boolean)`

Purpose: to determine whether a character precedes another (or is equal to it) in a case-insensitive manner

---

`char-ci<?` : `(char char ... -> boolean)`

Purpose: to determine whether a character precedes another in a case-insensitive manner

---

`char-ci=?` : `(char char ... -> boolean)`

Purpose: to determine whether two characters are equal in a case-insensitive manner

---

`char-ci>=?` : `(char char ... -> boolean)`

Purpose: to determine whether a character succeeds another (or is equal to it) in a case-insensitive manner

---

`char-ci>?` : `(char char ... -> boolean)`

Purpose: to determine whether a character succeeds another in a case-insensitive manner

---

`char-downcase` : `(char -> char)`

Purpose: to determine the equivalent lower-case character

`char-lower-case?` : `(char -> boolean)`

Purpose: to determine whether a character is a lower-case character

---

`char-numeric?` : `(char -> boolean)`

Purpose: to determine whether a character represents a digit

`char-upcase` : `(char -> char)`

Purpose: to determine the equivalent upper-case character

`char-upper-case?` : `(char -> boolean)`

Purpose: to determine whether a character is an upper-case character

---

`char-whitespace?` : `(char -> boolean)`

Purpose: to determine whether a character represents space

```
char<=? : (char char ... -> boolean)
```

Purpose: to determine whether a character precedes another (or is equal to it)

```
char<? : (char char ... -> boolean)
```

Purpose: to determine whether a character precedes another

```
char=? : (char char ... -> boolean)
```

Purpose: to determine whether two characters are equal

```
char>=? : (char char ... -> boolean)
```

Purpose: to determine whether a character succeeds another (or is equal to it)

```
char>? : (char char ... -> boolean)
```

Purpose: to determine whether a character succeeds another

```
char? : (any -> boolean)
```

Purpose: to determine whether a value is a character

```
format : (string any ... -> string)
```

Purpose: to format a string, possibly embedding values

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

Purpose: to convert a s list of characters into a string

```
make-string : (nat char -> string)
```

Purpose: to produce a string of given length from a single given character

---

`string` : `(char ... -> string)`

Purpose: (string c1 c2 ...) builds a string

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

Purpose: to convert a string into a list of characters

`string->number` : `(string -> (union number false))`

Purpose: to convert a string into a number, produce false if impossible

---

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

Purpose: to convert a string into a symbol

`string-append` : `(string ... -> string)`

Purpose: to juxtapose the characters of several strings

`string-ci<=?` : `(string string ... -> boolean)`

Purpose: to determine whether one string alphabetically precedes another (or is equal to it) in a case-insensitive manner

---

`string-ci<?` : `(string string ... -> boolean)`

Purpose: to determine whether one string alphabetically precedes another in a case-insensitive manner

---

`string-ci=?` : `(string string ... -> boolean)`

Purpose: to compare two strings character-wise in a case-insensitive manner

---

`string-ci>=?` : `(string string ... -> boolean)`

Purpose: to determine whether one string alphabetically succeeds another (or is equal to it) in a case-insensitive manner

---

`string-ci>?` : `(string string ... -> boolean)`

Purpose: to determine whether one string alphabetically succeeds another in a case-insensitive manner

---

`string-copy` : `(string -> string)`

Purpose: to copy a string

`string-length` : `(string -> nat)`

Purpose: to determine the length of a string

`string-ref` : `(string nat -> char)`

Purpose: to extract the i-the character from a string

`string<=?` : `(string string ... -> boolean)`

Purpose: to determine whether one string alphabetically precedes another (or is equal to it)

`string<?` : `(string string ... -> boolean)`

Purpose: to determine whether one string alphabetically precedes another

---

`string=?` : `(string string ... -> boolean)`

Purpose: to compare two strings character-wise

`string>=?` : `(string string ... -> boolean)`

Purpose: to determine whether one string alphabetically succeeds another (or is equal to it)

`string>?` : `(string string ... -> boolean)`

Purpose: to determine whether one string alphabetically succeeds another

---

`string?` : `(any -> boolean)`

Purpose: to determine whether a value is a string

`substring : (string nat nat -> string)`

Purpose: to extract the substring starting at a 0-based index up to the second 0-based index (exclusive)

`image=? : (image image -> boolean)`

Purpose: to determine whether two images are equal

`image? : (any -> boolean)`

Purpose: to determine whether a value is an image

`=~ : (real real non-negative-real -> boolean)`

Purpose: to check whether two real numbers are within some amount (the third argument) of either other

---

`eof : eof`

Purpose: the end-of-file value

`eof-object? : (any -> boolean)`

Purpose: to determine whether some value is the end-of-file value

---

`eq? : (any any -> boolean)`

Purpose: to compare two values

`equal? : (any any -> boolean)`

Purpose: to determine whether two values are structurally equal

---

`equal~? : (any any non-negative-real -> boolean)`

Purpose: to compare like equal? on the first two arguments, except using =~ in the case of

real numbers

```
eqv? : (any any -> boolean)
```

Purpose: to compare two values

```
error : (symbol string -> void)
```

Purpose: to signal an error

```
exit : (-> void)
```

Purpose: to exit the running program

```
identity : (any -> any)
```

Purpose: to return the argument unchanged

```
struct? : (any -> boolean)
```

Purpose: to determine whether some value is a structure

```
andmap : ((X -> boolean) (listof X) -> boolean)
```

Purpose: (andmap p (list x-1 ... x-n)) = (and (p x-1) (and ... (p x-n)))

```
apply : ((X-1 ... X-N -> Y)
         X-1
         ...
         X-i
         (list X-i+1 ... X-N)
         ->
         Y)
```

Purpose: to apply a function using items from a list as the arguments

---

```
build-list : (nat (nat -> X) -> (listof X))
```

Purpose: (build-list n f) = (list (f 0) ... (f (- n 1)))

```
build-string : (nat (nat -> char) -> string)
```

Purpose: (build-string n f) = (string (f 0) ... (f (- n 1)))

```
compose : ((Y-1 -> Z)
           ...
           (Y-N -> Y-N-1)
           (X-1 ... X-N -> Y-N)
           ->
           (X-1 ... X-N -> Z))
```

Purpose: to compose a sequence of procedures into a single procedure

---

```
filter : ((X -> boolean) (listof X) -> (listof X))
```

Purpose: to construct a list from all those items on a list for which the predicate holds

---

```
foldl : ((X Y -> Y) Y (listof X) -> Y)
```

Purpose: (foldl f base (list x-1 ... x-n)) = (f x-n ... (f x-1 base))

```
foldr : ((X Y -> Y) Y (listof X) -> Y)
```

Purpose: (foldr f base (list x-1 ... x-n)) = (f x-1 ... (f x-n base))

```
for-each : ((any ... -> any) (listof any) ... -> void)
```

Purpose: to apply a function to each item on one or more lists for effect only

---

```
map : ((X ... -> Z) (listof X) ... -> (listof Z))
```

Purpose: to construct a new list by applying a function to each item on one or more existing lists

---

```
memf : ((X -> boolean)
        (listof X)
        ->
        (union false (listof X)))
```

Purpose: to determine whether the first argument produces true for some value in the second argument

---

```
ormap : ((X -> boolean) (listof X) -> boolean)
```

Purpose: (ormap p (list x-1 ... x-n)) = (or (p x-1) (or ... (p x-n)))

```
procedure? : (any -> boolean)
```

Purpose: to determine if a value is a procedure

```
quicksort : ((listof X) (X X -> boolean) -> (listof X))
```

Purpose: to construct a list from all items on a list in an order according to a predicate

4.5 Unchanged Forms

```
(define-struct structid (fieldid ...))
```

The same as Intermediate's `define-struct`.

```
(local [definition ...] expr)
(letrec ([id expr-for-let] ...) expr)
(let ([id expr-for-let] ...) expr)
(let* ([id expr-for-let] ...) expr)
```

The same as Intermediate's `local`, `letrec`, `let`, and `let*`.

```
(cond [expr expr] ... [expr expr])
else
```

The same as Beginning's `cond`.

```
(if expr expr expr)
```

The same as Beginning's `if`.

```
(and expr expr expr ...)
```

```
(or expr expr expr ...)
```

The same as Beginning's `and` and `or`.

```
(time expr)
```

The same as Intermediate's `time`.

```
(check-expect expr expr)
(check-within expr expr expr)
(check-error expr expr)
```

The same as Beginning's `check-expect`, etc.

```
empty : empty?
true : boolean?
false : boolean?
```

Constants for the empty list, true, and false.

```
(require string)
```

The same as Beginning's `require`.

5 Advanced Student

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

def-or-expr = definition
            | expr
            | library-require

 definition = (define (id id id ...) expr)
            | (define id expr)
            | (define-struct id (id ...))

       expr = (begin expr expr ...)
            | (begin0 expr expr ...)
            | (set! id expr)
            | (delay expr)
            | (lambda (id id ...) expr)
            | (local [definition ...] expr)
            | (letrec ([id expr] ...) expr)
            | (shared ([id expr] ...) expr)
            | (let ([id expr] ...) expr)
            | (let id ([id expr] ...) expr)
            | (let* ([id expr] ...) expr)
            | (recur id ([id expr] ...) expr)
            | (expr expr expr ...) ; function call
            | (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])
            | (if expr expr expr)
            | (when expr expr)
            | (unless expr expr)
            | (and expr expr expr ...)
            | (or expr expr expr ...)
            | (time expr)
            | test-case
            | empty
            | id ; identifier
            | prim-op ; primitive operation
            | 'id
            | 'quoted ; quoted value
            | `quasiquoted ; quasiquote
            | number
```

```
                | true
                | false
                | string
                | character

        choice = id ; treated as a symbol
                | number

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

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

     test-case = (check-expect expr expr)
                | (check-within expr expr expr)
                | (check-error expr expr)

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

       package = (string string number number)
```

An *id* 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 *string* is enclosed by a pair of ". Unlike symbols, strings may be split into characters and manipulated by a variety of primitive 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.

A *prim-op* is one of:

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

```
* : (num num num ... -> num)
+ : (num num num ... -> num)
- : (num num ... -> num)
/ : (num num num ... -> num)
< : (real real real ... -> boolean)
<= : (real real real ... -> boolean)
= : (num num num ... -> boolean)
> : (real real real ... -> boolean)
>= : (real real ... -> boolean)
abs : (real -> real)
acos : (num -> num)
add1 : (number -> number)
angle : (num -> real)
asin : (num -> num)
atan : (num -> num)
ceiling : (real -> int)
complex? : (any -> boolean)
conjugate : (num -> num)
cos : (num -> num)
cosh : (num -> num)
current-seconds : (-> int)
denominator : (rat -> int)
e : real
even? : (integer -> boolean)
exact->inexact : (num -> num)
exact? : (num -> boolean)
exp : (num -> num)
expt : (num num -> num)
floor : (real -> int)
gcd : (int int ... -> int)
imag-part : (num -> real)
inexact->exact : (num -> num)
inexact? : (num -> boolean)
integer->char : (int -> char)
integer? : (any -> boolean)
lcm : (int int ... -> int)
log : (num -> num)
magnitude : (num -> real)
make-polar : (real real -> num)
```

```
max : (real real ... -> real)
min : (real real ... -> real)
modulo : (int int -> int)
negative? : (number -> boolean)
number->string : (num -> string)
number? : (any -> boolean)
numerator : (rat -> int)
odd? : (integer -> boolean)
pi : real
positive? : (number -> boolean)
quotient : (int int -> int)
random : (int -> int)
rational? : (any -> boolean)
real-part : (num -> real)
real? : (any -> boolean)
remainder : (int int -> int)
round : (real -> int)
sgn : (real -> (union 1 1.0 0 0.0 -1 -1.0))
sin : (num -> num)
sinh : (num -> num)
sqr : (num -> num)
sqrt : (num -> num)
sub1 : (number -> number)
tan : (num -> num)
zero? : (number -> boolean)
```

Booleans

```
boolean=? : (boolean boolean -> boolean)
boolean? : (any -> boolean)
false? : (any -> boolean)
not : (boolean -> boolean)
```

Symbols

```
symbol->string : (symbol -> string)
symbol=? : (symbol symbol -> boolean)
symbol? : (any -> boolean)
```

Lists

```
append : ((listof any) ... -> (listof any))
assq : (X
        (listof (cons X Y))
        ->
        (union false (cons X Y)))
caaar : ((cons
          (cons (cons W (listof Z)) (listof Y))
          (listof X))
         ->
         W)
```

```
caadr : ((cons
          (cons (cons W (listof Z)) (listof Y))
          (listof X))
         ->
         (listof Z))
caar : ((cons (cons Z (listof Y)) (listof X)) -> Z)
cadar : ((cons (cons W (cons Z (listof Y))) (listof X))
         ->
         Z)
cadddr : ((listof Y) -> Y)
caddr : ((cons W (cons Z (cons Y (listof X)))) -> Y)
cadr : ((cons Z (cons Y (listof X))) -> Y)
car : ((cons Y (listof X)) -> Y)
cdaar : ((cons
          (cons (cons W (listof Z)) (listof Y))
          (listof X))
         ->
         (listof Z))
cdadr : ((cons W (cons (cons Z (listof Y)) (listof X)))
         ->
         (listof Y))
cdar : ((cons (cons Z (listof Y)) (listof X))
        ->
        (listof Y))
cddar : ((cons (cons W (cons Z (listof Y))) (listof X))
         ->
         (listof Y))
cdddr : ((cons W (cons Z (cons Y (listof X))))
         ->
         (listof X))
cddr : ((cons Z (cons Y (listof X))) -> (listof X))
cdr : ((cons Y (listof X)) -> (listof X))
cons : (X (listof X) -> (listof X))
cons? : (any -> boolean)
eighth : ((listof Y) -> Y)
empty? : (any -> boolean)
fifth : ((listof Y) -> Y)
first : ((cons Y (listof X)) -> Y)
fourth : ((listof Y) -> Y)
length : (list -> number)
list : (any ... -> (listof any))
list-ref : ((listof X) natural-number -> X)
list? : (any -> boolean)
member : (any list -> (union false list))
memq : (any list -> (union false list))
memv : (any list -> (union false list))
```

```
null : empty
null? : (any -> boolean)
pair? : (any -> boolean)
rest : ((cons Y (listof X)) -> (listof X))
reverse : (list -> list)
second : ((cons Z (cons Y (listof X))) -> Y)
seventh : ((listof Y) -> Y)
sixth : ((listof Y) -> Y)
third : ((cons W (cons Z (cons Y (listof X)))) -> Y)
```

Posns

```
make-posn : (number number -> posn)
posn-x : (posn -> number)
posn-y : (posn -> number)
posn? : (anything -> boolean)
set-posn-x! : (posn number -> void)
set-posn-y! : (posn number -> void)
```

Characters

```
char->integer : (char -> integer)
char-alphabetic? : (char -> boolean)
char-ci<=? : (char char ... -> boolean)
char-ci<? : (char char ... -> boolean)
char-ci=? : (char char ... -> boolean)
char-ci>=? : (char char ... -> boolean)
char-ci>? : (char char ... -> boolean)
char-downcase : (char -> char)
char-lower-case? : (char -> boolean)
char-numeric? : (char -> boolean)
char-upcase : (char -> char)
char-upper-case? : (char -> boolean)
char-whitespace? : (char -> boolean)
char<=? : (char char ... -> boolean)
char<? : (char char ... -> boolean)
char=? : (char char ... -> boolean)
char>=? : (char char ... -> boolean)
char>? : (char char ... -> boolean)
char? : (any -> boolean)
```

Strings

```
format : (string any ... -> string)
list->string : ((listof char) -> string)
make-string : (nat char -> string)
string : (char ... -> string)
string->list : (string -> (listof char))
string->number : (string -> (union number false))
string->symbol : (string -> symbol)
string-append : (string ... -> string)
string-ci<=? : (string string ... -> boolean)
```

```
string-ci<? : (string string ... -> boolean)
string-ci=? : (string string ... -> boolean)
string-ci>=? : (string string ... -> boolean)
string-ci>? : (string string ... -> boolean)
string-copy : (string -> string)
string-length : (string -> nat)
string-ref : (string nat -> char)
string<=? : (string string ... -> boolean)
string<? : (string string ... -> boolean)
string=? : (string string ... -> boolean)
string>=? : (string string ... -> boolean)
string>? : (string string ... -> boolean)
string? : (any -> boolean)
substring : (string nat nat -> string)
```

Images

```
image=? : (image image -> boolean)
image? : (any -> boolean)
```

Misc

```
=~ : (real real non-negative-real -> boolean)
eof : eof
eof-object? : (any -> boolean)
eq? : (any any -> boolean)
equal? : (any any -> boolean)
equal~? : (any any non-negative-real -> boolean)
eqv? : (any any -> boolean)
error : (symbol string -> void)
exit : (-> void)
force : (delay -> any)
identity : (any -> any)
promise? : (any -> boolean)
struct? : (any -> boolean)
void : (-> void)
void? : (any -> boolean)
```

Higher-Order Functions

```
andmap : ((X -> boolean) (listof X) -> boolean)
apply : ((X-1 ... X-N -> Y)
         X-1
         ...
         X-i
         (list X-i+1 ... X-N)
         ->
         Y)
build-list : (nat (nat -> X) -> (listof X))
build-string : (nat (nat -> char) -> string)
```

```
compose : ((Y-1 -> Z)
           ...
           (Y-N -> Y-N-1)
           (X-1 ... X-N -> Y-N)
           ->
           (X-1 ... X-N -> Z))
filter : ((X -> boolean) (listof X) -> (listof X))
foldl : ((X Y -> Y) Y (listof X) -> Y)
foldr : ((X Y -> Y) Y (listof X) -> Y)
for-each : ((any ... -> any) (listof any) ... -> void)
map : ((X ... -> Z) (listof X) ... -> (listof Z))
memf : ((X -> boolean)
        (listof X)
        ->
        (union false (listof X)))
ormap : ((X -> boolean) (listof X) -> boolean)
procedure? : (any -> boolean)
quicksort : ((listof X) (X X -> boolean) -> (listof X))
```

Reading and Printing

```
display : (any -> void)
newline : (-> void)
pretty-print : (any -> void)
print : (any -> void)
printf : (string any ... -> void)
read : (-> sexp)
write : (any -> void)
```

Vectors

```
build-vector : (nat (nat -> X) -> (vectorof X))
make-vector : (number X -> (vectorof X))
vector : (X ... -> (vector X ...))
vector-length : ((vector X) -> nat)
vector-ref : ((vector X) nat -> X)
vector-set! : ((vectorof X) nat X -> void)
vector? : (any -> boolean)
```

Boxes

```
box : (any -> box)
box? : (any -> boolean)
set-box! : (box any -> void)
unbox : (box -> any)
```

5.1 `define`

```
(define (id id ...) expr)
(define id expr)
```

The same as Intermediate with Lambda's `define`, except that a function is allowed to accept zero arguments.

5.2 `define-struct`

```
(define-struct structid (fieldid ...))
```

The same as Intermediate's `define-struct`, but defines an additional set of operations:

- `make-structid` : takes a number of arguments equal to the number of fields in the structure type, and creates a new instance of the structure type.
- `set-structid-fieldid!` : takes an instance of the structure and a value, and changes the instance's field to the given value.

5.3 `lambda`

```
(lambda (id ...) expr)
```

The same as Intermediate with Lambda's `lambda`, except that a function is allowed to accept zero arguments.

5.4 `begin`

```
(begin expr expr ...)
```

Evaluates the *expr*s in order from left to right. The value of the `begin` expression is the value of the last *expr*.

5.5 `begin0`

```
(begin0 expr expr ...)
```

Evaluates the *expr*s in order from left to right. The value of the `begin` expression is the value of the first *expr*.

5.6 set!

```
(set! id expr)
```

Evaluates *expr*, and then changes the definition *id* to have *expr*'s value. The *id* must be defined or bound by `letrec`, `let`, or `let*`.

5.7 delay

```
(delay expr)
```

Produces a "promise" to evaluate *expr*. The *expr* is not evaluated until the promise is forced through the `force` operator; when the promise is forced, the result is recorded, so that any further `force` of the promise always produces the remembered value.

5.8 shared

```
(shared ([id expr] ...) expr)
```

Like `letrec`, but when an *expr* next to an *id* is a `cons`, `list`, `vector`, quasiquoted expression, or `make-`*structid* from a `define-struct`, the *expr* can refer directly to any *id*, not just *id*s defined earlier. Thus, `shared` can be used to create cyclic data structures.

5.9 let

```
(let ([id expr] ...) expr)
(let id ([id expr] ...) expr)
```

The first form of `let` is the same as Intermediate's `let`.

The second form is equivalent to a `recur` form.

5.10 recur

```
(recur id ([id expr] ...) expr)
```

A short-hand recursion construct. The first *id* corresponds to the name of the recursive function. The parenthesized *id*s are the function's arguments, and each corresponding *expr* is a value supplied for that argument in an initial starting call of the function. The last *expr* is the body of the function.

More precisely, a `recur` form

```
(recur func-id ([arg-id arg-expr] ...)
  body-expr)
```

is equivalent to

```
((local [(define (func-id arg-id ...)
           body-expr)]
   func-id)
  arg-expr ...)
```

5.11 `case`

`(case` *expr* `[(`*choice* `...)` *expr*`]` `...` `[(`*choice* `...)` *expr*`])`

A `case` form contains one or more "lines" that are surrounded by parentheses or square brackets. Each line contains a sequence of choices—numbers and names for symbols—and an answer *expr*. The initial *expr* is evaluated, and the resulting value is compared to the choices in each line, where the lines are considered in order. The first line that contains a matching choice provides an answer *expr* whose value is the result of the whole `case` expression. If none of the lines contains a matching choice, it is an error.

`(cond` *expr* `[(`*choice* `...)` *expr*`]` `...` `[else` *expr*`])`

This form of `case` is similar to the prior one, except that the final `else` clause is always taken if no prior line contains a choice matching the value of the initial *expr*. In other words, so there is no possibility to "fall off them end" of the `case` form.

5.12 `when` and `unless`

`(when` *expr* *expr*`)`

The first *expr* (known as the "test" expression) is evaluated. If it evaluates to `true`, the result of the `when` expression is the result of evaluating the second *expr*, otherwise the

result is (void) and the second *expr* is not evaluated. If the result of evaluating the test *expr* is neither true nor false, it is an error.

(unless *expr expr*)

Like when, but the second *expr* is evaluated when the first *expr* produces false instead of true.

5.13 Primitive Operations

---

* : (num num num ... -> num)

Purpose: to compute the product of all of the input numbers

+ : (num num num ... -> num)

Purpose: to compute the sum of the input numbers

- : (num num ... -> num)

Purpose: to subtract the second (and following) number(s) from the first; negate the number if there is only one argument

---

/ : (num num num ... -> num)

Purpose: to divide the first by the second (and all following) number(s); only the first number can be zero.

< : (real real real ... -> boolean)

Purpose: to compare real numbers for less-than

<= : (real real real ... -> boolean)

Purpose: to compare real numbers for less-than or equality

= : (num num num ... -> boolean)

Purpose: to compare numbers for equality

```
> : (real real real ... -> boolean)
```

Purpose: to compare real numbers for greater-than

```
>= : (real real ... -> boolean)
```

Purpose: to compare real numbers for greater-than or equality

---

```
abs : (real -> real)
```

Purpose: to compute the absolute value of a real number

```
acos : (num -> num)
```

Purpose: to compute the arccosine (inverse of cos) of a number

---

```
add1 : (number -> number)
```

Purpose: to compute a number one larger than a given number

```
angle : (num -> real)
```

Purpose: to extract the angle from a complex number

```
asin : (num -> num)
```

Purpose: to compute the arcsine (inverse of sin) of a number

```
atan : (num -> num)
```

Purpose: to compute the arctan (inverse of tan) of a number

```
ceiling : (real -> int)
```

Purpose: to determine the closest integer above a real number

---

```
complex? : (any -> boolean)
```

Purpose: to determine whether some value is complex

```
conjugate : (num -> num)
```

Purpose: to compute the conjugate of a complex number

```
cos : (num -> num)
```

Purpose: to compute the cosine of a number (radians)

```
cosh : (num -> num)
```

Purpose: to compute the hyperbolic cosine of a number

```
current-seconds : (-> int)
```

Purpose: to compute the current time in seconds elapsed (since a platform-specific starting date)

```
denominator : (rat -> int)
```

Purpose: to compute the denominator of a rational

```
e : real
```

Purpose: Euler's number

```
even? : (integer -> boolean)
```

Purpose: to determine if some value is even or not

```
exact->inexact : (num -> num)
```

Purpose: to convert an exact number to an inexact one

```
exact? : (num -> boolean)
```

Purpose: to determine whether some number is exact

`exp : (num -> num)`

Purpose: to compute e raised to a number

`expt : (num num -> num)`

Purpose: to compute the power of the first to the second number

---

`floor : (real -> int)`

Purpose: to determine the closest integer below a real number

---

`gcd : (int int ... -> int)`

Purpose: to compute the greatest common divisior

`imag-part : (num -> real)`

Purpose: to extract the imaginary part from a complex number

`inexact->exact : (num -> num)`

Purpose: to approximate an inexact number by an exact one

`inexact? : (num -> boolean)`

Purpose: to determine whether some number is inexact

`integer->char : (int -> char)`

Purpose: to lookup the character that corresponds to the given integer in the ASCII table (if any)

`integer? : (any -> boolean)`

Purpose: to determine whether some value is an integer (exact or inexact)

`lcm : (int int ... -> int)`

Purpose: to compute the least common multiple of two integers

---

`log : (num -> num)`

Purpose: to compute the base-e logarithm of a number

`magnitude : (num -> real)`

Purpose: to determine the magnitude of a complex number

`make-polar : (real real -> num)`

Purpose: to create a complex from a magnitude and angle

`max : (real real ... -> real)`

Purpose: to determine the largest number

`min : (real real ... -> real)`

Purpose: to determine the smallest number

`modulo : (int int -> int)`

Purpose: to compute first number modulo second number

`negative? : (number -> boolean)`

Purpose: to determine if some value is strictly smaller than zero

---

`number->string : (num -> string)`

Purpose: to convert a number to a string

`number? : (any -> boolean)`

Purpose: to determine whether some value is a number

```
numerator : (rat -> int)
```

Purpose: to compute the numerator of a rational

```
odd? : (integer -> boolean)
```

Purpose: to determine if some value is odd or not

```
pi : real
```

Purpose: the ratio of a circle's circumference to its diameter

---

```
positive? : (number -> boolean)
```

Purpose: to determine if some value is strictly larger than zero

---

```
quotient : (int int -> int)
```

Purpose: to compute the quotient of two integers

```
random : (int -> int)
```

Purpose: to generate a random natural number less than some given integer

---

```
rational? : (any -> boolean)
```

Purpose: to determine whether some value is a rational number

---

```
real-part : (num -> real)
```

Purpose: to extract the real part from a complex number

```
real? : (any -> boolean)
```

Purpose: to determine whether some value is a real number

```
remainder : (int int -> int)
```

Purpose: to compute the remainder of dividing the first by the second integer

---

```
round : (real -> int)
```

Purpose: to round a real number to an integer (rounds to even to break ties)

```
sgn : (real -> (union 1 1.0 0 0.0 -1 -1.0))
```

Purpose: to compute the sign of a real number

```
sin : (num -> num)
```

Purpose: to compute the sine of a number (radians)

```
sinh : (num -> num)
```

Purpose: to compute the hyperbolic sine of a number

```
sqr : (num -> num)
```

Purpose: to compute the square of a number

```
sqrt : (num -> num)
```

Purpose: to compute the square root of a number

```
sub1 : (number -> number)
```

Purpose: to compute a number one smaller than a given number

```
tan : (num -> num)
```

Purpose: to compute the tangent of a number (radians)

```
zero? : (number -> boolean)
```

Purpose: to determine if some value is zero or not

```
boolean=? : (boolean boolean -> boolean)
```

Purpose: to determine whether two booleans are equal

```
boolean? : (any -> boolean)
```

Purpose: to determine whether some value is a boolean

```
false? : (any -> boolean)
```

Purpose: to determine whether a value is false

```
not : (boolean -> boolean)
```

Purpose: to compute the negation of a boolean value

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

Purpose: to convert a symbol to a string

```
symbol=? : (symbol symbol -> boolean)
```

Purpose: to determine whether two symbols are equal

```
symbol? : (any -> boolean)
```

Purpose: to determine whether some value is a symbol

```
append : ((listof any) ... -> (listof any))
```

Purpose: to create a single list from several

```
assq : (X
        (listof (cons X Y))
        ->
        (union false (cons X Y)))
```

Purpose: to determine whether some item is the first item of a pair in a list of pairs

---

```
caaar : ((cons
          (cons (cons W (listof Z)) (listof Y))
          (listof X))
         ->
         W)
```

Purpose: to select the first item of the first list in the first list of a list

---

```
caadr : ((cons
          (cons (cons W (listof Z)) (listof Y))
          (listof X))
         ->
         (listof Z))
```

Purpose: to select the rest of the first list in the first list of a list

---

```
caar : ((cons (cons Z (listof Y)) (listof X)) -> Z)
```

Purpose: to select the first item of the first list in a list

---

```
cadar : ((cons (cons W (cons Z (listof Y))) (listof X))
         ->
         Z)
```

Purpose: to select the second item of the first list of a list

---

```
cadddr : ((listof Y) -> Y)
```

Purpose: to select the fourth item of a non-empty list

```
caddr : ((cons W (cons Z (cons Y (listof X)))) -> Y)
```

Purpose: to select the third item of a non-empty list

```
cadr : ((cons Z (cons Y (listof X))) -> Y)
```

Purpose: to select the second item of a non-empty list

```
car : ((cons Y (listof X)) -> Y)
```

Purpose: to select the first item of a non-empty list

```
cdaar : ((cons
          (cons (cons W (listof Z)) (listof Y))
          (listof X))
         ->
         (listof Z))
```

Purpose: to select the rest of the first list in the first list of a list

---

```
cdadr : ((cons W (cons (cons Z (listof Y)) (listof X)))
         ->
         (listof Y))
```

Purpose: to select the rest of the first list in the rest of a list

---

```
cdar : ((cons (cons Z (listof Y)) (listof X))
        ->
        (listof Y))
```

Purpose: to select the rest of a non-empty list in a list

```
cddar : ((cons (cons W (cons Z (listof Y))) (listof X))
         ->
         (listof Y))
```

Purpose: to select the rest of the rest of the first list of a list

---

```
cdddr : ((cons W (cons Z (cons Y (listof X))))
         ->
         (listof X))
```

Purpose: to select the rest of the rest of the rest of a list

---

```
cddr : ((cons Z (cons Y (listof X))) -> (listof X))
```

Purpose: to select the rest of the rest of a list

```
cdr : ((cons Y (listof X)) -> (listof X))
```

Purpose: to select the rest of a non-empty list

```
cons : (X (listof X) -> (listof X))
```

Purpose: to construct a list

```
cons? : (any -> boolean)
```

Purpose: to determine whether some value is a constructed list

---

```
eighth : ((listof Y) -> Y)
```

Purpose: to select the eighth item of a non-empty list

```
empty? : (any -> boolean)
```

Purpose: to determine whether some value is the empty list

```
fifth : ((listof Y) -> Y)
```

Purpose: to select the fifth item of a non-empty list

```
first : ((cons Y (listof X)) -> Y)
```

Purpose: to select the first item of a non-empty list

```
fourth : ((listof Y) -> Y)
```

Purpose: to select the fourth item of a non-empty list

```
length : (list -> number)
```

Purpose: to compute the number of items on a list

```
list : (any ... -> (listof any))
```

Purpose: to construct a list of its arguments

```
list-ref : ((listof X) natural-number -> X)
```

Purpose: to extract the indexed item from the list

```
list? : (any -> boolean)
```

Purpose: to determine whether some value is a list

```
member : (any list -> (union false list))
```

Purpose: to determine whether some value is on the list (comparing values with equal?)

```
memq : (any list -> (union false list))
```

Purpose: to determine whether some value is on some list (comparing values with eq?)

```
memv : (any list -> (union false list))
```

Purpose: to determine whether some value is on the list (comparing values with eqv?)

```
null : empty
```

Purpose: the empty list

```
null? : (any -> boolean)
```

Purpose: to determine whether some value is the empty list

```
pair? : (any -> boolean)
```

Purpose: to determine whether some value is a constructed list

---

```
rest : ((cons Y (listof X)) -> (listof X))
```

Purpose: to select the rest of a non-empty list

```
reverse : (list -> list)
```

Purpose: to create a reversed version of a list

```
second : ((cons Z (cons Y (listof X))) -> Y)
```

Purpose: to select the second item of a non-empty list

```
seventh : ((listof Y) -> Y)
```

Purpose: to select the seventh item of a non-empty list

```
sixth : ((listof Y) -> Y)
```

Purpose: to select the sixth item of a non-empty list

```
third : ((cons W (cons Z (cons Y (listof X)))) -> Y)
```

Purpose: to select the third item of a non-empty list

```
make-posn : (number number -> posn)
```

Purpose: to construct a posn

```
posn-x : (posn -> number)
```

Purpose: to extract the x component of a posn

```
posn-y : (posn -> number)
```

Purpose: to extract the y component of a posn

```
posn? : (anything -> boolean)
```

Purpose: to determine if its input is a posn

```
set-posn-x! : (posn number -> void)
```

Purpose: to update the x component of a posn

```
set-posn-y! : (posn number -> void)
```

Purpose: to update the x component of a posn

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

Purpose: to lookup the number that corresponds to the given character in the ASCII table (if any)

`char-alphabetic? : (char -> boolean)`

Purpose: to determine whether a character represents an alphabetic character

---

`char-ci<=? : (char char ... -> boolean)`

Purpose: to determine whether a character precedes another (or is equal to it) in a case-insensitive manner

---

`char-ci<? : (char char ... -> boolean)`

Purpose: to determine whether a character precedes another in a case-insensitive manner

---

`char-ci=? : (char char ... -> boolean)`

Purpose: to determine whether two characters are equal in a case-insensitive manner

---

`char-ci>=? : (char char ... -> boolean)`

Purpose: to determine whether a character succeeds another (or is equal to it) in a case-insensitive manner

---

`char-ci>? : (char char ... -> boolean)`

Purpose: to determine whether a character succeeds another in a case-insensitive manner

---

`char-downcase : (char -> char)`

Purpose: to determine the equivalent lower-case character

`char-lower-case? : (char -> boolean)`

Purpose: to determine whether a character is a lower-case character

---

`char-numeric? : (char -> boolean)`

Purpose: to determine whether a character represents a digit

`char-upcase : (char -> char)`

Purpose: to determine the equivalent upper-case character

`char-upper-case? : (char -> boolean)`

Purpose: to determine whether a character is an upper-case character

---

`char-whitespace? : (char -> boolean)`

Purpose: to determine whether a character represents space

`char<=? : (char char ... -> boolean)`

Purpose: to determine whether a character precedes another (or is equal to it)

`char<? : (char char ... -> boolean)`

Purpose: to determine whether a character precedes another

`char=? : (char char ... -> boolean)`

Purpose: to determine whether two characters are equal

`char>=? : (char char ... -> boolean)`

Purpose: to determine whether a character succeeds another (or is equal to it)

`char>? : (char char ... -> boolean)`

Purpose: to determine whether a character succeeds another

`char?` : `(any -> boolean)`

Purpose: to determine whether a value is a character

`format` : `(string any ... -> string)`

Purpose: to format a string, possibly embedding values

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

Purpose: to convert a s list of characters into a string

`make-string` : `(nat char -> string)`

Purpose: to produce a string of given length from a single given character

---

`string` : `(char ... -> string)`

Purpose: (string c1 c2 ...) builds a string

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

Purpose: to convert a string into a list of characters

`string->number` : `(string -> (union number false))`

Purpose: to convert a string into a number, produce false if impossible

---

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

Purpose: to convert a string into a symbol

`string-append` : `(string ... -> string)`

Purpose: to juxtapose the characters of several strings

`string-ci<=?` : `(string string ... -> boolean)`

Purpose: to determine whether one string alphabetically precedes another (or is equal to it)

in a case-insensitive manner

`string-ci<? : (string string ... -> boolean)`

Purpose: to determine whether one string alphabetically precedes another in a case-insensitive manner

---

`string-ci=? : (string string ... -> boolean)`

Purpose: to compare two strings character-wise in a case-insensitive manner

---

`string-ci>=? : (string string ... -> boolean)`

Purpose: to determine whether one string alphabetically succeeds another (or is equal to it) in a case-insensitive manner

---

`string-ci>? : (string string ... -> boolean)`

Purpose: to determine whether one string alphabetically succeeds another in a case-insensitive manner

---

`string-copy : (string -> string)`

Purpose: to copy a string

`string-length : (string -> nat)`

Purpose: to determine the length of a string

`string-ref : (string nat -> char)`

Purpose: to extract the i-the character from a string

`string<=? : (string string ... -> boolean)`

Purpose: to determine whether one string alphabetically precedes another (or is equal to it)

`string<? : (string string ... -> boolean)`

Purpose: to determine whether one string alphabetically precedes another

---

`string=?` : `(string string ... -> boolean)`

Purpose: to compare two strings character-wise

`string>=?` : `(string string ... -> boolean)`

Purpose: to determine whether one string alphabetically succeeds another (or is equal to it)

`string>?` : `(string string ... -> boolean)`

Purpose: to determine whether one string alphabetically succeeds another

---

`string?` : `(any -> boolean)`

Purpose: to determine whether a value is a string

`substring` : `(string nat nat -> string)`

Purpose: to extract the substring starting at a 0-based index up to the second 0-based index (exclusive)

`image=?` : `(image image -> boolean)`

Purpose: to determine whether two images are equal

`image?` : `(any -> boolean)`

Purpose: to determine whether a value is an image

`=~` : `(real real non-negative-real -> boolean)`

Purpose: to check whether two real numbers are within some amount (the third argument) of either other

---

`eof` : `eof`

Purpose: the end-of-file value

`eof-object?` : `(any -> boolean)`

Purpose: to determine whether some value is the end-of-file value

---

`eq?` : `(any any -> boolean)`

Purpose: to compare two values

`equal?` : `(any any -> boolean)`

Purpose: to determine whether two values are structurally equal

---

`equal~?` : `(any any non-negative-real -> boolean)`

Purpose: to compare like equal? on the first two arguments, except using $=\sim$ in the case of real numbers

---

`eqv?` : `(any any -> boolean)`

Purpose: to compare two values

`error` : `(symbol string -> void)`

Purpose: to signal an error

`exit` : `(-> void)`

Purpose: to exit the running program

`force` : `(delay -> any)`

Purpose: to find the delayed value; see also delay

`identity` : `(any -> any)`

Purpose: to return the argument unchanged

`promise?` : `(any -> boolean)`

Purpose: to determine if a value is delayed

```
struct? : (any -> boolean)
```

Purpose: to determine whether some value is a structure

```
void : (-> void)
```

Purpose: produces a void value

```
void? : (any -> boolean)
```

Purpose: to determine if a value is void

```
andmap : ((X -> boolean) (listof X) -> boolean)
```

Purpose: (andmap p (list x-1 ... x-n)) = (and (p x-1) (and ... (p x-n)))

```
apply : ((X-1 ... X-N -> Y)
         X-1
         ...
         X-i
         (list X-i+1 ... X-N)
         ->
         Y)
```

Purpose: to apply a function using items from a list as the arguments

---

```
build-list : (nat (nat -> X) -> (listof X))
```

Purpose: (build-list n f) = (list (f 0) ... (f (- n 1)))

```
build-string : (nat (nat -> char) -> string)
```

Purpose: (build-string n f) = (string (f 0) ... (f (- n 1)))

```
compose : ((Y-1 -> Z)
           ...
           (Y-N -> Y-N-1)
           (X-1 ... X-N -> Y-N)
           ->
           (X-1 ... X-N -> Z))
```

Purpose: to compose a sequence of procedures into a single procedure

---

```
filter : ((X -> boolean) (listof X) -> (listof X))
```

Purpose: to construct a list from all those items on a list for which the predicate holds

---

```
foldl : ((X Y -> Y) Y (listof X) -> Y)
```

Purpose: (foldl f base (list x-1 ... x-n)) = (f x-n ... (f x-1 base))

```
foldr : ((X Y -> Y) Y (listof X) -> Y)
```

Purpose: (foldr f base (list x-1 ... x-n)) = (f x-1 ... (f x-n base))

```
for-each : ((any ... -> any) (listof any) ... -> void)
```

Purpose: to apply a function to each item on one or more lists for effect only

---

```
map : ((X ... -> Z) (listof X) ... -> (listof Z))
```

Purpose: to construct a new list by applying a function to each item on one or more existing lists

---

```
memf : ((X -> boolean)
        (listof X)
        ->
        (union false (listof X)))
```

Purpose: to determine whether the first argument produces true for some value in the second argument

---

```
ormap : ((X -> boolean) (listof X) -> boolean)
```

Purpose: (ormap p (list x-1 ... x-n)) = (or (p x-1) (or ... (p x-n)))

```
procedure? : (any -> boolean)
```

Purpose: to determine if a value is a procedure

```
quicksort : ((listof X) (X X -> boolean) -> (listof X))
```

Purpose: to construct a list from all items on a list in an order according to a predicate

---

```
display : (any -> void)
```

Purpose: to print the argument to stdout (without quotes on symbols and strings, etc.)

```
newline : (-> void)
```

Purpose: to print a newline to stdout

```
pretty-print : (any -> void)
```

Purpose: like write, but with standard newlines and indentation

---

```
print : (any -> void)
```

Purpose: to print the argument as a value to stdout

```
printf : (string any ... -> void)
```

Purpose: to format the rest of the arguments according to the first argument and print it to stdout

---

```
read : (-> sexp)
```

Purpose: to read input from the user

```
write : (any -> void)
```

Purpose: to print the argument to stdout (in a traditional style that is somewhere between print and display)

`build-vector : (nat (nat -> X) -> (vectorof X))`

Purpose: to construct a vector

`make-vector : (number X -> (vectorof X))`

Purpose: to construct a vector

`vector : (X ... -> (vector X ...))`

Purpose: to construct a vector

`vector-length : ((vector X) -> nat)`

Purpose: to determine the length of a vector

`vector-ref : ((vector X) nat -> X)`

Purpose: to extract an element from a vector

`vector-set! : ((vectorof X) nat X -> void)`

Purpose: to update a vector

`vector? : (any -> boolean)`

Purpose: to determine if a value is a vector

`box : (any -> box)`

Purpose: to construct a box

`box? : (any -> boolean)`

Purpose: to determine if a value is a box

`set-box! : (box any -> void)`

Purpose: to update a box

```
unbox : (box -> any)
```

Purpose: to extract the boxed value

5.14 Unchanged Forms

```
(local [definition ...] expr)
(letrec ([id expr-for-let] ...) expr)
(let* ([id expr-for-let] ...) expr)
```

The same as Intermediate's `local`, `letrec`, and `let*`.

```
(cond [expr expr] ... [expr expr])
else
```

The same as Beginning's `cond`, except that `else` can be used with `case`.

```
(if expr expr expr)
```

The same as Beginning's `if`.

```
(and expr expr expr ...)
(or expr expr expr ...)
```

The same as Beginning's `and` and `or`.

```
(time expr)
```

The same as Intermediate's `time`.

```
(check-expect expr expr)
(check-within expr expr expr)
(check-error expr expr)
```

The same as Beginning's `check-expect`, etc.

```
empty : empty?
true : boolean?
false : boolean?
```

Constants for the empty list, true, and false.

```
(require string)
```

The same as Beginning's `require`.

Index