

# *How to Design Programs Languages*

Version 5.0

June 6, 2010

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

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

# Contents

<b>1</b>	<b>Beginning Student</b>	<b>5</b>
1.1	define . . . . .	11
1.2	define-struct . . . . .	11
1.3	Function Calls . . . . .	12
1.4	Primitive Calls . . . . .	12
1.5	cond . . . . .	13
1.6	if . . . . .	13
1.7	and . . . . .	13
1.8	or . . . . .	14
1.9	Test Cases . . . . .	14
1.10	empty . . . . .	15
1.11	Identifiers . . . . .	15
1.12	Symbols . . . . .	15
1.13	true and false . . . . .	15
1.14	require . . . . .	16
1.15	Primitive Operations . . . . .	16
<b>2</b>	<b>Beginning Student with List Abbreviations</b>	<b>37</b>
2.1	Quote . . . . .	43
2.2	Quasiquote . . . . .	44
2.3	Primitive Operations . . . . .	44
2.4	Unchanged Forms . . . . .	64
<b>3</b>	<b>Intermediate Student</b>	<b>66</b>
3.1	define . . . . .	73

3.2	define-struct . . . . .	73
3.3	local . . . . .	74
3.4	letrec, let, and let* . . . . .	74
3.5	Function Calls . . . . .	74
3.6	time . . . . .	75
3.7	Identifiers . . . . .	75
3.8	Primitive Operations . . . . .	75
3.9	Unchanged Forms . . . . .	97
<b>4</b>	<b>Intermediate Student with Lambda</b>	<b>99</b>
4.1	define . . . . .	106
4.2	lambda . . . . .	106
4.3	Function Calls . . . . .	107
4.4	Primitive Operation Names . . . . .	107
4.5	Unchanged Forms . . . . .	129
<b>5</b>	<b>Advanced Student</b>	<b>131</b>
5.1	define . . . . .	139
5.2	define-struct . . . . .	139
5.3	lambda . . . . .	140
5.4	Function Calls . . . . .	140
5.5	begin . . . . .	140
5.6	begin0 . . . . .	140
5.7	set! . . . . .	141
5.8	delay . . . . .	141
5.9	shared . . . . .	141

5.10 let . . . . .	141
5.11 recur . . . . .	141
5.12 case . . . . .	142
5.13 when and unless . . . . .	142
5.14 Primitive Operations . . . . .	143
5.15 Unchanged Forms . . . . .	168
<b>Index</b>	<b>170</b>

# 1 Beginning Student

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

def-or-expr = definition
            | expr
            | test-case
            | 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 ...)
      | empty
      | id
      | id ; identifier
      | 'id ; symbol
      | number
      | true
      | false
      | string
      | character

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

library-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**

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

```

integer->char : (integer -> char)
integer-sqrt : (number -> integer)
integer? : (any -> boolean)
lcm : (integer integer ... -> integer)
log : (number -> number)
magnitude : (number -> real)
make-polar : (real real -> number)
make-rectangular : (real real -> number)
max : (real real ... -> real)
min : (real real ... -> real)
modulo : (integer integer -> integer)
negative? : (number -> boolean)
number->string : (number -> string)
number? : (any -> boolean)
numerator : (rat -> integer)
odd? : (integer -> boolean)
pi : real
positive? : (number -> boolean)
quotient : (integer integer -> integer)
random : (integer -> integer)
rational? : (any -> boolean)
real-part : (number -> real)
real? : (any -> boolean)
remainder : (integer integer -> integer)
round : (real -> integer)
sgn : (real -> (union 1 1.0 0 0.0 -1 -1.0))
sin : (number -> number)
sinh : (number -> number)
sqr : (number -> number)
sqrt : (number -> number)
sub1 : (number -> number)
tan : (number -> number)
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)
caddr : ((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))
cddr : ((cons W (cons Z (cons Y (listof X))))
        ->
        (listof X))
caddr : ((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 : ((listof any) -> number)
list : (any ... -> (listof any))
list* : (any ... (listof any) -> (listof any))
list-ref : ((listof X) natural-number -> X)
make-list : (natural-number any -> (listof any))
member : (any (listof any) -> boolean)
member? : (any (listof any) -> boolean)
memq : (any (listof any) -> (union false list))
memv : (any (listof any) -> (union false list))
null : empty
null? : (any -> boolean)
pair? : (any -> boolean)
remove : (any (listof any) -> (listof any))
rest : ((cons Y (listof X)) -> (listof X))
reverse : ((listof any) -> 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 char ... -> boolean)
char-ci<? : (char char char ... -> boolean)
char-ci=? : (char char char ... -> boolean)
char-ci>=? : (char char char ... -> boolean)
char-ci>? : (char 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 char ... -> boolean)
char<? : (char char char ... -> boolean)
char=? : (char char char ... -> boolean)
char>=? : (char char char ... -> boolean)
char>? : (char char char ... -> boolean)
char? : (any -> boolean)
```

### Strings

```
explode : (string -> (listof string))
format : (string any ... -> string)
implode : ((listof string) -> string)
int->string : (integer -> string)
list->string : ((listof char) -> string)
make-string : (nat char -> string)
replicate : (nat string -> string)
string : (char ... -> string)
string->int : (string -> integer)
string->list : (string -> (listof char))
string->number : (string -> (union number false))
string->symbol : (string -> symbol)
string-alphabetic? : (string -> boolean)
string-append : (string ... -> string)
string-ci<=? : (string string string ... -> boolean)
string-ci<? : (string string string ... -> boolean)
string-ci=? : (string string string ... -> boolean)
string-ci>=? : (string string string ... -> boolean)
string-ci>? : (string string string ... -> boolean)
string-copy : (string -> string)
string-ith : (string nat -> string)
string-length : (string -> nat)
string-lower-case? : (string -> boolean)
string-numeric? : (string -> boolean)
string-ref : (string nat -> char)
string-upper-case? : (string -> boolean)
string-whitespace? : (string -> boolean)
string<=? : (string string string ... -> boolean)
string<? : (string string string ... -> boolean)
string=? : (string string string ... -> boolean)
string>=? : (string string string ... -> boolean)
string>? : (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 : (any ... -> 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 *ids* 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 *ids* 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 *fieldids* 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 *exprs* 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 *exprs* 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 *exprs* 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 `exprs`: 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 `exprs` 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 the 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 `exprs` 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 `exprs` 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 `exprs` 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 `exprs` 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`.

---

`(check-member-of expr expr expr ...)`

A test case to check that the first `expr` produces an element that is equivalent to one of the following `exprs`.

---

`(check-range expr expr expr)`

A test case to check that the first `expr` produces a number inbetween the numbers produced

by the second and third *exprs*, inclusive.

## 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 module-id)`

Accesses a file in an installed library. The library name is an identifier with the same constraints as for a relative-path string, with the additional constraint that it must not contain a `..`.

---

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

Accesses a file in an installed library, making its definitions available in the current module (i.e., current file). The first *string* names the library file, and the remaining *strings* 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 its definitions available in the current module (i.e., current file).

## 1.15 Primitive Operations

---

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

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

---

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

Purpose: to compute the sum of the input numbers

---

`- : (number number ... -> number)`

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

---

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

Purpose: to divide the first by the second (and all following) number(s); try `(/ 3 4)` and `(/ 3 2 2)` 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

---

`= : (number number number ... -> 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 : (number -> number)`

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` : (number -> real)

Purpose: to extract the angle from a complex number

---

`asin` : (number -> number)

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

---

`atan` : (number -> number)

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

---

`ceiling` : (real -> integer)

Purpose: to determine the closest integer (exact or inexact) above a real number

---

`complex?` : (any -> boolean)

Purpose: to determine whether some value is complex

---

`conjugate` : (number -> number)

Purpose: to compute the conjugate of a complex number

---

`cos` : (number -> number)

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

---

`cosh` : (number -> number)

Purpose: to compute the hyperbolic cosine of a number

---

`current-seconds` : (-> integer)

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

---

`denominator` : (rat -> integer)

Purpose: to compute the denominator of a rational

---

`e` : real

Purpose: Euler's number

---

`even?` : (integer -> boolean)

Purpose: to determine if some integer (exact or inexact) is even or not

---

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

Purpose: to convert an exact number to an inexact one

---

`exact?` : (number -> boolean)

Purpose: to determine whether some number is exact

---

`exp` : (number -> number)

Purpose: to compute e raised to a number

---

`expt` : (number number -> number)

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

---

`floor` : (real -> integer)

Purpose: to determine the closest integer (exact or inexact) below a real number

---

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

Purpose: to compute the greatest common divisor of two integers (exact or inexact)

---

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

Purpose: to extract the imaginary part from a complex number

---

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

Purpose: to approximate an inexact number by an exact one

---

`inexact?` : (number -> boolean)

Purpose: to determine whether some number is inexact

---

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

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

---

`integer-sqrt` : (number -> integer)

Purpose: to compute the integer (exact or inexact) square root of a number

---

`integer?` : (any -> boolean)

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

---

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

Purpose: to compute the least common multiple of two integers (exact or inexact)

---

`log` : (number -> number)

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

---

`magnitude` : (number -> real)

Purpose: to determine the magnitude of a complex number

---

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

Purpose: to create a complex from a magnitude and angle

---

`make-rectangular` : (real real -> number)

Purpose: to create a complex from a real and an imaginary part

---

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

Purpose: to determine the largest number

---

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

Purpose: to determine the smallest number

---

```
modulo : (integer integer -> integer)
```

Purpose: to find the remainder of the division of the first number by the second; try (modulo 4 3) (modulo 4 -3)

---

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

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

---

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

Purpose: to convert a number to a string

---

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

Purpose: to determine whether some value is a number

---

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

Purpose: to compute the numerator of a rational

---

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

Purpose: to determine if some integer (exact or inexact) 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` : (integer integer -> integer)

Purpose: to divide the first integer (exact or inexact) into the second; try (quotient 3 4) and (quotient 4 3)

---

`random` : (integer -> integer)

Purpose: to generate a random natural number less than some given integer (exact only!)

---

`rational?` : (any -> boolean)

Purpose: to determine whether some value is a rational number

---

`real-part` : (number -> 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` : (integer integer -> integer)

Purpose: to determine the remainder of dividing the first by the second integer (exact or inexact)

---

`round` : (real -> integer)

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` : (number -> number)

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

---

`sinh` : (number -> number)

Purpose: to compute the hyperbolic sine of a number

---

`sqr` : (number -> number)

Purpose: to compute the square of a number

---

`sqrt` : (number -> number)

Purpose: to compute the square root of a number

---

`sub1` : (number -> number)

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

---

`tan` : (number -> number)

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

---

```
caddr : ((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 : ((listof any) -> 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

---

```
make-list : (natural-number any -> (listof any))
```

Purpose: (make-list k x) constructs a list of k copies of x

---

```
member : (any (listof any) -> boolean)
```

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

---

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

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

---

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

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

---

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

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

---

```
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

---

```
remove : (any (listof any) -> (listof any))
```

Purpose: to construct a list like the given one with the first occurrence of the given item removed (comparing values with equal?)

---

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

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

---

```
reverse : ((listof any) -> 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 char ... -> boolean)

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

insensitive manner

---

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

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

---

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

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

---

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

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

---

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

Purpose: to determine whether a character precedes another

---

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

Purpose: to determine whether two characters are equal

---

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

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

---

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

Purpose: to determine whether a character succeeds another

---

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

Purpose: to determine whether a value is a character

---

```
explode : (string -> (listof string))
```

Purpose: to translate a string into a list of 1-letter strings

---

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

Purpose: to format a string, possibly embedding values

---

```
implode : ((listof string) -> string)
```

Purpose: to concatenate the list of 1-letter strings into one string

---

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

Purpose: to convert an integer in [0,55295] or [57344 1114111] to a 1-letter string

---

`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

---

`replicate` : (nat string -> string)

Purpose: to replicate the given string

---

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

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

---

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

Purpose: to convert a 1-letter string to an integer in [0,55295] or [57344, 1114111]

---

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

Purpose: to determine whether all 'letters' in the string are alphabetic

---

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

Purpose: to juxtapose the characters of several strings

---

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

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

---

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

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

---

`string-ci>=?` : (string 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 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-ith` : (string nat -> string)

Purpose: to extract the ith 1-letter substring from the given one

---

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

Purpose: to determine the length of a string

---

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

Purpose: to determine whether all 'letters' in the string are lower case

---

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

Purpose: to determine whether all 'letters' in the string are numeric

---

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

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

---

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

Purpose: to determine whether all 'letters' in the string are upper case

---

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

Purpose: to determine whether all 'letters' in the string are white space

---

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

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

---

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

Purpose: to determine whether one string alphabetically precedes another

---

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

Purpose: to compare two strings character-wise

---

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

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

---

`string>?` : (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 determine whether two values are equivalent from the computer's perspective (intensional)

---

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

Purpose: to determine whether two values are structurally equal where basic values are compared with the `eqv?` predicate

---

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

---

`error` : (any ... -> void)

Purpose: to signal an error, combining the given values into an error message.

If any of the values' printed representations is too long, it is truncated and “...” is put into the string. If the first value is a symbol, it is treated specially; it is suffixed with a colon and a space (the intention is that the symbol is the name of the function signalling the 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
            | test-case
            | 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 ...)
      | 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-member-of expr expr ...)
| (check-range expr expr expr)
| (check-error expr expr)

library-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**

```

* : (number number number ... -> number)
+ : (number number number ... -> number)
- : (number number ... -> number)
/ : (number number number ... -> number)
< : (real real real ... -> boolean)
<= : (real real real ... -> boolean)
= : (number number number ... -> boolean)
> : (real real real ... -> boolean)
>= : (real real ... -> boolean)
abs : (real -> real)
acos : (number -> number)
add1 : (number -> number)
angle : (number -> real)
asin : (number -> number)

```

```

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

```

```
sinh : (number -> number)
sqr  : (number -> number)
sqrt : (number -> number)
sub1 : (number -> number)
tan  : (number -> number)
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)

caddr : ((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))
cddddr : ((cons W (cons Z (cons Y (listof X))))
          ->
          (listof X))
caddr : ((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 : ((listof any) -> number)
list : (any ... -> (listof any))
list* : (any ... (listof any) -> (listof any))
list-ref : ((listof X) natural-number -> X)
make-list : (natural-number any -> (listof any))
member : (any (listof any) -> boolean)
member? : (any (listof any) -> boolean)
memq : (any (listof any) -> (union false list))
memv : (any (listof any) -> (union false list))
null : empty
null? : (any -> boolean)
pair? : (any -> boolean)
remove : (any (listof any) -> (listof any))
rest : ((cons Y (listof X)) -> (listof X))
reverse : ((listof any) -> 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 char ... -> boolean)
char-ci<? : (char char char ... -> boolean)
char-ci=? : (char char char ... -> boolean)
char-ci>=? : (char char char ... -> boolean)
char-ci>? : (char 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 char ... -> boolean)
char<? : (char char char ... -> boolean)
char=? : (char char char ... -> boolean)
char>=? : (char char char ... -> boolean)
char>? : (char char char ... -> boolean)
char? : (any -> boolean)
```

### Strings

```
explode : (string -> (listof string))
format : (string any ... -> string)
implode : ((listof string) -> string)
int->string : (integer -> string)
list->string : ((listof char) -> string)
make-string : (nat char -> string)
replicate : (nat string -> string)
string : (char ... -> string)
string->int : (string -> integer)
string->list : (string -> (listof char))
string->number : (string -> (union number false))
string->symbol : (string -> symbol)
string-alphabetic? : (string -> boolean)
string-append : (string ... -> string)
string-ci<=? : (string string string ... -> boolean)
string-ci<? : (string string string ... -> boolean)
string-ci=? : (string string string ... -> boolean)
string-ci>=? : (string string string ... -> boolean)
string-ci>? : (string string string ... -> boolean)
string-copy : (string -> string)
string-ith : (string nat -> string)
```

```

string-length : (string -> nat)
string-lower-case? : (string -> boolean)
string-numeric? : (string -> boolean)
string-ref : (string nat -> char)
string-upper-case? : (string -> boolean)
string-whitespace? : (string -> boolean)
string<=? : (string string string ... -> boolean)
string<? : (string string string ... -> boolean)
string=? : (string string string ... -> boolean)
string>=? : (string string string ... -> boolean)
string>? : (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 : (any ... -> void)
exit : (-> void)
identity : (any -> any)
struct? : (any -> boolean)

```

## 2.1 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

---

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

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

---

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

Purpose: to compute the sum of the input numbers

---

```
- : (number number ... -> number)
```

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

---

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

Purpose: to divide the first by the second (and all following) number(s); try (/ 3 4) and (/ 3 2) 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

---

```
= : (number number number ... -> 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 : (number -> number)
```

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` : (number -> real)

Purpose: to extract the angle from a complex number

---

`asin` : (number -> number)

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

---

`atan` : (number -> number)

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

---

`ceiling` : (real -> integer)

Purpose: to determine the closest integer (exact or inexact) above a real number

---

`complex?` : (any -> boolean)

Purpose: to determine whether some value is complex

---

`conjugate` : (number -> number)

Purpose: to compute the conjugate of a complex number

---

`cos` : (number -> number)

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

---

`cosh` : (number -> number)

Purpose: to compute the hyperbolic cosine of a number

---

`current-seconds` : (-> integer)

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

---

`denominator` : (rat -> integer)

Purpose: to compute the denominator of a rational

---

`e` : real

Purpose: Euler's number

---

`even?` : (integer -> boolean)

Purpose: to determine if some integer (exact or inexact) is even or not

---

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

Purpose: to convert an exact number to an inexact one

---

`exact?` : (number -> boolean)

Purpose: to determine whether some number is exact

---

`exp` : (number -> number)

Purpose: to compute e raised to a number

---

`expt` : (number number -> number)

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

---

`floor` : (real -> integer)

Purpose: to determine the closest integer (exact or inexact) below a real number

---

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

Purpose: to compute the greatest common divisor of two integers (exact or inexact)

---

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

Purpose: to extract the imaginary part from a complex number

---

---

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

Purpose: to approximate an inexact number by an exact one

---

`inexact?` : (number -> boolean)

Purpose: to determine whether some number is inexact

---

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

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

---

`integer-sqrt` : (number -> integer)

Purpose: to compute the integer (exact or inexact) square root of a number

---

`integer?` : (any -> boolean)

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

---

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

Purpose: to compute the least common multiple of two integers (exact or inexact)

---

`log` : (number -> number)

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

---

`magnitude` : (number -> real)

Purpose: to determine the magnitude of a complex number

---

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

Purpose: to create a complex from a magnitude and angle

---

`make-rectangular` : (real real -> number)

Purpose: to create a complex from a real and an imaginary part

---

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

Purpose: to determine the largest number

---

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

Purpose: to determine the smallest number

---

`modulo` : (integer integer -> integer)

Purpose: to find the remainder of the division of the first number by the second; try (modulo 4 3) (modulo 4 -3)

---

`negative?` : (number -> boolean)

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

---

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

Purpose: to convert a number to a string

---

`number?` : (any -> boolean)

Purpose: to determine whether some value is a number

---

`numerator` : (rat -> integer)

Purpose: to compute the numerator of a rational

---

`odd?` : (integer -> boolean)

Purpose: to determine if some integer (exact or inexact) 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` : (integer integer -> integer)

Purpose: to divide the first integer (exact or inexact) into the second; try (quotient 3 4) and (quotient 4 3)

---

`random` : (integer -> integer)

Purpose: to generate a random natural number less than some given integer (exact only!)

---

`rational?` : (any -> boolean)

Purpose: to determine whether some value is a rational number

---

`real-part` : (number -> 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` : (integer integer -> integer)

Purpose: to determine the remainder of dividing the first by the second integer (exact or inexact)

---

`round` : (real -> integer)

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` : (number -> number)

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

---

`sinh` : (number -> number)

Purpose: to compute the hyperbolic sine of a number

---

```
sqr : (number -> number)
```

Purpose: to compute the square of a number

---

```
sqrt : (number -> number)
```

Purpose: to compute the square root of a number

---

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

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

---

```
tan : (number -> number)
```

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

---

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

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

---

```
cadadr : ((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 : ((listof any) -> 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

---

```
make-list : (natural-number any -> (listof any))
```

Purpose: (make-list k x) constructs a list of k copies of x

---

```
member : (any (listof any) -> boolean)
```

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

---

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

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

---

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

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

---

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

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

---

```
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

---

```
remove : (any (listof any) -> (listof any))
```

Purpose: to construct a list like the given one with the first occurrence of the given item removed (comparing values with equal?)

---

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

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

---

```
reverse : ((listof any) -> 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 char ... -> boolean)

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

insensitive manner

---

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

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

---

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

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

---

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

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

---

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

Purpose: to determine whether a character precedes another

---

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

Purpose: to determine whether two characters are equal

---

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

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

---

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

Purpose: to determine whether a character succeeds another

---

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

Purpose: to determine whether a value is a character

---

```
explode : (string -> (listof string))
```

Purpose: to translate a string into a list of 1-letter strings

---

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

Purpose: to format a string, possibly embedding values

---

```
implode : ((listof string) -> string)
```

Purpose: to concatenate the list of 1-letter strings into one string

---

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

Purpose: to convert an integer in [0,55295] or [57344 1114111] to a 1-letter string

---

`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

---

`replicate` : (nat string -> string)

Purpose: to replicate the given string

---

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

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

---

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

Purpose: to convert a 1-letter string to an integer in [0,55295] or [57344, 1114111]

---

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

Purpose: to determine whether all 'letters' in the string are alphabetic

---

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

Purpose: to juxtapose the characters of several strings

---

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

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

---

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

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

---

```
string-ci>=? : (string 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 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-ith : (string nat -> string)
```

Purpose: to extract the ith 1-letter substring from the given one

---

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

Purpose: to determine the length of a string

---

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

Purpose: to determine whether all 'letters' in the string are lower case

---

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

Purpose: to determine whether all 'letters' in the string are numeric

---

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

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

---

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

Purpose: to determine whether all 'letters' in the string are upper case

---

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

Purpose: to determine whether all 'letters' in the string are white space

---

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

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

---

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

Purpose: to determine whether one string alphabetically precedes another

---

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

Purpose: to compare two strings character-wise

---

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

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

---

`string>?` : (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 determine whether two values are equivalent from the computer's perspective (intensional)

---

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

Purpose: to determine whether two values are structurally equal where basic values are compared with the `eqv?` predicate

---

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

---

`error` : (any ... -> void)

Purpose: to signal an error, combining the given values into an error message.

If any of the values' printed representations is too long, it is truncated and “...” is put into the string. If the first value is a symbol, it is treated specially; it is suffixed with a colon and a space (the intention is that the symbol is the name of the function signalling the 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)
```

```
(check-member-of expr expr expr ...)
```

```
(check-range expr 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 module-path)
```

The same as Beginning's require.

### 3 Intermediate Student

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

def-or-expr = definition
            | expr
            | test-case
            | 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)
      | empty
      | id ; identifier
      | prim-op ; primitive operation
      | '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-member-of expr expr ...)
| (check-range expr expr expr)
| (check-error expr expr)

library-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**

```

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

```

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

```

real? : (any -> boolean)
remainder : (integer integer -> integer)
round : (real -> integer)
sgn : (real -> (union 1 1.0 0 0.0 -1 -1.0))
sin : (number -> number)
sinh : (number -> number)
sqr : (number -> number)
sqrt : (number -> number)
sub1 : (number -> number)
tan : (number -> number)
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)
caddr : ((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))
cddddr : ((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 : ((listof any) -> number)
list : (any ... -> (listof any))
list* : (any ... (listof any) -> (listof any))
list-ref : ((listof X) natural-number -> X)
make-list : (natural-number any -> (listof any))
member : (any (listof any) -> boolean)
member? : (any (listof any) -> boolean)
memq : (any (listof any) -> (union false list))
memv : (any (listof any) -> (union false list))
null : empty
null? : (any -> boolean)
pair? : (any -> boolean)
remove : (any (listof any) -> (listof any))
rest : ((cons Y (listof X)) -> (listof X))
reverse : ((listof any) -> 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 char ... -> boolean)
char-ci<? : (char char char ... -> boolean)
char-ci=? : (char char char ... -> boolean)
char-ci>=? : (char char char ... -> boolean)
char-ci>? : (char 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 char ... -> boolean)
char<? : (char char char ... -> boolean)
char=? : (char char char ... -> boolean)
char>=? : (char char char ... -> boolean)
char>? : (char char char ... -> boolean)
char? : (any -> boolean)
```

### Strings

```
explode : (string -> (listof string))
format : (string any ... -> string)
implode : ((listof string) -> string)
int->string : (integer -> string)
list->string : ((listof char) -> string)
make-string : (nat char -> string)
replicate : (nat string -> string)
string : (char ... -> string)
string->int : (string -> integer)
string->list : (string -> (listof char))
string->number : (string -> (union number false))
string->symbol : (string -> symbol)
string-alphabetic? : (string -> boolean)
string-append : (string ... -> string)
string-ci<=? : (string string string ... -> boolean)
string-ci<? : (string string string ... -> boolean)
string-ci=? : (string string string ... -> boolean)
string-ci>=? : (string string string ... -> boolean)
string-ci>? : (string string string ... -> boolean)
string-copy : (string -> string)
string-ith : (string nat -> string)
```

```

string-length : (string -> nat)
string-lower-case? : (string -> boolean)
string-numeric? : (string -> boolean)
string-ref : (string nat -> char)
string-upper-case? : (string -> boolean)
string-whitespace? : (string -> boolean)
string<=? : (string string string ... -> boolean)
string<? : (string string string ... -> boolean)
string=? : (string string string ... -> boolean)
string>=? : (string string string ... -> boolean)
string>? : (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 : (any ... -> void)
exit : (-> void)
identity : (any -> any)
struct? : (any -> boolean)

```

### Numbers (relaxed conditions)

```

* : (number ... -> number)
+ : (number ... -> number)
- : (number ... -> number)
/ : (number ... -> number)

```

### 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)
argmax : ((X -> real) (listof X) -> X)
argmin : ((X -> real) (listof X) -> X)
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))
sort : ((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 *definitions* and the *expr*) may refer to the names defined by the *definitions*. 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 *ids* can be used only in the last *expr*, not the *expr-for-lets* next to the *ids*.

---

```
(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 locally 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.

(The reported time is measured as the number of milliseconds of CPU time required to obtain this result, the number of “real” milliseconds required for the result, and the number of milliseconds of CPU time (included in the first result) spent on garbage collection. The reliability of the timing numbers depends on the platform.)

### 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.

---

```
< : (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

---

`= : (number number number ... -> 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 : (number -> number)`

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 : (number -> real)`

Purpose: to extract the angle from a complex number

---

`asin : (number -> number)`

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

---

`atan : (number -> number)`

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

---

`ceiling` : (real -> integer)

Purpose: to determine the closest integer (exact or inexact) above a real number

---

`complex?` : (any -> boolean)

Purpose: to determine whether some value is complex

---

`conjugate` : (number -> number)

Purpose: to compute the conjugate of a complex number

---

`cos` : (number -> number)

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

---

`cosh` : (number -> number)

Purpose: to compute the hyperbolic cosine of a number

---

`current-seconds` : (-> integer)

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

---

`denominator` : (rat -> integer)

Purpose: to compute the denominator of a rational

---

`e` : real

Purpose: Euler's number

---

`even?` : (integer -> boolean)

Purpose: to determine if some integer (exact or inexact) is even or not

---

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

Purpose: to convert an exact number to an inexact one

---

`exact?` : (number -> boolean)

Purpose: to determine whether some number is exact

---

`exp` : (number -> number)

Purpose: to compute e raised to a number

---

`expt` : (number number -> number)

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

---

`floor` : (real -> integer)

Purpose: to determine the closest integer (exact or inexact) below a real number

---

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

Purpose: to compute the greatest common divisor of two integers (exact or inexact)

---

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

Purpose: to extract the imaginary part from a complex number

---

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

Purpose: to approximate an inexact number by an exact one

---

`inexact?` : (number -> boolean)

Purpose: to determine whether some number is inexact

---

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

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

---

`integer-sqrt` : (number -> integer)

Purpose: to compute the integer (exact or inexact) square root of a number

---

`integer?` : (any -> boolean)

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

---

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

Purpose: to compute the least common multiple of two integers (exact or inexact)

---

`log` : (number -> number)

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

---

`magnitude` : (number -> real)

Purpose: to determine the magnitude of a complex number

---

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

Purpose: to create a complex from a magnitude and angle

---

`make-rectangular` : (real real -> number)

Purpose: to create a complex from a real and an imaginary part

---

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

Purpose: to determine the largest number

---

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

Purpose: to determine the smallest number

---

`modulo` : (integer integer -> integer)

Purpose: to find the remainder of the division of the first number by the second; try (modulo 4 3) (modulo 4 -3)

---

`negative?` : (number -> boolean)

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

---

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

Purpose: to convert a number to a string

---

`number?` : (any -> boolean)

Purpose: to determine whether some value is a number

---

`numerator` : (rat -> integer)

Purpose: to compute the numerator of a rational

---

`odd?` : (integer -> boolean)

Purpose: to determine if some integer (exact or inexact) 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` : (integer integer -> integer)

Purpose: to divide the first integer (exact or inexact) into the second; try (quotient 3 4) and (quotient 4 3)

---

`random` : (integer -> integer)

Purpose: to generate a random natural number less than some given integer (exact only!)

---

`rational?` : (any -> boolean)

Purpose: to determine whether some value is a rational number

---

`real-part` : (number -> 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` : (integer integer -> integer)

Purpose: to determine the remainder of dividing the first by the second integer (exact or inexact)

---

`round` : (real -> integer)

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` : (number -> number)

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

---

`sinh` : (number -> number)

Purpose: to compute the hyperbolic sine of a number

---

`sqr` : (number -> number)

Purpose: to compute the square of a number

---

`sqrt` : (number -> number)

Purpose: to compute the square root of a number

---

`sub1` : (number -> number)

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

---

`tan` : (number -> number)

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, 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

---

```
caddr : ((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

---

```
cddr : ((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` : ((listof any) -> 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

---

`make-list` : (natural-number any -> (listof any))

Purpose: (make-list k x) constructs a list of k copies of x

---

`member` : (any (listof any) -> boolean)

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

---

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

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

---

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

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

---

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

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

---

`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

---

`remove` : (any (listof any) -> (listof any))

Purpose: to construct a list like the given one with the first occurrence of the given item removed (comparing values with equal?)

---

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

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

---

```
reverse : ((listof any) -> 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 char ... -> boolean)

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

---

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

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

---

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

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

---

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

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

---

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

Purpose: to determine whether a character precedes another

---

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

Purpose: to determine whether two characters are equal

---

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

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

---

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

Purpose: to determine whether a character succeeds another

---

`char?` : (any -> boolean)

Purpose: to determine whether a value is a character

---

`explode` : (string -> (listof string))

Purpose: to translate a string into a list of 1-letter strings

---

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

Purpose: to format a string, possibly embedding values

---

`implode` : ((listof string) -> string)

Purpose: to concatenate the list of 1-letter strings into one string

---

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

Purpose: to convert an integer in [0,55295] or [57344 1114111] to a 1-letter string

---

`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

---

`replicate` : (nat string -> string)

Purpose: to replicate the given string

---

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

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

---

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

Purpose: to convert a 1-letter string to an integer in [0,55295] or [57344, 1114111]

---

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

Purpose: to determine whether all 'letters' in the string are alphabetic

---

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

Purpose: to juxtapose the characters of several strings

---

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

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

---

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

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

---

`string-ci>=?` : (string 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 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-ith` : (string nat -> string)

Purpose: to extract the ith 1-letter substring from the given one

---

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

Purpose: to determine the length of a string

---

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

Purpose: to determine whether all 'letters' in the string are lower case

---

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

Purpose: to determine whether all 'letters' in the string are numeric

---

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

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

---

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

Purpose: to determine whether all 'letters' in the string are upper case

---

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

Purpose: to determine whether all 'letters' in the string are white space

---

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

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

---

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

Purpose: to determine whether one string alphabetically precedes another

---

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

Purpose: to compare two strings character-wise

---

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

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

---

`string>?` : (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 determine whether two values are equivalent from the computer's perspective (intensional)

---

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

Purpose: to determine whether two values are structurally equal where basic values are compared with the `eqv?` predicate

---

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

---

`error` : (any ... -> void)

Purpose: to signal an error, combining the given values into an error message.

If any of the values' printed representations is too long, it is truncated and "..." is put into the string. If the first value is a symbol, it is treated specially; it is suffixed with a colon and a space (the intention is that the symbol is the name of the function signalling the 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

---

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

Purpose: to multiply all given numbers

---

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

Purpose: to add all given numbers

---

`- : (number ... -> number)`

Purpose: to subtract from the first all remaining numbers

---

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

Purpose: to divide the first by all remaining numbers

---

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

Purpose:  $(\text{andmap } p \text{ (list } x-1 \dots x-n)) = (\text{and } (p \ x-1) \dots (p \ x-n))$

---

`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

---

`argmax : ((X -> real) (listof X) -> X)`

Purpose: to find the (first) element of the list that maximizes the output of the function

---

`argmin : ((X -> real) (listof X) -> X)`

Purpose: to find the (first) element of the list that minimizes the output of the function

---

`build-list : (nat (nat -> X) -> (listof X))`

Purpose:  $(\text{build-list } n \text{ } f) = (\text{list } (f \ 0) \dots (f \ (- \ n \ 1)))$

---

`build-string` :  $(\text{nat } (\text{nat } \rightarrow \text{char}) \rightarrow \text{string})$

Purpose:  $(\text{build-string } n \text{ } f) = (\text{string } (f \ 0) \dots (f \ (- \ n \ 1)))$

---

`compose` :  $((Y-1 \rightarrow Z)$   
     $\dots$   
     $(Y-N \rightarrow Y-N-1)$   
     $(X-1 \dots X-N \rightarrow Y-N)$   
     $\rightarrow$   
     $(X-1 \dots X-N \rightarrow Z))$

Purpose: to compose a sequence of procedures into a single procedure

---

`filter` :  $((X \rightarrow \text{boolean}) (\text{listof } X) \rightarrow (\text{listof } X))$

Purpose: to construct a list from all those items on a list for which the predicate holds

---

`foldl` :  $((X \ Y \rightarrow Y) \ Y \ (\text{listof } X) \rightarrow Y)$

Purpose:  $(\text{foldl } f \ \text{base} \ (\text{list } x-1 \dots x-n)) = (f \ x-n \dots (f \ x-1 \ \text{base}))$

---

`foldr` :  $((X \ Y \rightarrow Y) \ Y \ (\text{listof } X) \rightarrow Y)$

Purpose:  $(\text{foldr } f \ \text{base} \ (\text{list } x-1 \dots x-n)) = (f \ x-1 \dots (f \ x-n \ \text{base}))$

---

`for-each` :  $((\text{any } \dots \rightarrow \text{any}) (\text{listof } \text{any}) \dots \rightarrow \text{void})$

Purpose: to apply a function to each item on one or more lists for effect only

---

`map` :  $((X \ \dots \rightarrow Z) (\text{listof } X) \dots \rightarrow (\text{listof } Z))$

Purpose: to construct a new list by applying a function to each item on one or more existing lists

---

`memf` :  $((X \rightarrow \text{boolean})$   
     $(\text{listof } X)$   
     $\rightarrow$   
     $(\text{union } \text{false} \ (\text{listof } X)))$

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) ... (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

---

```
sort : ((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)  
(check-member-of expr expr expr ...)
```

```
(check-range expr 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 module-path)
```

The same as Beginning's require.

## 4 Intermediate Student with Lambda

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

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

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

expr = (lambda (id id ...) expr)
      | (λ (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)
      | empty
      | id ; identifier
      | prim-op ; primitive operation
      | '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-member-of expr expr ...)
| (check-range expr expr expr)
| (check-error expr expr)

library-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**

```

< : (real real real ... -> boolean)
<= : (real real real ... -> boolean)
= : (number number number ... -> boolean)
> : (real real real ... -> boolean)
>= : (real real ... -> boolean)
abs : (real -> real)
acos : (number -> number)
add1 : (number -> number)

```

```

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

```

```

sgn : (real -> (union 1 1.0 0 0.0 -1 -1.0))
sin : (number -> number)
sinh : (number -> number)
sqr : (number -> number)
sqrt : (number -> number)
sub1 : (number -> number)
tan : (number -> number)
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)
caddr : ((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))
cddr : ((cons W (cons Z (cons Y (listof X))))
->
(listof X))
cdr : ((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 : ((listof any) -> number)
list : (any ... -> (listof any))
list* : (any ... (listof any) -> (listof any))
list-ref : ((listof X) natural-number -> X)
make-list : (natural-number any -> (listof any))
member : (any (listof any) -> boolean)
member? : (any (listof any) -> boolean)
memq : (any (listof any) -> (union false list))
memv : (any (listof any) -> (union false list))
null : empty
null? : (any -> boolean)
pair? : (any -> boolean)
remove : (any (listof any) -> (listof any))
rest : ((cons Y (listof X)) -> (listof X))
reverse : ((listof any) -> 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 char ... -> boolean)
char-ci<? : (char char char ... -> boolean)
char-ci=? : (char char char ... -> boolean)
char-ci>=? : (char char char ... -> boolean)
char-ci>? : (char 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 char ... -> boolean)
char<? : (char char char ... -> boolean)
char=? : (char char char ... -> boolean)
char>=? : (char char char ... -> boolean)
char>? : (char char char ... -> boolean)
char? : (any -> boolean)

```

### Strings

```

explode : (string -> (listof string))
format : (string any ... -> string)
implode : ((listof string) -> string)
int->string : (integer -> string)
list->string : ((listof char) -> string)
make-string : (nat char -> string)
replicate : (nat string -> string)
string : (char ... -> string)
string->int : (string -> integer)
string->list : (string -> (listof char))
string->number : (string -> (union number false))
string->symbol : (string -> symbol)
string-alphabetic? : (string -> boolean)
string-append : (string ... -> string)
string-ci<=? : (string string string ... -> boolean)
string-ci<? : (string string string ... -> boolean)
string-ci=? : (string string string ... -> boolean)
string-ci>=? : (string string string ... -> boolean)
string-ci>? : (string string string ... -> boolean)
string-copy : (string -> string)
string-ith : (string nat -> string)
string-length : (string -> nat)
string-lower-case? : (string -> boolean)
string-numeric? : (string -> boolean)
string-ref : (string nat -> char)
string-upper-case? : (string -> boolean)

```

```

string-whitespace? : (string -> boolean)
string<=? : (string string string ... -> boolean)
string<? : (string string string ... -> boolean)
string=? : (string string string ... -> boolean)
string>=? : (string string string ... -> boolean)
string>? : (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 : (any ... -> void)
exit : (-> void)
identity : (any -> any)
struct? : (any -> boolean)

```

### Numbers (relaxed conditions)

```

* : (number ... -> number)
+ : (number ... -> number)
- : (number ... -> number)
/ : (number ... -> number)

```

### 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)
argmax : ((X -> real) (listof X) -> X)
argmin : ((X -> real) (listof X) -> X)
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))
sort : ((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 *ids*, and whose body is *expr*.

---

```

(λ (id id ...) expr)

```

The Greek letter  $\lambda$  is a synonym for `lambda`.

### 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 expr 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.

---

`< : (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

---

`= : (number number number ... -> 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 : (number -> number)`

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 : (number -> real)`

Purpose: to extract the angle from a complex number

---

`asin : (number -> number)`

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

---

`atan : (number -> number)`

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

---

`ceiling : (real -> integer)`

Purpose: to determine the closest integer (exact or inexact) above a real number

---

`complex? : (any -> boolean)`

Purpose: to determine whether some value is complex

---

`conjugate : (number -> number)`

Purpose: to compute the conjugate of a complex number

---

`cos : (number -> number)`

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

---

`cosh` : (number -> number)

Purpose: to compute the hyperbolic cosine of a number

---

`current-seconds` : (-> integer)

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

---

`denominator` : (rat -> integer)

Purpose: to compute the denominator of a rational

---

`e` : real

Purpose: Euler's number

---

`even?` : (integer -> boolean)

Purpose: to determine if some integer (exact or inexact) is even or not

---

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

Purpose: to convert an exact number to an inexact one

---

`exact?` : (number -> boolean)

Purpose: to determine whether some number is exact

---

`exp` : (number -> number)

Purpose: to compute e raised to a number

---

`expt` : (number number -> number)

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

---

`floor` : (real -> integer)

Purpose: to determine the closest integer (exact or inexact) below a real number

---

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

Purpose: to compute the greatest common divisor of two integers (exact or inexact)

---

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

Purpose: to extract the imaginary part from a complex number

---

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

Purpose: to approximate an inexact number by an exact one

---

`inexact?` : (number -> boolean)

Purpose: to determine whether some number is inexact

---

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

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

---

`integer-sqrt` : (number -> integer)

Purpose: to compute the integer (exact or inexact) square root of a number

---

`integer?` : (any -> boolean)

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

---

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

Purpose: to compute the least common multiple of two integers (exact or inexact)

---

`log` : (number -> number)

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

---

`magnitude` : (number -> real)

Purpose: to determine the magnitude of a complex number

---

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

Purpose: to create a complex from a magnitude and angle

---

`make-rectangular` : (real real -> number)

Purpose: to create a complex from a real and an imaginary part

---

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

Purpose: to determine the largest number

---

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

Purpose: to determine the smallest number

---

`modulo` : (integer integer -> integer)

Purpose: to find the remainder of the division of the first number by the second; try (modulo 4 3) (modulo 4 -3)

---

`negative?` : (number -> boolean)

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

---

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

Purpose: to convert a number to a string

---

`number?` : (any -> boolean)

Purpose: to determine whether some value is a number

---

`numerator` : (rat -> integer)

Purpose: to compute the numerator of a rational

---

`odd?` : (integer -> boolean)

Purpose: to determine if some integer (exact or inexact) 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` : (integer integer -> integer)

Purpose: to divide the first integer (exact or inexact) into the second; try (quotient 3 4) and (quotient 4 3)

---

`random` : (integer -> integer)

Purpose: to generate a random natural number less than some given integer (exact only!)

---

`rational?` : (any -> boolean)

Purpose: to determine whether some value is a rational number

---

`real-part` : (number -> 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` : (integer integer -> integer)

Purpose: to determine the remainder of dividing the first by the second integer (exact or inexact)

---

`round` : (real -> integer)

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 : (number -> number)
```

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

---

```
sinh : (number -> number)
```

Purpose: to compute the hyperbolic sine of a number

---

```
sqr : (number -> number)
```

Purpose: to compute the square of a number

---

```
sqrt : (number -> number)
```

Purpose: to compute the square root of a number

---

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

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

---

```
tan : (number -> number)
```

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, 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

---

```
caddr : ((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` : ((listof any) -> 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

---

`make-list` : (natural-number any -> (listof any))

Purpose: (make-list k x) constructs a list of k copies of x

---

`member` : (any (listof any) -> boolean)

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

---

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

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

---

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

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

---

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

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

---

`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

---

`remove` : (any (listof any) -> (listof any))

Purpose: to construct a list like the given one with the first occurrence of the given item removed (comparing values with equal?)

---

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

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

---

`reverse` : ((listof any) -> 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 char ... -> boolean)
```

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

---

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

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

---

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

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

---

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

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

---

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

Purpose: to determine whether a character precedes another

---

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

Purpose: to determine whether two characters are equal

---

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

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

---

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

Purpose: to determine whether a character succeeds another

---

`char?` : (any -> boolean)

Purpose: to determine whether a value is a character

---

`explode` : (string -> (listof string))

Purpose: to translate a string into a list of 1-letter strings

---

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

Purpose: to format a string, possibly embedding values

---

`implode` : ((listof string) -> string)

Purpose: to concatenate the list of 1-letter strings into one string

---

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

Purpose: to convert an integer in [0,55295] or [57344 1114111] to a 1-letter string

---

`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

---

`replicate` : (nat string -> string)

Purpose: to replicate the given string

---

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

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

---

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

Purpose: to convert a 1-letter string to an integer in [0,55295] or [57344, 1114111]

---

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

Purpose: to determine whether all 'letters' in the string are alphabetic

---

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

Purpose: to juxtapose the characters of several strings

---

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

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

---

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

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

---

`string-ci>=? : (string 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 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-ith : (string nat -> string)`

Purpose: to extract the ith 1-letter substring from the given one

---

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

Purpose: to determine the length of a string

---

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

Purpose: to determine whether all 'letters' in the string are lower case

---

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

Purpose: to determine whether all 'letters' in the string are numeric

---

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

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

---

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

Purpose: to determine whether all 'letters' in the string are upper case

---

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

Purpose: to determine whether all 'letters' in the string are white space

---

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

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

---

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

Purpose: to determine whether one string alphabetically precedes another

---

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

Purpose: to compare two strings character-wise

---

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

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

---

`string>? : (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 determine whether two values are equivalent from the computer's perspective (intensional)

---

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

Purpose: to determine whether two values are structurally equal where basic values are compared with the `eqv?` predicate

---

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

---

```
error : (any ... -> void)
```

Purpose: to signal an error, combining the given values into an error message.

If any of the values' printed representations is too long, it is truncated and “...” is put into the string. If the first value is a symbol, it is treated specially; it is suffixed with a colon and a space (the intention is that the symbol is the name of the function signalling the 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

---

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

Purpose: to multiply all given numbers

---

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

Purpose: to add all given numbers

---

`- : (number ... -> number)`

Purpose: to subtract from the first all remaining numbers

---

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

Purpose: to divide the first by all remaining numbers

---

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

Purpose:  $(\text{andmap } p \text{ (list } x-1 \dots x-n)) = (\text{and } (p \ x-1) \dots (p \ x-n))$

---

`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

---

`argmax : ((X -> real) (listof X) -> X)`

Purpose: to find the (first) element of the list that maximizes the output of the function

---

`argmin : ((X -> real) (listof X) -> X)`

Purpose: to find the (first) element of the list that minimizes the output of the function

---

`build-list : (nat (nat -> X) -> (listof X))`

Purpose:  $(\text{build-list } n \ f) = (\text{list } (f \ 0) \dots (f \ (- \ n \ 1)))$

---

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

Purpose:  $(\text{build-string } n \ f) = (\text{string } (f \ 0) \dots (f \ (- \ n \ 1)))$

---

```
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) ... (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

---

`sort` : ((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)`  
`(check-member-of expr expr expr ...)`  
`(check-range expr 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 module-path)`

The same as Beginning's require.

## 5 Advanced Student

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

def-or-expr = definition
            | expr
            | test-case
            | 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 ...) expr)
      | (λ (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 ...) ; 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)
      | empty
      | id ; identifier
      | prim-op ; primitive operation
      | '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-member-of expr expr ...)
| (check-range expr expr expr)
| (check-error expr expr)

library-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**

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

```

max : (real real ... -> real)
min : (real real ... -> real)
modulo : (integer integer -> integer)
negative? : (number -> boolean)
number->string : (number -> string)
number? : (any -> boolean)
numerator : (rat -> integer)
odd? : (integer -> boolean)
pi : real
positive? : (number -> boolean)
quotient : (integer integer -> integer)
random : (case->
          (integer -> integer)
          (-> (and/c real inexact? (>/c 0) (</c 1))))
rational? : (any -> boolean)
real-part : (number -> real)
real? : (any -> boolean)
remainder : (integer integer -> integer)
round : (real -> integer)
sgn : (real -> (union 1 1.0 0 0.0 -1 -1.0))
sin : (number -> number)
sinh : (number -> number)
sqr : (number -> number)
sqrt : (number -> number)
sub1 : (number -> number)
tan : (number -> number)
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))
assoc : (any (listof any) -> (listof any) or false)
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)
caddr : ((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))
cdddd : ((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 : ((listof any) -> number)
list : (any ... -> (listof any))

```

```

list* : (any ... (listof any) -> (listof any))
list-ref : ((listof X) natural-number -> X)
list? : (any -> boolean)
make-list : (natural-number any -> (listof any))
member : (any (listof any) -> boolean)
member? : (any (listof any) -> boolean)
memq : (any (listof any) -> (union false list))
memv : (any (listof any) -> (union false list))
null : empty
null? : (any -> boolean)
pair? : (any -> boolean)
remove : (any (listof any) -> (listof any))
rest : ((cons Y (listof X)) -> (listof X))
reverse : ((listof any) -> 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 char ... -> boolean)
char-ci<? : (char char char ... -> boolean)
char-ci=? : (char char char ... -> boolean)
char-ci>=? : (char char char ... -> boolean)
char-ci>? : (char 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 char ... -> boolean)
char<? : (char char char ... -> boolean)
char=? : (char char char ... -> boolean)
char>=? : (char char char ... -> boolean)
char>? : (char char char ... -> boolean)
char? : (any -> boolean)

```

### Strings

```

explode : (string -> (listof string))
format : (string any ... -> string)
implode : ((listof string) -> string)
int->string : (integer -> string)
list->string : ((listof char) -> string)
make-string : (nat char -> string)
replicate : (nat string -> string)
string : (char ... -> string)
string->int : (string -> integer)
string->list : (string -> (listof char))
string->number : (string -> (union number false))
string->symbol : (string -> symbol)
string-alphabetic? : (string -> boolean)
string-append : (string ... -> string)
string-ci<=? : (string string string ... -> boolean)
string-ci<? : (string string string ... -> boolean)
string-ci=? : (string string string ... -> boolean)
string-ci>=? : (string string string ... -> boolean)
string-ci>? : (string string string ... -> boolean)
string-copy : (string -> string)
string-ith : (string nat -> string)
string-length : (string -> nat)
string-lower-case? : (string -> boolean)
string-numeric? : (string -> boolean)
string-ref : (string nat -> char)
string-upper-case? : (string -> boolean)
string-whitespace? : (string -> boolean)
string<=? : (string string string ... -> boolean)
string<? : (string string string ... -> boolean)
string=? : (string string string ... -> boolean)
string>=? : (string string string ... -> boolean)
string>? : (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)
current-milliseconds : (-> exact-integer)
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 : (any ... -> void)
exit : (-> void)
force : (delay -> any)
gensym : (-> symbol?)
identity : (any -> any)
promise? : (any -> boolean)
sleep : (-> positive-number void)
struct? : (any -> boolean)
void : (-> void)
void? : (any -> boolean)
Numbers (relaxed conditions)
* : (number ... -> number)
+ : (number ... -> number)
- : (number ... -> number)
/ : (number ... -> number)
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)
argmax : ((X -> real) (listof X) -> X)
argmin : ((X -> real) (listof X) -> X)
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))

```

```
sort : ((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)
with-input-from-file : (string (-> any) -> any)
with-input-from-string : (string (-> any) -> any)
with-output-to-file : (string (-> any) -> any)
with-output-to-string : (string (-> any) -> any)
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:

- `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)  
(λ (id ...) expr)
```

The same as Intermediate with Lambda's lambda, except that a function is allowed to accept zero arguments.

### 5.4 Function Calls

---

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

A function call in Advanced is the same as an Intermediate with Lambda function call, except that zero arguments are allowed.

---

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

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

### 5.5 begin

---

```
(begin expr expr ...)
```

Evaluates the `exprs` in order from left to right. The value of the `begin` expression is the value of the last `expr`.

### 5.6 begin0

---

```
(begin0 expr expr ...)
```

Evaluates the `exprs` in order from left to right. The value of the `begin` expression is the value of the first `expr`.

## 5.7 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*`.

## 5.8 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.

## 5.9 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 *ids* defined earlier. Thus, `shared` can be used to create cyclic data structures.

## 5.10 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.

## 5.11 recur

---

```
(recur id ([id expr] ...) expr)
```

A short-hand recursion construct. The first *id* corresponds to the name of the recursive function. The parenthesized *ids* 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 ...)
```

## 5.12 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.

---

```
(case 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 the end” of the case form.

## 5.13 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`.

## 5.14 Primitive Operations

---

`< : (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

---

`= : (number number number ... -> 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 : (number -> number)`

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` : (number -> real)

Purpose: to extract the angle from a complex number

---

`asin` : (number -> number)

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

---

`atan` : (number -> number)

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

---

`ceiling` : (real -> integer)

Purpose: to determine the closest integer (exact or inexact) above a real number

---

`complex?` : (any -> boolean)

Purpose: to determine whether some value is complex

---

`conjugate` : (number -> number)

Purpose: to compute the conjugate of a complex number

---

`cos` : (number -> number)

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

---

`cosh` : (number -> number)

Purpose: to compute the hyperbolic cosine of a number

---

`current-seconds` : (-> integer)

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

---

`denominator` : (rat -> integer)

Purpose: to compute the denominator of a rational

---

`e` : real

Purpose: Euler's number

---

`even?` : (integer -> boolean)

Purpose: to determine if some integer (exact or inexact) is even or not

---

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

Purpose: to convert an exact number to an inexact one

---

`exact?` : (number -> boolean)

Purpose: to determine whether some number is exact

---

`exp` : (number -> number)

Purpose: to compute e raised to a number

---

`expt` : (number number -> number)

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

---

`floor` : (real -> integer)

Purpose: to determine the closest integer (exact or inexact) below a real number

---

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

Purpose: to compute the greatest common divisor of two integers (exact or inexact)

---

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

Purpose: to extract the imaginary part from a complex number

---

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

Purpose: to approximate an inexact number by an exact one

---

`inexact?` : (number -> boolean)

Purpose: to determine whether some number is inexact

---

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

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

---

`integer-sqrt` : (number -> integer)

Purpose: to compute the integer (exact or inexact) square root of a number

---

`integer?` : (any -> boolean)

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

---

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

Purpose: to compute the least common multiple of two integers (exact or inexact)

---

`log` : (number -> number)

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

---

`magnitude` : (number -> real)

Purpose: to determine the magnitude of a complex number

---

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

Purpose: to create a complex from a magnitude and angle

---

`make-rectangular` : (real real -> number)

Purpose: to create a complex from a real and an imaginary part

---

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

Purpose: to determine the largest number

---

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

Purpose: to determine the smallest number

---

```
modulo : (integer integer -> integer)
```

Purpose: to find the remainder of the division of the first number by the second; try (modulo 4 3) (modulo 4 -3)

---

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

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

---

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

Purpose: to convert a number to a string

---

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

Purpose: to determine whether some value is a number

---

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

Purpose: to compute the numerator of a rational

---

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

Purpose: to determine if some integer (exact or inexact) 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 : (integer integer -> integer)
```

Purpose: to divide the first integer (exact or inexact) into the second; try (quotient 3 4) and (quotient 4 3)

---

```
random : (case->
  (integer -> integer)
  (-> (and/c real inexact? (>/c 0) (</c 1))))
```

Purpose: to generate a random natural number less than some given integer, or to generate a random inexact number between 0.0 and 1.0 exclusive

---

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

Purpose: to determine whether some value is a rational number

---

```
real-part : (number -> 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 : (integer integer -> integer)
```

Purpose: to determine the remainder of dividing the first by the second integer (exact or inexact)

---

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

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 : (number -> number)
```

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

---

```
sinh : (number -> number)
```

Purpose: to compute the hyperbolic sine of a number

---

```
sqr : (number -> number)
```

Purpose: to compute the square of a number

---

```
sqrt : (number -> number)
```

Purpose: to compute the square root of a number

---

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

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

---

```
tan : (number -> number)
```

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

---

```
assoc : (any (listof any) -> (listof any) or false)
```

Purpose: to produce the first element on the list whose first is equal? to v; otherwise it produces false

---

```
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

---

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

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

---

```
cadadr : ((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 : ((listof any) -> 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

---

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

Purpose: to determine whether some value is a list

---

```
make-list : (natural-number any -> (listof any))
```

Purpose: (make-list k x) constructs a list of k copies of x

---

```
member : (any (listof any) -> boolean)
```

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

---

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

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

---

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

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

---

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

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

---

```
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

---

```
remove : (any (listof any) -> (listof any))
```

Purpose: to construct a list like the given one with the first occurrence of the given item removed (comparing values with equal?)

---

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

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

---

```
reverse : ((listof any) -> 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 char ... -> boolean)

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

---

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

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

---

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

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

---

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

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

---

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

Purpose: to determine whether a character precedes another

---

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

Purpose: to determine whether two characters are equal

---

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

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

---

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

Purpose: to determine whether a character succeeds another

---

`char?` : (any -> boolean)

Purpose: to determine whether a value is a character

---

`explode` : (string -> (listof string))

Purpose: to translate a string into a list of 1-letter strings

---

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

Purpose: to format a string, possibly embedding values

---

`implode` : ((listof string) -> string)

Purpose: to concatenate the list of 1-letter strings into one string

---

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

Purpose: to convert an integer in [0,55295] or [57344 1114111] to a 1-letter string

---

`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

---

`replicate` : (nat string -> string)

Purpose: to replicate the given string

---

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

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

---

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

Purpose: to convert a 1-letter string to an integer in [0,55295] or [57344, 1114111]

---

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

Purpose: to determine whether all 'letters' in the string are alphabetic

---

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

Purpose: to juxtapose the characters of several strings

---

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

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

---

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

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

---

`string-ci>=?` : (string 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 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-ith` : (string nat -> string)

Purpose: to extract the ith 1-letter substring from the given one

---

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

Purpose: to determine the length of a string

---

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

Purpose: to determine whether all 'letters' in the string are lower case

---

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

Purpose: to determine whether all 'letters' in the string are numeric

---

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

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

---

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

Purpose: to determine whether all 'letters' in the string are upper case

---

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

Purpose: to determine whether all 'letters' in the string are white space

---

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

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

---

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

Purpose: to determine whether one string alphabetically precedes another

---

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

Purpose: to compare two strings character-wise

---

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

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

---

`string>?` : (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

---

`current-milliseconds` : (-> exact-integer)

Purpose: to return the current “time” in fixnum milliseconds (possibly negative)

---

`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 determine whether two values are equivalent from the computer's perspective (intensional)

---

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

Purpose: to determine whether two values are structurally equal where basic values are compared with the `eqv?` predicate

---

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

---

`error` : (any ... -> void)

Purpose: to signal an error, combining the given values into an error message.

If any of the values' printed representations is too long, it is truncated and "..." is put into the string. If the first value is a symbol, it is treated specially; it is suffixed with a colon and a space (the intention is that the symbol is the name of the function signalling the error).

---

`exit` : (-> void)

Purpose: to exit the running program

---

`force` : (delay -> any)

Purpose: to find the delayed value; see also `delay`

---

`gensym` : (-> symbol?)

Purpose: to generate a new symbol, different from all symbols in the program

---

`identity` : (any -> any)

Purpose: to return the argument unchanged

---

`promise?` : (any -> boolean)

Purpose: to determine if a value is delayed

---

`sleep` : (-> positive-number void)

Purpose: to cause the program to sleep for the given number of seconds

---

`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

---

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

Purpose: to multiply all given numbers

---

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

Purpose: to add all given numbers

---

`-` : (number ... -> number)

Purpose: to subtract from the first all remaining numbers

---

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

Purpose: to divide the first by all remaining numbers

---

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

Purpose: (andmap p (list x-1 ... x-n)) = (and (p x-1) ... (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

---

`argmax` : ((X -> real) (listof X) -> X)

Purpose: to find the (first) element of the list that maximizes the output of the function

---

`argmin` : ((X -> real) (listof X) -> X)

Purpose: to find the (first) element of the list that minimizes the output of the function

---

`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) ... (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

---

```
sort : ((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

---

`with-input-from-file` : (string (-> any) -> any)

Purpose: to open the named input file and to extract all input from there

---

`with-input-from-string` : (string (-> any) -> any)

Purpose: to turn the given string into input for read\* operations

---

`with-output-to-file` : (string (-> any) -> any)

Purpose: to open the named output file and to put all output there

---

`with-output-to-string` : (string (-> any) -> any)

Purpose: to produce a string from all write/display/print operations

---

`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

## 5.15 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)
```

```
(check-member-of expr expr expr ...)  
(check-range expr 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 module-path)
```

The same as Beginning's `require`.

## Index

`##app`, 75  
`##app`, 140  
`##app`, 107  
`##app`, 12  
`*`, 44  
`*`, 126  
`*`, 16  
`*`, 95  
`*`, 163  
`+`, 163  
`+`, 44  
`+`, 95  
`+`, 126  
`+`, 17  
`-`, 17  
`-`, 95  
`-`, 127  
`-`, 45  
`-`, 163  
`/`, 163  
`/`, 17  
`/`, 45  
`/`, 95  
`/`, 127  
`<`, 75  
`<`, 143  
`<`, 17  
`<`, 45  
`<`, 107  
`<=`, 107  
`<=`, 76  
`<=`, 45  
`<=`, 17  
`<=`, 143  
`=`, 45  
`=`, 143  
`=`, 107  
`=`, 17  
`=`, 76  
`=~`, 63  
`=~`, 125  
`=~`, 93  
`=~`, 35  
`=~`, 161  
`>`, 143  
`>`, 107  
`>`, 17  
`>`, 76  
`>`, 45  
`>=`, 17  
`>=`, 107  
`>=`, 76  
`>=`, 143  
`>=`, 45  
`abs`, 76  
`abs`, 143  
`abs`, 17  
`abs`, 108  
`abs`, 45  
`acos`, 108  
`acos`, 76  
`acos`, 143  
`acos`, 45  
`acos`, 17  
`add1`, 76  
`add1`, 18  
`add1`, 108  
`add1`, 46  
`add1`, 143  
Advanced Student, 131  
`and`, 13  
`and`, 13  
`and`, 97  
`and`, 168  
`and`, 65  
`and`, 129  
`andmap`, 95  
`andmap`, 127  
`andmap`, 164  
`angle`, 46  
`angle`, 144  
`angle`, 18

angle, 76  
 angle, 108  
 append, 150  
 append, 114  
 append, 52  
 append, 24  
 append, 82  
 apply, 95  
 apply, 127  
 apply, 164  
 argmax, 95  
 argmax, 164  
 argmax, 127  
 argmin, 127  
 argmin, 164  
 argmin, 95  
 asin, 18  
 asin, 76  
 asin, 144  
 asin, 46  
 asin, 108  
 assoc, 150  
 assq, 24  
 assq, 82  
 assq, 52  
 assq, 114  
 assq, 150  
 atan, 144  
 atan, 46  
 atan, 18  
 atan, 108  
 atan, 76  
 begin, 140  
 begin, 140  
 begin0, 140  
 begin0, 140  
 Beginning Student, 5  
 Beginning Student with List Abbreviations,  
 37  
 boolean=?, 113  
 boolean=?, 51  
 boolean=?, 149  
 boolean=?, 23  
 boolean=?, 82  
 boolean?, 82  
 boolean?, 113  
 boolean?, 51  
 boolean?, 23  
 boolean?, 149  
 box, 167  
 box?, 167  
 build-list, 127  
 build-list, 164  
 build-list, 95  
 build-string, 96  
 build-string, 127  
 build-string, 164  
 build-vector, 167  
 caaar, 52  
 caaar, 24  
 caaar, 150  
 caaar, 114  
 caaar, 83  
 caadr, 115  
 caadr, 83  
 caadr, 24  
 caadr, 52  
 caadr, 150  
 caar, 25  
 caar, 151  
 caar, 115  
 caar, 83  
 caar, 53  
 cadar, 151  
 cadar, 115  
 cadar, 25  
 cadar, 83  
 cadar, 53  
 caddr, 83  
 caddr, 53  
 caddr, 151  
 caddr, 25  
 caddr, 115  
 caddr, 151

caddr, 83  
caddr, 53  
caddr, 25  
caddr, 115  
cadr, 83  
cadr, 115  
cadr, 53  
cadr, 151  
cadr, 25  
car, 115  
car, 151  
car, 25  
car, 53  
car, 83  
case, 142  
case, 142  
cdaar, 84  
cdaar, 115  
cdaar, 25  
cdaar, 53  
cdaar, 151  
cdadr, 25  
cdadr, 116  
cdadr, 151  
cdadr, 53  
cdadr, 84  
cdar, 54  
cdar, 26  
cdar, 84  
cdar, 116  
cdar, 152  
cddar, 26  
cddar, 152  
cddar, 116  
cddar, 84  
cddar, 54  
cdddr, 84  
cdddr, 54  
cdddr, 26  
cdddr, 116  
cdddr, 152  
cddr, 84  
cddr, 152  
cddr, 26  
cddr, 54  
cddr, 116  
cdr, 116  
cdr, 152  
cdr, 84  
cdr, 54  
cdr, 26  
ceiling, 144  
ceiling, 108  
ceiling, 18  
ceiling, 77  
ceiling, 46  
char->integer, 156  
char->integer, 119  
char->integer, 57  
char->integer, 88  
char->integer, 29  
char-alphabetic?, 156  
char-alphabetic?, 88  
char-alphabetic?, 57  
char-alphabetic?, 119  
char-alphabetic?, 29  
char-ci<=?, 88  
char-ci<=?, 57  
char-ci<=?, 120  
char-ci<=?, 29  
char-ci<=?, 156  
char-ci<?, 30  
char-ci<?, 88  
char-ci<?, 120  
char-ci<?, 156  
char-ci<?, 58  
char-ci=?, 88  
char-ci=?, 156  
char-ci=?, 58  
char-ci=?, 30  
char-ci=?, 120  
char-ci>=?, 30  
char-ci>=?, 156  
char-ci>=?, 88

char-ci>=?, 58  
char-ci>=?, 120  
char-ci>?, 58  
char-ci>?, 30  
char-ci>?, 120  
char-ci>?, 88  
char-ci>?, 156  
char-downcase, 58  
char-downcase, 156  
char-downcase, 120  
char-downcase, 30  
char-downcase, 88  
char-lower-case?, 88  
char-lower-case?, 30  
char-lower-case?, 120  
char-lower-case?, 156  
char-lower-case?, 58  
char-numeric?, 30  
char-numeric?, 89  
char-numeric?, 58  
char-numeric?, 156  
char-numeric?, 120  
char-upcase, 30  
char-upcase, 89  
char-upcase, 157  
char-upcase, 120  
char-upcase, 58  
char-upper-case?, 89  
char-upper-case?, 58  
char-upper-case?, 157  
char-upper-case?, 121  
char-upper-case?, 30  
char-whitespace?, 89  
char-whitespace?, 30  
char-whitespace?, 121  
char-whitespace?, 58  
char-whitespace?, 157  
char<=?, 157  
char<=?, 89  
char<=?, 59  
char<=?, 121  
char<=?, 31  
char<=?, 121  
char<=?, 59  
char<=?, 31  
char<=?, 157  
char<=?, 89  
char=?, 121  
char=?, 157  
char=?, 31  
char=?, 89  
char=?, 59  
char=?, 157  
check-error, 168  
check-error, 65  
check-error, 97  
check-error, 14  
check-error, 130  
check-expect, 97  
check-expect, 168  
check-expect, 130  
check-expect, 65  
check-expect, 14  
check-member-of, 65  
check-member-of, 14  
check-member-of, 130  
check-member-of, 97  
check-member-of, 169  
check-range, 98  
check-range, 14

check-range, 65  
 check-range, 169  
 check-range, 130  
 check-within, 65  
 check-within, 130  
 check-within, 97  
 check-within, 14  
 check-within, 168  
 complex?, 46  
 complex?, 144  
 complex?, 18  
 complex?, 108  
 complex?, 77  
 compose, 96  
 compose, 164  
 compose, 128  
 cond, 13  
 cond, 129  
 cond, 13  
 cond, 168  
 cond, 64  
 cond, 97  
 conjugate, 18  
 conjugate, 144  
 conjugate, 77  
 conjugate, 108  
 conjugate, 46  
 cons, 26  
 cons, 152  
 cons, 85  
 cons, 54  
 cons, 116  
 cons?, 85  
 cons?, 152  
 cons?, 116  
 cons?, 26  
 cons?, 54  
 cos, 46  
 cos, 108  
 cos, 18  
 cos, 144  
 cos, 77  
 cosh, 18  
 cosh, 144  
 cosh, 46  
 cosh, 109  
 cosh, 77  
 current-milliseconds, 161  
 current-seconds, 18  
 current-seconds, 109  
 current-seconds, 144  
 current-seconds, 46  
 current-seconds, 77  
 define, 139  
 define, 11  
 define, 73  
 define, 106  
 define, 106  
 define, 11  
 define, 64  
 define, 139  
 define, 73  
 define-struct, 139  
 define-struct, 11  
 define-struct, 73  
 define-struct, 139  
 define-struct, 11  
 define-struct, 129  
 define-struct, 64  
 define-struct, 73  
 delay, 141  
 delay, 141  
 denominator, 47  
 denominator, 109  
 denominator, 77  
 denominator, 19  
 denominator, 144  
 display, 166  
 e, 145  
 e, 47  
 e, 77  
 e, 109  
 e, 19  
 eighth, 85

eighth, 152  
eighth, 117  
eighth, 26  
eighth, 54  
else, 97  
else, 65  
else, 13  
else, 168  
else, 129  
empty, 15  
empty, 15  
empty, 65  
empty, 169  
empty, 130  
empty, 98  
empty?, 117  
empty?, 26  
empty?, 152  
empty?, 54  
empty?, 85  
eof, 125  
eof, 63  
eof, 161  
eof, 35  
eof, 93  
eof-object?, 125  
eof-object?, 93  
eof-object?, 161  
eof-object?, 35  
eof-object?, 63  
eq?, 94  
eq?, 35  
eq?, 162  
eq?, 63  
eq?, 125  
equal?, 63  
equal?, 35  
equal?, 162  
equal?, 125  
equal?, 94  
equal~?, 63  
equal~?, 162  
equal~?, 35  
equal~?, 94  
equal~?, 126  
eqv?, 36  
eqv?, 64  
eqv?, 94  
eqv?, 126  
eqv?, 162  
error, 94  
error, 126  
error, 36  
error, 162  
error, 64  
even?, 77  
even?, 19  
even?, 47  
even?, 109  
even?, 145  
exact->inexact, 47  
exact->inexact, 77  
exact->inexact, 109  
exact->inexact, 19  
exact->inexact, 145  
exact?, 109  
exact?, 47  
exact?, 19  
exact?, 78  
exact?, 145  
exit, 94  
exit, 162  
exit, 64  
exit, 36  
exit, 126  
exp, 78  
exp, 109  
exp, 47  
exp, 145  
exp, 19  
explode, 90  
explode, 31  
explode, 121  
explode, 59

explode, 157  
expt, 78  
expt, 145  
expt, 19  
expt, 109  
expt, 47  
false, 130  
false, 15  
false, 98  
false, 65  
false, 169  
false?, 113  
false?, 23  
false?, 82  
false?, 51  
false?, 149  
fifth, 55  
fifth, 27  
fifth, 85  
fifth, 117  
fifth, 153  
filter, 164  
filter, 96  
filter, 128  
first, 153  
first, 117  
first, 27  
first, 55  
first, 85  
floor, 109  
floor, 47  
floor, 78  
floor, 145  
floor, 19  
foldl, 96  
foldl, 165  
foldl, 128  
foldr, 128  
foldr, 165  
foldr, 96  
for-each, 96  
for-each, 165  
for-each, 128  
force, 162  
format, 31  
format, 158  
format, 59  
format, 90  
format, 121  
fourth, 55  
fourth, 117  
fourth, 85  
fourth, 153  
fourth, 27  
Function Calls, 140  
Function Calls, 12  
Function Calls, 74  
Function Calls, 107  
gcd, 19  
gcd, 78  
gcd, 47  
gcd, 110  
gcd, 145  
gensym, 162  
*How to Design Programs Languages*, 1  
Identifiers, 15  
Identifiers, 75  
identity, 64  
identity, 126  
identity, 36  
identity, 94  
identity, 163  
if, 13  
if, 129  
if, 65  
if, 13  
if, 168  
if, 97  
imag-part, 78  
imag-part, 19  
imag-part, 110  
imag-part, 47  
imag-part, 145  
image=?, 35

image=?, 125  
 image=?, 63  
 image=?, 93  
 image=?, 161  
 image?, 161  
 image?, 63  
 image?, 125  
 image?, 35  
 image?, 93  
 implode, 158  
 implode, 31  
 implode, 122  
 implode, 59  
 implode, 90  
 inexact->exact, 20  
 inexact->exact, 48  
 inexact->exact, 78  
 inexact->exact, 110  
 inexact->exact, 145  
 inexact?, 78  
 inexact?, 48  
 inexact?, 20  
 inexact?, 146  
 inexact?, 110  
 int->string, 31  
 int->string, 158  
 int->string, 59  
 int->string, 90  
 int->string, 122  
 integer->char, 48  
 integer->char, 110  
 integer->char, 20  
 integer->char, 146  
 integer->char, 78  
 integer-sqrt, 20  
 integer-sqrt, 78  
 integer-sqrt, 146  
 integer-sqrt, 48  
 integer-sqrt, 110  
 integer?, 48  
 integer?, 79  
 integer?, 146  
 integer?, 110  
 integer?, 20  
 Intermediate Student, 66  
 Intermediate Student with Lambda, 99  
 lambda, 140  
 lambda, 106  
 lambda, 11  
 lambda, 73  
 lambda, 140  
 lambda, 64  
 lambda, 106  
 lcm, 110  
 lcm, 79  
 lcm, 20  
 lcm, 146  
 lcm, 48  
 length, 117  
 length, 27  
 length, 55  
 length, 85  
 length, 153  
 let, 141  
 let, 74  
 let, 141  
 let, 129  
 let\*, 129  
 let\*, 168  
 let\*, 74  
 letrec, 74  
 letrec, 129  
 letrec, 168  
 letrec, let, and let\*, 74  
 list, 117  
 list, 27  
 list, 55  
 list, 153  
 list, 85  
 list\*, 55  
 list\*, 85  
 list\*, 153  
 list\*, 117  
 list\*, 27

list->string, 122  
list->string, 32  
list->string, 90  
list->string, 158  
list->string, 60  
list-ref, 153  
list-ref, 27  
list-ref, 86  
list-ref, 55  
list-ref, 117  
list?, 153  
local, 74  
local, 74  
local, 129  
local, 168  
log, 48  
log, 146  
log, 79  
log, 20  
log, 110  
magnitude, 20  
magnitude, 110  
magnitude, 79  
magnitude, 48  
magnitude, 146  
make-list, 153  
make-list, 55  
make-list, 86  
make-list, 27  
make-list, 117  
make-polar, 111  
make-polar, 48  
make-polar, 79  
make-polar, 146  
make-polar, 20  
make-posn, 29  
make-posn, 57  
make-posn, 155  
make-posn, 87  
make-posn, 119  
make-rectangular, 48  
make-rectangular, 146  
make-rectangular, 20  
make-rectangular, 79  
make-rectangular, 111  
make-string, 60  
make-string, 122  
make-string, 90  
make-string, 158  
make-string, 32  
make-vector, 167  
map, 128  
map, 96  
map, 165  
max, 49  
max, 111  
max, 146  
max, 21  
max, 79  
member, 118  
member, 153  
member, 27  
member, 55  
member, 86  
member?, 55  
member?, 154  
member?, 86  
member?, 118  
member?, 27  
memf, 128  
memf, 165  
memf, 96  
memq, 56  
memq, 154  
memq, 28  
memq, 86  
memq, 118  
memv, 56  
memv, 154  
memv, 28  
memv, 86  
memv, 118  
min, 21  
min, 49

min, 79  
min, 147  
min, 111  
modulo, 111  
modulo, 49  
modulo, 79  
modulo, 21  
modulo, 147  
negative?, 147  
negative?, 111  
negative?, 21  
negative?, 49  
negative?, 79  
newline, 166  
not, 51  
not, 23  
not, 149  
not, 82  
not, 114  
null, 56  
null, 154  
null, 86  
null, 118  
null, 28  
null?, 154  
null?, 28  
null?, 86  
null?, 118  
null?, 56  
number->string, 21  
number->string, 49  
number->string, 147  
number->string, 80  
number->string, 111  
number?, 49  
number?, 80  
number?, 21  
number?, 111  
number?, 147  
numerator, 80  
numerator, 147  
numerator, 21  
numerator, 49  
numerator, 111  
odd?, 111  
odd?, 49  
odd?, 80  
odd?, 147  
odd?, 21  
or, 14  
or, 168  
or, 129  
or, 65  
or, 14  
or, 97  
ormap, 97  
ormap, 165  
ormap, 128  
pair?, 154  
pair?, 28  
pair?, 118  
pair?, 56  
pair?, 86  
pi, 49  
pi, 147  
pi, 80  
pi, 21  
pi, 112  
positive?, 21  
positive?, 49  
positive?, 147  
positive?, 80  
positive?, 112  
posn-x, 119  
posn-x, 87  
posn-x, 29  
posn-x, 155  
posn-x, 57  
posn-y, 119  
posn-y, 57  
posn-y, 155  
posn-y, 29  
posn-y, 87  
posn?, 29

posn?, 57  
posn?, 119  
posn?, 87  
posn?, 155  
pretty-print, 166  
Primitive Calls, 12  
Primitive Operation Names, 107  
Primitive Operations, 143  
Primitive Operations, 44  
Primitive Operations, 16  
Primitive Operations, 75  
print, 166  
printf, 166  
procedure?, 165  
procedure?, 97  
procedure?, 129  
promise?, 163  
Quasiquote, 44  
quasiquote, 44  
quicksort, 97  
quicksort, 129  
quicksort, 165  
Quote, 43  
quote, 15  
quote, 43  
quotient, 112  
quotient, 50  
quotient, 22  
quotient, 147  
quotient, 80  
random, 148  
random, 112  
random, 80  
random, 22  
random, 50  
rational?, 112  
rational?, 50  
rational?, 22  
rational?, 80  
rational?, 148  
read, 166  
real-part, 50  
real-part, 148  
real-part, 22  
real-part, 80  
real-part, 112  
real?, 22  
real?, 50  
real?, 112  
real?, 148  
real?, 81  
recur, 141  
recur, 141  
remainder, 112  
remainder, 81  
remainder, 50  
remainder, 148  
remainder, 22  
remove, 28  
remove, 154  
remove, 56  
remove, 86  
remove, 118  
replicate, 122  
replicate, 60  
replicate, 32  
replicate, 158  
replicate, 90  
require, 16  
require, 169  
require, 98  
require, 65  
require, 16  
require, 130  
rest, 56  
rest, 154  
rest, 118  
rest, 87  
rest, 28  
reverse, 154  
reverse, 28  
reverse, 118  
reverse, 87  
reverse, 56

round, 22  
round, 148  
round, 50  
round, 112  
round, 81  
second, 87  
second, 119  
second, 28  
second, 155  
second, 56  
set!, 141  
set!, 141  
set-box!, 168  
set-posn-x!, 155  
set-posn-y!, 155  
seventh, 155  
seventh, 119  
seventh, 29  
seventh, 87  
seventh, 57  
sgn, 112  
sgn, 148  
sgn, 50  
sgn, 22  
sgn, 81  
shared, 141  
shared, 141  
sin, 22  
sin, 113  
sin, 148  
sin, 50  
sin, 81  
sinh, 50  
sinh, 23  
sinh, 81  
sinh, 148  
sinh, 113  
sixth, 155  
sixth, 29  
sixth, 119  
sixth, 57  
sixth, 87  
sleep, 163  
sort, 129  
sort, 165  
sort, 97  
sqr, 51  
sqr, 23  
sqr, 81  
sqr, 113  
sqr, 149  
sqrt, 23  
sqrt, 113  
sqrt, 81  
sqrt, 149  
sqrt, 51  
string, 90  
string, 122  
string, 32  
string, 60  
string, 158  
string->int, 90  
string->int, 122  
string->int, 158  
string->int, 60  
string->int, 32  
string->list, 32  
string->list, 90  
string->list, 60  
string->list, 158  
string->list, 122  
string->number, 158  
string->number, 60  
string->number, 91  
string->number, 32  
string->number, 122  
string->symbol, 122  
string->symbol, 60  
string->symbol, 159  
string->symbol, 91  
string->symbol, 32  
string-alphabetic?, 123  
string-alphabetic?, 32  
string-alphabetic?, 60

string-alphabetic?, 159  
string-alphabetic?, 91  
string-append, 60  
string-append, 91  
string-append, 159  
string-append, 32  
string-append, 123  
string-ci<=?, 33  
string-ci<=?, 159  
string-ci<=?, 91  
string-ci<=?, 61  
string-ci<=?, 123  
string-ci<?, 61  
string-ci<?, 159  
string-ci<?, 123  
string-ci<?, 33  
string-ci<?, 91  
string-ci=?, 33  
string-ci=?, 123  
string-ci=?, 61  
string-ci=?, 159  
string-ci=?, 91  
string-ci>=?, 123  
string-ci>=?, 159  
string-ci>=?, 91  
string-ci>=?, 61  
string-ci>=?, 33  
string-ci>?, 61  
string-ci>?, 123  
string-ci>?, 159  
string-ci>?, 33  
string-ci>?, 91  
string-copy, 123  
string-copy, 33  
string-copy, 159  
string-copy, 92  
string-copy, 61  
string-ith, 92  
string-ith, 33  
string-ith, 61  
string-ith, 123  
string-ith, 160  
string-length, 61  
string-length, 33  
string-length, 92  
string-length, 160  
string-length, 124  
string-lower-case?, 33  
string-lower-case?, 61  
string-lower-case?, 92  
string-lower-case?, 124  
string-lower-case?, 160  
string-numeric?, 124  
string-numeric?, 34  
string-numeric?, 62  
string-numeric?, 92  
string-numeric?, 160  
string-ref, 160  
string-ref, 62  
string-ref, 124  
string-ref, 92  
string-ref, 34  
string-upper-case?, 124  
string-upper-case?, 92  
string-upper-case?, 34  
string-upper-case?, 160  
string-upper-case?, 62  
string-whitespace?, 62  
string-whitespace?, 92  
string-whitespace?, 160  
string-whitespace?, 34  
string-whitespace?, 124  
string<=?, 92  
string<=?, 34  
string<=?, 62  
string<=?, 124  
string<=?, 160  
string<?, 92  
string<?, 160  
string<?, 62  
string<?, 124  
string<?, 34  
string=?, 93  
string=?, 34

string=?, 124  
string=?, 160  
string=?, 62  
string>=?, 161  
string>=?, 34  
string>=?, 93  
string>=?, 62  
string>=?, 124  
string>?, 62  
string>?, 125  
string>?, 93  
string>?, 34  
string>?, 161  
string?, 34  
string?, 125  
string?, 62  
string?, 161  
string?, 93  
struct?, 126  
struct?, 36  
struct?, 94  
struct?, 64  
struct?, 163  
sub1, 149  
sub1, 51  
sub1, 81  
sub1, 23  
sub1, 113  
substring, 93  
substring, 125  
substring, 63  
substring, 35  
substring, 161  
symbol->string, 24  
symbol->string, 114  
symbol->string, 82  
symbol->string, 51  
symbol->string, 149  
symbol=?, 114  
symbol=?, 82  
symbol=?, 150  
symbol=?, 24  
symbol=?, 52  
symbol?, 150  
symbol?, 114  
symbol?, 82  
symbol?, 24  
symbol?, 52  
Symbols, 15  
tan, 81  
tan, 51  
tan, 23  
tan, 113  
tan, 149  
Test Cases, 14  
third, 87  
third, 57  
third, 155  
third, 119  
third, 29  
time, 75  
time, 168  
time, 130  
time, 75  
true, 65  
true, 15  
true, 130  
true, 98  
true, 169  
true and false, 15  
unbox, 168  
Unchanged Forms, 64  
Unchanged Forms, 168  
Unchanged Forms, 129  
Unchanged Forms, 97  
unless, 143  
unquote, 44  
unquote-splicing, 44  
vector, 167  
vector-length, 167  
vector-ref, 167  
vector-set!, 167  
vector?, 167  
void, 163

`void?`, 163  
`when`, 142  
`when and unless`, 142  
`with-input-from-file`, 166  
`with-input-from-string`, 166  
`with-output-to-file`, 166  
`with-output-to-string`, 166  
`write`, 167  
`zero?`, 149  
`zero?`, 113  
`zero?`, 51  
`zero?`, 23  
`zero?`, 82  
 $\lambda$ , 140  
 $\lambda$ , 106