How to Design Programs Languages

Version 5.1.2

August 3, 2011

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

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

Contents

1 Beginning Student

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

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

 definition = (define (name variable variable ...) expr)
            | (define name expr)
            | (define name (lambda (variable variable ...) expr))
            | (define-struct name (name ...))

       expr = (name expr expr ...)
            | (cond [expr expr] ... [expr expr])
            | (cond [expr expr] ... [else expr])
            | (if expr expr expr)
            | (and expr expr expr ...)
            | (or expr expr expr ...)
            | name
            | 'name
            | number
            | string
            | character

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

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

    package = (string string number number)
```

An *name* or a *variable* 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 *symbol* is a quote character followed by a name. A symbol is a value, just like 0 or empty.

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

In function calls, the function appearing immediatly after the open parenthesis can be any functions defined with `define` or `define-struct`, or any one of the pre-defined functions.

1.1 Pre-Defined Variables

`empty : empty?`

The empty list.

`true : boolean?`

The true value.

`false : boolean?`

The false value.

1.2 Syntax

`(define (name variable variable ...) expression)`

Defines a function named *name*. The *expression* is the body of the function. When the function is called, the values of the arguments are inserted into the body in place of the *variable*s. The function returns the value of that new expression.

The function name's cannot be the same as that of another function or variable.

`(define name expression)`

Defines a variable called *name* with the the value of *expression*. The variable name's cannot be the same as that of another function or variable, and *name* itself must not appear in *expression*.

```
(define name (lambda (variable variable ...) expression))
```

An alternate way on defining functions. The *name* is the name of the function, which cannot be the same as that of another function or variable.

A `lambda` cannot be used outside of this alternate syntax.

```
'name
(quote name)
```

A quoted *name* is a symbol. A symbol is a value, just like `0` or `empty`.

```
(define-struct structure-name (field-name ...))
```

Defines a new structure called *structure-name*. The structure's fields are named by the *field-name*s. After the `define-struct`, the following new functions are available:

- `make-`*structure-name* : takes a number of arguments equal to the number of fields in the structure, and creates a new instance of that structure.
- *structure-name-field-name* : takes an instance of the structure and returns the value in the field named by *field-name*.
- *structure-name*`?` : takes any value, and returns `true` if the value is an instance of the structure.

The name of the new functions introduced by `define-struct` must not be the same as that of other functions or variables, otherwise `define-struct` reports an error.

```
(name expression expression ...)
```

Calls the function named *name*. The value of the call is the value of *name*'s body when every one of the function's variables are replaced by the values of the corresponding *expression*s.

The function named *name* must defined before it can be called. The number of argument *expression*s must be the same as the number of arguments expected by the function.

```
(cond [question-expression answer-expression] ...)
(cond [question-expression answer-expression]
      ...
      [else answer-expression])
```

Chooses a clause based on some condition. `cond` finds the first *question-expression* that evaluates to `true`, then evaluates the corresponding *answer-expression*.

If none of the *question-expression*s evaluates to `true`, `cond`'s value is the *answer-expression* of the `else` clause. If there is no `else`, `cond` reports an error. If the result of a *question-expression* is neither `true` nor `false`, `cond` also reports an error.

`else` cannot be used outside of `cond`.

`(if` *test-expression then-expression else-expression*`)`

When the value of the *test-expression* is `true`, `if` evaluates the *then-expression*. When the test is `false`, `if` evaluates the *else-expression*.

If the *test-expression* is neither `true` nor `false`, `if` reports an error.

`(or` *expression expression expression ...*`)`

Evaluates to `true` if all the *expression*s are `true`. If any *expression* is `false`, the `or` expression immediately evaluates to `false` (and the expressions to the right of that expression are not evaluated.)

If any of the expressions evaluate to a value other than `true` or `false`, `or` reports an error.

`(and` *expression expression expression ...*`)`

Evaluates to `true` as soon as one of the *expression*s is `true` (and the expressions to the right of that expression are not evaluated.) If all of the *expression*s are `false`, the `and` expression evaluates to `false`.

If any of the expressions evaluate to a value other than `true` or `false`, `and` reports an error.

`(check-expect` *expression expected-expression*`)`

Checks that the first *expression* evaluates to the same value as the *expected-expression*.

`(check-within` *expression expected-expression delta-expression*`)`

Checks that the first *expression* evaluates to a value within *delta-expression* of the *expected-expression*. If *delta-expression* is not a number, `check-within` reports an error.

```
(check-error expression match-expression)
(check-error expression)
```

Checks that the *expression* reports an error, where the error messages matches the value of matchexpression, if it is present.

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

Checks that the value of the first *expression* as that of one of the following *expression*s.

```
(check-range expression low-expression high-expression)
```

Checks that the value of the first *expression* is a number in between the value of the *low-expression* and the *high-expression*, inclusive.

```
(require string)
```

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

Accesses a file in an installed library. The library name is an identifier with the same constraints as for a relative-path string (though without the quotes), 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., the current file). The first *string* names the library file, and the remaining *string*s name the collection (and sub-collection, and so on) where the file is installed. Each string is constrained in the same way as for the (require *string*) form.

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

Accesses a library that is distributed on the internet via the PLaneT server, making it defini-

tions available in the current module (i.e., current file).

1.3 Pre-defined Functions

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

---

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

Evaluates the product of all of the input numbers.

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

Evaluates the sum of the input numbers.

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

subtracts the second (and following) number(s) from the first; negate the number if there is only one argument.

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

Divides 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)`

Compares real numbers for less-than.

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

Compares real numbers for less-than or equality.

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

Compares numbers for equality.

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

Compares real numbers for greater-than.

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

Compares real numbers for greater-than or equality.

`abs : (real -> real)`

Evaluates the absolute value of a real number.

`acos : (number -> number)`

Evaluates the arccosine (inverse of cos) of a number.

`add1 : (number -> number)`

Evaluates a number one larger than a given number.

`angle : (number -> real)`

Extracts the angle from a complex number.

`asin : (number -> number)`

Evaluates the arcsine (inverse of sin) of a number.

`atan : (number (number) -> number)`

Evaluates the arctan of the given number or the ratio of the two given numbers.

`ceiling : (real -> integer)`

Determines the closest integer (exact or inexact) above a real number.

`complex? : (any -> boolean)`

Determines whether some value is complex.

`conjugate : (number -> number)`

Evaluates the conjugate of a complex number.

`cos : (number -> number)`

Evaluates the cosine of a number (radians).

`cosh : (number -> number)`

Evaluates the hyperbolic cosine of a number.

`current-seconds : (-> integer)`

Evaluates the current time in seconds elapsed (since a platform-specific starting date).

`denominator : (rat -> integer)`

Evaluates the denominator of a rational.

`e : real`

Euler's number.

`even? : (integer -> boolean)`

Determines if some integer (exact or inexact) is even or not.

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

Converts an exact number to an inexact one.

`exact? : (number -> boolean)`

Determines whether some number is exact.

`exp : (number -> number)`

Evaluates e raised to a number.

`expt : (number number -> number)`

Evaluates the power of the first to the second number.

`floor : (real -> integer)`

Determines the closest integer (exact or inexact) below a real number.

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

Evaluates the greatest common divisior of two integers (exact or inexact).

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

Extracts the imaginary part from a complex number.

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

Approximates an inexact number by an exact one.

`inexact? : (number -> boolean)`

Determines whether some number is inexact.

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

Lookups the character that corresponds to the given integer (exact only!) in the ASCII table (if any).

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

Evaluates the integer (exact or inexact) square root of a number.

`integer? : (any -> boolean)`

Determines whether some value is an integer (exact or inexact).

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

Evaluates the least common multiple of two integers (exact or inexact).

`log : (number -> number)`

Evaluates the base-e logarithm of a number.

`magnitude : (number -> real)`

Determines the magnitude of a complex number.

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

Creates a complex from a magnitude and angle.

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

Creates a complex from a real and an imaginary part.

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

Determines the largest number.

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

Determines the smallest number.

`modulo : (integer integer -> integer)`

Finds the remainder of the division of the first number by the second; try (modulo 4 3) (modulo 4 -3).

`negative? : (number -> boolean)`

Determines if some value is strictly smaller than zero.

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

Converts a number to a string.

`number? : (any -> boolean)`

Determines whether some value is a number.

`numerator : (rat -> integer)`

Evaluates the numerator of a rational.

`odd? : (integer -> boolean)`

Determines if some integer (exact or inexact) is odd or not.

`pi : real`

The ratio of a circle's circumference to its diameter.

`positive? : (number -> boolean)`

Determines if some value is strictly larger than zero.

`quotient : (integer integer -> integer)`

Divides the first integer (exact or inexact) into the second; try (quotient 3 4) and (quotient 4 3).

`random : (integer -> integer)`

Generates a random natural number less than some given integer (exact only!).

`rational? : (any -> boolean)`

Determines whether some value is a rational number.

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

Extracts the real part from a complex number.

`real? : (any -> boolean)`

Determines whether some value is a real number.

`remainder : (integer integer -> integer)`

Determines the remainder of dividing the first by the second integer (exact or inexact).

`round : (real -> integer)`

Rounds a real number to an integer (rounds to even to break ties).

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

Evaluates the sign of a real number.

`sin : (number -> number)`

Evaluates the sine of a number (radians).

`sinh : (number -> number)`

Evaluates the hyperbolic sine of a number.

`sqr : (number -> number)`

Evaluates the square of a number.

`sqrt : (number -> number)`

Evaluates the square root of a number.

`sub1 : (number -> number)`

Evaluates a number one smaller than a given number.

`tan : (number -> number)`

Evaluates the tangent of a number (radians).

`zero? : (number -> boolean)`

Determines if some value is zero or not.

1.3.2 Booleans

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

Determines whether two booleans are equal.

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

Determines whether some value is a boolean.

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

Determines whether a value is false.

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

Evaluates the negation of a boolean value.

1.3.3 Symbols

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

Converts a symbol to a string.

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

Determines whether two symbols are equal.

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

Determines whether some value is a symbol.

1.3.4 Lists

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

Creates a single list from several, by juxtaposition of the items.

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

Determines 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)
```

Selects 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))
```

Selects the rest of the first list in the first list of a list.

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

Selects the first item of the first list in a list.

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

Selects the second item of the first list of a list.

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

Selects the fourth item of a non-empty list.

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

Selects the third item of a non-empty list.

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

Selects the second item of a non-empty list.

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

Selects the first item of a non-empty list.

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

Selects 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))
```

Selects the rest of the first list in the rest of a list.

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

Selects the rest of a non-empty list in a list.

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

Selects the rest of the rest of the first list of a list.

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

Selects the rest of the rest of the rest of a list.

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

Selects the rest of the rest of a list.

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

Selects the rest of a non-empty list.

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

Constructs a list.

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

Determines whether some value is a constructed list.

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

Selects the eighth item of a non-empty list.

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

Determines whether some value is the empty list.

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

Selects the fifth item of a non-empty list.

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

Selects the first item of a non-empty list.

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

Selects the fourth item of a non-empty list.

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

Evaluates the number of items on a list.

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

Constructs a list of its arguments.

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

Constructs a list by adding multiple items to a list.

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

Extracts the indexed item from the list.

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

Constructs a list of k (the first argument) copies of x (the second argument).

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

Determines whether some value is on the list (comparing values with equal?).

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

Determines whether some value is on the list (comparing values with equal?).

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

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

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

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

```
null : empty
```

The empty list.

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

Determines whether some value is the empty list.

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

Constructs 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))
```

Selects the rest of a non-empty list.

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

Creates a reversed version of a list.

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

Selects the second item of a non-empty list.

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

Selects the seventh item of a non-empty list.

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

Selects the sixth item of a non-empty list.

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

Selects the third item of a non-empty list.

1.3.5 Posns

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

Constructs a posn.

```
posn : signature
```

Signature for posns.

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

Extracts the x component of a posn.

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

Extracts the y component of a posn.

`posn? : (anything -> boolean)`

Determines if its input is a posn.

1.3.6 Characters

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

Lookups the number that corresponds to the given character in the ASCII table (if any).

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

Determines whether a character represents an alphabetic character.

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

Determines whether a character precedes another (or is equal to it) in a case-insensitive manner.

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

Determines whether a character precedes another in a case-insensitive manner.

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

Determines whether two characters are equal in a case-insensitive manner.

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

Determines whether a character succeeds another (or is equal to it) in a case-insensitive manner.

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

Determines whether a character succeeds another in a case-insensitive manner.

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

Determines the equivalent lower-case character.

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

Determines whether a character is a lower-case character.

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

Determines whether a character represents a digit.

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

Determines the equivalent upper-case character.

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

Determines whether a character is an upper-case character.

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

Determines whether a character represents space.

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

Determines whether a character precedes another (or is equal to it).

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

Determines whether a character precedes another.

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

Determines whether two characters are equal.

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

Determines whether a character succeeds another (or is equal to it).

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

Determines whether a character succeeds another.

`char? : (any -> boolean)`

Determines whether a value is a character.

1.3.7 Strings

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

Translates a string into a list of 1-letter strings.

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

Formats a string, possibly embedding values.

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

Concatenates the list of 1-letter strings into one string.

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

Converts an integer in [0,55295] or [57344 1114111] to a 1-letter string.

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

Converts a s list of characters into a string.

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

Produces a string of given length from a single given character.

`replicate : (nat string -> string)`

Replicates the given string.

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

Builds a string of the given characters.

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

Converts a 1-letter string to an integer in [0,55295] or [57344, 1114111].

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

Converts a string into a list of characters.

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

Converts a string into a number, produce false if impossible.

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

Converts a string into a symbol.

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

Determines whether all 'letters' in the string are alphabetic.

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

Juxtaposes the characters of several strings.

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

Determines whether one string alphabetically precedes another (or is equal to it) in a case-insensitive manner.

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

Determines whether one string alphabetically precedes another in a case-insensitive manner.

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

Compares two strings character-wise in a case-insensitive manner.

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

Determines whether one string alphabetically succeeds another (or is equal to it) in a case-insensitive manner.

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

Determines whether one string alphabetically succeeds another in a case-insensitive manner.

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

Copies a string.

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

Extracts the ith 1-letter substring from the given one.

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

Determines the length of a string.

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

Determines whether all 'letters' in the string are lower case.

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

Determines whether all 'letters' in the string are numeric.

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

Extracts the i-the character from a string.

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

Determines whether all 'letters' in the string are upper case.

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

Determines whether all 'letters' in the string are white space.

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

Determines whether one string alphabetically precedes another (or is equal to it).

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

Determines whether one string alphabetically precedes another.

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

Compares two strings character-wise.

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

Determines whether one string alphabetically succeeds another (or is equal to it).

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

Determines whether one string alphabetically succeeds another.

`string? : (any -> boolean)`

Determines whether a value is a string.

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

Extracts the substring starting at a 0-based index up to the second 0-based index (exclusive).

1.3.8 Images

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

Determines whether two images are equal.

`image? : (any -> boolean)`

Determines whether a value is an image.

1.3.9 Misc

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

Checks whether two numbers are within some amount (the third argument) of either other.

`eof : eof`

The end-of-file value.

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

Determines whether some value is the end-of-file value.

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

Determines whether two values are equivalent from the computer's perspective (intensional).

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

Determines whether two values are structurally equal where basic values are compared with the eqv? predicate.

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

Compares like equal? on the first two arguments, except using =~ in the case of numbers.

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

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

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

signals an error, combining the given values into an error message.

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

`exit : (-> void)`

Exits the running program.

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

Returns the argument unchanged.

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

Determines 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 (name variable variable ...) expr)
            | (define name expr)
            | (define name (lambda (variable variable ...) expr))
            | (define-struct name (name ...))

       expr = (name expr expr ...)
            | (prim-op expr ...)
            | (cond [expr expr] ... [expr expr])
            | (cond [expr expr] ... [else expr])
            | (if expr expr expr)
            | (and expr expr expr ...)
            | (or expr expr expr ...)
            | name
            | 'quoted
            | `quasiquoted
            | number
            | string
            | character

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

quasiquoted = name
            | number
            | string
            | character
            | (quasiquoted ...)
            | 'quasiquoted
            | `quasiquoted
```

```
                | ,expr
                | ,@expr

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

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

        package = (string string number number)
```

An *name* or a *variable* 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 *symbol* is a quote character followed by a name. A symbol is a value, just like 0 or empty.

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

In function calls, the function appearing immediatly after the open parenthesis can be any functions defined with `define` or `define-struct`, or any one of the pre-defined functions.

2.1 Pre-Defined Variables

`empty` : `empty?`

The empty list.

`true` : `boolean?`

The true value.

```
false : boolean?
```

The false value.

2.2 Syntaxes for Beginning Student with List Abbreviations

```
'name
'part
(quote name)
(quote part)
```

A quoted name is a symbol. A quoted part is an abbreviation for a nested lists.

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

```
`name
`part
(quasiquote name)
(quasiquote part)
```

Like `quote`, but also allows escaping to expression "unquotes."

Normally, quasi-quotations are written with a backquote, `` ` ``, like `` `(apple ,(+ 1 2)) ``, but they can also be written with `quasiquote`, like `(quasiquote-elem (apple ,(+ 1 2)))`.

```
,expression
(unquote expression)
```

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

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

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

```
,@expression
(unquote-splicing expression)
```

Under a single quasiquote, `,@`*expression* 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 Common Syntaxes

The following syntaxes behave the same in the *Beginner with List Abbreviations* level as they did in the §1 "Beginning Student" level.

`(define (`*name variable variable ...*`) `*expression*`)`

Defines a function named *name*. The *expression* is the body of the function. When the function is called, the values of the arguments are inserted into the body in place of the *variable*s. The function returns the value of that new expression.

The function name's cannot be the same as that of another function or variable.

`(define `*name expression*`)`

Defines a variable called *name* with the the value of *expression*. The variable name's cannot be the same as that of another function or variable, and *name* itself must not appear in *expression*.

`(define `*name* `(lambda (`*variable variable ...*`) `*expression*`))`

An alternate way on defining functions. The *name* is the name of the function, which cannot be the same as that of another function or variable.

A `lambda` cannot be used outside of this alternate syntax.

`(define-struct `*structure-name* `(`*field-name ...*`))`

Defines a new structure called *structure-name*. The structure's fields are named by the *field-name*s. After the `define-struct`, the following new functions are available:

- `make-`*structure-name* : takes a number of arguments equal to the number of fields in the structure, and creates a new instance of that structure.
- *structure-name-field-name* : takes an instance of the structure and returns the

value in the field named by *field-name*.

- *structure-name?* : takes any value, and returns true if the value is an instance of the structure.

The name of the new functions introduced by `define-struct` must not be the same as that of other functions or variables, otherwise `define-struct` reports an error.

```
(name expression expression ...)
```

Calls the function named *name*. The value of the call is the value of *name*'s body when every one of the function's variables are replaced by the values of the corresponding *expression*s.

The function named *name* must defined before it can be called. The number of argument *expression*s must be the same as the number of arguments expected by the function.

```
(cond [question-expression answer-expression] ...)
(cond [question-expression answer-expression]
      ...
      [else answer-expression])
```

Chooses a clause based on some condition. `cond` finds the first *question-expression* that evaluates to true, then evaluates the corresponding *answer-expression*.

If none of the *question-expression*s evaluates to true, `cond`'s value is the *answer-expression* of the `else` clause. If there is no `else`, `cond` reports an error. If the result of a *question-expression* is neither true nor false, `cond` also reports an error.

`else` cannot be used outside of `cond`.

```
(if test-expression then-expression else-expression)
```

When the value of the *test-expression* is true, `if` evaluates the *then-expression*. When the test is false, `if` evaluates the *else-expression*.

If the *test-expression* is neither true nor false, `if` reports an error.

```
(or expression expression expression ...)
```

Evaluates to true if all the *expression*s are true. If any *expression* is false, the `or` expression immediately evaluates to false (and the expressions to the right of that expression are not evaluated.)

If any of the expressions evaluate to a value other than true or false, `or` reports an error.

`(and expression expression expression ...)`

Evaluates to `true` as soon as one of the *expression*s is `true` (and the expressions to the right of that expression are not evaluated.) If all of the *expression*s are `false`, the `and` expression evaluates to `false`.

If any of the expressions evaluate to a value other than `true` or `false`, `and` reports an error.

`(check-expect expression expected-expression)`

Checks that the first *expression* evaluates to the same value as the *expected-expression*.

`(check-within expression expected-expression delta-expression)`

Checks that the first *expression* evaluates to a value within *delta-expression* of the *expected-expression*. If *delta-expression* is not a number, `check-within` reports an error.

`(check-error expression match-expression)`
`(check-error expression)`

Checks that the *expression* reports an error, where the error messages matches the value of `matchexpression`, if it is present.

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

Checks that the value of the first *expression* as that of one of the following *expression*s.

`(check-range expression low-expression high-expression)`

Checks that the value of the first *expression* is a number in between the value of the *low-expression* and the *high-expression*, inclusive.

`(require string)`

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

Accesses a file in an installed library. The library name is an identifier with the same constraints as for a relative-path string (though without the quotes), 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., the current file). The first *string* names the library file, and the remaining *string*s name the collection (and sub-collection, and so on) where the file is installed. Each string is constrained in the same way as for the `(require string)` form.

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

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

2.4 Pre-defined Functions

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

---

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

Evaluates the product of all of the input numbers.

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

Evaluates the sum of the input numbers.

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

subtracts the second (and following) number(s) from the first; negate the number if there is only one argument.

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

Divides 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)`

Compares real numbers for less-than.

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

Compares real numbers for less-than or equality.

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

Compares numbers for equality.

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

Compares real numbers for greater-than.

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

Compares real numbers for greater-than or equality.

`abs : (real -> real)`

Evaluates the absolute value of a real number.

`acos : (number -> number)`

Evaluates the arccosine (inverse of cos) of a number.

`add1 : (number -> number)`

Evaluates a number one larger than a given number.

`angle : (number -> real)`

Extracts the angle from a complex number.

`asin : (number -> number)`

Evaluates the arcsine (inverse of sin) of a number.

`atan : (number (number) -> number)`

Evaluates the arctan of the given number or the ratio of the two given numbers.

`ceiling : (real -> integer)`

Determines the closest integer (exact or inexact) above a real number.

`complex? : (any -> boolean)`

Determines whether some value is complex.

`conjugate : (number -> number)`

Evaluates the conjugate of a complex number.

`cos : (number -> number)`

Evaluates the cosine of a number (radians).

`cosh : (number -> number)`

Evaluates the hyperbolic cosine of a number.

`current-seconds : (-> integer)`

Evaluates the current time in seconds elapsed (since a platform-specific starting date).

`denominator : (rat -> integer)`

Evaluates the denominator of a rational.

`e : real`

Euler's number.

`even? : (integer -> boolean)`

Determines if some integer (exact or inexact) is even or not.

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

Converts an exact number to an inexact one.

`exact? : (number -> boolean)`

Determines whether some number is exact.

`exp : (number -> number)`

Evaluates e raised to a number.

`expt : (number number -> number)`

Evaluates the power of the first to the second number.

`floor : (real -> integer)`

Determines the closest integer (exact or inexact) below a real number.

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

Evaluates the greatest common divisior of two integers (exact or inexact).

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

Extracts the imaginary part from a complex number.

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

Approximates an inexact number by an exact one.

`inexact? : (number -> boolean)`

Determines whether some number is inexact.

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

Lookups the character that corresponds to the given integer (exact only!) in the ASCII table

(if any).

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

Evaluates the integer (exact or inexact) square root of a number.

`integer? : (any -> boolean)`

Determines whether some value is an integer (exact or inexact).

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

Evaluates the least common multiple of two integers (exact or inexact).

`log : (number -> number)`

Evaluates the base-e logarithm of a number.

`magnitude : (number -> real)`

Determines the magnitude of a complex number.

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

Creates a complex from a magnitude and angle.

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

Creates a complex from a real and an imaginary part.

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

Determines the largest number.

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

Determines the smallest number.

`modulo : (integer integer -> integer)`

Finds the remainder of the division of the first number by the second; try (modulo 4 3) (modulo 4 -3).

`negative? : (number -> boolean)`

Determines if some value is strictly smaller than zero.

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

Converts a number to a string.

`number? : (any -> boolean)`

Determines whether some value is a number.

`numerator : (rat -> integer)`

Evaluates the numerator of a rational.

`odd? : (integer -> boolean)`

Determines if some integer (exact or inexact) is odd or not.

`pi : real`

The ratio of a circle's circumference to its diameter.

`positive? : (number -> boolean)`

Determines if some value is strictly larger than zero.

`quotient : (integer integer -> integer)`

Divides the first integer (exact or inexact) into the second; try (quotient 3 4) and (quotient 4 3).

`random : (integer -> integer)`

Generates a random natural number less than some given integer (exact only!).

`rational? : (any -> boolean)`

Determines whether some value is a rational number.

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

Extracts the real part from a complex number.

`real? : (any -> boolean)`

Determines whether some value is a real number.

`remainder : (integer integer -> integer)`

Determines the remainder of dividing the first by the second integer (exact or inexact).

`round : (real -> integer)`

Rounds a real number to an integer (rounds to even to break ties).

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

Evaluates the sign of a real number.

`sin : (number -> number)`

Evaluates the sine of a number (radians).

`sinh : (number -> number)`

Evaluates the hyperbolic sine of a number.

`sqr : (number -> number)`

Evaluates the square of a number.

`sqrt : (number -> number)`

Evaluates the square root of a number.

`sub1 : (number -> number)`

Evaluates a number one smaller than a given number.

`tan : (number -> number)`

Evaluates the tangent of a number (radians).

`zero? : (number -> boolean)`

Determines if some value is zero or not.

2.4.2 Booleans

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

Determines whether two booleans are equal.

`boolean? : (any -> boolean)`

Determines whether some value is a boolean.

`false? : (any -> boolean)`

Determines whether a value is false.

`not : (boolean -> boolean)`

Evaluates the negation of a boolean value.

2.4.3 Symbols

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

Converts a symbol to a string.

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

Determines whether two symbols are equal.

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

Determines whether some value is a symbol.

2.4.4 Lists

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

Creates a single list from several, by juxtaposition of the items.

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

Determines 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)
```

Selects 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))
```

Selects the rest of the first list in the first list of a list.

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

Selects the first item of the first list in a list.

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

Selects the second item of the first list of a list.

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

Selects the fourth item of a non-empty list.

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

Selects the third item of a non-empty list.

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

Selects the second item of a non-empty list.

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

Selects the first item of a non-empty list.

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

Selects 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))
```

Selects the rest of the first list in the rest of a list.

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

Selects the rest of a non-empty list in a list.

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

Selects the rest of the rest of the first list of a list.

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

Selects the rest of the rest of the rest of a list.

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

Selects the rest of the rest of a list.

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

Selects the rest of a non-empty list.

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

Constructs a list.

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

Determines whether some value is a constructed list.

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

Selects the eighth item of a non-empty list.

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

Determines whether some value is the empty list.

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

Selects the fifth item of a non-empty list.

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

Selects the first item of a non-empty list.

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

Selects the fourth item of a non-empty list.

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

Evaluates the number of items on a list.

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

Constructs a list of its arguments.

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

Constructs a list by adding multiple items to a list.

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

Extracts the indexed item from the list.

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

Constructs a list of k (the first argument) copies of x (the second argument).

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

Determines whether some value is on the list (comparing values with equal?).

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

Determines whether some value is on the list (comparing values with equal?).

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

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

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

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

```
null : empty
```

The empty list.

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

Determines whether some value is the empty list.

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

Constructs 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))
```

Selects the rest of a non-empty list.

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

Creates a reversed version of a list.

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

Selects the second item of a non-empty list.

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

Selects the seventh item of a non-empty list.

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

Selects the sixth item of a non-empty list.

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

Selects the third item of a non-empty list.

2.4.5 Posns

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

Constructs a posn.

`posn : signature`

Signature for posns.

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

Extracts the x component of a posn.

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

Extracts the y component of a posn.

`posn? : (anything -> boolean)`

Determines if its input is a posn.

2.4.6 Characters

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

Lookups the number that corresponds to the given character in the ASCII table (if any).

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

Determines whether a character represents an alphabetic character.

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

Determines whether a character precedes another (or is equal to it) in a case-insensitive manner.

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

Determines whether a character precedes another in a case-insensitive manner.

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

Determines whether two characters are equal in a case-insensitive manner.

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

Determines whether a character succeeds another (or is equal to it) in a case-insensitive manner.

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

Determines whether a character succeeds another in a case-insensitive manner.

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

Determines the equivalent lower-case character.

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

Determines whether a character is a lower-case character.

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

Determines whether a character represents a digit.

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

Determines the equivalent upper-case character.

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

Determines whether a character is an upper-case character.

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

Determines whether a character represents space.

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

Determines whether a character precedes another (or is equal to it).

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

Determines whether a character precedes another.

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

Determines whether two characters are equal.

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

Determines whether a character succeeds another (or is equal to it).

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

Determines whether a character succeeds another.

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

Determines whether a value is a character.

2.4.7 Strings

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

Translates a string into a list of 1-letter strings.

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

Formats a string, possibly embedding values.

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

Concatenates the list of 1-letter strings into one string.

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

Converts an integer in [0,55295] or [57344 1114111] to a 1-letter string.

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

Converts a s list of characters into a string.

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

Produces a string of given length from a single given character.

`replicate : (nat string -> string)`

Replicates the given string.

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

Builds a string of the given characters.

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

Converts a 1-letter string to an integer in [0,55295] or [57344, 1114111].

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

Converts a string into a list of characters.

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

Converts a string into a number, produce false if impossible.

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

Converts a string into a symbol.

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

Determines whether all 'letters' in the string are alphabetic.

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

Juxtaposes the characters of several strings.

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

Determines whether one string alphabetically precedes another (or is equal to it) in a case-insensitive manner.

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

Determines whether one string alphabetically precedes another in a case-insensitive manner.

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

Compares two strings character-wise in a case-insensitive manner.

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

Determines whether one string alphabetically succeeds another (or is equal to it) in a case-insensitive manner.

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

Determines whether one string alphabetically succeeds another in a case-insensitive manner.

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

Copies a string.

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

Extracts the ith 1-letter substring from the given one.

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

Determines the length of a string.

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

Determines whether all 'letters' in the string are lower case.

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

Determines whether all 'letters' in the string are numeric.

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

Extracts the i-the character from a string.

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

Determines whether all 'letters' in the string are upper case.

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

Determines whether all 'letters' in the string are white space.

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

Determines whether one string alphabetically precedes another (or is equal to it).

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

Determines whether one string alphabetically precedes another.

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

Compares two strings character-wise.

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

Determines whether one string alphabetically succeeds another (or is equal to it).

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

Determines whether one string alphabetically succeeds another.

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

Determines whether a value is a string.

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

Extracts the substring starting at a 0-based index up to the second 0-based index (exclusive).

2.4.8 Images

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

Determines whether two images are equal.

`image? : (any -> boolean)`

Determines whether a value is an image.

2.4.9 Misc

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

Checks whether two numbers are within some amount (the third argument) of either other.

`eof : eof`

The end-of-file value.

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

Determines whether some value is the end-of-file value.

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

Determines whether two values are equivalent from the computer's perspective (intensional).

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

Determines whether two values are structurally equal where basic values are compared with the eqv? predicate.

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

Compares like equal? on the first two arguments, except using =~ in the case of numbers.

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

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

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

signals an error, combining the given values into an error message.

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

`exit` : `(-> void)`

Exits the running program.

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

Returns the argument unchanged.

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

Determines whether some value is a structure.

3 Intermediate Student

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

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

 definition = (define (name variable variable ...) expr)
            | (define name expr)
            | (define name (lambda (variable variable ...) expr))
            | (define-struct name (name ...))

       expr = (local [definition ...] expr)
            | (letrec ([name expr-for-let] ...) expr)
            | (let ([name expr-for-let] ...) expr)
            | (let* ([name expr-for-let] ...) expr)
            | (name expr expr ...)
            | (cond [expr expr] ... [expr expr])
            | (cond [expr expr] ... [else expr])
            | (if expr expr expr)
            | (and expr expr expr ...)
            | (or expr expr expr ...)
            | (time expr)
            | name
            | 'quoted
            | `quasiquoted
            | number
            | string
            | character

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

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

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

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

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

    package = (string string number number)
```

An *name* or a *variable* 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 *symbol* is a quote character followed by a name. A symbol is a value, just like 0 or empty.

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

In function calls, the function appearing immediatly after the open parenthesis can be any functions defined with `define` or `define-struct`, or any one of the pre-defined functions.

3.1 Pre-Defined Variables

`empty` : `empty?`

The empty list.

`true` : `boolean?`

The true value.

`false` : `boolean?`

The false value.

3.2 Syntax for Intermediate

`(local [`*`definition ...`*`] `*`expression`*`)`

Groups related definitions for use in *`expression`*. Each *`definition`* can be either a `define` or a `define-struct`.

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

`(letrec ([`*`name expr-for-let`*`] ...) `*`expression`*`)`

Like `local`, but with a simpler syntax. Each *`name`* defines a variable (or a function) with the value of the corresponding *`expr-for-let`*. If *`expr-for-let`* is a `lambda`, `letrec` defines a function, otherwise it defines a variable.

`(let* ([`*`name expr-for-let`*`] ...) `*`expression`*`)`

Like `letrec`, but each *`name`* can only be used in *`expression`*, and in *`expr-for-let`*s occuring after that *`name`*.

`(let ([`*`name expr-for-let`*`] ...) `*`expression`*`)`

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

(time *expression*)

Measures the time taken to evaluate *expression*. After evaluating *expression*, time prints out the time taken by the evaluation (including real time, time taken by the CPU, and the time spent collecting free memory). The value of time is the same as that of *expression*.

3.3 Common Syntaxes

The following syntaxes behave the same in the *Intermediate* level as they did in the §2 "Beginning Student with List Abbreviations" level.

'*name*
'*part*
(quote *name*)
(quote *part*)

A quoted name is a symbol. A quoted part is an abbreviation for a nested lists.

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

`*name*
`*part*
(quasiquote *name*)
(quasiquote *part*)

Like quote, but also allows escaping to expression "unquotes."

Normally, quasi-quotations are written with a backquote, `, like `(apple ,(+ 1 2)), but they can also be written with quasiquote, like (quasiquote-elem (apple ,(+ 1 2))).

,*expression*
(unquote *expression*)

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

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

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

`,@`*expression*
`(unquote-splicing` *expression*`)`

Under a single quasiquote, `,@`*expression* 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`.

`(define (`*name variable variable ...*`)` *expression*`)`

Defines a function named *name*. The *expression* is the body of the function. When the function is called, the values of the arguments are inserted into the body in place of the *variable*s. The function returns the value of that new expression.

The function name's cannot be the same as that of another function or variable.

`(define` *name expression*`)`

Defines a variable called *name* with the the value of *expression*. The variable name's cannot be the same as that of another function or variable, and *name* itself must not appear in *expression*.

`(define` *name* `(lambda (`*variable variable ...*`)` *expression*`))`

An alternate way on defining functions. The *name* is the name of the function, which cannot be the same as that of another function or variable.

A `lambda` cannot be used outside of this alternate syntax.

`(define-struct` *structure-name* `(`*field-name ...*`))`

Defines a new structure called *structure-name*. The structure's fields are named by the *field-name*s. After the `define-struct`, the following new functions are available:

- `make-`*structure-name* : takes a number of arguments equal to the number of fields

in the structure, and creates a new instance of that structure.

- *structure-name-field-name* : takes an instance of the structure and returns the value in the field named by *field-name*.
- *structure-name?* : takes any value, and returns `true` if the value is an instance of the structure.

The name of the new functions introduced by `define-struct` must not be the same as that of other functions or variables, otherwise `define-struct` reports an error.

```
(name expression expression ...)
```

Calls the function named *name*. The value of the call is the value of *name*'s body when every one of the function's variables are replaced by the values of the corresponding *expression*s.

The function named *name* must defined before it can be called. The number of argument *expression*s must be the same as the number of arguments expected by the function.

```
(cond [question-expression answer-expression] ...)
(cond [question-expression answer-expression]
      ...
      [else answer-expression])
```

Chooses a clause based on some condition. `cond` finds the first *question-expression* that evaluates to `true`, then evaluates the corresponding *answer-expression*.

If none of the *question-expression*s evaluates to `true`, `cond`'s value is the *answer-expression* of the `else` clause. If there is no `else`, `cond` reports an error. If the result of a *question-expression* is neither `true` nor `false`, `cond` also reports an error.

`else` cannot be used outside of `cond`.

```
(if test-expression then-expression else-expression)
```

When the value of the *test-expression* is `true`, `if` evaluates the *then-expression*. When the test is `false`, `if` evaluates the *else-expression*.

If the *test-expression* is neither `true` nor `false`, `if` reports an error.

```
(or expression expression expression ...)
```

Evaluates to `true` if all the *expression*s are `true`. If any *expression* is `false`, the `or` expression immediately evaluates to `false` (and the expressions to the right of that expres-

sion are not evaluated.)

If any of the expressions evaluate to a value other than `true` or `false`, `or` reports an error.

`(and` *`expression expression expression ...`*`)`

Evaluates to `true` as soon as one of the *`expression`*s is `true` (and the expressions to the right of that expression are not evaluated.) If all of the *`expression`*s are `false`, the `and` expression evaluates to `false`.

If any of the expressions evaluate to a value other than `true` or `false`, `and` reports an error.

`(check-expect` *`expression expected-expression`*`)`

Checks that the first *`expression`* evaluates to the same value as the *`expected-expression`*.

`(check-within` *`expression expected-expression delta-expression`*`)`

Checks that the first *`expression`* evaluates to a value within *`delta-expression`* of the *`expected-expression`*. If *`delta-expression`* is not a number, `check-within` reports an error.

`(check-error` *`expression match-expression`*`)`
`(check-error` *`expression`*`)`

Checks that the *`expression`* reports an error, where the error messages matches the value of `matchexpression`, if it is present.

`(check-member-of` *`expression expression expression ...`*`)`

Checks that the value of the first *`expression`* as that of one of the following *`expression`*s.

`(check-range` *`expression low-expression high-expression`*`)`

Checks that the value of the first *`expression`* is a number in between the value of the *`low-expression`* and the *`high-expression`*, inclusive.

`(require` *`string`*`)`

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

Accesses a file in an installed library. The library name is an identifier with the same constraints as for a relative-path string (though without the quotes), 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., the current file). The first *string* names the library file, and the remaining *string*s name the collection (and sub-collection, and so on) where the file is installed. Each string is constrained in the same way as for the `(require` *string*`)` form.

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

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

3.4 Pre-defined Functions

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

---

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

Compares real numbers for less-than.

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

Compares real numbers for less-than or equality.

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

Compares numbers for equality.

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

Compares real numbers for greater-than.

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

Compares real numbers for greater-than or equality.

`abs : (real -> real)`

Evaluates the absolute value of a real number.

`acos : (number -> number)`

Evaluates the arccosine (inverse of cos) of a number.

`add1 : (number -> number)`

Evaluates a number one larger than a given number.

`angle : (number -> real)`

Extracts the angle from a complex number.

`asin : (number -> number)`

Evaluates the arcsine (inverse of sin) of a number.

`atan : (number (number) -> number)`

Evaluates the arctan of the given number or the ratio of the two given numbers.

`ceiling : (real -> integer)`

Determines the closest integer (exact or inexact) above a real number.

`complex? : (any -> boolean)`

Determines whether some value is complex.

`conjugate : (number -> number)`

Evaluates the conjugate of a complex number.

`cos : (number -> number)`

Evaluates the cosine of a number (radians).

`cosh : (number -> number)`

Evaluates the hyperbolic cosine of a number.

`current-seconds : (-> integer)`

Evaluates the current time in seconds elapsed (since a platform-specific starting date).

`denominator : (rat -> integer)`

Evaluates the denominator of a rational.

`e : real`

Euler's number.

`even? : (integer -> boolean)`

Determines if some integer (exact or inexact) is even or not.

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

Converts an exact number to an inexact one.

`exact? : (number -> boolean)`

Determines whether some number is exact.

`exp : (number -> number)`

Evaluates e raised to a number.

`expt : (number number -> number)`

Evaluates the power of the first to the second number.

`floor : (real -> integer)`

Determines the closest integer (exact or inexact) below a real number.

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

Evaluates the greatest common divisior of two integers (exact or inexact).

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

Extracts the imaginary part from a complex number.

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

Approximates an inexact number by an exact one.

`inexact? : (number -> boolean)`

Determines whether some number is inexact.

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

Lookups the character that corresponds to the given integer (exact only!) in the ASCII table (if any).

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

Evaluates the integer (exact or inexact) square root of a number.

`integer? : (any -> boolean)`

Determines whether some value is an integer (exact or inexact).

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

Evaluates the least common multiple of two integers (exact or inexact).

`log : (number -> number)`

Evaluates the base-e logarithm of a number.

`magnitude : (number -> real)`

Determines the magnitude of a complex number.

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

Creates a complex from a magnitude and angle.

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

Creates a complex from a real and an imaginary part.

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

Determines the largest number.

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

Determines the smallest number.

`modulo : (integer integer -> integer)`

Finds the remainder of the division of the first number by the second; try (modulo 4 3) (modulo 4 -3).

`negative? : (number -> boolean)`

Determines if some value is strictly smaller than zero.

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

Converts a number to a string.

`number? : (any -> boolean)`

Determines whether some value is a number.

`numerator : (rat -> integer)`

Evaluates the numerator of a rational.

`odd? : (integer -> boolean)`

Determines if some integer (exact or inexact) is odd or not.

`pi : real`

The ratio of a circle's circumference to its diameter.

`positive? : (number -> boolean)`

Determines if some value is strictly larger than zero.

`quotient : (integer integer -> integer)`

Divides the first integer (exact or inexact) into the second; try (quotient 3 4) and (quotient 4 3).

`random : (integer -> integer)`

Generates a random natural number less than some given integer (exact only!).

`rational? : (any -> boolean)`

Determines whether some value is a rational number.

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

Extracts the real part from a complex number.

`real? : (any -> boolean)`

Determines whether some value is a real number.

`remainder : (integer integer -> integer)`

Determines the remainder of dividing the first by the second integer (exact or inexact).

`round : (real -> integer)`

Rounds a real number to an integer (rounds to even to break ties).

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

Evaluates the sign of a real number.

`sin : (number -> number)`

Evaluates the sine of a number (radians).

`sinh : (number -> number)`

Evaluates the hyperbolic sine of a number.

`sqr : (number -> number)`

Evaluates the square of a number.

`sqrt : (number -> number)`

Evaluates the square root of a number.

`sub1 : (number -> number)`

Evaluates a number one smaller than a given number.

`tan : (number -> number)`

Evaluates the tangent of a number (radians).

`zero? : (number -> boolean)`

Determines if some value is zero or not.

3.4.2 Booleans

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

Determines whether two booleans are equal.

`boolean? : (any -> boolean)`

Determines whether some value is a boolean.

`false? : (any -> boolean)`

Determines whether a value is false.

`not : (boolean -> boolean)`

Evaluates the negation of a boolean value.

3.4.3 Symbols

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

Converts a symbol to a string.

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

Determines whether two symbols are equal.

`symbol? : (any -> boolean)`

Determines whether some value is a symbol.

3.4.4 Lists

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

Creates a single list from several, by juxtaposition of the items.

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

Determines 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)
```

Selects 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))
```

Selects the rest of the first list in the first list of a list.

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

Selects the first item of the first list in a list.

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

Selects the second item of the first list of a list.

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

Selects the fourth item of a non-empty list.

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

Selects the third item of a non-empty list.

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

Selects the second item of a non-empty list.

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

Selects the first item of a non-empty list.

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

Selects 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))
```

Selects the rest of the first list in the rest of a list.

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

Selects the rest of a non-empty list in a list.

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

Selects the rest of the rest of the first list of a list.

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

Selects the rest of the rest of the rest of a list.

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

Selects the rest of the rest of a list.

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

Selects the rest of a non-empty list.

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

Constructs a list.

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

Determines whether some value is a constructed list.

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

Selects the eighth item of a non-empty list.

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

Determines whether some value is the empty list.

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

Selects the fifth item of a non-empty list.

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

Selects the first item of a non-empty list.

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

Selects the fourth item of a non-empty list.

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

Evaluates the number of items on a list.

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

Constructs a list of its arguments.

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

Constructs a list by adding multiple items to a list.

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

Extracts the indexed item from the list.

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

Constructs a list of k (the first argument) copies of x (the second argument).

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

Determines whether some value is on the list (comparing values with equal?).

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

Determines whether some value is on the list (comparing values with equal?).

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

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

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

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

`null` : `empty`

The empty list.

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

Determines whether some value is the empty list.

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

Constructs 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))
```

Selects the rest of a non-empty list.

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

Creates a reversed version of a list.

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

Selects the second item of a non-empty list.

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

Selects the seventh item of a non-empty list.

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

Selects the sixth item of a non-empty list.

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

Selects the third item of a non-empty list.

3.4.5 Posns

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

Constructs a posn.

```
posn : signature
```

Signature for posns.

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

Extracts the x component of a posn.

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

Extracts the y component of a posn.

`posn? : (anything -> boolean)`

Determines if its input is a posn.

3.4.6 Characters

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

Lookups the number that corresponds to the given character in the ASCII table (if any).

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

Determines whether a character represents an alphabetic character.

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

Determines whether a character precedes another (or is equal to it) in a case-insensitive manner.

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

Determines whether a character precedes another in a case-insensitive manner.

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

Determines whether two characters are equal in a case-insensitive manner.

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

Determines whether a character succeeds another (or is equal to it) in a case-insensitive manner.

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

Determines whether a character succeeds another in a case-insensitive manner.

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

Determines the equivalent lower-case character.

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

Determines whether a character is a lower-case character.

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

Determines whether a character represents a digit.

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

Determines the equivalent upper-case character.

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

Determines whether a character is an upper-case character.

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

Determines whether a character represents space.

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

Determines whether a character precedes another (or is equal to it).

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

Determines whether a character precedes another.

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

Determines whether two characters are equal.

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

Determines whether a character succeeds another (or is equal to it).

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

Determines whether a character succeeds another.

`char? : (any -> boolean)`

Determines whether a value is a character.

3.4.7 Strings

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

Translates a string into a list of 1-letter strings.

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

Formats a string, possibly embedding values.

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

Concatenates the list of 1-letter strings into one string.

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

Converts an integer in [0,55295] or [57344 1114111] to a 1-letter string.

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

Converts a s list of characters into a string.

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

Produces a string of given length from a single given character.

`replicate : (nat string -> string)`

Replicates the given string.

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

Builds a string of the given characters.

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

Converts a 1-letter string to an integer in [0,55295] or [57344, 1114111].

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

Converts a string into a list of characters.

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

Converts a string into a number, produce false if impossible.

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

Converts a string into a symbol.

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

Determines whether all 'letters' in the string are alphabetic.

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

Juxtaposes the characters of several strings.

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

Determines whether one string alphabetically precedes another (or is equal to it) in a case-insensitive manner.

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

Determines whether one string alphabetically precedes another in a case-insensitive manner.

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

Compares two strings character-wise in a case-insensitive manner.

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

Determines whether one string alphabetically succeeds another (or is equal to it) in a case-insensitive manner.

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

Determines whether one string alphabetically succeeds another in a case-insensitive manner.

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

Copies a string.

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

Extracts the ith 1-letter substring from the given one.

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

Determines the length of a string.

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

Determines whether all 'letters' in the string are lower case.

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

Determines whether all 'letters' in the string are numeric.

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

Extracts the i-the character from a string.

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

Determines whether all 'letters' in the string are upper case.

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

Determines whether all 'letters' in the string are white space.

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

Determines whether one string alphabetically precedes another (or is equal to it).

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

Determines whether one string alphabetically precedes another.

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

Compares two strings character-wise.

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

Determines whether one string alphabetically succeeds another (or is equal to it).

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

Determines whether one string alphabetically succeeds another.

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

Determines whether a value is a string.

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

Extracts the substring starting at a 0-based index up to the second 0-based index (exclusive).

3.4.8 Images

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

Determines whether two images are equal.

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

Determines whether a value is an image.

3.4.9 Misc

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

Checks whether two numbers are within some amount (the third argument) of either other.

`eof : eof`

The end-of-file value.

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

Determines whether some value is the end-of-file value.

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

Determines whether two values are equivalent from the computer's perspective (intensional).

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

Determines whether two values are structurally equal where basic values are compared with the eqv? predicate.

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

Compares like equal? on the first two arguments, except using =~ in the case of numbers.

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

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

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

signals an error, combining the given values into an error message.

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

`exit : (-> void)`

Exits the running program.

`identity : (any -> any)`

Returns the argument unchanged.

`struct? : (any -> boolean)`

Determines whether some value is a structure.

3.4.10 Numbers (relaxed conditions)

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

Multiplys all given numbers.

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

Adds all given numbers.

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

Subtracts from the first all remaining numbers.

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

Divides the first by all remaining numbers.

3.4.11 Higher-Order Functions

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

(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)
```

Applies a function using items from a list as the arguments.

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

Finds the (first) element of the list that maximizes the output of the function.

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

Finds the (first) element of the list that minimizes the output of the function.

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

(build-list n f) = (list (f 0) ... (f (- n 1)))

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

(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))
```

Composes a sequence of procedures into a single procedure.

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

Constructs a list from all those items on a list for which the predicate holds.

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

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

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

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

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

Applies a function to each item on one or more lists for effect only.

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

Constructs a new list by applying a function to each item on one or more existing lists.

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

Produces true if the function given as the first argument produces a non-false value for any item in the second argument.

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

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

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

Produces true if the value is a procedure.

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

Constructs a list from all items on a list in an order according to a predicate.

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

Constructs a list from all items on a list in an order according to a predicate.

4 Intermediate Student with Lambda

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

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

 definition = (define (name variable variable ...) expr)
            | (define name expr)
            | (define-struct name (name ...))

       expr = (lambda (variable variable ...) expr)
            | (λ (variable variable ...) expr)
            | (local [definition ...] expr)
            | (letrec ([name expr] ...) expr)
            | (let ([name expr] ...) expr)
            | (let* ([name expr] ...) expr)
            | (expr expr expr ...)
            | (cond [expr expr] ... [expr expr])
            | (cond [expr expr] ... [else expr])
            | (if expr expr expr)
            | (and expr expr expr ...)
            | (or expr expr expr ...)
            | (time expr)
            | name
            | prim-op
            | 'quoted
            | `quasiquoted
            | number
            | string
            | character

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

quasiquoted = name
```

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

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

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

        package = (string string number number)
```

An *name* or a *variable* 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 *symbol* is a quote character followed by a name. A symbol is a value, just like 0 or empty.

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

In function calls, the function appearing immediatly after the open parenthesis can be any functions defined with `define` or `define-struct`, or any one of the pre-defined functions.

4.1 Pre-Defined Variables

```
empty : empty?
```

The empty list.

`true` : `boolean?`

The true value.

`false` : `boolean?`

The false value.

4.2 Syntax for Intermediate with Lambda

(`lambda` (*`variable variable ...`*) *`expression`*)

Creates a function that takes as many arguments as given *`variable`*s, and whose body is *`expression`*.

(λ (*`variable variable ...`*) *`expression`*)

The Greek letter λ is a synonym for `lambda`.

(*`expression expression expression ...`*)

Calls the function that results from evaluating the first *`expression`*. The value of the call is the value of function's body when every instance of `name`'s variables are replaced by the values of the corresponding *`expression`*s.

The function being called must come from either a definition appearing before the function call, or from a `lambda` expression. The number of argument *`expression`*s must be the same as the number of arguments expected by the function.

(`local` [*`definition ...`*] *`expression`*)

Groups related definitions for use in *`expression`*. Each *`definition`* can be either a `define` or a `define-struct`.

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

the inner one.

```
(letrec ([name expr-for-let] ...) expression)
```

Like `local`, but with a simpler syntax. Each *name* defines a variable (or a function) with the value of the corresponding *expr-for-let*. If *expr-for-let* is a `lambda`, `letrec` defines a function, otherwise it defines a variable.

```
(let* ([name expr-for-let] ...) expression)
```

Like `letrec`, but each *name* can only be used in *expression*, and in *expr-for-let*s occuring after that *name*.

```
(let ([name expr-for-let] ...) expression)
```

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

```
(time expression)
```

Measures the time taken to evaluate *expression*. After evaluating *expression*, `time` prints out the time taken by the evaluation (including real time, time taken by the CPU, and the time spent collecting free memory). The value of `time` is the same as that of *expression*.

4.3 Common Syntaxes

The following syntaxes behave the same in the *Intermediate with Lambda* level as they did in the §3 “Intermediate Student” level.

```
(define (name variable variable ...) expression)
```

Defines a function named *name*. The *expression* is the body of the function. When the function is called, the values of the arguments are inserted into the body in place of the *variable*s. The function returns the value of that new expression.

The function name’s cannot be the same as that of another function or variable.

```
(define name expression)
```

Defines a variable called *name* with the the value of *expression*. The variable name’s

cannot be the same as that of another function or variable, and *name* itself must not appear in *expression*.

```
(define-struct structure-name (field-name ...))
```

Defines a new structure called *structure-name*. The structure's fields are named by the *field-name*s. After the `define-struct`, the following new functions are available:

- `make-`*structure-name* : takes a number of arguments equal to the number of fields in the structure, and creates a new instance of that structure.
- *structure-name-field-name* : takes an instance of the structure and returns the value in the field named by *field-name*.
- *structure-name*`?` : takes any value, and returns `true` if the value is an instance of the structure.

The name of the new functions introduced by `define-struct` must not be the same as that of other functions or variables, otherwise `define-struct` reports an error.

```
(cond [question-expression answer-expression] ...)
(cond [question-expression answer-expression]
      ...
      [else answer-expression])
```

Chooses a clause based on some condition. `cond` finds the first *question-expression* that evaluates to `true`, then evaluates the corresponding *answer-expression*.

If none of the *question-expression*s evaluates to `true`, `cond`'s value is the *answer-expression* of the `else` clause. If there is no `else`, `cond` reports an error. If the result of a *question-expression* is neither `true` nor `false`, `cond` also reports an error.

`else` cannot be used outside of `cond`.

```
(if test-expression then-expression else-expression)
```

When the value of the *test-expression* is `true`, `if` evaluates the *then-expression*. When the test is `false`, `if` evaluates the *else-expression*.

If the *test-expression* is neither `true` nor `false`, `if` reports an error.

```
(or expression expression expression ...)
```

Evaluates to `true` if all the *expression*s are `true`. If any *expression* is `false`, the `or`

expression immediately evaluates to `false` (and the expressions to the right of that expression are not evaluated.)

If any of the expressions evaluate to a value other than `true` or `false`, `or` reports an error.

```
(and expression expression expression ...)
```

Evaluates to `true` as soon as one of the *expression*s is `true` (and the expressions to the right of that expression are not evaluated.) If all of the *expression*s are `false`, the `and` expression evaluates to `false`.

If any of the expressions evaluate to a value other than `true` or `false`, `and` reports an error.

```
(check-expect expression expected-expression)
```

Checks that the first *expression* evaluates to the same value as the *expected-expression*.

```
(check-within expression expected-expression delta-expression)
```

Checks that the first *expression* evaluates to a value within *delta-expression* of the *expected-expression*. If *delta-expression* is not a number, `check-within` reports an error.

```
(check-error expression match-expression)
(check-error expression)
```

Checks that the *expression* reports an error, where the error messages matches the value of `matchexpression`, if it is present.

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

Checks that the value of the first *expression* as that of one of the following *expression*s.

```
(check-range expression low-expression high-expression)
```

Checks that the value of the first *expression* is a number in between the value of the *low-expression* and the *high-expression*, inclusive.

```
(require string)
```

Makes the definitions of the module specified by *string* available in the current module

(i.e., the current file), where *string* refers to a file relative to the current 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-name*)

Accesses a file in an installed library. The library name is an identifier with the same constraints as for a relative-path string (though without the quotes), 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., the current file). The first *string* names the library file, and the remaining *string*s name the collection (and sub-collection, and so on) where the file is installed. Each string is constrained in the same way as for the (require *string*) form.

(require (planet *string* (*string string number number*)))

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

4.4 Pre-defined Functions

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

---

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

Compares real numbers for less-than.

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

Compares real numbers for less-than or equality.

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

Compares numbers for equality.

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

Compares real numbers for greater-than.

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

Compares real numbers for greater-than or equality.

`abs : (real -> real)`

Evaluates the absolute value of a real number.

`acos : (number -> number)`

Evaluates the arccosine (inverse of cos) of a number.

`add1 : (number -> number)`

Evaluates a number one larger than a given number.

`angle : (number -> real)`

Extracts the angle from a complex number.

`asin : (number -> number)`

Evaluates the arcsine (inverse of sin) of a number.

`atan : (number (number) -> number)`

Evaluates the arctan of the given number or the ratio of the two given numbers.

`ceiling : (real -> integer)`

Determines the closest integer (exact or inexact) above a real number.

`complex? : (any -> boolean)`

Determines whether some value is complex.

`conjugate : (number -> number)`

Evaluates the conjugate of a complex number.

`cos : (number -> number)`

Evaluates the cosine of a number (radians).

`cosh : (number -> number)`

Evaluates the hyperbolic cosine of a number.

`current-seconds : (-> integer)`

Evaluates the current time in seconds elapsed (since a platform-specific starting date).

`denominator : (rat -> integer)`

Evaluates the denominator of a rational.

`e : real`

Euler's number.

`even? : (integer -> boolean)`

Determines if some integer (exact or inexact) is even or not.

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

Converts an exact number to an inexact one.

`exact? : (number -> boolean)`

Determines whether some number is exact.

`exp : (number -> number)`

Evaluates e raised to a number.

`expt : (number number -> number)`

Evaluates the power of the first to the second number.

`floor : (real -> integer)`

Determines the closest integer (exact or inexact) below a real number.

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

Evaluates the greatest common divisior of two integers (exact or inexact).

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

Extracts the imaginary part from a complex number.

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

Approximates an inexact number by an exact one.

`inexact? : (number -> boolean)`

Determines whether some number is inexact.

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

Lookups the character that corresponds to the given integer (exact only!) in the ASCII table (if any).

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

Evaluates the integer (exact or inexact) square root of a number.

`integer? : (any -> boolean)`

Determines whether some value is an integer (exact or inexact).

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

Evaluates the least common multiple of two integers (exact or inexact).

`log : (number -> number)`

Evaluates the base-e logarithm of a number.

`magnitude : (number -> real)`

Determines the magnitude of a complex number.

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

Creates a complex from a magnitude and angle.

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

Creates a complex from a real and an imaginary part.

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

Determines the largest number.

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

Determines the smallest number.

`modulo : (integer integer -> integer)`

Finds the remainder of the division of the first number by the second; try (modulo 4 3) (modulo 4 -3).

`negative? : (number -> boolean)`

Determines if some value is strictly smaller than zero.

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

Converts a number to a string.

`number? : (any -> boolean)`

Determines whether some value is a number.

`numerator : (rat -> integer)`

Evaluates the numerator of a rational.

`odd? : (integer -> boolean)`

Determines if some integer (exact or inexact) is odd or not.

`pi : real`

The ratio of a circle's circumference to its diameter.

`positive? : (number -> boolean)`

Determines if some value is strictly larger than zero.

`quotient : (integer integer -> integer)`

Divides the first integer (exact or inexact) into the second; try (quotient 3 4) and (quotient 4 3).

`random : (integer -> integer)`

Generates a random natural number less than some given integer (exact only!).

`rational? : (any -> boolean)`

Determines whether some value is a rational number.

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

Extracts the real part from a complex number.

`real? : (any -> boolean)`

Determines whether some value is a real number.

`remainder : (integer integer -> integer)`

Determines the remainder of dividing the first by the second integer (exact or inexact).

`round : (real -> integer)`

Rounds a real number to an integer (rounds to even to break ties).

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

Evaluates the sign of a real number.

`sin : (number -> number)`

Evaluates the sine of a number (radians).

`sinh : (number -> number)`

Evaluates the hyperbolic sine of a number.

`sqr : (number -> number)`

Evaluates the square of a number.

`sqrt : (number -> number)`

Evaluates the square root of a number.

`sub1 : (number -> number)`

Evaluates a number one smaller than a given number.

`tan : (number -> number)`

Evaluates the tangent of a number (radians).

`zero? : (number -> boolean)`

Determines if some value is zero or not.

4.4.2 Booleans

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

Determines whether two booleans are equal.

`boolean? : (any -> boolean)`

Determines whether some value is a boolean.

`false? : (any -> boolean)`

Determines whether a value is false.

`not : (boolean -> boolean)`

Evaluates the negation of a boolean value.

4.4.3 Symbols

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

Converts a symbol to a string.

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

Determines whether two symbols are equal.

`symbol? : (any -> boolean)`

Determines whether some value is a symbol.

4.4.4 Lists

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

Creates a single list from several, by juxtaposition of the items.

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

Determines 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)
```

Selects 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))
```

Selects the rest of the first list in the first list of a list.

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

Selects the first item of the first list in a list.

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

Selects the second item of the first list of a list.

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

Selects the fourth item of a non-empty list.

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

Selects the third item of a non-empty list.

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

Selects the second item of a non-empty list.

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

Selects the first item of a non-empty list.

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

Selects 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))
```

Selects the rest of the first list in the rest of a list.

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

Selects the rest of a non-empty list in a list.

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

Selects the rest of the rest of the first list of a list.

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

Selects the rest of the rest of the rest of a list.

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

Selects the rest of the rest of a list.

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

Selects the rest of a non-empty list.

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

Constructs a list.

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

Determines whether some value is a constructed list.

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

Selects the eighth item of a non-empty list.

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

Determines whether some value is the empty list.

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

Selects the fifth item of a non-empty list.

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

Selects the first item of a non-empty list.

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

Selects the fourth item of a non-empty list.

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

Evaluates the number of items on a list.

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

Constructs a list of its arguments.

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

Constructs a list by adding multiple items to a list.

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

Extracts the indexed item from the list.

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

Constructs a list of k (the first argument) copies of x (the second argument).

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

Determines whether some value is on the list (comparing values with equal?).

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

Determines whether some value is on the list (comparing values with equal?).

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

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

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

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

`null` : `empty`

The empty list.

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

Determines whether some value is the empty list.

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

Constructs 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))
```

Selects the rest of a non-empty list.

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

Creates a reversed version of a list.

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

Selects the second item of a non-empty list.

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

Selects the seventh item of a non-empty list.

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

Selects the sixth item of a non-empty list.

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

Selects the third item of a non-empty list.

4.4.5 Posns

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

Constructs a posn.

```
posn : signature
```

Signature for posns.

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

Extracts the x component of a posn.

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

Extracts the y component of a posn.

`posn? : (anything -> boolean)`

Determines if its input is a posn.

4.4.6 Characters

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

Lookups the number that corresponds to the given character in the ASCII table (if any).

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

Determines whether a character represents an alphabetic character.

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

Determines whether a character precedes another (or is equal to it) in a case-insensitive manner.

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

Determines whether a character precedes another in a case-insensitive manner.

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

Determines whether two characters are equal in a case-insensitive manner.

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

Determines whether a character succeeds another (or is equal to it) in a case-insensitive manner.

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

Determines whether a character succeeds another in a case-insensitive manner.

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

Determines the equivalent lower-case character.

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

Determines whether a character is a lower-case character.

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

Determines whether a character represents a digit.

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

Determines the equivalent upper-case character.

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

Determines whether a character is an upper-case character.

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

Determines whether a character represents space.

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

Determines whether a character precedes another (or is equal to it).

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

Determines whether a character precedes another.

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

Determines whether two characters are equal.

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

Determines whether a character succeeds another (or is equal to it).

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

Determines whether a character succeeds another.

`char? : (any -> boolean)`

Determines whether a value is a character.

4.4.7 Strings

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

Translates a string into a list of 1-letter strings.

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

Formats a string, possibly embedding values.

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

Concatenates the list of 1-letter strings into one string.

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

Converts an integer in [0,55295] or [57344 1114111] to a 1-letter string.

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

Converts a s list of characters into a string.

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

Produces a string of given length from a single given character.

`replicate : (nat string -> string)`

Replicates the given string.

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

Builds a string of the given characters.

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

Converts a 1-letter string to an integer in [0,55295] or [57344, 1114111].

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

Converts a string into a list of characters.

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

Converts a string into a number, produce false if impossible.

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

Converts a string into a symbol.

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

Determines whether all 'letters' in the string are alphabetic.

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

Juxtaposes the characters of several strings.

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

Determines whether one string alphabetically precedes another (or is equal to it) in a case-insensitive manner.

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

Determines whether one string alphabetically precedes another in a case-insensitive manner.

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

Compares two strings character-wise in a case-insensitive manner.

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

Determines whether one string alphabetically succeeds another (or is equal to it) in a case-insensitive manner.

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

Determines whether one string alphabetically succeeds another in a case-insensitive manner.

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

Copies a string.

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

Extracts the ith 1-letter substring from the given one.

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

Determines the length of a string.

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

Determines whether all 'letters' in the string are lower case.

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

Determines whether all 'letters' in the string are numeric.

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

Extracts the i-the character from a string.

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

Determines whether all 'letters' in the string are upper case.

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

Determines whether all 'letters' in the string are white space.

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

Determines whether one string alphabetically precedes another (or is equal to it).

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

Determines whether one string alphabetically precedes another.

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

Compares two strings character-wise.

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

Determines whether one string alphabetically succeeds another (or is equal to it).

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

Determines whether one string alphabetically succeeds another.

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

Determines whether a value is a string.

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

Extracts the substring starting at a 0-based index up to the second 0-based index (exclusive).

4.4.8 Images

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

Determines whether two images are equal.

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

Determines whether a value is an image.

4.4.9 Misc

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

Checks whether two numbers are within some amount (the third argument) of either other.

```
eof : eof
```

The end-of-file value.

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

Determines whether some value is the end-of-file value.

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

Determines whether two values are equivalent from the computer's perspective (intensional).

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

Determines whether two values are structurally equal where basic values are compared with the eqv? predicate.

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

Compares like equal? on the first two arguments, except using =~ in the case of numbers.

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

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

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

signals an error, combining the given values into an error message.

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

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

Exits the running program.

`identity : (any -> any)`

Returns the argument unchanged.

`struct? : (any -> boolean)`

Determines whether some value is a structure.

4.4.10 Numbers (relaxed conditions)

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

Multiplys all given numbers.

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

Adds all given numbers.

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

Subtracts from the first all remaining numbers.

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

Divides the first by all remaining numbers.

4.4.11 Higher-Order Functions

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

(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)
```

Applies a function using items from a list as the arguments.

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

Finds the (first) element of the list that maximizes the output of the function.

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

Finds the (first) element of the list that minimizes the output of the function.

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

(build-list n f) = (list (f 0) ... (f (- n 1)))

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

(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))
```

Composes a sequence of procedures into a single procedure.

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

Constructs a list from all those items on a list for which the predicate holds.

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

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

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

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

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

Applies a function to each item on one or more lists for effect only.

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

Constructs a new list by applying a function to each item on one or more existing lists.

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

Produces true if the function given as the first argument produces a non-false value for any item in the second argument.

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

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

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

Produces true if the value is a procedure.

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

Constructs a list from all items on a list in an order according to a predicate.

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

Constructs a list from all items on a list in an order according to a predicate.

5 Advanced Student

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

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

 definition = (define (name variable ...) expr)
            | (define name expr)
            | (define-struct name (name ...))
            | (define-datatype name (name name ...) ...)

       expr = (begin expr expr ...)
            | (begin0 expr expr ...)
            | (set! variable expr)
            | (delay expr)
            | (lambda (variable ...) expr)
            | (λ (variable ...) expr)
            | (local [definition ...] expr)
            | (letrec ([name expr] ...) expr)
            | (shared ([name expr] ...) expr)
            | (let ([name expr] ...) expr)
            | (let name ([name expr] ...) expr)
            | (let* ([name expr] ...) expr)
            | (recur name ([name expr] ...) expr)
            | (expr expr ...)
            | (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])
            | (match expr [pattern expr] ...)
            | (if expr expr expr)
            | (when expr expr)
            | (unless expr expr)
            | (and expr expr expr ...)
            | (or expr expr expr ...)
            | (time expr)
            | name
            | 'quoted
            | `quasiquoted
            | number
```

```
                | string
                | character

          choice = name
                | number

         pattern = _
                | name
                | number
                | true
                | false
                | string
                | character
                | 'quoted
                | `quasiquoted-pattern
                | (cons pattern pattern)
                | (list pattern ...)
                | (list* pattern ...)
                | (struct id (pattern ...))
                | (vector pattern ...)
                | (box pattern)

quasiquoted-pattern = name
                | number
                | string
                | character
                | (quasiquoted-pattern ...)
                | 'quasiquoted-pattern
                | `quasiquoted-pattern
                | ,pattern
                | ,@pattern

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

      quasiquoted = name
                | number
                | string
                | character
```

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

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

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

        package = (string string number number)
```

An *name* or a *variable* 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 *symbol* is a quote character followed by a name. A symbol is a value, just like 0 or empty.

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

In function calls, the function appearing immediatly after the open parenthesis can be any functions defined with `define` or `define-struct`, or any one of the pre-defined functions.

5.1 Pre-Defined Variables

`empty` : `empty?`

The empty list.

`true` : `boolean?`

The true value.

`false` : `boolean?`

The false value.

5.2 Syntax for Advanced

In Advanced, `set!` can be used to mutate variables, and `define-struct`'s structures are mutatable. `define` and `lambda` can define functions of zero arguments, and function calls can invoke functions of zero arguments.

(`lambda` (*variable* ...) *expression*)

Creates a function that takes as many arguments as given *variable*s, and whose body is *expression*.

(λ (*variable* ...) *expression*)

The Greek letter λ is a synonym for `lambda`.

(*expression expression* ...)

Calls the function that results from evaluating the first *expression*. The value of the call is the value of function's body when every instance of `name`'s variables are replaced by the values of the corresponding *expression*s.

The function being called must come from either a definition appearing before the function call, or from a `lambda` expression. The number of argument *expression*s must be the same as the number of arguments expected by the function.

(`define-datatype` *datatype-name* [*variant-name field-name* ...] ...)

A short-hand for defining a group of related structures. The following `define-datatype`:

```
(define-datatype datatype-name
  [variant-name field-name ...]
  ...)
```

is equivalent to:

```
(define (datatype-name? x)
  (or (variant-name? x) ...))
(define-struct variant-name (field-name ...))
...
```

`(begin expression expression ...)`

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

`(begin0 expression expression ...)`

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

`(set! variable expression)`

Evaluates *expression*, and then mutates the *variable* to have *expression*'s value. The *variable* must be defined by `define`, `letrec`, `let*`, or `let`.

`(delay expression)`

Produces a "promise" to evaluate *expression*. The *expression* is not evaluated until the promise is forced with `force`; when the promise is forced, the result is recorded, so that any further `force` of the promise immediately produces the remembered value.

`(shared ([name expression] ...) expression)`

Like `letrec`, but when an *expression* next to an `id` is a `cons`, `list`, `vector`, quasiquoted expression, or `make-struct-name` from a `define-struct`, the *expression* can refer directly to any *name*, not just *name*s defined earlier. Thus, `shared` can be used to create cyclic data structures.

`(recur name ([name expression] ...) expression)`

A short-hand syntax for recursive loops. The first *name* corresponds to the name of the recursive function. The *name*s in the parenthesis are the function's arguments, and each corresponding *expression* is a value supplied for that argument in an initial starting call of the function. The last *expression* is the body of the function.

More precisely, the following `recur`:

```
(recur func-name ([arg-name arg-expression] ...)
  body-expression)
```

is equivalent to:

```
(local [(define (func-name arg-name ...) body-expression)]
  (func-name arg-expression ...))
```

(let name ([name expression] ...) expression)

An alternate syntax for `recur`.

(case *expression* [(*choice* ...) *expression*] ... [(*choice* ...) *expression*])

A `case` form contains one or more clauses. Each clause contains a choices (in parentheses)—either numbers or names—and an answer *expression*. The initial *expression* is evaluated, and its value is compared to the choices in each clause, where the lines are considered in order. The first line that contains a matching choice provides an answer *expression* whose value is the result of the whole `case` expression. Numbers match with the numbers in the choices, and symbols match with the names. If none of the lines contains a matching choice, it is an error.

(case *expression* [(*choice* ...) *expression*] ... [else *expression*])

This form of `case` is similar to the prior one, except that the final `else` clause is taken if no clause contains a choice matching the value of the initial *expression*.

(match *expression* [*pattern* *expression*] ...)

A `match` form contains one or more clauses that are surrounded by square brackets. Each clause contains a pattern—a description of a value—and an answer *expression*. The initial *expression* is evaluated, and its value is matched against the pattern in each clause, where the clauses are considered in order. The first clause that contains a matching pattern provides an answer *expression* whose value is the result of the whole `match` expression. This *expression* may reference identifiers defined in the matching pattern. If none of the clauses contains a matching pattern, it is an error.

(when *test-expression* *body-expression*)

If *test-expression* evaluates to `true`, the result of the `when` expression is the result of evaluating the *body-expression*, otherwise the result is `(void)` and the *body-*

expression is not evaluated. If the result of evaluating the *test-expression* is neither `true` nor `false`, it is an error.

`(unless` *test-expression body-expression*`)`

Like `when`, but the *body-expression* is evaluated when the *test-expression* produces `false` instead of `true`.

5.3 Common Syntaxes

The following syntaxes behave the same in the *Advanced* level as they did in the §4 "Intermediate Student with Lambda" level.

`(local [`*definition ...*`]` *expression*`)`

Groups related definitions for use in *expression*. Each *definition* can be either a `define` or a `define-struct`.

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

`(letrec ([`*name expr-for-let*`] ...)` *expression*`)`

Like `local`, but with a simpler syntax. Each *name* defines a variable (or a function) with the value of the corresponding *expr-for-let*. If *expr-for-let* is a `lambda`, `letrec` defines a function, otherwise it defines a variable.

`(let* ([`*name expr-for-let*`] ...)` *expression*`)`

Like `letrec`, but each *name* can only be used in *expression*, and in *expr-for-let*s occuring after that *name*.

`(let ([`*name expr-for-let*`] ...)` *expression*`)`

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

`(time expression)`

Measures the time taken to evaluate *expression*. After evaluating *expression*, `time` prints out the time taken by the evaluation (including real time, time taken by the CPU, and the time spent collecting free memory). The value of `time` is the same as that of *expression*.

`(define (name variable variable ...) expression)`

Defines a function named *name*. The *expression* is the body of the function. When the function is called, the values of the arguments are inserted into the body in place of the *variable*s. The function returns the value of that new expression.

The function name's cannot be the same as that of another function or variable.

`(define name expression)`

Defines a variable called *name* with the the value of *expression*. The variable name's cannot be the same as that of another function or variable, and *name* itself must not appear in *expression*.

`(define-struct structure-name (field-name ...))`

Defines a new structure called *structure-name*. The structure's fields are named by the *field-name*s. After the `define-struct`, the following new functions are available:

- `make-`*structure-name* : takes a number of arguments equal to the number of fields in the structure, and creates a new instance of that structure.
- *structure-name-field-name* : takes an instance of the structure and returns the value in the field named by *field-name*.
- *structure-name*`?` : takes any value, and returns `true` if the value is an instance of the structure.

The name of the new functions introduced by `define-struct` must not be the same as that of other functions or variables, otherwise `define-struct` reports an error.

In Advanced, `define-struct` introduces one additional function:

- `set-`*structure-name-field-name*`!` : takes an instance of the structure and a value, and mutates the instance's field to the given value.

```
(cond [question-expression answer-expression] ...)
(cond [question-expression answer-expression]
      ...
      [else answer-expression])
```

Chooses a clause based on some condition. `cond` finds the first *question-expression* that evaluates to `true`, then evaluates the corresponding *answer-expression*.

If none of the *question-expression*s evaluates to `true`, `cond`'s value is the *answer-expression* of the `else` clause. If there is no `else`, `cond` reports an error. If the result of a *question-expression* is neither `true` nor `false`, `cond` also reports an error.

`else` cannot be used outside of `cond`.

```
(if test-expression then-expression else-expression)
```

When the value of the *test-expression* is `true`, `if` evaluates the *then-expression*. When the test is `false`, `if` evaluates the *else-expression*.

If the *test-expression* is neither `true` nor `false`, `if` reports an error.

```
(or expression expression expression ...)
```

Evaluates to `true` if all the *expression*s are `true`. If any *expression* is `false`, the `or` expression immediately evaluates to `false` (and the expressions to the right of that expression are not evaluated.)

If any of the expressions evaluate to a value other than `true` or `false`, `or` reports an error.

```
(and expression expression expression ...)
```

Evaluates to `true` as soon as one of the *expression*s is `true` (and the expressions to the right of that expression are not evaluated.) If all of the *expression*s are `false`, the `and` expression evaluates to `false`.

If any of the expressions evaluate to a value other than `true` or `false`, `and` reports an error.

```
(check-expect expression expected-expression)
```

Checks that the first *expression* evaluates to the same value as the *expected-expression*.

```
(check-within expression expected-expression delta-expression)
```

Checks that the first *expression* evaluates to a value within *delta-expression* of the *expected-expression*. If *delta-expression* is not a number, `check-within` reports an error.

```
(check-error expression match-expression)
(check-error expression)
```

Checks that the *expression* reports an error, where the error messages matches the value of `matchexpression`, if it is present.

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

Checks that the value of the first *expression* as that of one of the following *expression*s.

```
(check-range expression low-expression high-expression)
```

Checks that the value of the first *expression* is a number in between the value of the *low-expression* and the *high-expression*, inclusive.

```
(require string)
```

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

Accesses a file in an installed library. The library name is an identifier with the same constraints as for a relative-path string (though without the quotes), 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., the current file). The first *string* names the library file, and the remaining *string*s name the collection (and sub-collection, and so on) where the file is installed. Each string is constrained in the same way as for the `(require` *string*`)` form.

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

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

5.4 Pre-Defined Functions

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

---

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

Compares real numbers for less-than.

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

Compares real numbers for less-than or equality.

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

Compares numbers for equality.

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

Compares real numbers for greater-than.

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

Compares real numbers for greater-than or equality.

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

Evaluates the absolute value of a real number.

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

Evaluates the arccosine (inverse of cos) of a number.

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

Evaluates a number one larger than a given number.

`angle : (number -> real)`

Extracts the angle from a complex number.

`asin : (number -> number)`

Evaluates the arcsine (inverse of sin) of a number.

`atan : (number (number) -> number)`

Evaluates the arctan of the given number or the ratio of the two given numbers.

`ceiling : (real -> integer)`

Determines the closest integer (exact or inexact) above a real number.

`complex? : (any -> boolean)`

Determines whether some value is complex.

`conjugate : (number -> number)`

Evaluates the conjugate of a complex number.

`cos : (number -> number)`

Evaluates the cosine of a number (radians).

`cosh : (number -> number)`

Evaluates the hyperbolic cosine of a number.

`current-seconds : (-> integer)`

Evaluates the current time in seconds elapsed (since a platform-specific starting date).

`denominator : (rat -> integer)`

Evaluates the denominator of a rational.

`e : real`

Euler's number.

`even? : (integer -> boolean)`

Determines if some integer (exact or inexact) is even or not.

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

Converts an exact number to an inexact one.

`exact? : (number -> boolean)`

Determines whether some number is exact.

`exp : (number -> number)`

Evaluates e raised to a number.

`expt : (number number -> number)`

Evaluates the power of the first to the second number.

`floor : (real -> integer)`

Determines the closest integer (exact or inexact) below a real number.

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

Evaluates the greatest common divisior of two integers (exact or inexact).

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

Extracts the imaginary part from a complex number.

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

Approximates an inexact number by an exact one.

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

Determines whether some number is inexact.

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

Lookups the character that corresponds to the given integer (exact only!) in the ASCII table (if any).

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

Evaluates the integer (exact or inexact) square root of a number.

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

Determines whether some value is an integer (exact or inexact).

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

Evaluates the least common multiple of two integers (exact or inexact).

`log` : `(number -> number)`

Evaluates the base-e logarithm of a number.

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

Determines the magnitude of a complex number.

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

Creates a complex from a magnitude and angle.

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

Creates a complex from a real and an imaginary part.

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

Determines the largest number.

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

Determines the smallest number.

`modulo : (integer integer -> integer)`

Finds the remainder of the division of the first number by the second; try (modulo 4 3) (modulo 4 -3).

`negative? : (number -> boolean)`

Determines if some value is strictly smaller than zero.

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

Converts a number to a string.

`number? : (any -> boolean)`

Determines whether some value is a number.

`numerator : (rat -> integer)`

Evaluates the numerator of a rational.

`odd? : (integer -> boolean)`

Determines if some integer (exact or inexact) is odd or not.

`pi : real`

The ratio of a circle's circumference to its diameter.

`positive? : (number -> boolean)`

Determines if some value is strictly larger than zero.

`quotient : (integer integer -> integer)`

Divides the first integer (exact or inexact) into the second; try (quotient 3 4) and (quotient 4

3).

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

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

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

Determines whether some value is a rational number.

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

Extracts the real part from a complex number.

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

Determines whether some value is a real number.

```
remainder : (integer integer -> integer)
```

Determines the remainder of dividing the first by the second integer (exact or inexact).

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

Rounds a real number to an integer (rounds to even to break ties).

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

Evaluates the sign of a real number.

```
sin : (number -> number)
```

Evaluates the sine of a number (radians).

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

Evaluates the hyperbolic sine of a number.

`sqr : (number -> number)`

Evaluates the square of a number.

`sqrt : (number -> number)`

Evaluates the square root of a number.

`sub1 : (number -> number)`

Evaluates a number one smaller than a given number.

`tan : (number -> number)`

Evaluates the tangent of a number (radians).

`zero? : (number -> boolean)`

Determines if some value is zero or not.

5.4.2 Booleans

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

Determines whether two booleans are equal.

`boolean? : (any -> boolean)`

Determines whether some value is a boolean.

`false? : (any -> boolean)`

Determines whether a value is false.

`not : (boolean -> boolean)`

Evaluates the negation of a boolean value.

5.4.3 Symbols

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

Converts a symbol to a string.

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

Determines whether two symbols are equal.

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

Determines whether some value is a symbol.

5.4.4 Lists

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

Creates a single list from several.

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

Produces the first element on the list whose first is equal? to v; otherwise it produces false.

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

Determines 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)
```

Selects 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))
```

Selects the rest of the first list in the first list of a list.

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

Selects the first item of the first list in a list.

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

Selects the second item of the first list of a list.

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

Selects the fourth item of a non-empty list.

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

Selects the third item of a non-empty list.

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

Selects the second item of a non-empty list.

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

Selects the first item of a non-empty list.

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

Selects 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))
```

Selects the rest of the first list in the rest of a list.

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

Selects the rest of a non-empty list in a list.

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

Selects the rest of the rest of the first list of a list.

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

Selects the rest of the rest of the rest of a list.

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

Selects the rest of the rest of a list.

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

Selects the rest of a non-empty list.

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

Constructs a list.

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

Determines whether some value is a constructed list.

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

Selects the eighth item of a non-empty list.

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

Determines whether some value is the empty list.

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

Selects the fifth item of a non-empty list.

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

Selects the first item of a non-empty list.

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

Selects the fourth item of a non-empty list.

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

Evaluates the number of items on a list.

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

Constructs a list of its arguments.

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

Constructs a list by adding multiple items to a list.

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

Extracts the indexed item from the list.

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

Determines whether some value is a list.

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

Constructs a list of k (the first argument) copies of x (the second argument).

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

Determines whether some value is on the list (comparing values with equal?).

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

Determines whether some value is on the list (comparing values with equal?).

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

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

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

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

```
null : empty
```

The empty list.

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

Determines whether some value is the empty list.

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

Constructs 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))
```

Selects the rest of a non-empty list.

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

Creates a reversed version of a list.

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

Selects the second item of a non-empty list.

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

Selects the seventh item of a non-empty list.

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

Selects the sixth item of a non-empty list.

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

Selects the third item of a non-empty list.

5.4.5 Posns

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

Constructs a posn.

`posn : signature`

Signature for posns.

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

Extracts the x component of a posn.

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

Extracts the y component of a posn.

`posn? : (anything -> boolean)`

Determines if its input is a posn.

`set-posn-x! : (posn number -> void)`

Updates the x component of a posn.

`set-posn-y!` : `(posn number -> void)`

Updates the x component of a posn.

5.4.6 Characters

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

Lookups the number that corresponds to the given character in the ASCII table (if any).

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

Determines whether a character represents an alphabetic character.

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

Determines whether a character precedes another (or is equal to it) in a case-insensitive manner.

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

Determines whether a character precedes another in a case-insensitive manner.

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

Determines whether two characters are equal in a case-insensitive manner.

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

Determines whether a character succeeds another (or is equal to it) in a case-insensitive manner.

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

Determines whether a character succeeds another in a case-insensitive manner.

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

Determines the equivalent lower-case character.

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

Determines whether a character is a lower-case character.

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

Determines whether a character represents a digit.

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

Determines the equivalent upper-case character.

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

Determines whether a character is an upper-case character.

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

Determines whether a character represents space.

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

Determines whether a character precedes another (or is equal to it).

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

Determines whether a character precedes another.

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

Determines whether two characters are equal.

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

Determines whether a character succeeds another (or is equal to it).

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

Determines whether a character succeeds another.

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

Determines whether a value is a character.

5.4.7 Strings

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

Translates a string into a list of 1-letter strings.

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

Formats a string, possibly embedding values.

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

Concatenates the list of 1-letter strings into one string.

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

Converts an integer in [0,55295] or [57344 1114111] to a 1-letter string.

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

Converts a s list of characters into a string.

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

Produces a string of given length from a single given character.

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

Replicates the given string.

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

Builds a string of the given characters.

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

Converts a 1-letter string to an integer in [0,55295] or [57344, 1114111].

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

Converts a string into a list of characters.

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

Converts a string into a number, produce false if impossible.

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

Converts a string into a symbol.

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

Determines whether all 'letters' in the string are alphabetic.

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

Juxtaposes the characters of several strings.

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

Determines whether one string alphabetically precedes another (or is equal to it) in a case-insensitive manner.

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

Determines whether one string alphabetically precedes another in a case-insensitive manner.

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

Compares two strings character-wise in a case-insensitive manner.

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

Determines whether one string alphabetically succeeds another (or is equal to it) in a case-

insensitive manner.

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

Determines whether one string alphabetically succeeds another in a case-insensitive manner.

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

Copies a string.

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

Extracts the ith 1-letter substring from the given one.

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

Determines the length of a string.

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

Determines whether all 'letters' in the string are lower case.

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

Determines whether all 'letters' in the string are numeric.

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

Extracts the i-the character from a string.

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

Determines whether all 'letters' in the string are upper case.

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

Determines whether all 'letters' in the string are white space.

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

Determines whether one string alphabetically precedes another (or is equal to it).

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

Determines whether one string alphabetically precedes another.

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

Compares two strings character-wise.

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

Determines whether one string alphabetically succeeds another (or is equal to it).

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

Determines whether one string alphabetically succeeds another.

`string? : (any -> boolean)`

Determines whether a value is a string.

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

Extracts the substring starting at a 0-based index up to the second 0-based index (exclusive).

5.4.8 Images

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

Determines whether two images are equal.

`image? : (any -> boolean)`

Determines whether a value is an image.

5.4.9 Misc

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

Checks whether two numbers are within some amount (the third argument) of either other.

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

Returns the current "time" in fixnum milliseconds (possibly negative).

`eof : eof`

The end-of-file value.

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

Determines whether some value is the end-of-file value.

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

Determines whether two values are equivalent from the computer's perspective (intensional).

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

Determines whether two values are structurally equal where basic values are compared with the eqv? predicate.

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

Compares like equal? on the first two arguments, except using =~ in the case of numbers.

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

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

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

signals an error, combining the given values into an error message.

If any of the values' printed representations is too long, it is truncated and "..." is put into

the string. If the first value is a symbol, it is treated specially; it is suffixed with a colon and a space (the intention is that the symbol is the name of the function signalling the error).

`exit : (-> void)`

Exits the running program.

`force : (delay -> any)`

Finds the delayed value; see also delay.

`gensym : (-> symbol?)`

Generates a new symbol, different from all symbols in the program.

`identity : (any -> any)`

Returns the argument unchanged.

`promise? : (any -> boolean)`

Determines if a value is delayed.

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

Causes the program to sleep for the given number of seconds.

`struct? : (any -> boolean)`

Determines whether some value is a structure.

`void : (-> void)`

Produces a void value.

`void? : (any -> boolean)`

Determines if a value is void.

5.4.10 Numbers (relaxed conditions)

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

Multiplys all given numbers.

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

Adds all given numbers.

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

Subtracts from the first all remaining numbers.

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

Divides the first by all remaining numbers.

5.4.11 Higher-Order Functions

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

(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)
```

Applies a function using items from a list as the arguments.

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

Finds the (first) element of the list that maximizes the output of the function.

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

Finds the (first) element of the list that minimizes the output of the function.

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

(build-list n f) = (list (f 0) ... (f (- n 1)))

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

(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))
```

Composes a sequence of procedures into a single procedure.

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

Constructs a list from all those items on a list for which the predicate holds.

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

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

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

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

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

Applies a function to each item on one or more lists for effect only.

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

Constructs a new list by applying a function to each item on one or more existing lists.

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

Produces true if the function given as the first argument produces a non-false value for any item in the second argument.

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

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

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

Produces true if the value is a procedure.

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

Constructs a list from all items on a list in an order according to a predicate.

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

Constructs a list from all items on a list in an order according to a predicate.

5.4.12 Reading and Printing

```
display : (any -> void)
```

Prints the argument to stdout (without quotes on symbols and strings, etc.).

```
newline : (-> void)
```

Prints a newline to stdout.

```
pretty-print : (any -> void)
```

Like write, but with standard newlines and indentation.

```
print : (any -> void)
```

Prints the argument as a value to stdout.

```
printf : (string any ... -> void)
```

Formats the rest of the arguments according to the first argument and print it to stdout.

```
read : (-> sexp)
```

Reads input from the user.

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

Opens the named input file and to extract all input from there.

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

Turns the given string into input for read* operations.

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

Opens the named output file and to put all output there.

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

Produces a string from all write/display/print operations.

```
write : (any -> void)
```

Prints the argument to stdout (in a traditional style that is somewhere between print and display).

5.4.13 Vectors

```
build-vector : (nat (nat -> X) -> (vectorof X))
```

Constructs a vector.

```
make-vector : (number X -> (vectorof X))
```

Constructs a vector.

`vector : (X ... -> (vector X ...))`

Constructs a vector.

`vector-length : ((vector X) -> nat)`

Determines the length of a vector.

`vector-ref : ((vector X) nat -> X)`

Extracts an element from a vector.

`vector-set! : ((vectorof X) nat X -> void)`

Updates a vector.

`vector? : (any -> boolean)`

Determines if a value is a vector.

5.4.14 Boxes

`box : (any -> box)`

Constructs a box.

`box? : (any -> boolean)`

Determines if a value is a box.

`set-box! : (box any -> void)`

Updates a box.

`unbox : (box -> any)`

Extracts the boxed value.

5.4.15 Hash Tables

```
hash-copy : (hash -> hash)
```

Copies a hash table.

```
hash-count : (hash -> integer)
```

Determines the number of keys mapped by a hash table.

```
hash-eq? : (hash -> boolean)
```

Determines if a hash table uses eq? for comparisions.

```
hash-equal? : (hash -> boolean)
```

Determines if a hash table uses equal? for comparisions.

```
hash-eqv? : (hash -> boolean)
```

Determines if a hash table uses eqv? for comparisions.

```
hash-for-each : ((hash X Y) (X Y -> any) -> void)
```

Applies a function to each mapping of a hash table for effect only.

```
hash-has-key? : ((hash X Y) X -> boolean)
```

Determines if a key is associated with a value in a hash table.

```
hash-map : ((hash X Y) (X Y -> A) -> (listof A))
```

Constructs a new list by applying a function to each mapping of a hash table.

```
hash-ref : (case->
            ((hash X Y) X -> Y)
            ((hash X Y) X Y -> Y)
            ((hash X Y) X (-> Y) -> Y))
```

Extracts the value associated with a key from a hash table; the three argument case allows a

default value or default value computation.

```
hash-ref! : (case->
             ((hash X Y) X Y -> Y)
             ((hash X Y) X (-> Y) -> Y))
```

Extracts the value associated with a key from a mutable hash table; if the key does not have an mapping, the third argument is used as the value (or used to compute the value) and is added to the hash table associated with the key.

```
hash-remove : ((hash X Y) X -> (hash X Y))
```

Constructs an immutable hash table with one less mapping than an existing immutable hash table.

```
hash-remove! : ((hash X Y) X -> void)
```

Removes an mapping from a mutable hash table.

```
hash-set : ((hash X Y) X Y -> (hash X Y))
```

Constructs an immutable hash table with one new mapping from an existing immutable hash table.

```
hash-set! : ((hash X Y) X Y -> void)
```

Updates a mutable hash table with a new mapping.

```
hash-update : (case->
               ((hash X Y) X (Y -> Y) -> (hash X Y))
               ((hash X Y) X (Y -> Y) Y -> (hash X Y))
               ((hash X Y) X (Y -> Y) (-> Y) -> (hash X Y)))
```

Composes hash-ref and hash-set to update an existing mapping; the third argument is used to compute the new mapping value; the fourth argument is used as the third argument to hash-ref.

```
hash-update! : (case->
                ((hash X Y) X (Y -> Y) -> void)
                ((hash X Y) X (Y -> Y) Y -> void)
                ((hash X Y) X (Y -> Y) (-> Y) -> void))
```

Composes hash-ref and hash-set! to update an existing mapping; the third argument is used to compute the new mapping value; the fourth argument is used as the third argument to hash-ref.

```
hash? : (any -> boolean)
```

Determines if a value is a hash table.

```
make-hash : (case->
             (-> (hash X Y))
             ((listof (list X Y)) -> (hash X Y)))
```

Constructs a mutable hash table from an optional list of mappings that uses equal? for comparisions.

```
make-hasheq : (case->
               (-> (hash X Y))
               ((listof (list X Y)) -> (hash X Y)))
```

Constructs a mutable hash table from an optional list of mappings that uses eq? for comparisions.

```
make-hasheqv : (case->
                (-> (hash X Y))
                ((listof (list X Y)) -> (hash X Y)))
```

Constructs a mutable hash table from an optional list of mappings that uses eqv? for comparisions.

```
make-immutable-hash : (case->
                       (-> (hash X Y))
                       ((listof (list X Y)) -> (hash X Y)))
```

Constructs an immutable hash table from an optional list of mappings that uses equal? for comparisions.

```
make-immutable-hasheq : (case->
                         (-> (hash X Y))
                         ((listof (list X Y)) -> (hash X Y)))
```

Constructs an immutable hash table from an optional list of mappings that uses eq? for comparisions.

```
make-immutable-hasheqv : (case->
                          (-> (hash X Y))
                          ((listof (list X Y)) -> (hash X Y)))
```

Constructs an immutable hash table from an optional list of mappings that uses eqv? for comparisions.

Index