

How to Design Programs Languages

Version 4.2.2

October 4, 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	13
1.6	if	13
1.7	and	13
1.8	or	14
1.9	Test Cases	14
1.10	empty	14
1.11	Identifiers	15
1.12	Symbols	15
1.13	true and false	15
1.14	require	15
1.15	Primitive Operations	16
2	Beginning Student with List Abbreviations	37
2.1	Quote	43
2.2	Quasiquote	43
2.3	Primitive Operations	44
2.4	Unchanged Forms	64
3	Intermediate Student	66
3.1	define	73

3.2	define-struct	73
3.3	local	74
3.4	letrec, let, and let*	74
3.5	Function Calls	74
3.6	time	75
3.7	Identifiers	75
3.8	Primitive Operations	75
3.9	Unchanged Forms	97
4	Intermediate Student with Lambda	98
4.1	define	105
4.2	lambda	105
4.3	Function Calls	105
4.4	Primitive Operation Names	106
4.5	Unchanged Forms	127
5	Advanced Student	129
5.1	define	137
5.2	define-struct	137
5.3	lambda	137
5.4	Function Calls	138
5.5	begin	138
5.6	begin0	138
5.7	set!	138
5.8	delay	138
5.9	shared	139

5.10 let	139
5.11 recur	139
5.12 case	140
5.13 when and unless	140
5.14 Primitive Operations	140
5.15 Unchanged Forms	164
Index	166

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

```



```

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

```

Posns

```

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

```

Characters

```

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

```

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

Strings

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

Images

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

Misc

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

```
eof-object? : (any -> boolean)
eq? : (any any -> boolean)
equal? : (any any -> boolean)
equal~? : (any any non-negative-real -> boolean)
eqv? : (any any -> boolean)
error : (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 text `expr` evaluates to `false`, the result of the `if` expression is the result of evaluating the third `expr` (known as the “else” `expr`). If the result of evaluating the test `expr` is neither `true` nor `false`, it is an error.

1.7 and

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

The `exprs` are evaluated from left to right. If the first `expr` evaluates to `false`, the `and`

expression immediately evaluates to `false`. If the first `expr` evaluates to `true`, the next expression is considered. If all `exprs` evaluate to `true`, the `and` expression evaluates to `true`. If any of the expressions evaluate to a value other than `true` or `false`, it is an error.

1.8 or

`(or expr expr expr ...)`

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

1.9 Test Cases

`(check-expect expr expr)`

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

`(check-within expr expr expr)`

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

`(check-error expr expr)`

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

1.10 empty

`empty : empty?`

The empty list.

1.11 Identifiers

`id`

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

1.12 Symbols

`'id`
`(quote id)`

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

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

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

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

1.13 true and false

`true` : `boolean?`

The true value.

`false` : `boolean?`

The false value.

1.14 require

`(require string)`

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

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

```
(require module-id)
```

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

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

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

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

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

1.15 Primitive Operations

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

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

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

Purpose: to compute the sum of the input numbers

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

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

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


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

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

Purpose: to compare real numbers for less-than

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

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

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

Purpose: to compare numbers for equality

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

Purpose: to compare real numbers for greater-than

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

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

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

Purpose: to compute the absolute value of a real number

```
acos : (number -> number)
```

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

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

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

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

Purpose: to extract the angle from a complex number

```
asin : (number -> number)
```

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

`atan` : (number -> number)

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

`ceiling` : (real -> integer)

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

`complex?` : (any -> boolean)

Purpose: to determine whether some value is complex

`conjugate` : (number -> number)

Purpose: to compute the conjugate of a complex number

`cos` : (number -> number)

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

`cosh` : (number -> number)

Purpose: to compute the hyperbolic cosine of a number

`current-seconds` : (-> integer)

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

`denominator` : (rat -> integer)

Purpose: to compute the denominator of a rational

`e` : real

Purpose: Euler's number

`even?` : (integer -> boolean)

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

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

Purpose: to convert an exact number to an inexact one

`exact?` : (number -> boolean)

Purpose: to determine whether some number is exact

`exp` : (number -> number)

Purpose: to compute e raised to a number

`expt` : (number number -> number)

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

`floor` : (real -> integer)

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

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

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

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

Purpose: to extract the imaginary part from a complex number

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

Purpose: to approximate an inexact number by an exact one

`inexact?` : (number -> boolean)

Purpose: to determine whether some number is inexact

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

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

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

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

`integer?` : (any -> boolean)

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

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

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

`log` : (number -> number)

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

`magnitude` : (number -> real)

Purpose: to determine the magnitude of a complex number

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

Purpose: to create a complex from a magnitude and angle

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

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

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

Purpose: to determine the largest number

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

Purpose: to determine the smallest number

`modulo` : (integer integer -> integer)

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

`negative?` : (number -> boolean)

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

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

Purpose: to convert a number to a string

`number?` : (any -> boolean)

Purpose: to determine whether some value is a number

`numerator` : (rat -> integer)

Purpose: to compute the numerator of a rational

`odd?` : (integer -> boolean)

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

`pi` : real

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

`positive?` : (number -> boolean)

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

`quotient` : (integer integer -> integer)

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

`random` : (integer -> integer)

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

`rational?` : (any -> boolean)

Purpose: to determine whether some value is a rational number

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

Purpose: to extract the real part from a complex number

`real?` : (any -> boolean)

Purpose: to determine whether some value is a real number

`remainder` : (integer integer -> integer)

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

`round` : (real -> integer)

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

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

Purpose: to compute the sign of a real number

`sin` : (number -> number)

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

`sinh` : (number -> number)

Purpose: to compute the hyperbolic sine of a number

`sqr` : (number -> number)

Purpose: to compute the square of a number

`sqrt` : (number -> number)

Purpose: to compute the square root of a number

`sub1` : (number -> number)

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

`tan` : (number -> number)

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

`zero?` : (number -> boolean)

Purpose: to determine if some value is zero or not

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

Purpose: to determine whether two booleans are equal

`boolean?` : (any -> boolean)

Purpose: to determine whether some value is a boolean

`false?` : (any -> boolean)

Purpose: to determine whether a value is false

`not` : (boolean -> boolean)

Purpose: to compute the negation of a boolean value

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

Purpose: to convert a symbol to a string

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

Purpose: to determine whether two symbols are equal

`symbol?` : (any -> boolean)

Purpose: to determine whether some value is a symbol

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

Purpose: to construct a list

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

Purpose: to determine whether some value is a constructed list

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

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

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

Purpose: to determine whether some value is the empty list

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

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

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

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

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

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

```
length : ((listof any) -> number)
```

Purpose: to compute the number of items on a list

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

Purpose: to construct a list of its arguments

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

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

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

Purpose: to extract the indexed item from the list

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

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

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

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

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

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

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

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

```
null : empty
```

Purpose: the empty list

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

Purpose: to determine whether some value is the empty list

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

Purpose: to determine whether some value is a constructed list

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

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

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

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

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

Purpose: to create a reversed version of a list

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

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

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

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

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

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

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

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

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

Purpose: to construct a posn

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

Purpose: to extract the x component of a posn

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

Purpose: to extract the y component of a posn

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

Purpose: to determine if its input is a posn

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

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

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

Purpose: to determine whether a character represents an alphabetic character

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

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

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

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

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

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

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

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

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

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

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

Purpose: to determine the equivalent lower-case character

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

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

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

Purpose: to determine whether a character represents a digit

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

Purpose: to determine the equivalent upper-case character

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

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

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

Purpose: to determine whether a character represents space

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

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

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

Purpose: to determine whether a character precedes another

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

Purpose: to determine whether two characters are equal

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

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

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

Purpose: to determine whether a character succeeds another

`char?` : (any -> boolean)

Purpose: to determine whether a value is a character

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

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

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

Purpose: to format a string, possibly embedding values

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

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

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

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

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

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

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

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

`replicate` : (nat string -> string)

Purpose: to replicate the given string

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

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

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

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

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

Purpose: to convert a string into a list of characters

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

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

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

Purpose: to convert a string into a symbol

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

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

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

Purpose: to juxtapose the characters of several strings

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

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

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

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

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

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

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

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

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

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

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

Purpose: to copy a string

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

Purpose: to extract the *i*th 1-letter substring from the given one

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

Purpose: to determine the length of a string

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

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

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

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

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

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

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

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

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

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

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

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

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

Purpose: to determine whether one string alphabetically precedes another

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

Purpose: to compare two strings character-wise

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

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

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

Purpose: to determine whether one string alphabetically succeeds another

`string?` : (any -> boolean)

Purpose: to determine whether a value is a string

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

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

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

Purpose: to determine whether two images are equal

`image?` : (any -> boolean)

Purpose: to determine whether a value is an image

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

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

of either other

`eof` : eof

Purpose: the end-of-file value

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

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

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

Purpose: to compare two values

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

Purpose: to determine whether two values are structurally equal

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

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

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

Purpose: to compare two values

`error` : (symbol string -> void)

Purpose: to signal an error

`exit` : (-> void)

Purpose: to exit the running program

`identity` : (any -> any)

Purpose: to return the argument unchanged

`struct?` : (any -> boolean)

Purpose: to determine whether some value is a structure

2 Beginning Student with List Abbreviations

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

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

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

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

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

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

```

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

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

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

package = (string string number number)

```

An *id* is a sequence of characters not including a space or one of the following:

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

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

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

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

A *prim-op* is one of:

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

```

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

```

```

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

```

```
sqrt : (number -> number)
sub1 : (number -> number)
tan : (number -> number)
zero? : (number -> boolean)
```

Booleans

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

Symbols

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

Lists

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

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

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

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

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

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



```

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

```

Posns

```

make-posn : (number number -> posn)

```

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

Characters

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

Strings

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

```
string-lower-case? : (string -> boolean)
string-numeric? : (string -> boolean)
string-ref : (string nat -> char)
string-upper-case? : (string -> boolean)
string-whitespace? : (string -> boolean)
string<=? : (string string string ... -> boolean)
string<? : (string string string ... -> boolean)
string=? : (string string string ... -> boolean)
string>=? : (string string string ... -> boolean)
string>? : (string string string ... -> boolean)
string? : (any -> boolean)
substring : (string nat nat -> string)
```

Images

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

Misc

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

2.1 Quote

```
'quoted

```

Creates symbols and abbreviates nested lists.

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

2.2 Quasiquote

```
`quasiquoted
```

```
(quasiquote quasiquoted)
```

Creates symbols and abbreviates nested lists, but also allows escaping to expression “un-quotes.”

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

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

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

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

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

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

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

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

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

`null` : empty

Purpose: the empty list

`null?` : (any -> boolean)

Purpose: to determine whether some value is the empty list

`pair?` : (any -> boolean)

Purpose: to determine whether some value is a constructed list

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

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

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

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

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

Purpose: to create a reversed version of a list

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

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

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

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

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

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

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

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

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

Purpose: to construct a posn

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

Purpose: to extract the x component of a posn

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

Purpose: to extract the y component of a posn

`posn?` : (anything -> boolean)

Purpose: to determine if its input is a posn

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

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

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

Purpose: to determine whether a character represents an alphabetic character

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

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

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

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

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

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

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

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

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

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

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

Purpose: to determine the equivalent lower-case character

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

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

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

Purpose: to determine whether a character represents a digit

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

Purpose: to determine the equivalent upper-case character

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

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

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

Purpose: to determine whether a character represents space

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

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

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

Purpose: to determine whether a character precedes another

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

Purpose: to determine whether two characters are equal

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

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

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

Purpose: to determine whether a character succeeds another

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

Purpose: to determine whether a value is a character

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

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

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

Purpose: to format a string, possibly embedding values

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

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

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

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

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

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

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

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

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

Purpose: to replicate the given string

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

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

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

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

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

Purpose: to convert a string into a list of characters

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

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

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

Purpose: to convert a string into a symbol

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

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

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

Purpose: to juxtapose the characters of several strings

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

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

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

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

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

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

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

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

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

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

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

Purpose: to copy a string

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

Purpose: to extract the *i*th 1-letter substring from the given one

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

Purpose: to determine the length of a string

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

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

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

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

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

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

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

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

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

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

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

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

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

Purpose: to determine whether one string alphabetically precedes another

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

Purpose: to compare two strings character-wise

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

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

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

Purpose: to determine whether one string alphabetically succeeds another

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

Purpose: to determine whether a value is a string

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

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

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

Purpose: to determine whether two images are equal

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

Purpose: to determine whether a value is an image

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

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

```
eof : eof
```

Purpose: the end-of-file value

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

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

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

Purpose: to compare two values

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

Purpose: to determine whether two values are structurally equal

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

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

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

Purpose: to compare two values

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

Purpose: to signal an error

```
exit : (-> void)
```

Purpose: to exit the running program

`identity` : (any -> any)

Purpose: to return the argument unchanged

`struct?` : (any -> boolean)

Purpose: to determine whether some value is a structure

2.4 Unchanged Forms

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

The same as Beginning's `define`.

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

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

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

The same as Beginning's `cond`.

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

The same as Beginning's `if`.

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

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

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


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

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

Constants for the empty list, true, and false.

```
(require module-path)
```

The same as Beginning's require.

3 Intermediate Student

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

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

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

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

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

quoted = id
        | number
        | string
        | character
        | (quoted ...)
```

```

| 'quoted
| 'quoted
| ,quoted
| ,@quoted

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

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

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

package = (string string number number)

```

An *id* is a sequence of characters not including a space or one of the following:

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

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

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

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

A *prim-op* is one of:

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

```

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

```

```

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

```

```

round : (real -> integer)
sgn : (real -> (union 1 1.0 0 0.0 -1 -1.0))
sin : (number -> number)
sinh : (number -> number)
sqr : (number -> number)
sqrt : (number -> number)
sub1 : (number -> number)
tan : (number -> number)
zero? : (number -> boolean)

```

Booleans

```

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

```

Symbols

```

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

```

Lists

```

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

```

```

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

```

Posns

```

make-posn : (number number -> posn)

```

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

Characters

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

Strings

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

```

string-lower-case? : (string -> boolean)
string-numeric? : (string -> boolean)
string-ref : (string nat -> char)
string-upper-case? : (string -> boolean)
string-whitespace? : (string -> boolean)
string<=? : (string string string ... -> boolean)
string<? : (string string string ... -> boolean)
string=? : (string string string ... -> boolean)
string>=? : (string string string ... -> boolean)
string>? : (string string string ... -> boolean)
string? : (any -> boolean)
substring : (string nat nat -> string)

```

Images

```

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

```

Misc

```

=~ : (real real non-negative-real -> boolean)
eof : eof
eof-object? : (any -> boolean)
eq? : (any any -> boolean)
equal? : (any any -> boolean)
equal~? : (any any non-negative-real -> boolean)
eqv? : (any any -> boolean)
error : (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))
```

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

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

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

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

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

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

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

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

Purpose: to construct a list

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

Purpose: to determine whether some value is a constructed list

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

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

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

Purpose: to determine whether some value is the empty list

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

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

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

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

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

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

```
length : ((listof any) -> number)
```

Purpose: to compute the number of items on a list

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

Purpose: to construct a list of its arguments

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

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

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

Purpose: to extract the indexed item from the list

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

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

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

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

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

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

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

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

```
null : empty
```

Purpose: the empty list

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

Purpose: to determine whether some value is the empty list

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

Purpose: to determine whether some value is a constructed list

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

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

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

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

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

Purpose: to create a reversed version of a list

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

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

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

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

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

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

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

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

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

Purpose: to construct a posn

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

Purpose: to extract the x component of a posn

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

Purpose: to extract the y component of a posn

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

Purpose: to determine if its input is a posn

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

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

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

Purpose: to determine whether a character represents an alphabetic character

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

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

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

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

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

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

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

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

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

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

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

Purpose: to determine the equivalent lower-case character

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

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

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

Purpose: to determine whether a character represents a digit

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

Purpose: to determine the equivalent upper-case character

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

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

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

Purpose: to determine whether a character represents space

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

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

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

Purpose: to determine whether a character precedes another

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

Purpose: to determine whether two characters are equal

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

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

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

Purpose: to determine whether a character succeeds another

`char?` : (any -> boolean)

Purpose: to determine whether a value is a character

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

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

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

Purpose: to format a string, possibly embedding values

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

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

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

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

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

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

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

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

`replicate` : (nat string -> string)

Purpose: to replicate the given string

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

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

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

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

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

Purpose: to convert a string into a list of characters

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

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

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

Purpose: to convert a string into a symbol

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

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

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

Purpose: to juxtapose the characters of several strings

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

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

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

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

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

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

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

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

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

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

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

Purpose: to copy a string

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

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

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

Purpose: to determine the length of a string

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

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

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

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

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

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

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

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

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

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

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

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

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

Purpose: to determine whether one string alphabetically precedes another

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

Purpose: to compare two strings character-wise

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

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

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

Purpose: to determine whether one string alphabetically succeeds another

`string? : (any -> boolean)`

Purpose: to determine whether a value is a string

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

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

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

Purpose: to determine whether two images are equal

`image? : (any -> boolean)`

Purpose: to determine whether a value is an image

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

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

`eof : eof`

Purpose: the end-of-file value

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

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

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

Purpose: to compare two values

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

Purpose: to determine whether two values are structurally equal

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

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

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

Purpose: to compare two values

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

Purpose: to signal an error

```
exit : (-> void)
```

Purpose: to exit the running program

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

Purpose: to return the argument unchanged

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

Purpose: to determine whether some value is a structure

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

Purpose: to multiply all given numbers

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

Purpose: to add all given numbers

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

Purpose: to subtract from the first all remaining numbers

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

Purpose: to divide the first by all remaining numbers

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

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

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

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

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

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

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

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

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

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

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

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

```
compose : ((Y-1 -> Z)
           ...
           (Y-N -> Y-N-1)
           (X-1 ... X-N -> Y-N)
           ->
           (X-1 ... X-N -> Z))
```

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

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

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

`foldl` : ((X Y -> Y) Y (listof X) -> Y)

Purpose: (foldl f base (list x-1 ... x-n)) = (f x-n ... (f x-1 base))

`foldr` : ((X Y -> Y) Y (listof X) -> Y)

Purpose: (foldr f base (list x-1 ... x-n)) = (f x-1 ... (f x-n base))

`for-each` : ((any ... -> any) (listof any) ... -> void)

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

`map` : ((X ... -> Z) (listof X) ... -> (listof Z))

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

`memf` : ((X -> boolean)
 (listof X)
 ->
 (union false (listof X)))

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

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

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

`procedure?` : (any -> boolean)

Purpose: to determine if a value is a procedure

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

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

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

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

3.9 Unchanged Forms

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

The same as Beginning's cond.

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

The same as Beginning's if.

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

The same as Beginning's and and or.

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

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

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

Constants for the empty list, true, and false.

```
(require module-path)
```

The same as Beginning's require.

4 Intermediate Student with Lambda

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

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

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

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

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

```

| ,@quoted

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

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

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

package = (string string number number)

```

An *id* is a sequence of characters not including a space or one of the following:

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

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

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

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

A *prim-op* is one of:

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

```

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

```

```

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

```

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

Booleans

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

Symbols

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

Lists

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

```

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

```

Posns

```

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

```

Characters

```

char->integer : (char -> integer)

```

```

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

```

Strings

```

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

```

```

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

```

Images

```

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

```

Misc

```

≈~ : (real real non-negative-real -> boolean)
eof : eof
eof-object? : (any -> boolean)
eq? : (any any -> boolean)
equal? : (any any -> boolean)
equal~? : (any any non-negative-real -> boolean)
eqv? : (any any -> boolean)
error : (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))
```

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

Purpose: to construct a list

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

Purpose: to determine whether some value is a constructed list

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

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

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

Purpose: to determine whether some value is the empty list

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

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

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

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

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

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

`length` : ((listof any) -> number)

Purpose: to compute the number of items on a list

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

Purpose: to construct a list of its arguments

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

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

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

Purpose: to extract the indexed item from the list

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

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

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

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

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

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

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

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

`null` : empty

Purpose: the empty list

`null?` : (any -> boolean)

Purpose: to determine whether some value is the empty list

`pair?` : (any -> boolean)

Purpose: to determine whether some value is a constructed list

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

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

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

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

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

Purpose: to create a reversed version of a list

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

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

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

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

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

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

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

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

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

Purpose: to construct a posn

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

Purpose: to extract the x component of a posn

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

Purpose: to extract the y component of a posn

`posn?` : (anything -> boolean)

Purpose: to determine if its input is a posn

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

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

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

Purpose: to determine whether a character represents an alphabetic character

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

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

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

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

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

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

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

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

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

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

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

Purpose: to determine the equivalent lower-case character

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

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

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

Purpose: to determine whether a character represents a digit

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

Purpose: to determine the equivalent upper-case character

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

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

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

Purpose: to determine whether a character represents space

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

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

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

Purpose: to determine whether a character precedes another

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

Purpose: to determine whether two characters are equal

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

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

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

Purpose: to determine whether a character succeeds another

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

Purpose: to determine whether a value is a character

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

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

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

Purpose: to format a string, possibly embedding values

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

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

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

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

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

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

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

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

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


Purpose: to replicate the given string

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

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

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

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

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

Purpose: to convert a string into a list of characters

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

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

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

Purpose: to convert a string into a symbol

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

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

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

Purpose: to juxtapose the characters of several strings

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

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

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

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

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

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

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

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

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

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

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

Purpose: to copy a string

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

Purpose: to extract the *i*th 1-letter substring from the given one

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

Purpose: to determine the length of a string

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

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

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

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

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

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

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

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

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

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

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

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

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

Purpose: to determine whether one string alphabetically precedes another

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

Purpose: to compare two strings character-wise

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

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

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

Purpose: to determine whether one string alphabetically succeeds another

`string? : (any -> boolean)`

Purpose: to determine whether a value is a string

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

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

`image=? : (image image -> boolean)`

Purpose: to determine whether two images are equal

`image? : (any -> boolean)`

Purpose: to determine whether a value is an image

```
=~ : (real real non-negative-real -> boolean)
```

Purpose: to check whether two real numbers are within some amount (the third argument) of either other

```
eof : eof
```

Purpose: the end-of-file value

```
eof-object? : (any -> boolean)
```

Purpose: to determine whether some value is the end-of-file value

```
eq? : (any any -> boolean)
```

Purpose: to compare two values

```
equal? : (any any -> boolean)
```

Purpose: to determine whether two values are structurally equal

```
equal~? : (any any non-negative-real -> boolean)
```

Purpose: to compare like equal? on the first two arguments, except using =~ in the case of real numbers

```
eqv? : (any any -> boolean)
```

Purpose: to compare two values

```
error : (symbol string -> void)
```

Purpose: to signal an error

```
exit : (-> void)
```

Purpose: to exit the running program

`identity` : (any -> any)

Purpose: to return the argument unchanged

`struct?` : (any -> boolean)

Purpose: to determine whether some value is a structure

`*` : (number ... -> number)

Purpose: to multiply all given numbers

`+` : (number ... -> number)

Purpose: to add all given numbers

`-` : (number ... -> number)

Purpose: to subtract from the first all remaining numbers

`/` : (number ... -> number)

Purpose: to divide the first by all remaining numbers

`andmap` : ((X -> boolean) (listof X) -> boolean)

Purpose: (andmap p (list x-1 ... x-n)) = (and (p x-1) ... (p x-n))

`apply` : ((X-1 ... X-N -> Y)
X-1
...
X-i
(list X-i+1 ... X-N)
->
Y)

Purpose: to apply a function using items from a list as the arguments

`argmax` : ((X -> real) (listof X) -> X)

Purpose: to find the (first) element of the list that maximizes the output of the function

```
argmin : ((X -> real) (listof X) -> X)
```

Purpose: to find the (first) element of the list that minimizes the output of the function

```
build-list : (nat (nat -> X) -> (listof X))
```

Purpose: (build-list n f) = (list (f 0) ... (f (- n 1)))

```
build-string : (nat (nat -> char) -> string)
```

Purpose: (build-string n f) = (string (f 0) ... (f (- n 1)))

```
compose : ((Y-1 -> Z)
           ...
           (Y-N -> Y-N-1)
           (X-1 ... X-N -> Y-N)
           ->
           (X-1 ... X-N -> Z))
```

Purpose: to compose a sequence of procedures into a single procedure

```
filter : ((X -> boolean) (listof X) -> (listof X))
```

Purpose: to construct a list from all those items on a list for which the predicate holds

```
foldl : ((X Y -> Y) Y (listof X) -> Y)
```

Purpose: (foldl f base (list x-1 ... x-n)) = (f x-n ... (f x-1 base))

```
foldr : ((X Y -> Y) Y (listof X) -> Y)
```

Purpose: (foldr f base (list x-1 ... x-n)) = (f x-1 ... (f x-n base))

```
for-each : ((any ... -> any) (listof any) ... -> void)
```

Purpose: to apply a function to each item on one or more lists for effect only

```
map : ((X ... -> Z) (listof X) ... -> (listof Z))
```

Purpose: to construct a new list by applying a function to each item on one or more existing lists

```
memf : ((X -> boolean)
        (listof X)
        ->
        (union false (listof X)))
```

Purpose: to determine whether the first argument produces true for some value in the second argument

```
ormap : ((X -> boolean) (listof X) -> boolean)
```

Purpose: (ormap p (list x-1 ... x-n)) = (or (p x-1) ... (p x-n))

```
procedure? : (any -> boolean)
```

Purpose: to determine if a value is a procedure

```
quicksort : ((listof X) (X X -> boolean) -> (listof X))
```

Purpose: to construct a list from all items on a list in an order according to a predicate

```
sort : ((listof X) (X X -> boolean) -> (listof X))
```

Purpose: to construct a list from all items on a list in an order according to a predicate

4.5 Unchanged Forms

```
(define-struct structid (fieldid ...))
```

The same as Intermediate's `define-struct`.

```
(local [definition ...] expr)
(letrec ([id expr-for-let] ...) expr)
(let ([id expr-for-let] ...) expr)
(let* ([id expr-for-let] ...) expr)
```

The same as Intermediate's `local`, `letrec`, `let`, and `let*`.

```
(cond [expr expr] ... [expr expr])  
else
```

The same as Beginning's cond.

```
(if expr expr expr)
```

The same as Beginning's if.

```
(and expr expr expr ...)  
(or expr expr expr ...)
```

The same as Beginning's and and or.

```
(time expr)
```

The same as Intermediate's time.

```
(check-expect expr expr)  
(check-within expr expr expr)  
(check-error expr expr)
```

The same as Beginning's check-expect, etc.

```
empty : empty?  
true  : boolean?  
false : boolean?
```

Constants for the empty list, true, and false.

```
(require module-path)
```

The same as Beginning's require.

5 Advanced Student

```
program = def-or-expr ...

def-or-expr = definition
            | expr
            | test-case
            | library-require

definition = (define (id id id ...) expr)
            | (define id expr)
            | (define-struct id (id ...))

expr = (begin expr expr ...)
      | (begin0 expr expr ...)
      | (set! id expr)
      | (delay expr)
      | (lambda (id ...) expr)
      | (λ (id ...) expr)
      | (local [definition ...] expr)
      | (letrec ([id expr] ...) expr)
      | (shared ([id expr] ...) expr)
      | (let ([id expr] ...) expr)
      | (let id ([id expr] ...) expr)
      | (let* ([id expr] ...) expr)
      | (recur id ([id expr] ...) expr)
      | (expr expr ...) ; function call
      | (cond [expr expr] ... [expr expr])
      | (cond [expr expr] ... [else expr])
      | (case expr [(choice choice ...) expr] ...
          [(choice choice ...) expr])
      | (case expr [(choice choice ...) expr] ...
          [else expr])
      | (if expr expr expr)
      | (when expr expr)
      | (unless expr expr)
      | (and expr expr expr ...)
      | (or expr expr expr ...)
      | (time expr)
      | empty
      | id ; identifier
      | prim-op ; primitive operation
      | 'quoted ; quoted value
      | 'quasiquoted ; quasiquote
      | number
```

```

| true
| false
| string
| character

choice = id ; treated as a symbol
| number

quoted = id
| number
| string
| character
| (quoted ...)
| 'quoted
| 'quoted
| ,quoted
| ,@quoted

quasiquoted = id
| number
| string
| character
| (quasiquoted ...)
| 'quasiquoted
| 'quasiquoted
| ,expr
| ,@expr

test-case = (check-expect expr expr)
| (check-within expr expr expr)
| (check-error expr expr)

library-require = (require string)
| (require (lib string string ...))
| (require (planet string package))

package = (string string number number)

```

An *id* is a sequence of characters not including a space or one of the following:

```
" , ' ( ) [ ] { } | ; #
```

A *number* is a number such as 123, 3/2, or 5.5.

A *string* is enclosed by a pair of `"`. Unlike symbols, strings may be split into characters and manipulated by a variety of primitive functions. For example, "abcdef", "This is a

string", and "This is a string with \" inside" are all strings.

A *character* begins with #\ and has the name of the character. For example, #\a, #\b, and #\space are characters.

A *prim-op* is one of:

Numbers: Integers, Rationals, Reals, Complex, Exacts, Inexacts

```
< : (real real real ... -> boolean)
<= : (real real real ... -> boolean)
= : (number number number ... -> boolean)
> : (real real real ... -> boolean)
>= : (real real ... -> boolean)
abs : (real -> real)
acos : (number -> number)
add1 : (number -> number)
angle : (number -> real)
asin : (number -> number)
atan : (number -> number)
ceiling : (real -> integer)
complex? : (any -> boolean)
conjugate : (number -> number)
cos : (number -> number)
cosh : (number -> number)
current-seconds : (-> integer)
denominator : (rat -> integer)
e : real
even? : (integer -> boolean)
exact->inexact : (number -> number)
exact? : (number -> boolean)
exp : (number -> number)
expt : (number number -> number)
floor : (real -> integer)
gcd : (integer integer ... -> integer)
imag-part : (number -> real)
inexact->exact : (number -> number)
inexact? : (number -> boolean)
integer->char : (integer -> char)
integer-sqrt : (number -> integer)
integer? : (any -> boolean)
lcm : (integer integer ... -> integer)
log : (number -> number)
magnitude : (number -> real)
make-polar : (real real -> number)
make-rectangular : (real real -> number)
max : (real real ... -> real)
min : (real real ... -> real)
```

```

modulo : (integer integer -> integer)
negative? : (number -> boolean)
number->string : (number -> string)
number? : (any -> boolean)
numerator : (rat -> integer)
odd? : (integer -> boolean)
pi : real
positive? : (number -> boolean)
quotient : (integer integer -> integer)
random : (integer -> integer)
rational? : (any -> boolean)
real-part : (number -> real)
real? : (any -> boolean)
remainder : (integer integer -> integer)
round : (real -> integer)
sgn : (real -> (union 1 1.0 0 0.0 -1 -1.0))
sin : (number -> number)
sinh : (number -> number)
sqr : (number -> number)
sqrt : (number -> number)
sub1 : (number -> number)
tan : (number -> number)
zero? : (number -> boolean)

```

Booleans

```

boolean=? : (boolean boolean -> boolean)
boolean? : (any -> boolean)
false? : (any -> boolean)
not : (boolean -> boolean)

```

Symbols

```

symbol->string : (symbol -> string)
symbol=? : (symbol symbol -> boolean)
symbol? : (any -> boolean)

```

Lists

```

append : ((listof any) ... -> (listof any))
assq : (X
      (listof (cons X Y))
      ->
      (union false (cons X Y)))
caaar : ((cons
      (cons (cons W (listof Z)) (listof Y))
      (listof X))
      ->
      W)

```

```

caadr : ((cons
         (cons (cons W (listof Z)) (listof Y))
         (listof X))
        ->
        (listof Z))
caar : ((cons (cons Z (listof Y)) (listof X)) -> Z)
cadar : ((cons (cons W (cons Z (listof Y))) (listof X))
        ->
        Z)
caddr : ((listof Y) -> Y)
caddr : ((cons W (cons Z (cons Y (listof X)))) -> Y)
cadr : ((cons Z (cons Y (listof X))) -> Y)
car : ((cons Y (listof X)) -> Y)
cdaar : ((cons
         (cons (cons W (listof Z)) (listof Y))
         (listof X))
        ->
        (listof Z))
cdadr : ((cons W (cons (cons Z (listof Y)) (listof X)))
        ->
        (listof Y))
cdar : ((cons (cons Z (listof Y)) (listof X))
        ->
        (listof Y))
cddar : ((cons (cons W (cons Z (listof Y))) (listof X))
        ->
        (listof Y))
cddddr : ((cons W (cons Z (cons Y (listof X))))
        ->
        (listof X))
cddr : ((cons Z (cons Y (listof X))) -> (listof X))
cdr : ((cons Y (listof X)) -> (listof X))
cons : (X (listof X) -> (listof X))
cons? : (any -> boolean)
eighth : ((listof Y) -> Y)
empty? : (any -> boolean)
fifth : ((listof Y) -> Y)
first : ((cons Y (listof X)) -> Y)
fourth : ((listof Y) -> Y)
length : ((listof any) -> number)
list : (any ... -> (listof any))
list-ref : ((listof X) natural-number -> X)
list? : (any -> boolean)
make-list : (natural-number any -> (listof any))
member : (any (listof any) -> boolean)
memq : (any (listof any) -> (union false list))

```

```

memv : (any (listof any) -> (union false list))
null : empty
null? : (any -> boolean)
pair? : (any -> boolean)
remove : (any (listof any) -> (listof any))
rest : ((cons Y (listof X)) -> (listof X))
reverse : ((listof any) -> list)
second : ((cons Z (cons Y (listof X))) -> Y)
seventh : ((listof Y) -> Y)
sixth : ((listof Y) -> Y)
third : ((cons W (cons Z (cons Y (listof X)))) -> Y)

```

Posns

```

make-posn : (number number -> posn)
posn-x : (posn -> number)
posn-y : (posn -> number)
posn? : (anything -> boolean)
set-posn-x! : (posn number -> void)
set-posn-y! : (posn number -> void)

```

Characters

```

char->integer : (char -> integer)
char-alphabetic? : (char -> boolean)
char-ci<=? : (char char char ... -> boolean)
char-ci<? : (char char char ... -> boolean)
char-ci=? : (char char char ... -> boolean)
char-ci>=? : (char char char ... -> boolean)
char-ci>? : (char char char ... -> boolean)
char-downcase : (char -> char)
char-lower-case? : (char -> boolean)
char-numeric? : (char -> boolean)
char-upcase : (char -> char)
char-upper-case? : (char -> boolean)
char-whitespace? : (char -> boolean)
char<=? : (char char char ... -> boolean)
char<? : (char char char ... -> boolean)
char=? : (char char char ... -> boolean)
char>=? : (char char char ... -> boolean)
char>? : (char char char ... -> boolean)
char? : (any -> boolean)

```

Strings

```

explode : (string -> (listof string))
format : (string any ... -> string)
implode : ((listof string) -> string)
int->string : (integer -> string)
list->string : ((listof char) -> string)
make-string : (nat char -> string)
replicate : (nat string -> string)

```

```

string : (char ... -> string)
string->int : (string -> integer)
string->list : (string -> (listof char))
string->number : (string -> (union number false))
string->symbol : (string -> symbol)
string-alphabetic? : (string -> boolean)
string-append : (string ... -> string)
string-ci<=? : (string string string ... -> boolean)
string-ci<? : (string string string ... -> boolean)
string-ci=? : (string string string ... -> boolean)
string-ci>=? : (string string string ... -> boolean)
string-ci>? : (string string string ... -> boolean)
string-copy : (string -> string)
string-ith : (string nat -> string)
string-length : (string -> nat)
string-lower-case? : (string -> boolean)
string-numeric? : (string -> boolean)
string-ref : (string nat -> char)
string-upper-case? : (string -> boolean)
string-whitespace? : (string -> boolean)
string<=? : (string string string ... -> boolean)
string<? : (string string string ... -> boolean)
string=? : (string string string ... -> boolean)
string>=? : (string string string ... -> boolean)
string>? : (string string string ... -> boolean)
string? : (any -> boolean)
substring : (string nat nat -> string)

```

Images

```

image=? : (image image -> boolean)
image? : (any -> boolean)

```

Misc

```

=~ : (real real non-negative-real -> boolean)
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)
(λ (id ...) expr)
```

The same as Intermediate with Lambda's `lambda`, except that a function is allowed to accept zero arguments.

5.4 Function Calls

```
(expr expr ...)
```

A function call in Advanced is the same as an Intermediate with Lambda function call, except that zero arguments are allowed.

```
(#%app expr expr ...)
```

A function call can be written with `#%app`, though it's practically never written that way.

5.5 begin

```
(begin expr expr ...)
```

Evaluates the *exprs* in order from left to right. The value of the `begin` expression is the value of the last *expr*.

5.6 begin0

```
(begin0 expr expr ...)
```

Evaluates the *exprs* in order from left to right. The value of the `begin` expression is the value of the first *expr*.

5.7 set!

```
(set! id expr)
```

Evaluates *expr*, and then changes the definition *id* to have *expr*'s value. The *id* must be defined or bound by `letrec`, `let`, or `let*`.

5.8 delay

```
(delay expr)
```

Produces a “promise” to evaluate *expr*. The *expr* is not evaluated until the promise is forced through the `force` operator; when the promise is forced, the result is recorded, so that any further `force` of the promise always produces the remembered value.

5.9 shared

```
(shared ([id expr] ...) expr)
```

Like `letrec`, but when an *expr* next to an *id* is a `cons`, `list`, `vector`, quasiquoted expression, or `make-structid` from a `define-struct`, the *expr* can refer directly to any *id*, not just *ids* defined earlier. Thus, `shared` can be used to create cyclic data structures.

5.10 let

```
(let ([id expr] ...) expr)
(let id ([id expr] ...) expr)
```

The first form of `let` is the same as Intermediate’s `let`.

The second form is equivalent to a `recur` form.

5.11 recur

```
(recur id ([id expr] ...) expr)
```

A short-hand recursion construct. The first *id* corresponds to the name of the recursive function. The parenthesized *ids* are the function’s arguments, and each corresponding *expr* is a value supplied for that argument in an initial starting call of the function. The last *expr* is the body of the function.

More precisely, a `recur` form

```
(recur func-id ([arg-id arg-expr] ...)
  body-expr)
```

is equivalent to

```
((local [(define (func-id arg-id ...)
  body-expr)]
  func-id)
```

```
arg-expr ...)
```

5.12 case

```
(case expr [(choice ...) expr] ... [(choice ...) expr])
```

A case form contains one or more “lines” that are surrounded by parentheses or square brackets. Each line contains a sequence of choices—numbers and names for symbols—and an answer *expr*. The initial *expr* is evaluated, and the resulting value is compared to the choices in each line, where the lines are considered in order. The first line that contains a matching choice provides an answer *expr* whose value is the result of the whole case expression. If none of the lines contains a matching choice, it is an error.

```
(case expr [(choice ...) expr] ... [else expr])
```

This form of case is similar to the prior one, except that the final `else` clause is always taken if no prior line contains a choice matching the value of the initial *expr*. In other words, so there is no possibility to “fall off the end” of the case form.

5.13 when and unless

```
(when expr expr)
```

The first *expr* (known as the “test” expression) is evaluated. If it evaluates to `true`, the result of the when expression is the result of evaluating the second *expr*, otherwise the result is `(void)` and the second *expr* is not evaluated. If the result of evaluating the test *expr* is neither `true` nor `false`, it is an error.

```
(unless expr expr)
```

Like when, but the second *expr* is evaluated when the first *expr* produces `false` instead of `true`.

5.14 Primitive Operations

```
< : (real real real ... -> boolean)
```

Purpose: to compare real numbers for less-than

`<= : (real real real ... -> boolean)`

Purpose: to compare real numbers for less-than or equality

`= : (number number number ... -> boolean)`

Purpose: to compare numbers for equality

`> : (real real real ... -> boolean)`

Purpose: to compare real numbers for greater-than

`>= : (real real ... -> boolean)`

Purpose: to compare real numbers for greater-than or equality

`abs : (real -> real)`

Purpose: to compute the absolute value of a real number

`acos : (number -> number)`

Purpose: to compute the arccosine (inverse of cos) of a number

`add1 : (number -> number)`

Purpose: to compute a number one larger than a given number

`angle : (number -> real)`

Purpose: to extract the angle from a complex number

`asin : (number -> number)`

Purpose: to compute the arcsine (inverse of sin) of a number

`atan : (number -> number)`

Purpose: to compute the arctan (inverse of tan) of a number

`ceiling` : (real -> integer)

Purpose: to determine the closest integer (exact or inexact) above a real number

`complex?` : (any -> boolean)

Purpose: to determine whether some value is complex

`conjugate` : (number -> number)

Purpose: to compute the conjugate of a complex number

`cos` : (number -> number)

Purpose: to compute the cosine of a number (radians)

`cosh` : (number -> number)

Purpose: to compute the hyperbolic cosine of a number

`current-seconds` : (-> integer)

Purpose: to compute the current time in seconds elapsed (since a platform-specific starting date)

`denominator` : (rat -> integer)

Purpose: to compute the denominator of a rational

`e` : real

Purpose: Euler's number

`even?` : (integer -> boolean)

Purpose: to determine if some integer (exact or inexact) is even or not

`exact->inexact` : (number -> number)

Purpose: to convert an exact number to an inexact one

`exact?` : (number -> boolean)

Purpose: to determine whether some number is exact

`exp` : (number -> number)

Purpose: to compute e raised to a number

`expt` : (number number -> number)

Purpose: to compute the power of the first to the second number

`floor` : (real -> integer)

Purpose: to determine the closest integer (exact or inexact) below a real number

`gcd` : (integer integer ... -> integer)

Purpose: to compute the greatest common divisor of two integers (exact or inexact)

`imag-part` : (number -> real)

Purpose: to extract the imaginary part from a complex number

`inexact->exact` : (number -> number)

Purpose: to approximate an inexact number by an exact one

`inexact?` : (number -> boolean)

Purpose: to determine whether some number is inexact

`integer->char` : (integer -> char)

Purpose: to lookup the character that corresponds to the given integer (exact only!) in the ASCII table (if any)

`integer-sqrt` : (number -> integer)

Purpose: to compute the integer (exact or inexact) square root of a number

`integer?` : (any -> boolean)

Purpose: to determine whether some value is an integer (exact or inexact)

`lcm` : (integer integer ... -> integer)

Purpose: to compute the least common multiple of two integers (exact or inexact)

`log` : (number -> number)

Purpose: to compute the base-e logarithm of a number

`magnitude` : (number -> real)

Purpose: to determine the magnitude of a complex number

`make-polar` : (real real -> number)

Purpose: to create a complex from a magnitude and angle

`make-rectangular` : (real real -> number)

Purpose: to create a complex from a real and an imaginary part

`max` : (real real ... -> real)

Purpose: to determine the largest number

`min` : (real real ... -> real)

Purpose: to determine the smallest number

`modulo` : (integer integer -> integer)

Purpose: to find the remainder of the division of the first number by the second; try (modulo 4 3) (modulo 4 -3)

`negative?` : (number -> boolean)

Purpose: to determine if some value is strictly smaller than zero

`number->string` : (number -> string)

Purpose: to convert a number to a string

`number?` : (any -> boolean)

Purpose: to determine whether some value is a number

`numerator` : (rat -> integer)

Purpose: to compute the numerator of a rational

`odd?` : (integer -> boolean)

Purpose: to determine if some integer (exact or inexact) is odd or not

`pi` : real

Purpose: the ratio of a circle's circumference to its diameter

`positive?` : (number -> boolean)

Purpose: to determine if some value is strictly larger than zero

`quotient` : (integer integer -> integer)

Purpose: to divide the first integer (exact or inexact) into the second; try (quotient 3 4) and (quotient 4 3)

`random` : (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

```
cddr : ((cons W (cons Z (cons Y (listof X))))
        ->
        (listof X))
```

Purpose: to select the rest of the rest of the rest of a list

```
cddr : ((cons Z (cons Y (listof X))) -> (listof X))
```

Purpose: to select the rest of the rest of a list

```
cdr : ((cons Y (listof X)) -> (listof X))
```

Purpose: to select the rest of a non-empty list

`cons` : (X (listof X) -> (listof X))

Purpose: to construct a list

`cons?` : (any -> boolean)

Purpose: to determine whether some value is a constructed list

`eighth` : ((listof Y) -> Y)

Purpose: to select the eighth item of a non-empty list

`empty?` : (any -> boolean)

Purpose: to determine whether some value is the empty list

`fifth` : ((listof Y) -> Y)

Purpose: to select the fifth item of a non-empty list

`first` : ((cons Y (listof X)) -> Y)

Purpose: to select the first item of a non-empty list

`fourth` : ((listof Y) -> Y)

Purpose: to select the fourth item of a non-empty list

`length` : ((listof any) -> number)

Purpose: to compute the number of items on a list

`list` : (any ... -> (listof any))

Purpose: to construct a list of its arguments

`list-ref` : ((listof X) natural-number -> X)

Purpose: to extract the indexed item from the list

`list?` : (any -> boolean)

Purpose: to determine whether some value is a list

`make-list` : (natural-number any -> (listof any))

Purpose: (make-list k x) constructs a list of k copies of x

`member` : (any (listof any) -> boolean)

Purpose: to determine whether some value is on the list (comparing values with equal?)

`memq` : (any (listof any) -> (union false list))

Purpose: to determine whether some value is on some list (comparing values with eq?)

`memv` : (any (listof any) -> (union false list))

Purpose: to determine whether some value is on the list (comparing values with eqv?)

`null` : empty

Purpose: the empty list

`null?` : (any -> boolean)

Purpose: to determine whether some value is the empty list

`pair?` : (any -> boolean)

Purpose: to determine whether some value is a constructed list

`remove` : (any (listof any) -> (listof any))

Purpose: to construct a list like the given one with the first occurrence of the given item removed (comparing values with equal?)

`rest` : ((cons Y (listof X)) -> (listof X))

Purpose: to select the rest of a non-empty list

`reverse` : ((listof any) -> list)

Purpose: to create a reversed version of a list

`second` : ((cons Z (cons Y (listof X))) -> Y)

Purpose: to select the second item of a non-empty list

`seventh` : ((listof Y) -> Y)

Purpose: to select the seventh item of a non-empty list

`sixth` : ((listof Y) -> Y)

Purpose: to select the sixth item of a non-empty list

`third` : ((cons W (cons Z (cons Y (listof X)))) -> Y)

Purpose: to select the third item of a non-empty list

`make-posn` : (number number -> posn)

Purpose: to construct a posn

`posn-x` : (posn -> number)

Purpose: to extract the x component of a posn

`posn-y` : (posn -> number)

Purpose: to extract the y component of a posn

`posn?` : (anything -> boolean)

Purpose: to determine if its input is a posn

`set-posn-x!` : (posn number -> void)

Purpose: to update the x component of a posn

`set-posn-y!` : (posn number -> void)

Purpose: to update the x component of a posn

`char->integer` : (char -> integer)

Purpose: to lookup the number that corresponds to the given character in the ASCII table (if any)

`char-alphabetic?` : (char -> boolean)

Purpose: to determine whether a character represents an alphabetic character

`char-ci<=?` : (char char char ... -> boolean)

Purpose: to determine whether a character precedes another (or is equal to it) in a case-insensitive manner

`char-ci<?` : (char char char ... -> boolean)

Purpose: to determine whether a character precedes another in a case-insensitive manner

`char-ci=?` : (char char char ... -> boolean)

Purpose: to determine whether two characters are equal in a case-insensitive manner

`char-ci>=?` : (char char char ... -> boolean)

Purpose: to determine whether a character succeeds another (or is equal to it) in a case-insensitive manner

`char-ci>?` : (char char char ... -> boolean)

Purpose: to determine whether a character succeeds another in a case-insensitive manner

`char-downcase` : (char -> char)

Purpose: to determine the equivalent lower-case character

`char-lower-case?` : (char -> boolean)

Purpose: to determine whether a character is a lower-case character

`char-numeric?` : (char -> boolean)

Purpose: to determine whether a character represents a digit

`char-upcase` : (char -> char)

Purpose: to determine the equivalent upper-case character

`char-upper-case?` : (char -> boolean)

Purpose: to determine whether a character is an upper-case character

`char-whitespace?` : (char -> boolean)

Purpose: to determine whether a character represents space

`char<=?` : (char char char ... -> boolean)

Purpose: to determine whether a character precedes another (or is equal to it)

`char<?` : (char char char ... -> boolean)

Purpose: to determine whether a character precedes another

`char=?` : (char char char ... -> boolean)

Purpose: to determine whether two characters are equal

`char>=?` : (char char char ... -> boolean)

Purpose: to determine whether a character succeeds another (or is equal to it)

`char>?` : (char char char ... -> boolean)

Purpose: to determine whether a character succeeds another

`char?` : (any -> boolean)

Purpose: to determine whether a value is a character

`explode` : (string -> (listof string))

Purpose: to translate a string into a list of 1-letter strings

`format` : (string any ... -> string)

Purpose: to format a string, possibly embedding values

`implode` : ((listof string) -> string)

Purpose: to concatenate the list of 1-letter strings into one string

`int->string` : (integer -> string)

Purpose: to convert an integer in [0,55295] or [57344 1114111] to a 1-letter string

`list->string` : ((listof char) -> string)

Purpose: to convert a s list of characters into a string

`make-string` : (nat char -> string)

Purpose: to produce a string of given length from a single given character

`replicate` : (nat string -> string)

Purpose: to replicate the given string

`string` : (char ... -> string)

Purpose: (string c1 c2 ...) builds a string

`string->int` : (string -> integer)

Purpose: to convert a 1-letter string to an integer in [0,55295] or [57344, 1114111]

`string->list` : (string -> (listof char))

Purpose: to convert a string into a list of characters

```
string->number : (string -> (union number false))
```

Purpose: to convert a string into a number, produce false if impossible

```
string->symbol : (string -> symbol)
```

Purpose: to convert a string into a symbol

```
string-alphabetic? : (string -> boolean)
```

Purpose: to determine whether all 'letters' in the string are alphabetic

```
string-append : (string ... -> string)
```

Purpose: to juxtapose the characters of several strings

```
string-ci<=? : (string string string ... -> boolean)
```

Purpose: to determine whether one string alphabetically precedes another (or is equal to it) in a case-insensitive manner

```
string-ci<? : (string string string ... -> boolean)
```

Purpose: to determine whether one string alphabetically precedes another in a case-insensitive manner

```
string-ci=? : (string string string ... -> boolean)
```

Purpose: to compare two strings character-wise in a case-insensitive manner

```
string-ci>=? : (string string string ... -> boolean)
```

Purpose: to determine whether one string alphabetically succeeds another (or is equal to it) in a case-insensitive manner

```
string-ci>? : (string string string ... -> boolean)
```

Purpose: to determine whether one string alphabetically succeeds another in a case-insensitive manner

`string-copy` : (string -> string)

Purpose: to copy a string

`string-ith` : (string nat -> string)

Purpose: to extract the ith 1-letter substring from the given one

`string-length` : (string -> nat)

Purpose: to determine the length of a string

`string-lower-case?` : (string -> boolean)

Purpose: to determine whether all 'letters' in the string are lower case

`string-numeric?` : (string -> boolean)

Purpose: to determine whether all 'letters' in the string are numeric

`string-ref` : (string nat -> char)

Purpose: to extract the i-th character from a string

`string-upper-case?` : (string -> boolean)

Purpose: to determine whether all 'letters' in the string are upper case

`string-whitespace?` : (string -> boolean)

Purpose: to determine whether all 'letters' in the string are white space

`string<=?` : (string string string ... -> boolean)

Purpose: to determine whether one string alphabetically precedes another (or is equal to it)

`string<?` : (string string string ... -> boolean)

Purpose: to determine whether one string alphabetically precedes another

`string=?` : (string string string ... -> boolean)

Purpose: to compare two strings character-wise

`string>=?` : (string string string ... -> boolean)

Purpose: to determine whether one string alphabetically succeeds another (or is equal to it)

`string>?` : (string string string ... -> boolean)

Purpose: to determine whether one string alphabetically succeeds another

`string?` : (any -> boolean)

Purpose: to determine whether a value is a string

`substring` : (string nat nat -> string)

Purpose: to extract the substring starting at a 0-based index up to the second 0-based index (exclusive)

`image=?` : (image image -> boolean)

Purpose: to determine whether two images are equal

`image?` : (any -> boolean)

Purpose: to determine whether a value is an image

`=~` : (real real non-negative-real -> boolean)

Purpose: to check whether two real numbers are within some amount (the third argument) of either other

`eof` : eof

Purpose: the end-of-file value

`eof-object?` : (any -> boolean)

Purpose: to determine whether some value is the end-of-file value

`eq?` : (any any -> boolean)

Purpose: to 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: (andmap p (list x-1 ... x-n)) = (and (p x-1) ... (p x-n))

```
apply : ((X-1 ... X-N -> Y)
         X-1
         ...
         X-i
         (list X-i+1 ... X-N)
         ->
         Y)
```

Purpose: to apply a function using items from a list as the arguments

`argmax` : ((X -> real) (listof X) -> X)

Purpose: to find the (first) element of the list that maximizes the output of the function

`argmin` : ((X -> real) (listof X) -> X)

Purpose: to find the (first) element of the list that minimizes the output of the function

`build-list` : (nat (nat -> X) -> (listof X))

Purpose: (build-list n f) = (list (f 0) ... (f (- n 1)))

`build-string` : (nat (nat -> char) -> string)

Purpose: (build-string n f) = (string (f 0) ... (f (- n 1)))

`compose` : ((Y-1 -> Z)
...
(Y-N -> Y-N-1)
(X-1 ... X-N -> Y-N)
->
(X-1 ... X-N -> Z))

Purpose: to compose a sequence of procedures into a single procedure

`filter` : ((X -> boolean) (listof X) -> (listof X))

Purpose: to construct a list from all those items on a list for which the predicate holds

`foldl` : ((X Y -> Y) Y (listof X) -> Y)

Purpose: (foldl f base (list x-1 ... x-n)) = (f x-n ... (f x-1 base))

`foldr` : ((X Y -> Y) Y (listof X) -> Y)

Purpose: (foldr f base (list x-1 ... x-n)) = (f x-1 ... (f x-n base))

`for-each` : ((any ... -> any) (listof any) ... -> void)

Purpose: to apply a function to each item on one or more lists for effect only

```
map : ((X ... -> Z) (listof X) ... -> (listof Z))
```

Purpose: to construct a new list by applying a function to each item on one or more existing lists

```
memf : ((X -> boolean)
        (listof X)
        ->
        (union false (listof X)))
```

Purpose: to determine whether the first argument produces true for some value in the second argument

```
ormap : ((X -> boolean) (listof X) -> boolean)
```

Purpose: (ormap p (list x-1 ... x-n)) = (or (p x-1) ... (p x-n))

```
procedure? : (any -> boolean)
```

Purpose: to determine if a value is a procedure

```
quicksort : ((listof X) (X X -> boolean) -> (listof X))
```

Purpose: to construct a list from all items on a list in an order according to a predicate

```
sort : ((listof X) (X X -> boolean) -> (listof X))
```

Purpose: to construct a list from all items on a list in an order according to a predicate

```
display : (any -> void)
```

Purpose: to print the argument to stdout (without quotes on symbols and strings, etc.)

```
newline : (-> void)
```

Purpose: to print a newline to stdout

```
pretty-print : (any -> void)
```

Purpose: like write, but with standard newlines and indentation

`print` : (any -> void)

Purpose: to print the argument as a value to stdout

`printf` : (string any ... -> void)

Purpose: to format the rest of the arguments according to the first argument and print it to stdout

`read` : (-> sexp)

Purpose: to read input from the user

`write` : (any -> void)

Purpose: to print the argument to stdout (in a traditional style that is somewhere between print and display)

`build-vector` : (nat (nat -> X) -> (vectorof X))

Purpose: to construct a vector

`make-vector` : (number X -> (vectorof X))

Purpose: to construct a vector

`vector` : (X ... -> (vector X ...))

Purpose: to construct a vector

`vector-length` : ((vector X) -> nat)

Purpose: to determine the length of a vector

`vector-ref` : ((vector X) nat -> X)

Purpose: to extract an element from a vector

`vector-set!` : ((vectorof X) nat X -> void)

Purpose: to update a vector

```
vector? : (any -> boolean)
```

Purpose: to determine if a value is a vector

```
box : (any -> box)
```

Purpose: to construct a box

```
box? : (any -> boolean)
```

Purpose: to determine if a value is a box

```
set-box! : (box any -> void)
```

Purpose: to update a box

```
unbox : (box -> any)
```

Purpose: to extract the boxed value

5.15 Unchanged Forms

```
(local [definition ...] expr)  
(letrec ([id expr-for-let] ...) expr)  
(let* ([id expr-for-let] ...) expr)
```

The same as Intermediate's local, letrec, and let*.

```
(cond [expr expr] ... [expr expr])  
else
```

The same as Beginning's cond, except that else can be used with case.

```
(if expr expr expr)
```

The same as Beginning's if.

```
(and expr expr expr ...)  
(or expr expr expr ...)
```

The same as Beginning's and and or.

```
(time expr)
```

The same as Intermediate's time.

```
(check-expect expr expr)  
(check-within expr expr expr)  
(check-error expr expr)
```

The same as Beginning's check-expect, etc.

```
empty : empty?  
true : boolean?  
false : boolean?
```

Constants for the empty list, true, and false.

```
(require module-path)
```

The same as Beginning's require.

Index

`##app`, 12
`##app`, 138
`##app`, 106
`##app`, 75
`*`, 94
`*`, 125
`*`, 16
`*`, 44
`*`, 160
`+`, 44
`+`, 16
`+`, 94
`+`, 160
`+`, 125
`-`, 94
`-`, 160
`-`, 44
`-`, 125
`-`, 16
`/`, 160
`/`, 16
`/`, 45
`/`, 94
`/`, 125
`<`, 140
`<`, 75
`<`, 106
`<`, 45
`<`, 17
`<=`, 17
`<=`, 45
`<=`, 106
`<=`, 141
`<=`, 75
`=`, 17
`=`, 75
`=`, 45
`=`, 106
`=`, 141
`=~`, 34
`=~`, 124
`=~`, 158
`=~`, 63
`=~`, 93
`>`, 45
`>`, 17
`>`, 76
`>`, 106
`>`, 141
`>=`, 141
`>=`, 106
`>=`, 45
`>=`, 76
`>=`, 17
`abs`, 76
`abs`, 17
`abs`, 141
`abs`, 106
`abs`, 45
`acos`, 17
`acos`, 106
`acos`, 45
`acos`, 141
`acos`, 76
`add1`, 17
`add1`, 76
`add1`, 45
`add1`, 141
`add1`, 107
Advanced Student, 129
`and`, 13
`and`, 64
`and`, 97
`and`, 128
`and`, 165
`and`, 13
`andmap`, 160
`andmap`, 95
`andmap`, 125
`angle`, 107
`angle`, 46
`angle`, 141

angle, 17
angle, 76
append, 24
append, 113
append, 147
append, 82
append, 52
apply, 95
apply, 125
apply, 160
argmax, 161
argmax, 125
argmax, 95
argmin, 126
argmin, 95
argmin, 161
asin, 76
asin, 107
asin, 141
asin, 17
asin, 46
assq, 24
assq, 52
assq, 82
assq, 113
assq, 147
atan, 141
atan, 107
atan, 18
atan, 46
atan, 76
begin, 138
begin, 138
begin0, 138
begin0, 138
Beginning Student, 5
Beginning Student with List Abbreviations,
37
boolean=?, 147
boolean=?, 81
boolean=?, 112
boolean=?, 51
boolean=?, 23
boolean?, 147
boolean?, 23
boolean?, 112
boolean?, 82
boolean?, 51
box, 164
box?, 164
build-list, 161
build-list, 95
build-list, 126
build-string, 95
build-string, 161
build-string, 126
build-vector, 163
caaar, 24
caaar, 52
caaar, 83
caaar, 113
caaar, 148
caadr, 52
caadr, 83
caadr, 148
caadr, 24
caadr, 113
caar, 52
caar, 83
caar, 24
caar, 113
caar, 148
cadar, 24
cadar, 114
cadar, 148
cadar, 83
cadar, 53
caddr, 148
caddr, 25
caddr, 53
caddr, 114
caddr, 83
caddr, 114
caddr, 83

caddr, 53
caddr, 148
caddr, 25
cadr, 83
cadr, 25
cadr, 114
cadr, 148
cadr, 53
car, 83
car, 53
car, 114
car, 148
car, 25
case, 140
case, 140
cdaar, 114
cdaar, 25
cdaar, 53
cdaar, 149
cdaar, 84
cdadr, 84
cdadr, 53
cdadr, 149
cdadr, 114
cdadr, 25
cdar, 114
cdar, 53
cdar, 84
cdar, 25
cdar, 149
cddar, 84
cddar, 115
cddar, 149
cddar, 25
cddar, 54
cdddr, 149
cdddr, 54
cdddr, 84
cdddr, 115
cdddr, 26
cddr, 115
cddr, 26
cddr, 149
cddr, 84
cddr, 54
cdr, 84
cdr, 115
cdr, 54
cdr, 26
cdr, 149
ceiling, 18
ceiling, 76
ceiling, 107
ceiling, 46
ceiling, 142
char->integer, 118
char->integer, 153
char->integer, 57
char->integer, 29
char->integer, 87
char-alphabetic?, 57
char-alphabetic?, 118
char-alphabetic?, 29
char-alphabetic?, 87
char-alphabetic?, 153
char-ci<=?, 57
char-ci<=?, 29
char-ci<=?, 153
char-ci<=?, 118
char-ci<=?, 88
char-ci<?, 29
char-ci<?, 88
char-ci<?, 57
char-ci<?, 118
char-ci<?, 153
char-ci=?, 57
char-ci=?, 88
char-ci=?, 153
char-ci=?, 118
char-ci=?, 29
char-ci>=?, 29
char-ci>=?, 57
char-ci>=?, 153
char-ci>=?, 88

char-ci>=?, 118
char-ci>?, 88
char-ci>?, 119
char-ci>?, 58
char-ci>?, 153
char-ci>?, 29
char-downcase, 153
char-downcase, 30
char-downcase, 119
char-downcase, 58
char-downcase, 88
char-lower-case?, 119
char-lower-case?, 88
char-lower-case?, 58
char-lower-case?, 30
char-lower-case?, 153
char-numeric?, 88
char-numeric?, 154
char-numeric?, 30
char-numeric?, 58
char-numeric?, 119
char-upcase, 88
char-upcase, 58
char-upcase, 30
char-upcase, 119
char-upcase, 154
char-upper-case?, 30
char-upper-case?, 154
char-upper-case?, 119
char-upper-case?, 89
char-upper-case?, 58
char-whitespace?, 119
char-whitespace?, 89
char-whitespace?, 30
char-whitespace?, 58
char-whitespace?, 154
char<=?, 89
char<=?, 58
char<=?, 119
char<=?, 30
char<=?, 154
char<?, 89
char<?, 119
char<?, 58
char<?, 30
char<?, 154
char=?, 30
char=?, 58
char=?, 119
char=?, 89
char=?, 154
char>=?, 120
char>=?, 30
char>=?, 154
char>=?, 89
char>=?, 59
char>?, 31
char>?, 154
char>?, 89
char>?, 59
char>?, 120
char?, 31
char?, 59
char?, 154
char?, 89
char?, 120
check-error, 14
check-error, 97
check-error, 64
check-error, 165
check-error, 128
check-expect, 128
check-expect, 14
check-expect, 64
check-expect, 165
check-expect, 97
check-within, 14
check-within, 64
check-within, 97
check-within, 128
check-within, 165
complex?, 142
complex?, 46
complex?, 107

[complex?](#), 76
[complex?](#), 18
[compose](#), 95
[compose](#), 161
[compose](#), 126
[cond](#), 13
[cond](#), 128
[cond](#), 164
[cond](#), 13
[cond](#), 97
[cond](#), 64
[conjugate](#), 142
[conjugate](#), 107
[conjugate](#), 77
[conjugate](#), 46
[conjugate](#), 18
[cons](#), 115
[cons](#), 26
[cons](#), 150
[cons](#), 84
[cons](#), 54
[cons?](#), 85
[cons?](#), 26
[cons?](#), 115
[cons?](#), 54
[cons?](#), 150
[cos](#), 142
[cos](#), 18
[cos](#), 107
[cos](#), 46
[cos](#), 77
[cosh](#), 77
[cosh](#), 46
[cosh](#), 142
[cosh](#), 18
[cosh](#), 107
[current-seconds](#), 107
[current-seconds](#), 77
[current-seconds](#), 18
[current-seconds](#), 46
[current-seconds](#), 142
[define](#), 137
[define](#), 11
[define](#), 73
[define](#), 105
[define](#), 105
[define](#), 11
[define](#), 64
[define](#), 137
[define](#), 73
[define-struct](#), 137
[define-struct](#), 11
[define-struct](#), 73
[define-struct](#), 64
[define-struct](#), 127
[define-struct](#), 11
[define-struct](#), 137
[define-struct](#), 73
[delay](#), 138
[delay](#), 138
[denominator](#), 142
[denominator](#), 46
[denominator](#), 77
[denominator](#), 18
[denominator](#), 108
[display](#), 162
[e](#), 108
[e](#), 47
[e](#), 18
[e](#), 77
[e](#), 142
[eighth](#), 150
[eighth](#), 54
[eighth](#), 85
[eighth](#), 26
[eighth](#), 115
[else](#), 128
[else](#), 164
[else](#), 13
[else](#), 97
[else](#), 64
[empty](#), 14
[empty](#), 14
[empty](#), 97

empty, 65
empty, 128
empty, 165
empty?, 26
empty?, 150
empty?, 54
empty?, 115
empty?, 85
eof, 63
eof, 93
eof, 35
eof, 124
eof, 158
eof-object?, 63
eof-object?, 93
eof-object?, 158
eof-object?, 35
eof-object?, 124
eq?, 159
eq?, 93
eq?, 63
eq?, 124
eq?, 35
equal?, 159
equal?, 63
equal?, 93
equal?, 124
equal?, 35
equal~?, 94
equal~?, 124
equal~?, 35
equal~?, 159
equal~?, 63
eqv?, 63
eqv?, 94
eqv?, 35
eqv?, 159
eqv?, 124
error, 124
error, 63
error, 94
error, 159
error, 35
even?, 47
even?, 18
even?, 77
even?, 108
even?, 142
exact->inexact, 19
exact->inexact, 108
exact->inexact, 47
exact->inexact, 77
exact->inexact, 142
exact?, 19
exact?, 77
exact?, 108
exact?, 143
exact?, 47
exit, 159
exit, 124
exit, 94
exit, 35
exit, 63
exp, 47
exp, 19
exp, 143
exp, 108
exp, 77
explode, 120
explode, 59
explode, 155
explode, 31
explode, 89
expt, 143
expt, 78
expt, 47
expt, 19
expt, 108
false, 65
false, 15
false, 97
false, 165
false, 128
false?, 82

false?, 23
false?, 112
false?, 147
false?, 51
fifth, 54
fifth, 150
fifth, 85
fifth, 115
fifth, 26
filter, 126
filter, 161
filter, 96
first, 116
first, 85
first, 150
first, 26
first, 55
floor, 108
floor, 47
floor, 143
floor, 19
floor, 78
foldl, 126
foldl, 96
foldl, 161
foldr, 161
foldr, 96
foldr, 126
for-each, 126
for-each, 96
for-each, 161
force, 159
format, 59
format, 31
format, 120
format, 155
format, 89
fourth, 55
fourth, 116
fourth, 26
fourth, 85
fourth, 150
Function Calls, 138
Function Calls, 12
Function Calls, 74
Function Calls, 105
gcd, 108
gcd, 19
gcd, 143
gcd, 47
gcd, 78
How to Design Programs Languages, 1
Identifiers, 15
Identifiers, 75
identity, 94
identity, 159
identity, 125
identity, 64
identity, 35
if, 13
if, 64
if, 164
if, 97
if, 13
if, 128
imag-part, 19
imag-part, 47
imag-part, 143
imag-part, 108
imag-part, 78
image=?, 34
image=?, 62
image=?, 158
image=?, 93
image=?, 123
image?, 158
image?, 123
image?, 93
image?, 34
image?, 62
implode, 31
implode, 155
implode, 59
implode, 120

implode, 90
inexact->exact, 143
inexact->exact, 19
inexact->exact, 47
inexact->exact, 109
inexact->exact, 78
inexact?, 48
inexact?, 19
inexact?, 143
inexact?, 109
inexact?, 78
int->string, 155
int->string, 59
int->string, 120
int->string, 31
int->string, 90
integer->char, 78
integer->char, 48
integer->char, 143
integer->char, 109
integer->char, 19
integer-sqrt, 109
integer-sqrt, 78
integer-sqrt, 48
integer-sqrt, 143
integer-sqrt, 20
integer?, 109
integer?, 20
integer?, 144
integer?, 78
integer?, 48
Intermediate Student, 66
Intermediate Student with Lambda, 98
lambda, 137
lambda, 105
lambda, 64
lambda, 137
lambda, 105
lambda, 73
lambda, 11
lcm, 78
lcm, 48
lcm, 144
lcm, 109
lcm, 20
length, 116
length, 85
length, 150
length, 27
length, 55
let, 139
let, 74
let, 139
let, 127
let*, 74
let*, 127
let*, 164
letrec, 74
letrec, 164
letrec, 127
letrec, let, and let*, 74
list, 27
list, 55
list, 150
list, 116
list, 85
list*, 85
list*, 116
list*, 27
list*, 55
list->string, 31
list->string, 59
list->string, 155
list->string, 90
list->string, 120
list-ref, 116
list-ref, 150
list-ref, 27
list-ref, 55
list-ref, 85
list?, 151
local, 74
local, 164
local, 74

local, 127
log, 144
log, 79
log, 109
log, 20
log, 48
magnitude, 20
magnitude, 48
magnitude, 109
magnitude, 144
magnitude, 79
make-list, 55
make-list, 116
make-list, 86
make-list, 151
make-list, 27
make-polar, 20
make-polar, 48
make-polar, 109
make-polar, 144
make-polar, 79
make-posn, 152
make-posn, 57
make-posn, 118
make-posn, 87
make-posn, 28
make-rectangular, 48
make-rectangular, 79
make-rectangular, 144
make-rectangular, 20
make-rectangular, 109
make-string, 59
make-string, 120
make-string, 31
make-string, 90
make-string, 155
make-vector, 163
map, 162
map, 126
map, 96
max, 20
max, 79
max, 144
max, 110
max, 48
member, 151
member, 116
member, 27
member, 86
member, 55
memf, 96
memf, 162
memf, 127
memq, 27
memq, 116
memq, 86
memq, 55
memq, 151
memv, 86
memv, 151
memv, 116
memv, 55
memv, 27
min, 20
min, 49
min, 144
min, 79
min, 110
modulo, 49
modulo, 110
modulo, 144
modulo, 20
modulo, 79
negative?, 144
negative?, 110
negative?, 79
negative?, 21
negative?, 49
newline, 162
not, 51
not, 147
not, 23
not, 82
not, 112

null, 86
 null, 117
 null, 27
 null, 151
 null, 56
 null?, 56
 null?, 86
 null?, 117
 null?, 151
 null?, 27
 number->string, 79
 number->string, 49
 number->string, 110
 number->string, 145
 number->string, 21
 number?, 49
 number?, 110
 number?, 79
 number?, 21
 number?, 145
 numerator, 80
 numerator, 21
 numerator, 49
 numerator, 110
 numerator, 145
 odd?, 145
 odd?, 49
 odd?, 80
 odd?, 21
 odd?, 110
 or, 14
 or, 165
 or, 64
 or, 97
 or, 128
 or, 14
 ormap, 127
 ormap, 96
 ormap, 162
 pair?, 151
 pair?, 86
 pair?, 117
 pair?, 56
 pair?, 28
 pi, 21
 pi, 145
 pi, 110
 pi, 80
 pi, 49
 positive?, 110
 positive?, 80
 positive?, 145
 positive?, 21
 positive?, 49
 posn-x, 87
 posn-x, 28
 posn-x, 57
 posn-x, 118
 posn-x, 152
 posn-y, 152
 posn-y, 57
 posn-y, 118
 posn-y, 29
 posn-y, 87
 posn?, 29
 posn?, 87
 posn?, 118
 posn?, 57
 posn?, 152
 pretty-print, 162
 Primitive Calls, 12
 Primitive Operation Names, 106
 Primitive Operations, 140
 Primitive Operations, 44
 Primitive Operations, 16
 Primitive Operations, 75
 print, 163
 printf, 163
 procedure?, 162
 procedure?, 127
 procedure?, 96
 promise?, 159
 Quasiquote, 43
 quasiquote, 44

quicksort, 96
quicksort, 162
quicksort, 127
Quote, 43
quote, 43
quote, 15
quotient, 49
quotient, 111
quotient, 21
quotient, 145
quotient, 80
random, 111
random, 80
random, 145
random, 50
random, 21
rational?, 145
rational?, 50
rational?, 111
rational?, 22
rational?, 80
read, 163
real-part, 80
real-part, 111
real-part, 145
real-part, 50
real-part, 22
real?, 80
real?, 22
real?, 50
real?, 146
real?, 111
recur, 139
recur, 139
remainder, 111
remainder, 50
remainder, 80
remainder, 146
remainder, 22
remove, 56
remove, 28
remove, 151
remove, 117
remove, 86
replicate, 155
replicate, 120
replicate, 59
replicate, 90
replicate, 31
require, 15
require, 128
require, 15
require, 97
require, 165
require, 65
rest, 86
rest, 151
rest, 28
rest, 117
rest, 56
reverse, 86
reverse, 117
reverse, 28
reverse, 56
reverse, 152
round, 146
round, 22
round, 81
round, 111
round, 50
second, 117
second, 152
second, 87
second, 28
second, 56
set!, 138
set!, 138
set-box!, 164
set-posn-x!, 152
set-posn-y!, 153
seventh, 28
seventh, 117
seventh, 87
seventh, 56

seventh, 152
 sgn, 81
 sgn, 50
 sgn, 22
 sgn, 146
 sgn, 111
 shared, 139
 shared, 139
 sin, 50
 sin, 111
 sin, 22
 sin, 81
 sin, 146
 sinh, 111
 sinh, 146
 sinh, 50
 sinh, 81
 sinh, 22
 sixth, 56
 sixth, 152
 sixth, 117
 sixth, 87
 sixth, 28
 sort, 162
 sort, 97
 sort, 127
 sqr, 50
 sqr, 146
 sqr, 112
 sqr, 22
 sqr, 81
 sqrt, 51
 sqrt, 22
 sqrt, 146
 sqrt, 81
 sqrt, 112
 string, 60
 string, 90
 string, 31
 string, 121
 string, 155
 string->int, 60
 string->int, 90
 string->int, 32
 string->int, 155
 string->int, 121
 string->list, 155
 string->list, 60
 string->list, 121
 string->list, 32
 string->list, 90
 string->number, 90
 string->number, 32
 string->number, 121
 string->number, 60
 string->number, 156
 string->symbol, 156
 string->symbol, 121
 string->symbol, 90
 string->symbol, 60
 string->symbol, 32
 string-alphabetic?, 32
 string-alphabetic?, 91
 string-alphabetic?, 60
 string-alphabetic?, 156
 string-alphabetic?, 121
 string-append, 91
 string-append, 156
 string-append, 60
 string-append, 32
 string-append, 121
 string-ci<=?, 156
 string-ci<=?, 91
 string-ci<=?, 32
 string-ci<=?, 60
 string-ci<=?, 121
 string-ci<?, 60
 string-ci<?, 91
 string-ci<?, 32
 string-ci<?, 156
 string-ci<?, 121
 string-ci=?, 156
 string-ci=?, 122
 string-ci=?, 61

string-ci=?, 91
string-ci=?, 32
string-ci>=?, 122
string-ci>=?, 32
string-ci>=?, 156
string-ci>=?, 61
string-ci>=?, 91
string-ci>?, 91
string-ci>?, 61
string-ci>?, 122
string-ci>?, 156
string-ci>?, 33
string-copy, 122
string-copy, 91
string-copy, 157
string-copy, 33
string-copy, 61
string-ith, 33
string-ith, 61
string-ith, 122
string-ith, 91
string-ith, 157
string-length, 33
string-length, 92
string-length, 157
string-length, 122
string-length, 61
string-lower-case?, 33
string-lower-case?, 122
string-lower-case?, 61
string-lower-case?, 157
string-lower-case?, 92
string-numeric?, 122
string-numeric?, 92
string-numeric?, 33
string-numeric?, 157
string-numeric?, 61
string-ref, 157
string-ref, 122
string-ref, 33
string-ref, 61
string-ref, 92
string-upper-case?, 157
string-upper-case?, 122
string-upper-case?, 61
string-upper-case?, 33
string-upper-case?, 92
string-whitespace?, 62
string-whitespace?, 157
string-whitespace?, 92
string-whitespace?, 33
string-whitespace?, 123
string<=?, 157
string<=?, 34
string<=?, 123
string<=?, 62
string<=?, 92
string<?, 62
string<?, 123
string<?, 92
string<?, 34
string<?, 157
string=?, 62
string=?, 158
string=?, 34
string=?, 92
string=?, 123
string>=?, 123
string>=?, 34
string>=?, 92
string>=?, 62
string>=?, 158
string>?, 93
string>?, 158
string>?, 34
string>?, 62
string>?, 123
string?, 62
string?, 123
string?, 34
string?, 93
string?, 158
struct?, 35
struct?, 64

`struct?`, 125
`struct?`, 94
`struct?`, 159
`sub1`, 146
`sub1`, 51
`sub1`, 112
`sub1`, 23
`sub1`, 81
`substring`, 93
`substring`, 158
`substring`, 34
`substring`, 123
`substring`, 62
`symbol->string`, 82
`symbol->string`, 23
`symbol->string`, 147
`symbol->string`, 112
`symbol->string`, 51
`symbol=?`, 147
`symbol=?`, 82
`symbol=?`, 23
`symbol=?`, 113
`symbol=?`, 51
`symbol?`, 147
`symbol?`, 52
`symbol?`, 113
`symbol?`, 82
`symbol?`, 23
Symbols, 15
`tan`, 23
`tan`, 51
`tan`, 81
`tan`, 146
`tan`, 112
Test Cases, 14
`third`, 117
`third`, 87
`third`, 28
`third`, 152
`third`, 56
`time`, 75
`time`, 128
`time`, 165
`time`, 75
`true`, 65
`true`, 15
`true`, 165
`true`, 97
`true`, 128
`true` and `false`, 15
`unbox`, 164
Unchanged Forms, 64
Unchanged Forms, 164
Unchanged Forms, 127
Unchanged Forms, 97
`unless`, 140
`unquote`, 44
`unquote-splicing`, 44
`vector`, 163
`vector-length`, 163
`vector-ref`, 163
`vector-set!`, 163
`vector?`, 164
`void`, 160
`void?`, 160
`when`, 140
`when` and `unless`, 140
`write`, 163
`zero?`, 147
`zero?`, 51
`zero?`, 81
`zero?`, 23
`zero?`, 112
 λ , 105
 λ , 137