

# *How to Design Programs Languages*

Version 4.2.1

July 30, 2009

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

When programs in these languages are run in DrScheme, 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 . . . . .	14
1.11	Identifiers . . . . .	15
1.12	Symbols . . . . .	15
1.13	true and false . . . . .	15
1.14	require . . . . .	15
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 . . . . .	43
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	<code>define-struct</code> . . . . .	73
3.3	<code>local</code> . . . . .	74
3.4	<code>letrec</code> , <code>let</code> , and <code>let*</code> . . . . .	74
3.5	Function Calls . . . . .	74
3.6	<code>time</code> . . . . .	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	<code>define</code> . . . . .	106
4.2	<code>lambda</code> . . . . .	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	<code>define</code> . . . . .	139
5.2	<code>define-struct</code> . . . . .	139
5.3	<code>lambda</code> . . . . .	139
5.4	Function Calls . . . . .	140
5.5	<code>begin</code> . . . . .	140
5.6	<code>begin0</code> . . . . .	140
5.7	<code>set!</code> . . . . .	140
5.8	<code>delay</code> . . . . .	141
5.9	<code>shared</code> . . . . .	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 . . . . .	167
<b>Index</b>	<b>169</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-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))
cdddr : ((cons W (cons Z (cons Y (listof X))))
        ->
        (listof X))
cddr : ((cons Z (cons Y (listof X))) -> (listof X))

```



```

cdr : ((cons Y (listof X)) -> (listof X))
cons : (X (listof X) -> (listof X))
cons? : (any -> boolean)
eighth : ((listof Y) -> Y)
empty? : (any -> boolean)
fifth : ((listof Y) -> Y)
first : ((cons Y (listof X)) -> Y)
fourth : ((listof Y) -> Y)
length : ((listof any) -> number)
list : (any ... -> (listof any))
list* : (any ... (listof any) -> (listof any))
list-ref : ((listof X) natural-number -> X)
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 : (string nat -> 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 : (symbol string -> void)
exit : (-> void)
identity : (any -> any)
struct? : (any -> boolean)
```

## 1.1 define

---

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

Defines a function. The first *id* inside the parentheses is the name of the function. All remaining *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 `expr`s: a question `expr` and an answer `expr`.

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

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

---

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

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

## 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) 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

---

```
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 (comparing values with eq?)

---

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

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

---

```
null : empty
```

Purpose: the empty list

---

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

Purpose: to determine whether some value is the empty list

---

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

Purpose: to determine whether some value is a constructed list

---

```
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` : (string nat -> 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 *i*th 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 compare two values

---

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

Purpose: to determine whether two values are structurally equal

---

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

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

---

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

Purpose: to compare two values

---

`error` : (symbol string -> void)

Purpose: to signal an error

---

`exit` : (-> void)

Purpose: to exit the running program

---

`identity` : (any -> any)

Purpose: to return the argument unchanged

---

`struct?` : (any -> boolean)

Purpose: to determine whether some value is a structure

## 2 Beginning Student with List Abbreviations

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

def-or-expr = definition
            | expr
            | 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-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))
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)
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 : (string nat -> 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 : (symbol string -> 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 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

---

```
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 (comparing values with eq?)

---

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

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

---

`null` : empty

Purpose: the empty list

---

`null?` : (any -> boolean)

Purpose: to determine whether some value is the empty list

---

`pair?` : (any -> boolean)

Purpose: to determine whether some value is a constructed list

---

`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` : (string nat -> 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 *i*th 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 compare two values

---

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

Purpose: to determine whether two values are structurally equal

---

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

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

---

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

Purpose: to compare two values

---

`error` : (symbol string -> void)

Purpose: to signal an error

---

`exit` : (-> void)

Purpose: to exit the running program

---

`identity` : (any -> any)

Purpose: to return the argument unchanged

---

`struct?` : (any -> boolean)

Purpose: to determine whether some value is a structure

## 2.4 Unchanged Forms

---

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

The same as Beginning's `define`.

---

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

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

---

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

The same as Beginning's `cond`.

---

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

The same as Beginning's `if`.



---

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

The same as Beginning's and and or.

---

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

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

---

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

Constants for the empty list, true, and false.

---

```
(require 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-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)
          (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)
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 : (string nat -> 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 : (symbol string -> 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.

### 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)
          (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

---

```
cdadar : ((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

---

`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 (comparing values with eq?)

---

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

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

---

`null` : empty

Purpose: the empty list

---

`null?` : (any -> boolean)

Purpose: to determine whether some value is the empty list

---

`pair?` : (any -> boolean)

Purpose: to determine whether some value is a constructed list

---

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

Purpose: to extract the y component of a posn

---

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

Purpose: to determine if its input is a posn

---

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

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

---

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

Purpose: to determine whether a character represents an alphabetic character

---

```
char-alphabetic? : (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` : (string nat -> 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 *i*th 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 compare two values

---

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

Purpose: to determine whether two values are structurally equal

---

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

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

---

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

Purpose: to compare two values

---

`error` : (symbol string -> void)

Purpose: to signal an error

---

`exit` : (-> void)

Purpose: to exit the running program

---

`identity` : (any -> any)

Purpose: to return the argument unchanged

---

`struct?` : (any -> boolean)

Purpose: to determine whether some value is a structure

---

`*` : (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 minimizes 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:  $(\text{ormap } p \text{ (list } x-1 \dots x-n)) = (\text{or } (p \ x-1) \dots (p \ x-n))$

---

```
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)
```

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-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)
          (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)
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 : (string nat -> 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 : (symbol string -> 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)  
          (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

---

`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 (comparing values with eq?)

---

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

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

---

`null` : empty

Purpose: the empty list

---

`null?` : (any -> boolean)

Purpose: to determine whether some value is the empty list

---

`pair?` : (any -> boolean)

Purpose: to determine whether some value is a constructed list

---

`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` : (string nat -> 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 *i*th 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 compare two values

---

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

Purpose: to determine whether two values are structurally equal

---

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

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

---

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

Purpose: to compare two values

---

`error` : (symbol string -> void)

Purpose: to signal an error

---

`exit` : (-> void)

Purpose: to exit the running program

---

`identity` : (any -> any)

Purpose: to return the argument unchanged

---

`struct?` : (any -> boolean)

Purpose: to determine whether some value is a structure

---

`*` : (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 minimizes 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:  $(\text{ormap } p \text{ (list } x-1 \dots x-n)) = (\text{or } (p \ x-1) \dots (p \ x-n))$

---

`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)
```

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-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-ref : ((listof X) natural-number -> X)
list? : (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 : (string nat -> 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 : (symbol string -> void)
exit : (-> void)
force : (delay -> any)
identity : (any -> any)
promise? : (any -> boolean)
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)
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)
( $\lambda$  (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` : (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

---

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

Purpose: to extract the indexed item from the list

---

`list?` : (any -> boolean)

Purpose: to determine whether some value is a list

---

`member` : (any (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 (comparing values with eq?)

---

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

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

---

`null` : empty

Purpose: the empty list

---

`null?` : (any -> boolean)

Purpose: to determine whether some value is the empty list

---

`pair?` : (any -> boolean)

Purpose: to determine whether some value is a constructed list

---

`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` : (string nat -> 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 *i*th 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 compare two values

---

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

Purpose: to determine whether two values are structurally equal

---

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

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

---

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

Purpose: to compare two values

---

`error` : (symbol string -> void)

Purpose: to signal an error

---

`exit` : (-> void)

Purpose: to exit the running program

---

`force` : (delay -> any)

Purpose: to find the delayed value; see also delay

---

`identity` : (any -> any)

Purpose: to return the argument unchanged

---

`promise?` : (any -> boolean)

Purpose: to determine if a value is delayed

---

`struct?` : (any -> boolean)

Purpose: to determine whether some value is a structure

---

`void` : (-> void)

Purpose: produces a void value

---

`void?` : (any -> boolean)

Purpose: to determine if a value is void

---

`*` : (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 minimizes 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:  $(\text{ormap } p \text{ (list } x-1 \dots x-n)) = (\text{or } (p \ x-1) \dots (p \ x-n))$

---

`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

---

`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)
```

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, 140  
#%app, 75  
#%app, 107  
#%app, 12  
\*, 126  
\*, 44  
\*, 162  
\*, 16  
\*, 95  
+, 44  
+, 95  
+, 16  
+, 162  
+, 126  
-, 163  
-, 16  
-, 44  
-, 95  
-, 126  
/, 17  
/, 127  
/, 45  
/, 95  
/, 163  
<, 143  
<, 107  
<, 45  
<, 75  
<, 17  
<=, 45  
<=, 17  
<=, 75  
<=, 107  
<=, 143  
=, 17  
=, 143  
=, 76  
=, 45  
=, 107  
=~ , 35  
=~ , 161  
=~ , 63  
=~ , 125  
=~ , 93  
>, 17  
>, 45  
>, 107  
>, 76  
>, 143  
>=, 76  
>=, 107  
>=, 143  
>=, 17  
>=, 45  
abs, 76  
abs, 108  
abs, 143  
abs, 45  
abs, 17  
acos, 108  
acos, 76  
acos, 17  
acos, 143  
acos, 45  
add1, 108  
add1, 143  
add1, 76  
add1, 45  
add1, 17  
Advanced Student, 131  
and, 13  
and, 129  
and, 97  
and, 167  
and, 13  
and, 65  
andmap, 127  
andmap, 163  
andmap, 95  
angle, 108  
angle, 46  
angle, 17

[angle](#), 76  
[angle](#), 143  
[append](#), 114  
[append](#), 150  
[append](#), 83  
[append](#), 52  
[append](#), 24  
[apply](#), 95  
[apply](#), 127  
[apply](#), 163  
[argmax](#), 163  
[argmax](#), 95  
[argmax](#), 127  
[argmin](#), 127  
[argmin](#), 163  
[argmin](#), 95  
[asin](#), 18  
[asin](#), 46  
[asin](#), 108  
[asin](#), 143  
[asin](#), 76  
[assq](#), 83  
[assq](#), 52  
[assq](#), 114  
[assq](#), 150  
[assq](#), 24  
[atan](#), 46  
[atan](#), 108  
[atan](#), 144  
[atan](#), 76  
[atan](#), 18  
[begin](#), 140  
[begin](#), 140  
[begin0](#), 140  
[begin0](#), 140  
[Beginning Student](#), 5  
[Beginning Student with List Abbreviations](#),  
37  
[boolean=?](#), 51  
[boolean=?](#), 149  
[boolean=?](#), 113  
[boolean=?](#), 23  
[boolean=?](#), 82  
[boolean?](#), 149  
[boolean?](#), 23  
[boolean?](#), 114  
[boolean?](#), 82  
[boolean?](#), 51  
[box](#), 166  
[box?](#), 166  
[build-list](#), 127  
[build-list](#), 163  
[build-list](#), 96  
[build-string](#), 163  
[build-string](#), 127  
[build-string](#), 96  
[build-vector](#), 166  
[caaar](#), 83  
[caaar](#), 150  
[caaar](#), 24  
[caaar](#), 115  
[caaar](#), 52  
[caadr](#), 52  
[caadr](#), 83  
[caadr](#), 24  
[caadr](#), 150  
[caadr](#), 115  
[caar](#), 53  
[caar](#), 83  
[caar](#), 115  
[caar](#), 25  
[caar](#), 150  
[cadar](#), 53  
[cadar](#), 150  
[cadar](#), 25  
[cadar](#), 115  
[cadar](#), 83  
[caddr](#), 25  
[caddr](#), 151  
[caddr](#), 53  
[caddr](#), 84  
[caddr](#), 115  
[caddr](#), 115  
[caddr](#), 151

caddr, 84  
caddr, 53  
caddr, 25  
cadr, 53  
cadr, 84  
cadr, 25  
cadr, 151  
cadr, 115  
car, 115  
car, 25  
car, 84  
car, 53  
car, 151  
case, 142  
case, 142  
cdaar, 84  
cdaar, 116  
cdaar, 25  
cdaar, 151  
cdaar, 53  
cdadr, 25  
cdadr, 84  
cdadr, 53  
cdadr, 116  
cdadr, 151  
cdar, 26  
cdar, 116  
cdar, 151  
cdar, 54  
cdar, 84  
cddar, 116  
cddar, 54  
cddar, 84  
cddar, 152  
cddar, 26  
cdddr, 85  
cdddr, 116  
cdddr, 54  
cdddr, 26  
cdddr, 152  
cddr, 54  
cddr, 116  
cddr, 26  
cddr, 85  
cddr, 152  
cdr, 152  
cdr, 85  
cdr, 54  
cdr, 26  
cdr, 116  
ceiling, 76  
ceiling, 18  
ceiling, 46  
ceiling, 108  
ceiling, 144  
char->integer, 88  
char->integer, 29  
char->integer, 119  
char->integer, 155  
char->integer, 57  
char-alphabetic?, 29  
char-alphabetic?, 88  
char-alphabetic?, 120  
char-alphabetic?, 155  
char-alphabetic?, 57  
char-ci<=?, 120  
char-ci<=?, 88  
char-ci<=?, 29  
char-ci<=?, 155  
char-ci<=?, 57  
char-ci<?, 120  
char-ci<?, 155  
char-ci<?, 29  
char-ci<?, 88  
char-ci<?, 57  
char-ci=?, 120  
char-ci=?, 29  
char-ci=?, 57  
char-ci=?, 88  
char-ci=?, 156  
char-ci>=?, 58  
char-ci>=?, 156  
char-ci>=?, 30  
char-ci>=?, 88

char-ci>=?, 120  
char-ci>?, 30  
char-ci>?, 88  
char-ci>?, 58  
char-ci>?, 120  
char-ci>?, 156  
char-downcase, 58  
char-downcase, 120  
char-downcase, 89  
char-downcase, 30  
char-downcase, 156  
char-lower-case?, 120  
char-lower-case?, 89  
char-lower-case?, 58  
char-lower-case?, 30  
char-lower-case?, 156  
char-numeric?, 58  
char-numeric?, 156  
char-numeric?, 120  
char-numeric?, 30  
char-numeric?, 89  
char-upcase, 156  
char-upcase, 58  
char-upcase, 30  
char-upcase, 120  
char-upcase, 89  
char-upper-case?, 30  
char-upper-case?, 156  
char-upper-case?, 58  
char-upper-case?, 89  
char-upper-case?, 121  
char-whitespace?, 156  
char-whitespace?, 58  
char-whitespace?, 121  
char-whitespace?, 30  
char-whitespace?, 89  
char<=?, 156  
char<=?, 58  
char<=?, 30  
char<=?, 89  
char<=?, 121  
char<=?, 59  
char<?, 89  
char<?, 121  
char<?, 157  
char<?, 31  
char=?, 31  
char=?, 59  
char=?, 89  
char=?, 121  
char=?, 157  
char>=?, 121  
char>=?, 157  
char>=?, 31  
char>=?, 59  
char>=?, 89  
char>?, 59  
char>?, 157  
char>?, 90  
char>?, 121  
char>?, 31  
char?, 157  
char?, 121  
char?, 31  
char?, 59  
char?, 90  
check-error, 130  
check-error, 98  
check-error, 14  
check-error, 167  
check-error, 65  
check-expect, 65  
check-expect, 167  
check-expect, 98  
check-expect, 14  
check-expect, 130  
check-within, 65  
check-within, 167  
check-within, 130  
check-within, 14  
check-within, 98  
complex?, 18  
complex?, 46  
complex?, 144

[complex?](#), 108  
[complex?](#), 77  
[compose](#), 164  
[compose](#), 96  
[compose](#), 128  
[cond](#), 13  
[cond](#), 97  
[cond](#), 129  
[cond](#), 64  
[cond](#), 167  
[cond](#), 13  
[conjugate](#), 46  
[conjugate](#), 108  
[conjugate](#), 144  
[conjugate](#), 77  
[conjugate](#), 18  
[cons](#), 54  
[cons](#), 117  
[cons](#), 152  
[cons](#), 26  
[cons](#), 85  
[cons?](#), 54  
[cons?](#), 26  
[cons?](#), 85  
[cons?](#), 152  
[cons?](#), 117  
[cos](#), 144  
[cos](#), 77  
[cos](#), 108  
[cos](#), 46  
[cos](#), 18  
[cosh](#), 109  
[cosh](#), 18  
[cosh](#), 144  
[cosh](#), 46  
[cosh](#), 77  
[current-seconds](#), 109  
[current-seconds](#), 18  
[current-seconds](#), 77  
[current-seconds](#), 144  
[current-seconds](#), 46  
[define](#), 139  
[define](#), 11  
[define](#), 73  
[define](#), 106  
[define](#), 106  
[define](#), 11  
[define](#), 73  
[define](#), 139  
[define](#), 64  
[define-struct](#), 139  
[define-struct](#), 11  
[define-struct](#), 73  
[define-struct](#), 64  
[define-struct](#), 11  
[define-struct](#), 139  
[define-struct](#), 129  
[define-struct](#), 73  
[delay](#), 141  
[delay](#), 141  
[denominator](#), 77  
[denominator](#), 18  
[denominator](#), 144  
[denominator](#), 109  
[denominator](#), 46  
[display](#), 165  
[e](#), 47  
[e](#), 77  
[e](#), 19  
[e](#), 144  
[e](#), 109  
[eighth](#), 26  
[eighth](#), 152  
[eighth](#), 54  
[eighth](#), 85  
[eighth](#), 117  
[else](#), 129  
[else](#), 64  
[else](#), 13  
[else](#), 97  
[else](#), 167  
[empty](#), 14  
[empty](#), 168  
[empty](#), 130

empty, 65  
empty, 14  
empty, 98  
empty?, 54  
empty?, 85  
empty?, 152  
empty?, 117  
empty?, 26  
eof, 161  
eof, 94  
eof, 125  
eof, 35  
eof, 63  
eof-object?, 63  
eof-object?, 125  
eof-object?, 161  
eof-object?, 94  
eof-object?, 35  
eq?, 161  
eq?, 94  
eq?, 35  
eq?, 63  
eq?, 125  
equal?, 35  
equal?, 126  
equal?, 63  
equal?, 161  
equal?, 94  
equal~?, 161  
equal~?, 126  
equal~?, 94  
equal~?, 35  
equal~?, 63  
eqv?, 35  
eqv?, 94  
eqv?, 126  
eqv?, 63  
eqv?, 161  
error, 94  
error, 162  
error, 64  
error, 36  
error, 126  
even?, 77  
even?, 19  
even?, 109  
even?, 145  
even?, 47  
exact->inexact, 19  
exact->inexact, 77  
exact->inexact, 145  
exact->inexact, 47  
exact->inexact, 109  
exact?, 19  
exact?, 109  
exact?, 145  
exact?, 77  
exact?, 47  
exit, 126  
exit, 162  
exit, 36  
exit, 64  
exit, 94  
exp, 78  
exp, 109  
exp, 19  
exp, 47  
exp, 145  
explode, 59  
explode, 121  
explode, 157  
explode, 90  
explode, 31  
expt, 78  
expt, 145  
expt, 19  
expt, 47  
expt, 109  
false, 130  
false, 98  
false, 15  
false, 168  
false, 65  
false?, 51

false?, 23  
false?, 149  
false?, 82  
false?, 114  
fifth, 27  
fifth, 117  
fifth, 55  
fifth, 152  
fifth, 85  
filter, 96  
filter, 164  
filter, 128  
first, 55  
first, 117  
first, 85  
first, 27  
first, 153  
floor, 109  
floor, 78  
floor, 145  
floor, 47  
floor, 19  
foldl, 128  
foldl, 96  
foldl, 164  
foldr, 96  
foldr, 128  
foldr, 164  
for-each, 164  
for-each, 96  
for-each, 128  
force, 162  
format, 31  
format, 59  
format, 90  
format, 157  
format, 121  
fourth, 55  
fourth, 27  
fourth, 153  
fourth, 117  
fourth, 86  
Function Calls, 140  
Function Calls, 12  
Function Calls, 74  
Function Calls, 107  
gcd, 145  
gcd, 78  
gcd, 19  
gcd, 47  
gcd, 110  
*How to Design Programs Languages*, 1  
Identifiers, 15  
Identifiers, 75  
identity, 64  
identity, 94  
identity, 162  
identity, 36  
identity, 126  
if, 13  
if, 129  
if, 167  
if, 64  
if, 13  
if, 97  
imag-part, 19  
imag-part, 145  
imag-part, 110  
imag-part, 78  
imag-part, 47  
image=?, 35  
image=?, 161  
image=?, 63  
image=?, 93  
image=?, 125  
image?, 35  
image?, 161  
image?, 125  
image?, 63  
image?, 93  
implode, 122  
implode, 157  
implode, 90  
implode, 59

implode, 31  
inexact->exact, 47  
inexact->exact, 145  
inexact->exact, 19  
inexact->exact, 110  
inexact->exact, 78  
inexact?, 110  
inexact?, 145  
inexact?, 78  
inexact?, 48  
inexact?, 20  
int->string, 90  
int->string, 59  
int->string, 157  
int->string, 122  
int->string, 31  
integer->char, 78  
integer->char, 146  
integer->char, 20  
integer->char, 48  
integer->char, 110  
integer-sqrt, 48  
integer-sqrt, 20  
integer-sqrt, 110  
integer-sqrt, 146  
integer-sqrt, 78  
integer?, 20  
integer?, 79  
integer?, 48  
integer?, 110  
integer?, 146  
Intermediate Student, 66  
Intermediate Student with Lambda, 99  
lambda, 139  
lambda, 106  
lambda, 139  
lambda, 73  
lambda, 106  
lambda, 11  
lambda, 64  
lcm, 79  
lcm, 20  
lcm, 110  
lcm, 146  
lcm, 48  
length, 117  
length, 27  
length, 55  
length, 86  
length, 153  
let, 141  
let, 74  
let, 129  
let, 141  
let\*, 74  
let\*, 129  
let\*, 167  
letrec, 167  
letrec, 74  
letrec, 129  
letrec, let, and let\*, 74  
list, 27  
list, 153  
list, 55  
list, 86  
list, 117  
list\*, 27  
list\*, 117  
list\*, 86  
list\*, 55  
list->string, 90  
list->string, 122  
list->string, 157  
list->string, 59  
list->string, 31  
list-ref, 86  
list-ref, 55  
list-ref, 27  
list-ref, 118  
list-ref, 153  
list?, 153  
local, 74  
local, 167  
local, 74



local, 129  
log, 146  
log, 48  
log, 79  
log, 110  
log, 20  
magnitude, 111  
magnitude, 146  
magnitude, 79  
magnitude, 48  
magnitude, 20  
make-polar, 20  
make-polar, 48  
make-polar, 146  
make-polar, 111  
make-polar, 79  
make-posn, 87  
make-posn, 154  
make-posn, 119  
make-posn, 29  
make-posn, 57  
make-rectangular, 79  
make-rectangular, 146  
make-rectangular, 20  
make-rectangular, 48  
make-rectangular, 111  
make-string, 122  
make-string, 90  
make-string, 32  
make-string, 158  
make-string, 60  
make-vector, 166  
map, 128  
map, 164  
map, 96  
max, 49  
max, 111  
max, 79  
max, 146  
max, 20  
member, 55  
member, 153  
member, 27  
member, 86  
member, 118  
memf, 128  
memf, 164  
memf, 97  
memq, 55  
memq, 153  
memq, 118  
memq, 86  
memq, 27  
memv, 153  
memv, 27  
memv, 86  
memv, 55  
memv, 118  
min, 21  
min, 49  
min, 146  
min, 79  
min, 111  
modulo, 111  
modulo, 21  
modulo, 147  
modulo, 49  
modulo, 79  
negative?, 79  
negative?, 147  
negative?, 21  
negative?, 49  
negative?, 111  
newline, 165  
not, 51  
not, 114  
not, 149  
not, 23  
not, 82  
null, 86  
null, 153  
null, 118  
null, 28  
null, 56

null?, 28  
null?, 86  
null?, 118  
null?, 56  
null?, 154  
number->string, 111  
number->string, 49  
number->string, 80  
number->string, 147  
number->string, 21  
number?, 80  
number?, 111  
number?, 21  
number?, 147  
number?, 49  
numerator, 49  
numerator, 80  
numerator, 111  
numerator, 147  
numerator, 21  
odd?, 80  
odd?, 147  
odd?, 21  
odd?, 49  
odd?, 112  
or, 14  
or, 167  
or, 129  
or, 97  
or, 65  
or, 14  
ormap, 97  
ormap, 164  
ormap, 128  
pair?, 87  
pair?, 56  
pair?, 154  
pair?, 118  
pair?, 28  
pi, 49  
pi, 112  
pi, 147  
pi, 80  
pi, 21  
positive?, 112  
positive?, 80  
positive?, 49  
positive?, 147  
positive?, 21  
posn-x, 29  
posn-x, 57  
posn-x, 87  
posn-x, 119  
posn-x, 155  
posn-y, 119  
posn-y, 57  
posn-y, 29  
posn-y, 88  
posn-y, 155  
posn?, 57  
posn?, 119  
posn?, 155  
posn?, 29  
posn?, 88  
pretty-print, 165  
Primitive Calls, 12  
Primitive Operation Names, 107  
Primitive Operations, 143  
Primitive Operations, 44  
Primitive Operations, 16  
Primitive Operations, 75  
print, 165  
printf, 165  
procedure?, 97  
procedure?, 165  
procedure?, 129  
promise?, 162  
Quasiquote, 43  
quasiquote, 43  
quicksort, 97  
quicksort, 165  
quicksort, 129  
Quote, 43  
quote, 43

quote, 15  
quotient, 50  
quotient, 112  
quotient, 22  
quotient, 80  
quotient, 147  
random, 148  
random, 112  
random, 22  
random, 50  
random, 80  
rational?, 148  
rational?, 50  
rational?, 22  
rational?, 112  
rational?, 80  
read, 165  
real-part, 112  
real-part, 148  
real-part, 81  
real-part, 22  
real-part, 50  
real?, 112  
real?, 81  
real?, 22  
real?, 50  
real?, 148  
recur, 141  
recur, 141  
remainder, 22  
remainder, 112  
remainder, 50  
remainder, 148  
remainder, 81  
remove, 56  
remove, 28  
remove, 87  
remove, 154  
remove, 118  
replicate, 90  
replicate, 60  
replicate, 32  
replicate, 158  
replicate, 122  
require, 15  
require, 130  
require, 168  
require, 15  
require, 65  
require, 98  
rest, 154  
rest, 28  
rest, 87  
rest, 56  
rest, 118  
reverse, 154  
reverse, 28  
reverse, 87  
reverse, 56  
reverse, 118  
round, 50  
round, 148  
round, 113  
round, 81  
round, 22  
second, 119  
second, 56  
second, 28  
second, 154  
second, 87  
set!, 140  
set!, 140  
set-box!, 167  
set-posn-x!, 155  
set-posn-y!, 155  
seventh, 87  
seventh, 154  
seventh, 119  
seventh, 56  
seventh, 28  
sgn, 22  
sgn, 50  
sgn, 81  
sgn, 113

sgn, 148  
 shared, 141  
 shared, 141  
 sin, 113  
 sin, 50  
 sin, 148  
 sin, 81  
 sin, 22  
 sinh, 148  
 sinh, 81  
 sinh, 22  
 sinh, 113  
 sinh, 51  
 sixth, 87  
 sixth, 28  
 sixth, 56  
 sixth, 119  
 sixth, 154  
 sort, 165  
 sort, 97  
 sort, 129  
 sqr, 113  
 sqr, 51  
 sqr, 23  
 sqr, 148  
 sqr, 81  
 sqrt, 149  
 sqrt, 51  
 sqrt, 113  
 sqrt, 81  
 sqrt, 23  
 string, 90  
 string, 158  
 string, 60  
 string, 32  
 string, 122  
 string->int, 158  
 string->int, 122  
 string->int, 32  
 string->int, 91  
 string->int, 60  
 string->list, 122  
 string->list, 91  
 string->list, 32  
 string->list, 158  
 string->list, 60  
 string->number, 158  
 string->number, 32  
 string->number, 60  
 string->number, 122  
 string->number, 91  
 string->symbol, 122  
 string->symbol, 91  
 string->symbol, 158  
 string->symbol, 60  
 string->symbol, 32  
 string-alphabetic?, 32  
 string-alphabetic?, 91  
 string-alphabetic?, 60  
 string-alphabetic?, 158  
 string-alphabetic?, 123  
 string-append, 32  
 string-append, 158  
 string-append, 123  
 string-append, 60  
 string-append, 91  
 string-ci<=?, 91  
 string-ci<=?, 158  
 string-ci<=?, 60  
 string-ci<=?, 32  
 string-ci<=?, 123  
 string-ci<?, 91  
 string-ci<?, 159  
 string-ci<?, 61  
 string-ci<?, 33  
 string-ci<?, 123  
 string-ci=?, 91  
 string-ci=?, 61  
 string-ci=?, 159  
 string-ci=?, 33  
 string-ci=?, 123  
 string-ci>=?, 159  
 string-ci>=?, 91  
 string-ci>=?, 61

string-ci>=?, 123  
 string-ci>=?, 33  
 string-ci>?, 159  
 string-ci>?, 123  
 string-ci>?, 61  
 string-ci>?, 92  
 string-ci>?, 33  
 string-copy, 92  
 string-copy, 159  
 string-copy, 33  
 string-copy, 123  
 string-copy, 61  
 string-ith, 33  
 string-ith, 92  
 string-ith, 123  
 string-ith, 159  
 string-ith, 61  
 string-length, 124  
 string-length, 33  
 string-length, 92  
 string-length, 61  
 string-length, 159  
 string-lower-case?, 33  
 string-lower-case?, 159  
 string-lower-case?, 61  
 string-lower-case?, 124  
 string-lower-case?, 92  
 string-numeric?, 33  
 string-numeric?, 124  
 string-numeric?, 61  
 string-numeric?, 92  
 string-numeric?, 159  
 string-ref, 62  
 string-ref, 160  
 string-ref, 124  
 string-ref, 92  
 string-ref, 34  
 string-upper-case?, 92  
 string-upper-case?, 62  
 string-upper-case?, 124  
 string-upper-case?, 160  
 string-upper-case?, 34  
 string-whitespace?, 92  
 string-whitespace?, 62  
 string-whitespace?, 160  
 string-whitespace?, 124  
 string-whitespace?, 34  
 string<=?, 34  
 string<=?, 62  
 string<=?, 124  
 string<=?, 160  
 string<=?, 93  
 string<?, 160  
 string<?, 124  
 string<?, 93  
 string<?, 62  
 string<?, 34  
 string=?, 62  
 string=?, 34  
 string=?, 124  
 string=?, 160  
 string=?, 93  
 string>=?, 93  
 string>=?, 160  
 string>=?, 124  
 string>=?, 34  
 string>=?, 62  
 string>?, 93  
 string>?, 34  
 string>?, 62  
 string>?, 125  
 string>?, 160  
 string?, 62  
 string?, 160  
 string?, 125  
 string?, 93  
 string?, 34  
 struct?, 64  
 struct?, 162  
 struct?, 95  
 struct?, 126  
 struct?, 36  
 sub1, 149  
 sub1, 23

`sub1`, 113  
`sub1`, 81  
`sub1`, 51  
`substring`, 34  
`substring`, 62  
`substring`, 125  
`substring`, 93  
`substring`, 160  
`symbol->string`, 114  
`symbol->string`, 52  
`symbol->string`, 149  
`symbol->string`, 23  
`symbol->string`, 82  
`symbol=?`, 114  
`symbol=?`, 82  
`symbol=?`, 52  
`symbol=?`, 24  
`symbol=?`, 149  
`symbol?`, 52  
`symbol?`, 114  
`symbol?`, 24  
`symbol?`, 82  
`symbol?`, 150  
Symbols, 15  
`tan`, 82  
`tan`, 113  
`tan`, 23  
`tan`, 149  
`tan`, 51  
Test Cases, 14  
`third`, 57  
`third`, 154  
`third`, 29  
`third`, 119  
`third`, 87  
`time`, 75  
`time`, 130  
`time`, 75  
`time`, 167  
`true`, 130  
`true`, 65  
`true`, 15  
`true`, 98  
`true`, 168  
`true` and `false`, 15  
`unbox`, 167  
Unchanged Forms, 64  
Unchanged Forms, 167  
Unchanged Forms, 129  
Unchanged Forms, 97  
`unless`, 142  
`unquote`, 44  
`unquote-splicing`, 44  
`vector`, 166  
`vector-length`, 166  
`vector-ref`, 166  
`vector-set!`, 166  
`vector?`, 166  
`void`, 162  
`void?`, 162  
`when`, 142  
`when` and `unless`, 142  
`write`, 166  
`zero?`, 82  
`zero?`, 51  
`zero?`, 23  
`zero?`, 149  
`zero?`, 113  
 $\lambda$ , 139  
 $\lambda$ , 106