

How to Design Programs Languages

Version 4.2

June 1, 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

1	Beginning Student	5
1.1	define	11
1.2	define-struct	11
1.3	Function Calls	12
1.4	Primitive Calls	12
1.5	cond	12
1.6	if	13
1.7	and	13
1.8	or	13
1.9	Test Cases	14
1.10	empty	14
1.11	Identifiers	14
1.12	Symbols	15
1.13	true and false	15
1.14	require	15
1.15	Primitive Operations	16
2	Beginning Student with List Abbreviations	36
2.1	Quote	42
2.2	Quasiquote	42
2.3	Primitive Operations	43
2.4	Unchanged Forms	63
3	Intermediate Student	65
3.1	define	72

3.2	<code>define-struct</code>	72
3.3	<code>local</code>	73
3.4	<code>letrec</code> , <code>let</code> , and <code>let*</code>	73
3.5	Function Calls	73
3.6	<code>time</code>	74
3.7	Identifiers	74
3.8	Primitive Operations	74
3.9	Unchanged Forms	96
4	Intermediate Student with Lambda	97
4.1	<code>define</code>	104
4.2	<code>lambda</code>	104
4.3	Function Calls	104
4.4	Primitive Operation Names	105
4.5	Unchanged Forms	126
5	Advanced Student	128
5.1	<code>define</code>	136
5.2	<code>define-struct</code>	136
5.3	<code>lambda</code>	136
5.4	Function Calls	136
5.5	<code>begin</code>	137
5.6	<code>begin0</code>	137
5.7	<code>set!</code>	137
5.8	<code>delay</code>	137
5.9	<code>shared</code>	138

5.10 let	138
5.11 recur	138
5.12 case	139
5.13 when and unless	139
5.14 Primitive Operations	139
5.15 Unchanged Forms	163
Index	165

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))
cdddd : ((cons W (cons Z (cons Y (listof X))))
        ->
        (listof X))
cddr : ((cons Z (cons Y (listof X))) -> (listof X))
cdr : ((cons Y (listof X)) -> (listof X))
cons : (X (listof X) -> (listof X))
cons? : (any -> boolean)
eighth : ((listof Y) -> Y)
empty? : (any -> boolean)
fifth : ((listof Y) -> Y)

```

```

first : ((cons Y (listof X)) -> Y)
fourth : ((listof Y) -> Y)
length : ((listof any) -> number)
list : (any ... -> (listof any))
list* : (any ... (listof any) -> (listof any))
list-ref : ((listof X) natural-number -> X)
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)
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 -> 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 *exprs*: a question *expr* and an answer *expr*.

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

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

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

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

`else`

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

1.6 if

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

The first *expr* (known as the “test” *expr*) is evaluated. If it evaluates to **true**, the result of the `if` expression is the result of evaluating the second *expr* (often called the “then” *expr*). If the test *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

```
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 -> string)

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

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

Purpose: to determine the length of a string

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

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

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

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

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

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

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

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

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

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

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

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

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

Purpose: to determine whether one string alphabetically precedes another

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

Purpose: to compare two strings character-wise

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

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

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

Purpose: to determine whether one string alphabetically succeeds another

`string?` : (any -> boolean)

Purpose: to determine whether a value is a string

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

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

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

Purpose: to determine whether two images are equal

`image?` : (any -> boolean)

Purpose: to determine whether a value is an image

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

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

`eof` : eof

Purpose: the end-of-file value

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

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

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

Purpose: to 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)
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 -> 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
 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

`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 -> 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
      | (quote id)
      | 'quoted ; quoted value
      | 'quasiquoted ; quasiquote
      | number
      | true
      | false
      | string
      | character

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

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

```

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

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

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

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)
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 -> 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

```
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

`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 -> string)

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

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

Purpose: to determine the length of a string

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

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

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

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

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

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

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

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

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

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

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

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

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

Purpose: to determine whether one string alphabetically precedes another

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

Purpose: to compare two strings character-wise

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

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

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

Purpose: to determine whether one string alphabetically succeeds another

`string?` : (any -> boolean)

Purpose: to determine whether a value is a string

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

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

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

Purpose: to determine whether two images are equal

`image?` : (any -> boolean)

Purpose: to determine whether a value is an image

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

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

`eof` : eof

Purpose: the end-of-file value

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

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

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

Purpose: to 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) (and ... (p x-n)))

```
apply : ((X-1 ... X-N -> Y)
         X-1
         ...
         X-i
         (list X-i+1 ... X-N)
         ->
         Y)
```

Purpose: to apply a function using items from a list as the arguments

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

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

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

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

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

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

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

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

Purpose: to determine whether the first argument produces true for some value in the second argument

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

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

`procedure?` : $(\text{any } \rightarrow \text{boolean})$

Purpose: to determine if a value is a procedure

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

Purpose: to construct a list from all items on a list in an order according to a predicate

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

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
      | (quote 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

```

< : (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)
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 -> 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 λ 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

`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 -> 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) (and ... (p x-n)))

`apply` : ((X-1 ... X-N -> Y)
X-1
...
X-i
(list X-i+1 ... X-N)
->
Y)

Purpose: to apply a function using items from a list as the arguments

`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: (ormap p (list x-1 ... x-n)) = (or (p x-1) (or ... (p x-n)))

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

Purpose: to determine if a value is a procedure

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

Purpose: to construct a list from all items on a list in an order according to a predicate

```
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
      | (quote id)
      | 'quoted ; quoted value
      | `quasiquoted ; quasiquote
```

```

| number
| true
| false
| string
| character

choice = id ; treated as a symbol
| number

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

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

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

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)
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 -> 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

`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 -> string)`

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

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

Purpose: to determine the length of a string

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

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

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

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

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

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

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

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

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

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

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

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

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

Purpose: to determine whether one string alphabetically precedes another

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

Purpose: to compare two strings character-wise

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

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

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

Purpose: to determine whether one string alphabetically succeeds another

`string?` : (any -> boolean)

Purpose: to determine whether a value is a string

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

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

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

Purpose: to determine whether two images are equal

`image?` : (any -> boolean)

Purpose: to determine whether a value is an image

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

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

`eof` : eof

Purpose: the end-of-file value

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

Purpose: to determine whether the first argument produces true for some value in the second argument

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

Purpose: (ormap p (list x-1 ... x-n)) = (or (p x-1) (or ... (p x-n)))

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

Purpose: to determine if a value is a procedure

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

Purpose: to construct a list from all items on a list in an order according to a predicate

```
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` : (\rightarrow `sexp`)

Purpose: to read input from the user

`write` : (`any` \rightarrow `void`)

Purpose: to print the argument to stdout (in a traditional style that is somewhere between print and display)

`build-vector` : (`nat` (`nat` \rightarrow `X`) \rightarrow (`vectorof` `X`))

Purpose: to construct a vector

`make-vector` : (`number` `X` \rightarrow (`vectorof` `X`))

Purpose: to construct a vector

`vector` : (`X` ... \rightarrow (`vector` `X` ...))

Purpose: to construct a vector

`vector-length` : ((`vector` `X`) \rightarrow `nat`)

Purpose: to determine the length of a vector

`vector-ref` : ((`vector` `X`) `nat` \rightarrow `X`)

Purpose: to extract an element from a vector

`vector-set!` : ((`vectorof` `X`) `nat` `X` \rightarrow `void`)

Purpose: to update a vector

`vector?` : (`any` \rightarrow `boolean`)

Purpose: to determine if a value is a vector

`box` : (`any` \rightarrow `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`, 137
`##app`, 74
`##app`, 105
`##app`, 12
`*`, 43
`*`, 124
`*`, 16
`*`, 93
`*`, 159
`+`, 43
`+`, 16
`+`, 159
`+`, 124
`+`, 93
`-`, 124
`-`, 93
`-`, 159
`-`, 43
`-`, 16
`/`, 44
`/`, 93
`/`, 124
`/`, 16
`/`, 159
`<`, 44
`<`, 74
`<`, 105
`<`, 16
`<`, 139
`<=`, 140
`<=`, 17
`<=`, 74
`<=`, 44
`<=`, 105
`=`, 17
`=`, 105
`=`, 44
`=`, 140
`=`, 74
`=~`, 123
`=~`, 61
`=~`, 157
`=~`, 34
`=~`, 92
`>`, 17
`>`, 140
`>`, 44
`>`, 75
`>`, 105
`>=`, 105
`>=`, 140
`>=`, 75
`>=`, 44
`>=`, 17
`abs`, 17
`abs`, 140
`abs`, 105
`abs`, 44
`abs`, 75
`acos`, 17
`acos`, 106
`acos`, 44
`acos`, 140
`acos`, 75
`add1`, 75
`add1`, 106
`add1`, 44
`add1`, 140
`add1`, 17
Advanced Student, 128
`and`, 13
`and`, 13
`and`, 127
`and`, 163
`and`, 63
`and`, 96
`andmap`, 93
`andmap`, 159
`andmap`, 124
`angle`, 106
`angle`, 44
`angle`, 17

[angle](#), 140
[angle](#), 75
[append](#), 51
[append](#), 146
[append](#), 23
[append](#), 81
[append](#), 112
[apply](#), 124
[apply](#), 94
[apply](#), 159
[argmax](#), 94
[argmax](#), 159
[argmax](#), 124
[argmin](#), 159
[argmin](#), 94
[argmin](#), 125
[asin](#), 140
[asin](#), 45
[asin](#), 106
[asin](#), 75
[asin](#), 17
[assq](#), 24
[assq](#), 146
[assq](#), 81
[assq](#), 51
[assq](#), 112
[atan](#), 45
[atan](#), 140
[atan](#), 75
[atan](#), 17
[atan](#), 106
[begin](#), 137
[begin](#), 137
[begin0](#), 137
[begin0](#), 137
[Beginning Student](#), 5
[Beginning Student with List Abbreviations](#),
36
[boolean=?](#), 80
[boolean=?](#), 111
[boolean=?](#), 146
[boolean=?](#), 50
[boolean=?](#), 23
[boolean?](#), 146
[boolean?](#), 23
[boolean?](#), 81
[boolean?](#), 50
[boolean?](#), 111
[box](#), 162
[box?](#), 163
[build-list](#), 159
[build-list](#), 125
[build-list](#), 94
[build-string](#), 94
[build-string](#), 160
[build-string](#), 125
[build-vector](#), 162
[caaar](#), 82
[caaar](#), 147
[caaar](#), 51
[caaar](#), 112
[caaar](#), 24
[caadr](#), 24
[caadr](#), 82
[caadr](#), 113
[caadr](#), 147
[caadr](#), 51
[caar](#), 51
[caar](#), 24
[caar](#), 113
[caar](#), 82
[caar](#), 147
[cadar](#), 52
[cadar](#), 82
[cadar](#), 113
[cadar](#), 24
[cadar](#), 147
[caddr](#), 82
[caddr](#), 113
[caddr](#), 24
[caddr](#), 52
[caddr](#), 147
[caddr](#), 113
[caddr](#), 52

caddr, 24
caddr, 147
caddr, 82
cadr, 52
cadr, 147
cadr, 25
cadr, 82
cadr, 113
car, 82
car, 25
car, 113
car, 147
car, 52
case, 139
case, 139
cdaar, 83
cdaar, 25
cdaar, 52
cdaar, 148
cdaar, 113
cdadr, 83
cdadr, 52
cdadr, 148
cdadr, 25
cdadr, 114
cdar, 83
cdar, 114
cdar, 52
cdar, 25
cdar, 148
cddar, 25
cddar, 114
cddar, 53
cddar, 83
cddar, 148
cdddr, 83
cdddr, 114
cdddr, 148
cdddr, 53
cdddr, 25
cddr, 83
cddr, 148
cddr, 25
cddr, 53
cddr, 114
cdr, 148
cdr, 114
cdr, 53
cdr, 83
cdr, 26
ceiling, 141
ceiling, 75
ceiling, 106
ceiling, 18
ceiling, 45
char->integer, 28
char->integer, 56
char->integer, 151
char->integer, 86
char->integer, 117
char-alphabetic?, 86
char-alphabetic?, 28
char-alphabetic?, 152
char-alphabetic?, 117
char-alphabetic?, 56
char-ci<=?, 152
char-ci<=?, 86
char-ci<=?, 56
char-ci<=?, 117
char-ci<=?, 29
char-ci<?, 29
char-ci<?, 87
char-ci<?, 117
char-ci<?, 152
char-ci<?, 56
char-ci=?, 87
char-ci=?, 56
char-ci=?, 152
char-ci=?, 29
char-ci=?, 117
char-ci>=?, 152
char-ci>=?, 29
char-ci>=?, 87
char-ci>=?, 56

char-ci>=?, 118
char-ci>?, 152
char-ci>?, 29
char-ci>?, 118
char-ci>?, 87
char-ci>?, 56
char-downcase, 118
char-downcase, 29
char-downcase, 56
char-downcase, 87
char-downcase, 152
char-lower-case?, 118
char-lower-case?, 87
char-lower-case?, 57
char-lower-case?, 152
char-lower-case?, 29
char-numeric?, 118
char-numeric?, 29
char-numeric?, 87
char-numeric?, 152
char-numeric?, 57
char-upcase, 152
char-upcase, 87
char-upcase, 29
char-upcase, 57
char-upcase, 118
char-upper-case?, 30
char-upper-case?, 57
char-upper-case?, 118
char-upper-case?, 153
char-upper-case?, 87
char-whitespace?, 30
char-whitespace?, 57
char-whitespace?, 118
char-whitespace?, 153
char-whitespace?, 87
char<=?, 118
char<=?, 153
char<=?, 30
char<=?, 88
char<=?, 57
char<?, 30
char<?, 153
char<?, 57
char<?, 118
char<?, 88
char=?, 153
char=?, 88
char=?, 57
char=?, 30
char=?, 119
char>=?, 57
char>=?, 119
char>=?, 30
char>=?, 88
char>=?, 153
char>?, 153
char>?, 88
char>?, 57
char>?, 119
char>?, 30
char?, 119
char?, 58
char?, 153
char?, 88
char?, 30
check-error, 14
check-error, 127
check-error, 164
check-error, 63
check-error, 96
check-expect, 14
check-expect, 127
check-expect, 63
check-expect, 163
check-expect, 96
check-within, 164
check-within, 127
check-within, 96
check-within, 63
check-within, 14
complex?, 18
complex?, 141
complex?, 45

[complex?](#), 75
[complex?](#), 106
[compose](#), 94
[compose](#), 160
[compose](#), 125
[cond](#), 12
[cond](#), 163
[cond](#), 127
[cond](#), 63
[cond](#), 12
[cond](#), 96
[conjugate](#), 141
[conjugate](#), 18
[conjugate](#), 45
[conjugate](#), 76
[conjugate](#), 106
[cons](#), 114
[cons](#), 53
[cons](#), 83
[cons](#), 26
[cons](#), 149
[cons?](#), 84
[cons?](#), 114
[cons?](#), 26
[cons?](#), 149
[cons?](#), 53
[cos](#), 76
[cos](#), 45
[cos](#), 18
[cos](#), 141
[cos](#), 106
[cosh](#), 76
[cosh](#), 106
[cosh](#), 141
[cosh](#), 45
[cosh](#), 18
[current-seconds](#), 18
[current-seconds](#), 141
[current-seconds](#), 107
[current-seconds](#), 76
[current-seconds](#), 45
[define](#), 136
[define](#), 11
[define](#), 72
[define](#), 104
[define](#), 11
[define](#), 104
[define](#), 63
[define](#), 136
[define](#), 72
[define-struct](#), 136
[define-struct](#), 11
[define-struct](#), 72
[define-struct](#), 11
[define-struct](#), 136
[define-struct](#), 63
[define-struct](#), 126
[define-struct](#), 72
[delay](#), 137
[delay](#), 137
[denominator](#), 107
[denominator](#), 45
[denominator](#), 141
[denominator](#), 18
[denominator](#), 76
[display](#), 161
[e](#), 45
[e](#), 76
[e](#), 107
[e](#), 18
[e](#), 141
[eighth](#), 149
[eighth](#), 26
[eighth](#), 115
[eighth](#), 84
[eighth](#), 53
[else](#), 96
[else](#), 163
[else](#), 13
[else](#), 127
[else](#), 63
[empty](#), 14
[empty](#), 63
[empty](#), 14

empty, 127
empty, 96
empty, 164
empty?, 149
empty?, 53
empty?, 84
empty?, 26
empty?, 115
eof, 123
eof, 34
eof, 157
eof, 92
eof, 61
eof-object?, 92
eof-object?, 62
eof-object?, 157
eof-object?, 34
eof-object?, 123
eq?, 92
eq?, 123
eq?, 157
eq?, 34
eq?, 62
equal?, 123
equal?, 92
equal?, 157
equal?, 62
equal?, 34
equal~?, 62
equal~?, 92
equal~?, 158
equal~?, 35
equal~?, 123
eqv?, 123
eqv?, 158
eqv?, 62
eqv?, 92
eqv?, 35
error, 62
error, 123
error, 93
error, 158
error, 35
even?, 18
even?, 107
even?, 76
even?, 46
even?, 141
exact->inexact, 141
exact->inexact, 46
exact->inexact, 18
exact->inexact, 107
exact->inexact, 76
exact?, 107
exact?, 46
exact?, 76
exact?, 19
exact?, 142
exit, 93
exit, 35
exit, 62
exit, 158
exit, 123
exp, 142
exp, 107
exp, 76
exp, 19
exp, 46
explode, 88
explode, 30
explode, 58
explode, 153
explode, 119
expt, 142
expt, 19
expt, 77
expt, 46
expt, 107
false, 63
false, 127
false, 164
false, 96
false, 15
false?, 23

false?, 81
 false?, 146
 false?, 50
 false?, 111
 fifth, 84
 fifth, 26
 fifth, 115
 fifth, 53
 fifth, 149
 filter, 94
 filter, 125
 filter, 160
 first, 149
 first, 26
 first, 84
 first, 54
 first, 115
 floor, 46
 floor, 142
 floor, 19
 floor, 77
 floor, 107
 foldl, 94
 foldl, 160
 foldl, 125
 foldr, 125
 foldr, 95
 foldr, 160
 for-each, 160
 for-each, 125
 for-each, 95
 force, 158
 format, 30
 format, 119
 format, 58
 format, 153
 format, 88
 fourth, 26
 fourth, 149
 fourth, 84
 fourth, 115
 fourth, 54
 Function Calls, 136
 Function Calls, 12
 Function Calls, 73
 Function Calls, 104
 gcd, 77
 gcd, 46
 gcd, 107
 gcd, 142
 gcd, 19
How to Design Programs Languages, 1
 Identifiers, 14
 Identifiers, 74
 identity, 124
 identity, 62
 identity, 93
 identity, 158
 identity, 35
 if, 13
 if, 163
 if, 96
 if, 63
 if, 127
 if, 13
 imag-part, 108
 imag-part, 142
 imag-part, 77
 imag-part, 19
 imag-part, 46
 image=?, 157
 image=?, 61
 image=?, 92
 image=?, 122
 image=?, 34
 image?, 123
 image?, 34
 image?, 157
 image?, 61
 image?, 92
 implode, 154
 implode, 58
 implode, 88
 implode, 31

implode, 119
inexact->exact, 142
inexact->exact, 77
inexact->exact, 19
inexact->exact, 108
inexact->exact, 46
inexact?, 46
inexact?, 19
inexact?, 77
inexact?, 142
inexact?, 108
int->string, 88
int->string, 119
int->string, 58
int->string, 31
int->string, 154
integer->char, 47
integer->char, 108
integer->char, 142
integer->char, 19
integer->char, 77
integer-sqrt, 47
integer-sqrt, 108
integer-sqrt, 19
integer-sqrt, 77
integer-sqrt, 142
integer?, 77
integer?, 143
integer?, 47
integer?, 20
integer?, 108
Intermediate Student, 65
Intermediate Student with Lambda, 97
lambda, 136
lambda, 104
lambda, 11
lambda, 72
lambda, 63
lambda, 104
lambda, 136
lcm, 143
lcm, 20
lcm, 77
lcm, 47
lcm, 108
length, 115
length, 54
length, 149
length, 26
length, 84
let, 138
let, 126
let, 138
let, 73
let*, 73
let*, 126
let*, 163
letrec, 126
letrec, 163
letrec, 73
letrec, let, and let*, 73
list, 115
list, 149
list, 26
list, 84
list, 54
list*, 84
list*, 27
list*, 54
list*, 115
list->string, 154
list->string, 119
list->string, 31
list->string, 89
list->string, 58
list-ref, 54
list-ref, 149
list-ref, 115
list-ref, 27
list-ref, 84
list?, 150
local, 73
local, 163
local, 126

local, 73
log, 78
log, 108
log, 47
log, 143
log, 20
magnitude, 78
magnitude, 143
magnitude, 47
magnitude, 20
magnitude, 108
make-polar, 20
make-polar, 143
make-polar, 47
make-polar, 108
make-polar, 78
make-posn, 151
make-posn, 55
make-posn, 28
make-posn, 86
make-posn, 117
make-rectangular, 47
make-rectangular, 78
make-rectangular, 20
make-rectangular, 143
make-rectangular, 109
make-string, 119
make-string, 58
make-string, 31
make-string, 154
make-string, 89
make-vector, 162
map, 125
map, 160
map, 95
max, 78
max, 143
max, 109
max, 47
max, 20
member, 150
member, 85
member, 27
member, 115
member, 54
memf, 126
memf, 95
memf, 160
memq, 27
memq, 85
memq, 54
memq, 116
memq, 150
memv, 85
memv, 27
memv, 54
memv, 150
memv, 116
min, 47
min, 20
min, 109
min, 78
min, 143
modulo, 143
modulo, 78
modulo, 20
modulo, 48
modulo, 109
negative?, 143
negative?, 78
negative?, 48
negative?, 20
negative?, 109
newline, 161
not, 23
not, 111
not, 50
not, 146
not, 81
null, 54
null, 27
null, 150
null, 85
null, 116

null?, 55
 null?, 116
 null?, 85
 null?, 27
 null?, 150
 number->string, 144
 number->string, 78
 number->string, 21
 number->string, 48
 number->string, 109
 number?, 78
 number?, 109
 number?, 21
 number?, 144
 number?, 48
 numerator, 21
 numerator, 79
 numerator, 109
 numerator, 144
 numerator, 48
 odd?, 21
 odd?, 144
 odd?, 48
 odd?, 79
 odd?, 109
 or, 13
 or, 63
 or, 14
 or, 163
 or, 96
 or, 127
 ormap, 161
 ormap, 126
 ormap, 95
 pair?, 150
 pair?, 27
 pair?, 85
 pair?, 116
 pair?, 55
 pi, 109
 pi, 144
 pi, 21
 pi, 48
 pi, 79
 positive?, 21
 positive?, 48
 positive?, 144
 positive?, 110
 positive?, 79
 posn-x, 117
 posn-x, 55
 posn-x, 151
 posn-x, 86
 posn-x, 28
 posn-y, 56
 posn-y, 117
 posn-y, 28
 posn-y, 86
 posn-y, 151
 posn?, 56
 posn?, 117
 posn?, 151
 posn?, 86
 posn?, 28
 pretty-print, 161
 Primitive Calls, 12
 Primitive Operation Names, 105
 Primitive Operations, 139
 Primitive Operations, 43
 Primitive Operations, 16
 Primitive Operations, 74
 print, 161
 printf, 161
 procedure?, 126
 procedure?, 161
 procedure?, 95
 promise?, 158
 Quasiquote, 42
 quasiquote, 42
 quicksort, 126
 quicksort, 95
 quicksort, 161
 Quote, 42
 quote, 42

quote, 15
quotient, 110
quotient, 144
quotient, 48
quotient, 21
quotient, 79
random, 21
random, 79
random, 144
random, 110
random, 49
rational?, 49
rational?, 144
rational?, 110
rational?, 79
rational?, 21
read, 162
real-part, 49
real-part, 110
real-part, 21
real-part, 144
real-part, 79
real?, 22
real?, 110
real?, 79
real?, 145
real?, 49
recur, 138
recur, 138
remainder, 145
remainder, 79
remainder, 110
remainder, 22
remainder, 49
replicate, 31
replicate, 120
replicate, 89
replicate, 58
replicate, 154
require, 15
require, 96
require, 15
require, 127
require, 63
require, 164
rest, 150
rest, 27
rest, 55
rest, 85
rest, 116
reverse, 150
reverse, 85
reverse, 55
reverse, 27
reverse, 116
round, 145
round, 80
round, 110
round, 22
round, 49
second, 55
second, 85
second, 116
second, 150
second, 28
set!, 137
set!, 137
set-box!, 163
set-posn-x!, 151
set-posn-y!, 151
seventh, 85
seventh, 28
seventh, 55
seventh, 151
seventh, 116
sgn, 22
sgn, 80
sgn, 49
sgn, 145
sgn, 110
shared, 138
shared, 138
sin, 49
sin, 110

sin, 22
 sin, 145
 sin, 80
 sinh, 145
 sinh, 111
 sinh, 49
 sinh, 22
 sinh, 80
 sixth, 151
 sixth, 28
 sixth, 116
 sixth, 86
 sixth, 55
 sort, 95
 sort, 161
 sort, 126
 sqr, 22
 sqr, 111
 sqr, 80
 sqr, 49
 sqr, 145
 sqrt, 22
 sqrt, 111
 sqrt, 145
 sqrt, 80
 sqrt, 50
 string, 89
 string, 154
 string, 120
 string, 58
 string, 31
 string->int, 31
 string->int, 58
 string->int, 120
 string->int, 154
 string->int, 89
 string->list, 89
 string->list, 154
 string->list, 59
 string->list, 120
 string->list, 31
 string->number, 120
 string->number, 31
 string->number, 59
 string->number, 89
 string->number, 154
 string->symbol, 89
 string->symbol, 154
 string->symbol, 120
 string->symbol, 59
 string->symbol, 31
 string-alphabetic?, 155
 string-alphabetic?, 120
 string-alphabetic?, 89
 string-alphabetic?, 32
 string-alphabetic?, 59
 string-append, 155
 string-append, 59
 string-append, 32
 string-append, 89
 string-append, 120
 string-ci<=?, 155
 string-ci<=?, 32
 string-ci<=?, 59
 string-ci<=?, 120
 string-ci<=?, 90
 string-ci<?, 90
 string-ci<?, 32
 string-ci<?, 120
 string-ci<?, 59
 string-ci<?, 155
 string-ci=?, 155
 string-ci=?, 121
 string-ci=?, 90
 string-ci=?, 59
 string-ci=?, 32
 string-ci>=?, 155
 string-ci>=?, 59
 string-ci>=?, 121
 string-ci>=?, 90
 string-ci>=?, 32
 string-ci>?, 121
 string-ci>?, 32
 string-ci>?, 90

string-ci>?, 60	string<=?, 156
string-ci>?, 155	string<=?, 122
string-copy, 155	string<=?, 33
string-copy, 32	string<=?, 60
string-copy, 121	string<=?, 91
string-copy, 60	string<?, 156
string-copy, 90	string<?, 33
string-ith, 90	string<?, 91
string-ith, 155	string<?, 61
string-ith, 60	string<?, 122
string-ith, 32	string=?, 91
string-ith, 121	string=?, 61
string-length, 32	string=?, 33
string-length, 156	string=?, 122
string-length, 121	string=?, 156
string-length, 60	string>=?, 61
string-length, 90	string>=?, 156
string-lower-case?, 90	string>=?, 91
string-lower-case?, 60	string>=?, 122
string-lower-case?, 121	string>=?, 33
string-lower-case?, 33	string>?, 157
string-lower-case?, 156	string>?, 122
string-numeric?, 90	string>?, 61
string-numeric?, 60	string>?, 33
string-numeric?, 33	string>?, 91
string-numeric?, 156	string?, 61
string-numeric?, 121	string?, 157
string-ref, 33	string?, 122
string-ref, 91	string?, 91
string-ref, 60	string?, 34
string-ref, 156	struct?, 124
string-ref, 121	struct?, 62
string-upper-case?, 33	struct?, 93
string-upper-case?, 156	struct?, 158
string-upper-case?, 91	struct?, 35
string-upper-case?, 60	sub1, 111
string-upper-case?, 122	sub1, 22
string-whitespace?, 91	sub1, 145
string-whitespace?, 33	sub1, 50
string-whitespace?, 122	sub1, 80
string-whitespace?, 60	substring, 61
string-whitespace?, 156	substring, 34

`substring`, 122
`substring`, 157
`substring`, 91
`symbol->string`, 50
`symbol->string`, 112
`symbol->string`, 146
`symbol->string`, 81
`symbol->string`, 23
`symbol=?`, 23
`symbol=?`, 112
`symbol=?`, 81
`symbol=?`, 146
`symbol=?`, 50
`symbol?`, 51
`symbol?`, 23
`symbol?`, 81
`symbol?`, 112
`symbol?`, 146
Symbols, 15
`tan`, 111
`tan`, 145
`tan`, 22
`tan`, 80
`tan`, 50
Test Cases, 14
`third`, 86
`third`, 28
`third`, 117
`third`, 151
`third`, 55
`time`, 74
`time`, 163
`time`, 127
`time`, 74
`true`, 96
`true`, 164
`true`, 15
`true`, 63
`true`, 127
`true` and `false`, 15
`unbox`, 163
Unchanged Forms, 63
Unchanged Forms, 163
Unchanged Forms, 126
Unchanged Forms, 96
`unless`, 139
`unquote`, 43
`unquote-splicing`, 43
`vector`, 162
`vector-length`, 162
`vector-ref`, 162
`vector-set!`, 162
`vector?`, 162
`void`, 158
`void?`, 158
`when`, 139
`when` and `unless`, 139
`write`, 162
`zero?`, 23
`zero?`, 111
`zero?`, 80
`zero?`, 146
`zero?`, 50
 λ , 104
 λ , 136