SteelSeries GoLisp v1.1: Language Reference

Dave Astels

December 19, 2016

Abstract

Steelseries GoLisp is a Lisp-like language inspired by and based on the opensource MIT/GNU Scheme. As such, much of the functionallity is identical and much of the text herein is taken more or less verbatim from the associated reference manual [6]. Only functions documented here are included, and any differences from the MIT/GNU Scheme equivalents are noted.

1 Running GoLisp

Before GoLisp will work, the environment variable `GOLISPHOME` must be defined as the path where the GoLisp `lisp` and `tools` directories are located.

```
golisp [options] [files-to-load] [- program-args]
```

`files-to-load` is a series of filenames or directories to be loaded at startup (after the core library files are loaded). Any files, and contained files, that end in ".scm" are loaded.

Supported `options`:

- `-r` after files have been loaded, drop into the REPL. This will cause GoLisp to NOT use the `main` function if it is present.
- `-t test-file-or-directory` loads and runs tests contained in `test-file-or-directory` (files ending in ".scm").

- `-v` use with `-t` to request verbose test output.
- `-d "symbol=value"` define something in the global environment. The lisp equivalent is `(define symbol value)`. `value` must be a literal constant.
- `-e "code"` after startup (i.e. the standard library has loaded as well as any files specified on the command line), evaluate `code`. The result is printed and, unless the REPL has been requested, GoLisp exits. E.g.

  ```
  >:golisp -e "(+ 2 3)"
  ==> 5
  ```

Once options are processed and neither testing mode nor REPL mode have been requested and no expression has been supplied to be evaluated, then a function named `main` is look for. If found it is applied to the values, if any, that follow the `-`.

2 Data types

Booleans represent true and false. Boolean literals are `#t` and `#f` for true and false, respectively. The only thing that is considered to be logically false is `#f`. Everything else is logically true, inluding 0 and the empty list, which may surprise some.

Integers are sixtyfour bit signed integers. Both decimal ,hexadecimal, and binary formats are supported. E.g. `26`, `#x1a`, `#x1A`, `#b00011001`.

Floats are Go `float32` numbers. Accordingly they are signed. All arithmetic functions with the exception of modulus work as expected for both integers and floats. Numbers are coerced to floats as required, specifically if any arguments are float, all are converted to float and the result will be a float.

Characters are single characters. This is differen than strings with length 1.

Strings are any sequence of characters other than `"` enclosed by a pair of `"`, e.g. `"string"`. If you need to have `"` in a string, use `\"`.

Symbols are simple identifiers, e.g. `function-name`. Symbols follow the follow 3 simple rules:

- can only contain graphic characters (i.e. no control characters)

- can not contain any of the characters: `();,"‘&[]{}\`
- can not begin with a number or single quote
- can not contain whitespace

Typically, `-` is used to separate words in a symbol, `_` is used in special symbols (such as system use) to separate words and as a prefix and suffix. The characters `?`, `!`, and `*` are typically used as the final character of a function name to denote:

? a predicate, e.g. `nil?`

! a mutating function (changes the argument rather than returning a modified copy), e.g. `set-car!`

***** a variant of the primary function, e.g. `flatten` (which does a one level flattening of a list) and `flatten*` (which is a recursive `flatten`)

If a symbol ends with `:` it is what is called a *naked symbol*. It has no value other than itself. If it is evaluated, the result is the symbol itself. This feature is utilized by frames.

Cons Cells, aka **Lists** are the central data type in classic Lisp used both as dotted pairs `(a . b)` and general lists `(a b)`. For an overview of cons cells and how to use them see [1], [3], or `http://cs.gmu.edu/~sean/lisp/cons/`. Note that dotted pairs *require* spaces around the period; `(a.b)` is a list containing the symbol `a.b`, not a cons cell with car of `a` and cdr of `b`.

Bytearrays are simply objects that encapsulate `[]byte` objects. The difference is that there is syntactic support for them. Use square braces surrounding a list of numbers between 0 and 255, inclusive. For example: `[1 2 3 4 5]`. That format will parse to an `object` containing a the Go bytearray (i.e. `[]byte`). Bytearrays evaluate to themselves. There are also functions for doing bytearray manipulation.

Channels are simple objects that encapsulate Go ‘channel’ objects.

Vectors are a more efficient alternative to lists when the number of elements are known ahead of time and static. A vector can be grown, but it’s done explicitly with a function. Many enumeration and access functions that accept a list (or lists) can be passed a vector (or vectors) instead.

Ports provide access to files for reading and writing.

Frames are sets of named slots that hold arbitrary values, including functions. Frames can *inherit* from other frames in a prototypical manner. They are inspired by the languages Self [4] and Newtonscript [5].

Functions are user defined procedures. They are covered in detail later.

Macros are user defined syntactic extensions. Note that GoLisp macros are not yet hygenic, so use caution.

You can create an manipulate all of the above types in GoLisp. There are two more types that are usable in GoLisp, but can only be created in Go.

Primitives are just as they are in Lisp or Smalltalk: functions written in the implementation language, in this case Go, and exposed as functions in Lisp. The combinaion of primitives and objects allow you to integrate with the underlying Go program. **Special Forms** are almost identical to primitives, except that they use normal evaluation order instead of applicative order which functions and primitives use.

Objects allow you to encapsulate a Go object (struct) in a Lisp data object. There is no way to do this from Lisp itself, but is useful when writing primitive functions (see below). The objects can be used as any other object, but are opaque outside of primitives built to use them.

3 Special Forms

A special form is an expression that follows special evaluation rules. This section describes the basic GoLisp special forms.

3.1 Lambda Expressions

3.1.1 (lambda *formals sexpr...*)

A **lambda** expression evaluates to a procedure. The environment in effect when the **lambda** expression is evaluated is remembered as part of the procedure; it is called the *closing environment*. When the procedure is later called with some arguments, the closing environment is extended by binding the variables in the formal parameter list to fresh locations, and the locations are filled with the arguments according to rules about to be given. The new environment created by this process is referred to as the *invocation environment*.

Once the invocation environment has been constructed, the *sexprs* in the body of the **lambda** expression are evaluated sequentially in that environment. This means that the region of the variables bound by the **lambda** expression is all of the *sexprs* in the body. The result of evaluating the last *sexpr* in the body is returned as the result of the procedure call.

formals, the formal parameter list, is often referred to as a *lambda list*.

The process of matching up formal parameters with arguments is somewhat involved, but simpler than Scheme. There are two types of parameters, and the matching treats each in sequence:

Required

All of the *required* parameters are matched against the arguments first. If there are fewer arguments than required parameters, a `wrong number of arguments` error is signalled; this error is also signalled if there are more arguments than required parameters and there are no further parameters.

Rest

Finally, if there is a *rest* parameter (there can only be one), any remaining arguments are made into a list, and the *rest* parameter is bound to it. (If there are no remaining arguments, the *rest* parameter is bound to the empty list.)

In Scheme, unlike some other Lisp implementations, the list to which a *rest* parameter is bound is always freshly allocated. It has infinite extent and may be modified without affecting the procedure's caller.

A period, i.e. ".", is used to separate the *rest* parameter (if there is one) from the *required* parameters. The "." **must** be surrounded by spaces.

`(a b c)`: `a`, `b`, and `c` are all required. The procedure must be passed exactly three arguments.

`(a b . c)`: `a` and `b` are required and `c` is rest. The procedure may be passed two or more arguments.

Some examples of **lambda** expressions:

```
(lambda (x) (+ x x))            ⇒  <function: unnamed>

((lambda (x) (+ x x)) 4)        ⇒  8

(define reverse-subtract
  (lambda (x y)
    (- y x)))
(reverse-subtract 7 10)         ⇒  3

(define foo
  (let ((x 4))
    (lambda (y) (+ x y))))
(foo 6)                         ⇒  10
```

3.2 Lexical Binding

The three binding constructs `let`, `let*`, and `letrec`, give Scheme block structure. The syntax of the three constructs is identical, but they differ in the regions they establish for their variable bindings. In a `let` expression, the initial values are computed before any of the variables become bound. In a `let*` expression, the evaluations and bindings are sequentially interleaved. And in a `letrec` expression, all the bindings are in effect while the initial values are being computed (thus allowing mutually recursive definitions).

3.2.1 (let ((*variable init*)...) *sexpr*...)

The *inits* are evaluated in the current environment (in some unspecified order), the *variables* are bound to fresh locations holding the results, the *sexprs* are evaluated sequentially in the extended environment, and the value of the last *sexpr* is returned. Each binding of a *variable* has the sequence of *sexpr* as its region.

GoLisp allows any of the *inits* to be omitted, in which case the corresponding *variables* are unassigned.

Note that the following are equivalent:

```
(let ((variable init)...) expression...)
((lambda (variable...) expression...) init...)
```

Some examples:

```
(let ((x 2) (y 3))
  (* x y))                              ⇒  6

(let ((x 2) (y 3))
  (let ((foo (lambda (z) (+ x y z)))
        (x 7))
    (foo 4)))                           ⇒  9
```

3.2.2 (let* ((*variable init*)...) *sexpr*...)

`let*` is similar to `let`, but the bindings are performed sequentially from left to right, and the region of a binding is that part of the 'let*' expression to the right of the binding. Thus the second binding is done in an environment in which the first binding is visible, and so on.

Note that the following are equivalent:

```
(let* ((variable1 init1)
       (variable2 init2)
       ...
       (variablen initn))
   expression...)

(let ((variable1 init1))
  (let ((variable2 init2))
    ...
      (let ((variablen initn))
        expression...)
    ...))
```

An example:

```
(let ((x 2) (y 3))
  (let* ((x 7)
         (z (+ x y)))
    (* z x)))                           ⇒  70
```

3.2.3 (letrec ((*variable init*)...) *sexpr*...)

The *variables* are bound to fresh locations holding unassigned values, the *inits* are evaluated in the extended environment (in some unspecified order), each *variable* is assigned to the result of the corresponding *init*, the *expressions* are evaluated sequentially in the extended environment, and the value of the last *expression* is returned. Each binding of a *variable* has the entire `letrec` expression as its region, making it possible to define mutually recursive procedures.

Any of the *inits* may be omitted, in which case the corresponding *variable* is unassigned.

```
(letrec ((even? (lambda (n)
                  (if (zero? n)
                      #t
                      (odd? (- n 1)))))
         (odd? (lambda (n)
                 (if (zero? n)
                     #f
                     (even? (- n 1))))))
(even? 88))                                    ⇒  #t
```

One restriction on `letrec` is very important: it shall be possible to evaluated each *init* without assigning or referring to the value of any *variable*. If this restriction is violated, then it is an error. The restriction is necessary because Scheme passes arguments by value rather than by name. In the most common uses of `letrec`, all the *inits* are **lambda** expressions and the restriction is satisfied automatically.

3.3 Definitions

3.3.1 (define *variable* [*sexpr*])

3.3.2 (define *formals* [*doc-string*] *sexpr*...)

Definitions are valid in some but not all contexts where expressions are allowed. Definitions may only occur at the top level of a program and at the beginning of a lambda body (that is, the body of a `lambda`, `let`, `let*`, `letrec`, or procedure `define` expression). A definition that occurs at the top level of a program is

called a *top-level definition*, and a definition that occurs at the beginning of a body is called an *internal definition*.

The second form is used as a shorthand to define a procedure. The first "required parameter" in *formals* is not a parameter but the *name* of the resulting procedure; thus *formals* must have at least one required parameter.

Hence the following are identical.

```
(define inc (lambda (x) (+ x 1)))
(define (inc x) (+ x 1))
```

Using this form of define, a function that accepts a completely option set of arguments can be made:

```
(define (f . args) args)

(f) ⇒ ()
(f 1) ⇒ (1)
(f 1 2 3) ⇒ (1 2 3)
```

Please note: You can not currently define a lambda with completely optional arguments.

Finally, you can also provide a documentation string between the *formals* and the body. For example:

```
(define (inc x)
  "Increment the argument."
  (+ x 1))
```

3.3.3 (doc *func*)

This retrieves the documentation for the function *func*.

3.4 Type Signatures

GoLisp provides basic, and optional, type checking for the arguments and return values of user defined functions. Additionally, primtive functions also have type checking on arguments, as appropriate.

3.4.1 (typedef *fname arg-types...* [-> *return-type*])

This is similar to defining a function: *fname* is the name of a function that will be defined later (typicaly the next form) and *arg-types* correspond to its arguments). But with `typedef` these are argument type specification, not argument names.

Argunment type specifications can take two forms: *type* which can be a string or symbol, or a set of types separated by a pipe (E.g. `"integer|string"`) with no spaces. The latter must be a string.

When a function is passed a type that does not match its specified type(s) an error is raised, similar to:

```
> (typedef less-than number number)
> (define (less-than x y) (\textless{} x y))
> (less-than 1 4.3)
==> #t
> (less-than 1 'a)
Error in evaluation:
Evaling (less-than 1 'a). less-than argument 1 has the wrong type,
                          expected float or integer but was given symbol
```

A type specification can also include a type specification of the result of the function. Note that the `->` is required. If a return type is not provided, `anytype` is the default.

```
> (typedef less-than number number -> boolean)
> (define (less-than x y) (if (< x y) 'yes 'no))
> (less-than 1 4.3)
Error in evaluation:
Evaling (less-than 1 4.3). less-than returns the wrong type,
                           expected boolean but returned symbol
```

The following types are supported:

- list
- vector
- sequence (equivalent to list|vector)

- integer
- float
- number (equivalent to integer|float)
- boolean
- string
- character
- symbol
- stringy (equivalent to string|symbol)
- function
- macro
- primitive
- procedure (equivalent to function|primitive)
- boxedobject
- frame
- environment
- port
- anytype (equivalent to all the above typed combined)

Note that the *list* type just requires a ConsCell; if a proper list or other specific type is required, then either `http://daveastels.typed.com/blog/code-contracts-in-golisp` pre-conditions or explicit tests will be needed.

Putting all this together, a complete definition of a function would look like:

```
(typedef inc number -> number)
(define (inc x)
  "Increment the argument."
  (+ x 1))
```

3.4.2 (type *func*)

This retrieves the type signature for the function *func*.

```
(type inc)
==> (number -> number)
```

3.5 Top-Level Definitions

A top-level definition,

```
(define variable sexpr)
```

has essentially the same effect as this assignment expression, if *variable* is bound:

```
(set! variable expression)
```

If *variable* is not bound, however, `define` binds *variable* to a new location in the current environment before performing the assignment (it is an error to perform a `set!` on an unbound variable).

```
(define add3
  (lambda (x) (+ x 3)))                ⇒  unspecified
(add3 3)                                ⇒  6

(define first car)                      ⇒  unspecified
(first '(1 2))                          ⇒  1

(define bar)                            ⇒  unspecified
bar                                     error--> Unassigned variable
```

3.6 Internal Definitions

An *internal definition* is a definition that occurs at the beginning of a *body* (that is, the body of a `lambda`, `let`, `let*`, `letrec`, or procedure `define` expression), rather than at the top level of a program. The variable defined by an internal definition is local to the *body*. That is, *variable* is bound rather than assigned, and the region of the binding is the entire *body*. For example,

```
(let ((x 5))
  (define foo (lambda (y) (bar x y)))
  (define bar (lambda (a b) (+ (* a b) a)))
  (foo (+ x 3)))                          ⇒  45
```

A *body* containing internal definitions can always be converted into a completely equivalent `letrec` expression. For example, the `let` expression in the above example is equivalent to

```
(let ((x 5))
  (letrec ((foo (lambda (y) (bar x y)))
           (bar (lambda (a b) (+ (* a b) a))))
    (foo (+ x 3))))
```

3.7 Assignments

3.7.1 (set! *variable object*)

expression is evaluated and the resulting value is stored in the location to which *variable* is bound. The value of the `set!` expression is unspecified.

variable must be bound either in some region enclosing the `set!` expression, or at the top level. However, **variable** is permitted to be unassigned when the 'set!' form is entered.

```
(define x 2)                              ⇒  unspecified
(+ x 1)                                   ⇒  3
(set! x 4)                                ⇒  unspecified
(+ x 1)                                   ⇒  5
```

3.8 Quoting

This section describes the expressions that are used to modify or prevent the evaluation of objects.

3.8.1 (quote *datum*)

(quote *datum*) evaluates to *datum*. *datum* may be any external representation of a GoLisp object. Use `quote` to include literal constants in Scheme code.

```
(quote a)                                ⇒  a
(quote #(a b c))                         ⇒  #(a b c)
(quote (+ 1 2))                          ⇒  (+ 1 2)
```

`(quote datum)` may be abbreviated as `'datum`. The two notations are equivalent in all respects.

```
'a                                       ⇒  a
'#(a b c)                                ⇒  #(a b c)
'(+ 1 2)                                 ⇒  (+ 1 2)
'(quote a)                               ⇒  (quote a)
''a                                      ⇒  (quote a)
```

Numeric constants, string constants, character constants, and boolean constants evaluate to themselves, so they don't need to be quoted.

```
'"abc"                                   ⇒  "abc"
"abc"                                    ⇒  "abc"
'145932                                  ⇒  145932
145932                                   ⇒  145932
'#t                                      ⇒  #t
#t                                       ⇒  #t
```

3.8.2 (quasiquote *template*)

"Backquote" or "quasiquote" expressions are useful for constructing a list or vector structure when most but not all of the desired structure is known in advance. If no commas appear within the *template*, the result of evaluating is equivalent (in the sense of `equal?`) to the result of evaluating `'template`. If a comma appears within the *template*, however, the expression following the comma is evaluated ("unquoted") and its result is inserted into the structure instead of the comma and the expression. If a comma appears followed immediately by an at-sign (@), then the following expression shall evaluate to a list; the opening and closing parentheses of the list are then "stripped away" and the elements of the list are inserted in place of the comma at-sign expression sequence.

```
‘(list ,(+ 1 2) 4)                          ⇒  (list 3 4)

(let ((name 'a)) ‘(list ,name ',name))      ⇒  (list a 'a)

‘(a ,(+ 1 2) ,@(map abs '(4 -5 6)) b)       ⇒  (a 3 4 5 6 b)

‘((foo ,(- 10 3)) ,@(cdr '(c)) . ,(car '(cons)))
                                            ⇒  ((foo 7) . cons)

‘#(10 5 ,(sqrt 4) ,@(map sqrt '(16 9)) 8)
                                            ⇒  #(10 5 2 4 3 8)

‘,(+ 2 3)                                   ⇒  5
```

Quasiquote forms may be nested. Substitutions are made only for unquoted components appearing at the same nesting level as the outermost backquote. The nesting level increases by one inside each successive quasiquotation, and decreases by one inside each unquotation.

```
‘(a ‘(b ,(+ 1 2) ,(foo ,(+ 1 3) d) e) f)
     ⇒  (a ‘(b ,(+ 1 2) ,(foo 4 d) e) f)

(let ((name1 'x)
      (name2 'y))
  ‘(a ‘(b ,,name1 ,',name2 d) e))       ⇒  (a ‘(b ,x ,'y d) e)
```

The above notations and `(quasiquote template)` are identical in all respects. is identical to and is identical to .

```
(quasiquote (list (unquote (+ 1 2)) 4))
          ⇒  (list 3 4)

'(quasiquote (list (unquote (+ 1 2)) 4))
          ⇒  `(list ,(+ 1 2) 4)
          _i.e.,_ (quasiquote (list (unquote (+ 1 2)) 4))
```

Unpredictable behavior can result if any of the symbols `quasiquote`, `unquote`, or `unquote-splicing` appear in a *template* in ways otherwise than as described above.

3.9 Macros

3.9.1 (define-macro (*formals*) *template*)

3.9.2 (defmacro (*formals*) *template*)

The former is from standard Scheme and is the preferred name. The latter is retained for backward compatibility with earlier versions of GoLisp.

Create a named macro:

formals is the same as in a procedure definition: a name followed by formal parameters, if any. **NOTE** that the arguments to a macro invocation are **not** evaluated, but are passed as is to the macro to do with as it wishes.

template the template expression that is processed when the macro is invoked. The result of evaluating the processed template expression becomes the value of the macro's invocation. *template* is typically (even always) a quasiquoted expression using the formal parameter names for purposes of unquotiing in order to fill in the template.

```
(define-macro (double x)
   `(+ ,x ,x))

(double 5) ⇒ 10
```

3.9.3 (expand *name* [*object...*])

Expands the macro named by *name*, passing the evaluated sequence of *object* as arguments. **NOTE:** whereas invoking the macro (in the same way you invoke a function) expands and evaluates, `expand` (as you would expect) only expands the macro, resulting in the expanded template sexpr. This can then be evaluated as desired.

```
(define-macro (double x)
   `(+ ,x ,x))

(expand double 5) ⇒ (+ 5 5)
```

3.10 Sequencing

The `begin` special form is used to evaluate expressions in a particular order.

3.10.1 (begin *expression ...*)

The *expressions* are evaluated sequentially from left to right, and the value of the last *expression* is returned. This expression type is used to sequence side effects such as input and output. Keep in mind, `begin` does **not** create a nested lexical environment.

```
(define x 0)
(begin (set! x 5)
       (+ x 1))                ⇒  6

(begin (display "4 plus 1 equals ")
       (display (+ 4 1)))
                               -|  4 plus 1 equals 5
                               ⇒  unspecified
```

Often the use of `begin` is unnecessary, because many special forms already support sequences of expressions (that is, they have an implicit `begin`). Some of these special forms are:

- case

- cond
- define ;"procedure define" only
- do
- lambda
- let
- let*
- letrec

3.10.2 (-> *value sexpr|symbol...*)

This creates a function chain. *value* (evaluated first) is used as the first argument to the first *sexpr*. The result of each *sexpr* is used as the first argument of the next, and the result of the final *sexpr* is the value of the `->` form. If a *sexpr* would take a single argument (which would be provided by the *value* or the result of the previous *sexpr*, just the function name can be used.

The form `(-> 0 a b c)` is equivalent to `(c (b (a 0)))`.

```
(-> 1 (+ 3) (- 2))     ⇒ 2     ; (- (+ 1 3) 2)
(-> 1 (+ 3) (- 2) str) ⇒ "2"   ; (str (- (+ 1 3) 2))
```

The major advantage of this form is avoiding having to create a sequence on intermediate values/bindings purely to support the data flow. `->` lets you create a pipeline that you can put one value into and get the final result out the other end.

3.10.3 (=> *value sexpr|symbol...*)

This operates similarly to `->` with two differences:

1. *value* (evaluated **once** at the beginning) is used as the initial argument to **each** function, and they are independent and do not pass results one to another.
2. *value* is the result of the form.

The expression

```
(=> 1 a b c)
```

is equivalent to

```
(begin
  (a 1)
  (b 1)
  (c 1)
  1)
```

and

```
(=> (+ x 1) a b c)
```

is the same as

```
(let ((y (+ x 1)))
  (a y)
  (b y)
  (c y)
  y)
```

3.11 Conditionals

The behavior of the "conditional expressions" is determined by whether objects are true or false. The conditional expressions count only `#f` as false. They count everything else, including `#t`, pairs, symbols, numbers, strings, vectors, and procedures as true.

In the descriptions that follow, we say that an object has "a true value" or "is true" when the conditional expressions treat it as true, and we say that an object has "a false value" or "is false" when the conditional expressions treat it as false.

3.11.1 (cond *clause...*)

Each *clause* has this form:

```
(predicate expression...)
```

where *predicate* is any expression. The last *clause* may be an "`else` clause", which has the form:

```
(else expression...)
```

A `cond` expression does the following:

1. Evaluates the *predicate* expressions of successive *clauses* in order, until one of the *predicates* evaluates to a true value.
2. When a *predicate* evaluates to a true value, `cond` evaluates the *expressions* in the associated *clause* in left to right order, and returns the result of evaluating the last *expression* in the *clause* as the result of the entire `cond` expression. If the selected *clause* contains only the *predicate* and no *expressions*, `cond` returns the value of the *predicate* as the result.
3. If all *predicates* evaluate to false values, and there is no `else` clause, the result of the conditional expression is unspecified; if there is an `else` clause, `cond` evaluates its *expressions* (left to right) and returns the value of the last one.

```
(cond ((> 3 2) 'greater)
      ((< 3 2) 'less))                ⇒  greater

(cond ((> 3 3) 'greater)
      ((< 3 3) 'less)
      (else 'equal))                  ⇒  equal
```

Normally, programs should not depend on the value of a `cond` expression that has no `else` clause. However, some Scheme programmers prefer to write `cond` expressions in which at least one of the *predicates* is always true. In this style, the final *clause* is equivalent to an `else` clause.

GoLisp (and Scheme) supports an alternative clause syntax:

```
(predicate => recipient)
```

where *recipient* is an expression. If *predicate* evaluates to a true value, then *recipient* is evaluated. Its value must be a procedure of one argument; this procedure is then invoked on the value of the *predicate*.

```
(cond ((assv 'b '((a 1) (b 2))) => cadr)
      (else #f))                          ⇒  2
```

3.11.2 (case *key clause...*)

key may be any expression. Each *clause* has this form:

```
((object...) expression...)
```

No *object* is evaluated, and all the *objects* must be distinct. The last *clause* may be an "**else** clause", which has the form:

```
(else expression...)
```

A `case` expression does the following:

1. Evaluates *key* and compares the result with each *object*.
2. If the result of evaluating *key* is equivalent (in the sense of `eqv?`) to an *object*, `case` evaluates the *expressions* in the corresponding *clause* from left to right and returns the result of evaluating the last *expression* in the *clause* as the result of the `case` expression.
3. If the result of evaluating *key* is different from every *object*, and if there's an `else` clause, `case` evaluates its *expressions* and returns the result of the last one as the result of the `case` expression. If there's no `else` clause, `case` returns an unspecified result. Programs should not depend on the value of a `case` expression that has no `else` clause.

For example,

```
(case (* 2 3)
   ((2 3 5 7) 'prime)
   ((1 4 6 8 9) 'composite)) ⇒  composite

(case (car '(c d))
   ((a) 'a)
   ((b) 'b))                 ⇒  unspecified

(case (car '(c d))
   ((a e i o u) 'vowel)
   ((w y) 'semivowel)
   (else 'consonant))        ⇒  consonant
```

3.11.3 (and *expression...*)

The *expressions* are evaluated from left to right, and the value of the first *expression* that evaluates to a false value is returned. Any remaining *expressions* are not evaluated. If all the *expressions* evaluate to true values, the value of the last *expression* is returned. If there are no *expressions* then `#t` is returned.

```
(and (= 2 2) (> 2 1)) ⇒  #t
(and (= 2 2) (< 2 1)) ⇒  #f
(and 1 2 'c '(f g))   ⇒  (f g)
(and)                 ⇒  #t
```

3.11.4 (or *expression...*)

The *expressions* are evaluated from left to right, and the value of the first *expression* that evaluates to a true value is returned. Any remaining *expressions* are not evaluated. If all *expressions* evaluate to false values, the value of the last *expression* is returned. If there are no *expressions* then `#f` is returned.

```
(or (= 2 2) (> 2 1))           ⇒  #t
(or (= 2 2) (< 2 1))           ⇒  #t
(or #f #f #f)                  ⇒  #f
(or (memq 'b '(a b c)) (/ 3 0)) ⇒  (b c)
```

3.11.5 (if *predicate consequent* [*alternative*])

`if` is a macro based on `cond`.

predicate, *consequent*, and *alternative* are expressions. An `if` expression is evaluated as follows: first, *predicate* is evaluated. If it yields a true value, then *consequent* is evaluated and its value is returned. Otherwise *alternative* is evaluated and its value is returned. If *predicate* yields a false value and no *alternative* is specified, then the result of the expression is unspecified.

An `if` expression evaluates either *consequent* or *alternative*, never both. Programs should not depend on the value of an `if` expression that has no *alternative*.

```
(if (> 3 2) 'yes 'no)                    ⇒  yes
(if (> 2 3) 'yes 'no)                    ⇒  no
(if (> 3 2)
    (- 3 2)
    (+ 3 2))                             ⇒  1
```

3.11.6 (when *predicate expression...*)

`when` is a macro based on `cond`.

If *predicate* evaluates to logically `true`, the sequence of *expresions* is evaluated and the result of the last one is the result of the `when` form, otherwise `nil` is the result.

```
(when (> x 5)
  (write-line "greater")
  (+ x 2))
```

The above is equivalent to the following, but is simpler and clearer.

```
(if (> x 5)
    (begin (write-line "greater")
           (+ x 2)))
```

3.11.7 (unless *predicate expression...*)

`unless` is a macro based on `cond`.

If *predicate* evaluates to logically `false`, the sequence of *expresions* is evaluated and the result of the last one is the result of the `unless` form, otherwise `nil` is the result.

```
(unless (> x 5)
  (write-line "greater")
  (+ x 2))
```

The above is equivalent to the following, but is much simpler and clearer.

```
(if (> x 5)
    ()
    (begin (write-line "greater")
           (+ x 2)))
```

3.12 Iteration

The "iteration expressions" are: "named `let`" and `do`. They are also binding expressions, but are more commonly referred to as iteration expressions.

3.12.1 (let *name* ((*variable init*)...) _expression...)

GoLisp permits a variant on the syntax of *let* called "named `let`" which provides a more general looping construct than `do`, and may also be used to express recursions.

Named `let` has the same syntax and semantics as ordinary `let` except that *name* is bound within the *expressions* to a procedure whose formal arguments are the *variables* and whose body is the *expressions*. Thus the execution of the *expressions* may be repeated by invoking the procedure named by *name*.

GoLisp allows any of the *inits* to be omitted, in which case the corresponding *variables* are unassigned.

Note: the following expressions are equivalent:

```
(let name ((variable init)...)
  expression...)

((letrec ((name
           (lambda (variable...)
             expression...)))
   name)
 init...)
```

Here is an example:

```
(let loop
     ((numbers '(3 -2 1 6 -5))
      (nonneg '())
      (neg '()))
  (cond ((null? numbers)
         (list nonneg neg))
        ((>= (car numbers) 0)
         (loop (cdr numbers)
               (cons (car numbers) nonneg)
               neg))
        (else
         (loop (cdr numbers)
               nonneg
               (cons (car numbers) neg)))))

     ⇒  ((6 1 3) (-5 -2))
```

3.12.2 (do ((*variable init step*)...) (*test expression*...) *command*...)

`do` is an iteration construct. It specifies a set of variables to be bound, how they are to be initialized at the start, and how they are to be updated on each iteration. When a termination condition is met, the loop exits with a specified result value.

`do` expressions are evaluated as follows: The *init* expressions are evaluated (in some unspecified order), the *variables* are bound to fresh locations, the results of the *init* expressions are stored in the bindings of the *variables*, and then the iteration phase begins.

Each iteration begins by evaluating *test*; if the result is false, then the *command* expressions are evaluated in order for effect, the *step* expressions are evaluated

in some unspecified order, the *variables* are bound to fresh locations, the results of the *steps* are stored in the bindings of the *variables*, and the next iteration begins.

If *test* evaluates to a true value, then the *expressions* are evaluated from left to right and the value of the last *expression* is returned as the value of the `do` expression. If no *expressions* are present, then the value of the `do` expression is the empty list (i.e. `nil`).

The region of the binding of a *variable* consists of the entire `do` expression except for the *inits*. It is an error for a *variable* to appear more than once in the list of `do` variables.

A *step* may be omitted, in which case the effect is the same as if `(variable init variable)` had been written instead of `(variable init)`.

```
(do ((vec (make-vector 5))
     (i 0 (+ i 1)))
    ((= i 5) vec)
  (vector-set! vec i i))        ⇒  #(0 1 2 3 4)

(let ((x '(1 3 5 7 9)))
  (do ((x x (cdr x))
       (sum 0 (+ sum (car x))))
      ((null? x) sum)))         ⇒  25
```

3.13 Eval/Apply

3.13.1 (apply *function object...*)

Apply the function that results from evaluating *function* to the argument list resulting from evaluating each *object*.

Each initial *object* can be any type of object, but the final one (and there must be at least one *object*) must be a list.

```
(apply + 1 2 '(3 4)) ⇒ 10
(apply + '(1 2 3 4)) ⇒ 10
```

3.13.2 (eval *expression*)

Evaluate *expression* in the current environment.

```
(eval '(+ 1 2 3 4)) ⇒ 10
```

3.13.3 (definition-of *function*)

Fetch the definition of *function*. This returns an expression that can be evaluated to define it. One use of this is to copy definitions to a source file.

```
(define (square x)
  (* x x))

(definition-of square) ⇒ (define (square x) (* x x))

(define square (lambda (x)
                 (* x x)))

(definition-of square) ⇒ (define square (lambda (x) (* x x)))
```

4 Type tests

4.0.1 (atom? *object*)

Returns whether *object* is an atom, i.e. a number, string, boolean, or symbol.

4.0.2 (list? *object*)

Returns whether *object* is a list.

4.0.3 (pair? *object*)

Returns whether *object* is a pair, i.e. a list or a dotted pair.

4.0.4 (alist? *object*)

Returns whether *object* is an association list.

4.0.5 (vector? *object*)

Returns `#t` if `object` is a vector; otherwise returns `#f`.

4.0.6 (nil? *object*)

4.0.7 (null? *object*)

Returns whether *object* is an empty list, i.e. the nil value.

4.0.8 (notnil? *object*)

4.0.9 (notnull? *object*)

Returns whether *object* is not an empty list, i.e. the nil value.

4.0.10 (symbol? *object*)

Returns whether *object* is a symbol.

4.0.11 (string? *object*)

Returns whether *object* is a string.

4.0.12 (number? *object*)

Returns whether *object* is a number.

4.0.13 (integer? *object*)

Returns whether *object* is an integer.

4.0.14 (float? *object*)

Returns whether *object* is a floating point number.

4.0.15 (function? *object*)

Returns whether *object* is a function.

4.0.16 (macro? *object*)

Returns whether *object* is a macro.

4.0.17 (frame? *object*)

Returns whether *object* is a frame.

4.0.18 (bytearray? *object*)

Returns whether *object* is a bytearray.

4.0.19 (port? *object*)

Returns whether *object* is a port.

4.0.20 (channel? *object*)

Returns whether *object* is a channel.

4.1 Numerical Operations

4.1.1 (+ *number...*)

4.1.2 (* *number ...*)

These procedures return the sum or product of their arguments.

```
(+ 3 4)                 ⇒  7
(+ 3)                   ⇒  3
(+)                     ⇒  0
(* 4)                   ⇒  4
(*)                     ⇒  1
```

4.1.3 (- *number...*)

4.1.4 (/ *number...*)

With two or more arguments, these procedures return the difference or quotient of their arguments, associating to the left. With one argument, however, they return the additive or multiplicative inverse of their argument.

```
(- 3 4)                 ⇒  -1
(- 3 4 5)               ⇒  -6
(- 3)                   ⇒  -3
(/ 3 4 5)               ⇒  0.15
(/ 4)                   ⇒  0.25
```

Note that `/` always preforms floating point division. If the quotient is a whole number it will be returned as an integer.

```
(/ 12 5)                ⇒  2.4
(/ 12 2.4)              ⇒  5
```

4.1.5 (succ *integer*)

4.1.6 (1+ *integer*)

Equivalent to `(+ integer 1)`.

4.1.7 (pred *integer*)

4.1.8 (-1+ *integer*)

Equivalent to `(- integer 1)`.

4.1.9 (quotient *n1* _n2)

4.1.10 (remiander *n1 n2*)

4.1.11 (modulo *n1 n2*)

These procedures implement number-theoretic (integer) division: for positive integers *n1* and *n2*, if *n3* and *n4* are integers such that

```
n1 = (n2 * n3) + n4

0 <= n4 < n2
```

then

```
(quotient n1 n2)        ⇒  n3
(remainder n1 n2)       ⇒  n4
(modulo n1 n2)          ⇒  n4
```

for integers *n1* and *n2* with *n2* not equal to 0,

```
(= n1
   (+ (* n2 (quotient n1 n2))
      (remainder n1 n2)))
                          ⇒  #t
```

The value returned by `quotient` always has the sign of the product of its arguments. `remainder` and `modulo` differ on negative arguments – the `remainder` always has the sign of the dividend, the `modulo` always has the sign of the divisor:

```
(modulo 13 4)           ⇒  1
(remainder 13 4)        ⇒  1

(modulo -13 4)          ⇒  3
(remainder -13 4)       ⇒  -1

(modulo 13 -4)          ⇒  -3
(remainder 13 -4)       ⇒  1

(modulo -13 -4)         ⇒  -1
(remainder -13 -4)      ⇒  -1
```

The `%` function is an alias for `remainder`.

4.1.12 (floor *number*)

Returns the greatest integer value less than or equal to *number*. *number* can be an integer or float. Return value is a float.

```
(floor 3.4)             ⇒ 3.0
(floor -3.4)            ⇒ -4.0
(floor 3)               ⇒ 3.0
```

4.1.13 (ceiling *number*)

Returns the largest integer value greater than or equal to *number*. *number* can be an integer or float. Return value is a float.

```
(ceiling 3.4)           ⇒ 4.0
(ceiling -3.4)          ⇒ -3.0
(ceiling 3)             ⇒ 3.0
```

4.1.14 (integer *number*)

Returns the integer value of *number*. If it is an integer, it is simply returned. However, if it is a float the integer part is returned.

```
(integer 5)             ⇒ 5
(integer 5.2)           ⇒ 5
(integer -5.8)          ⇒ -5
```

4.1.15 (float *number*)

Returns the float value of *number*. If it is a float, it is simply returned. However, if it is an integer the corresponding float is returned.

```
(float 5)               ⇒ 5.0
```

Note that converting a float to a string for printing using the format `%g` to use the minimum number of characters so `5.0` will actually print as `5`.

4.1.16 (number->string *number* [*base*])

Converts *number* (first converted to an integer) to a string, in the given *base*. Allowed bases are 2, 8, 10, and 16. If the base is omitted, 10 is used. No base prefixes (e.g. `0x` for base 16) are added.

```
(number->string 42)    ⇒ "42"
(number->string 42 2)  ⇒ "101010"
(number->string 42 8)  ⇒ "52"
(number->string 42 10) ⇒ "42"
(number->string 42 16) ⇒ "2a"
(number->string 42 15) ⇒ ERROR number->string: unsupported base 15
```

4.1.17 (string->number *numeric-string* [*base*])

Converts *numeric-string* to an integer, in the given base. Allowed bases are 2, 8, 10, and 16. If the base is omitted, 10 is used. No base prefixes (e.g. `0x` for base 16) are allowed. Specifying an unsupported base will result in `0`.

```
(string->number "42")       ⇒ 42
(string->number "101010" 2) ⇒ 42
(string->number "52" 8)     ⇒ 42
(string->number "42" 10)    ⇒ 42
(string->number "2a" 16)    ⇒ 42
(string->number "42" 15)    ⇒ ERROR number->string: unsupported base 15
```

4.2 Comparisons

All comparison operations work with floating point numbers as well.

4.2.1 (== *number1 number2*)

4.2.2 (= *number1 number2*)

4.2.3 (!= *number1 number2*)

4.2.4 (/= *number1 number2*)

4.2.5 (< *number1 number2 ...*)

4.2.6 (> *number1 number2 ...*)

4.2.7 (<= *number1 number2 ...*)

4.2.8 (>= *number1 number2 ...*)

These procedures return `#t` if their arguments are (respectively): equal (two alternatives), not equal (two alternatives), monotonically increasing, monotonically decreasing, monotonically nondecreasing, or monotonically nonincreasing. They return `#f` otherwise. Note that `<`, `>`, `<=`, and `>=` can take more than 2 arguments.

4.2.9 (zero? *number*)

4.2.10 (positive? *number*)

4.2.11 (negative? *number*)

4.2.12 (odd? *number*)

4.2.13 (even? *number*)

These numerical predicates test a number for a particular property, returning `#t` or `#f`.

4.2.14 (min *number...*)

4.2.15 (min (*number...*))

4.2.16 (max *number...*)

4.2.17 (max (*number...*))

These procedures return the maximum or minimum of their arguments. Note that the arguments can be a series of numbers or a list of numbers:

```
(min 3 7 1 2)      ⇒ 1
(min '(3 7 1 2))   ⇒ 1
```

4.2.18 (log *number*)

This computes the natural logarithm of *number* **(not the base ten logarithm)**. An integer argument will be converted to a float. The result is always a float.

5 Equivalence Predicates

A "predicate" is a procedure that always returns a boolean value (`#t` or `#f`). An "equivalence predicate" is the computational analogue of a mathematical equivalence relation (it is symmetric, reflexive, and transitive). Of the equivalence predicates described in this section, `eq?` is the finest or most discriminating, and `equal?` is the coarsest. `eqv?` is slightly less discriminating than `eq?`.

5.0.1 (eqv? *obj1 obj2*)

The `eqv?` procedure defines a useful equivalence relation on objects. Briefly, it returns `#t` if *obj1* and *obj2* should normally be regarded as the same object.

The `eqv?` procedure returns `#t` if:

- *obj1* and *obj2* are both `#t` or both `#f`.
- *obj1* and *obj2* are both interned symbols and

```
(string=? (symbol->string obj1)
          (symbol->string obj2))
      ⇒ #t
```

- *obj1* and *obj2* are both numbers, are numerically equal according to the `=` procedure.
- both *obj1* and *obj2* are the empty list.
- *obj1* and *obj2* are procedures whose location tags are equal.
- *obj1* and *obj2* are pairs, vectors, strings, byte arrays, records, cells, or weak pairs that denote the same locations in the store.

The `eqv?` procedure returns `#f` if:

- *obj1* and *obj2* are of different types.
- one of *obj1* and *obj2* is `#t` but the other is `#f`.
- *obj1* and *obj2* are symbols but

```
(string=? (symbol->string obj1)
          (symbol->string obj2))
     ⇒ #f
```

- *obj1* and *obj2* are numbers for which the `=` procedure returns `#f`.
- one of *obj1* and *obj2* is the empty list but the other is not.
- *obj1* and *obj2* are procedures which have distinct underlying representations.
- *obj1* and *obj2* are pairs, vectors, strings, byte arrays, or frames that denote distinct locations.

Some examples:

```
(eqv? 'a 'a)                 ⇒  #t
(eqv? 'a 'b)                 ⇒  #f
(eqv? 2 2)                   ⇒  #t
(eqv? '() '())               ⇒  #t
(eqv? 100000000 100000000)   ⇒  #t
(eqv? (cons 1 2) (cons 1 2)) ⇒  #f
(eqv? (lambda () 1)
      (lambda () 2))         ⇒  #f
(eqv? #f 'nil)               ⇒  #f
(let ((p (lambda (x) x)))
  (eqv? p p))                ⇒  #t
```

The following examples illustrate how GoLisp behaves in cases where MIT Scheme's rules do not fully specify the behavior of 'eqv?'.

```
(eqv? "" "")                 ⇒  #f
(eqv? '#() '#())             ⇒  #f
(eqv? (lambda (x) x)
      (lambda (x) x))        ⇒  #f
(eqv? (lambda (x) x)
      (lambda (y) y))        ⇒  #f
```

Objects of distinct types must never be regarded as the same object.

Since it is an error to modify constant objects (those returned by literal expressions), the implementation may share structure between constants where appropriate. Thus the value of `eqv?` on constants is sometimes unspecified, however, the following cases hold.

```
(let ((x '(a)))
  (eqv? x x))                ⇒  #t
(eqv? '(a) '(a))             ⇒  #f
(eqv? "a" "a")               ⇒  #f
(eqv? '(b) (cdr '(a b)))     ⇒  #f
```

5.0.2 (eq? *obj1 obj2*)

`eq?` is similar to `eqv?` except that in some cases it is capable of discerning distinctions finer than those detectable by `eqv?`.

`eq?` and `eqv?` are guaranteed to have the same behavior on symbols, booleans, the empty list, pairs, records, and non-empty strings and vectors. `eq?`'s behavior on numbers and characters is implementation-dependent, but it will always return either true or false, and will return true only when `eqv?` would also return true. `eq?` may also behave differently from `eqv?` on empty vectors and empty strings.

```
(eq? 'a 'a)                  ⇒  #t
(eq? '(a) '(a))              ⇒  #f
(eq? (list 'a) (list 'a))    ⇒  #f
(eq? "a" "a")                ⇒  #t
(eq? "" "")                  ⇒  #t
(eq? '() '())                ⇒  #t
(eq? 2 2)                    ⇒  #t
(eq? car car)                ⇒  #t
(let ((n (+ 2 3)))
  (eq? n n))                 ⇒  #t
(let ((x '(a)))
  (eq? x x))                 ⇒  #t
(let ((x '#()))
  (eq? x x))                 ⇒  #t
(let ((p (lambda (x) x)))
  (eq? p p))                 ⇒  #t
```

5.0.3 (equal? *obj1 obj2*)

`equal?` recursively compares the contents of pairs, vectors, and strings, applying `eqv?` on other objects such as numbers, symbols, and booleans. A rule of thumb is that objects are generally `equal?` if they print the same. `equal?` may fail to terminate if its arguments are circular data structures.

```
(equal? 'a 'a)               ⇒  #t
(equal? '(a) '(a))           ⇒  #t
(equal? '#(a) '#(a))         ⇒  #t
(equal? '(a (b) c)
        '(a (b) c))          ⇒  #t
(equal? "abc" "abc")         ⇒  #t
(equal? 2 2)                 ⇒  #t
(equal? (make-vector 5 'a)
        (make-vector 5 'a))  ⇒  #t
(equal? (lambda (x) x)
        (lambda (y) y))      ⇒  #f
```

5.0.4 (neqv? *object object*)

5.0.5 (neq? *object object*)

5.0.6 (nequal? *object object*)

Each is the logical negation of the corresponding equivalence predicate.

6 Logical

6.0.1 (boolean=? *object1 object2*)

Returns whether *object1* and *object2* are both truthy or are both falsy.

6.0.2 (boolean/and *object...*)

Each *object* is evaluated; if all of the evaluated *objects* are truthy, returns the value of the final one

6.0.3 (boolean/or *object...*)

Each *object* is evaluated; if any of the evaluated *objects* are truthy, returns the first one that is

6.0.4 (not *object*)

6.0.5 (false? *object*)

Returns the boolean negation of the argument.

```
(not #t)    ⇒ #f
(false? #f) ⇒ #t
```

7 Binary

7.0.1 (binary-and *int-1 int-2*)

Performs a bitwise AND of *int-1* and *int-2*, returning the result.

```
(number->string (binary-and 0xaa 0x0f) 16) ⇒ "a"
(number->string (binary-and 0xaa 0xf0) 16) ⇒ "a0"
```

7.0.2 (binary-or *int-1 int-2*)

Performs a bitwise OR of *int-1* and *int-2*, returning the result.

```
(number->string (binary-or 0xaa 0x0f) 16) ⇒ "af"
(number->string (binary-or 0xaa 0xf0) 16) ⇒ "fa"
```

7.0.3 (binary-not *int*)

Performs a bitwise NOT of *int*, returning the result.

```
(number->string (binary-not 0x000000aa) 16) ⇒ "ffffff55"
```

7.0.4 (left-shift *int count*)

Shifts *int* left by *count* bits, returning the result.

```
(number->string (left-shift (string->number "10101010" 2) 1) 2) ⇒ "101010100"
(number->string (left-shift (string->number "10101010" 2) 3) 2) ⇒ "10101010000"
```

7.0.5 (right-shift *int count*)

Shifts *int* right by *count* bits, returning the result.

```
(number->string (right-shift (string->number "10000" 2) 1) 2) ⇒ "1000"
(number->string (right-shift (string->number "10000" 2) 4) 2) ⇒ "1"
```

8 Characters

GoLisp has minimal support for characters: just that required for basic string manipulation. this may be expanded in the future.

8.1 External Representation of Characters

Characters are written using the notation `\#CHARACTER` or `#\CHARACTER-NAME`. For example:

```
#\a                     ; lowercase letter
#\A                     ; uppercase letter
#\(                     ; left parenthesis
#\space                 ; the space character
#\newline               ; the newline character
```

Case is significant in `#\CHARACTER`, but not in `#\CHARACTER-NAME`. If CHARACTER in `#\CHARACTER` is a letter, *CHARACTER* must be followed by a delimiter character such as a space or closing parenthesis, bracket, or brace. Characters written in the `\#` notation are self-evaluating; you don't need to quote them.

The following *CHARACTER-NAME*s are supported, shown here with their ASCII equivalents:

```
Character Name          ASCII Name
 --------------          ----------

 altmode                 ESC
 backspace               BS
 esc                     ESC
 linefeed                LF
 page                    FF
 return                  CR
 rubout                  DEL
 space
 tab                     HT
```

9 Strings

A "string" is a immutable sequence of characters.

A string is written as a sequence of characters enclosed within double quotes " ". To include a double quote inside a string, precede the double quote with a backslash ‘’ (escape it), as in

```
"The word \"recursion\" has many meanings."
```

The printed representation of this string is

```
The word "recursion" has many meanings.
```

To include a backslash inside a string, precede it with another backslash; for example,

```
"Use #\\Control-q to quit."
```

The printed representation of this string is

```
Use #\Control-q to quit.
```

The effect of a backslash that doesn't precede a double quote or backslash is unspecified in standard Scheme, but GoLisp specifies the effect for three other characters: `\t`, `\n`, and `\f`. These escape sequences are respectively translated into tab, newline, and page characters. whose ISO-8859-1 code is those digits.

If a string literal is continued from one line to another, the string will contain the newline character at the line break. Standard Scheme does not specify what appears in a string literal at a line break.

The "length" of a string is the number of characters that it contains. This number is a non-negative integer that is established when the string is created. Each character in a string has an "index", which is a number that indicates the character's position in the string. The index of the first (leftmost) character in a string is 0, and the index of the last character is one less than the length of the string. The "valid indexes" of a string are the non-negative integers less than the length of the string.

```
0 <= start <= end <= (string-length string)
```

9.1 Construction of Strings

9.1.1 (make-string *k* [*char*])

Returns a newly allocated string of length *k*. If you specify *char*, all elements of the string are initialized to *char*, otherwise the contents of the string are unspecified.

```
(make-string 10 #\x)              ⇒  "xxxxxxxxxx"
```

9.1.2 (string *char...*)

Returns a newly allocated string consisting of the specified characters. The arguments should be single character strings.

```
(string "a")                          ⇒  "a"
(string "a" "b" #\c)                  ⇒  "abc"
(string #\a #\space #\b #\space #\c)  ⇒  "a b c"
(string)                              ⇒  undefined
```

9.1.3 (list->string *char-list*)

char-list must be a list of strings. `list->string` returns a newly allocated string formed by concatenating the elements of *char-list*. This is equivalent to `(apply string char-list)`. The inverse of this operation is `string->list`.

```
(list->string '(#\a #\b))       ⇒  "ab"
(string->list "Hello")          ⇒  (#\H #\e #\l #\l #\o)
```

9.1.4 (string-copy *string*)

Returns a newly allocated copy of *string*.

9.2 Selecting String Components

9.2.1 (string? *object*)

Returns `#t` if *object* is a string; otherwise returns `#f`.

```
(string? "Hi")                  ⇒  #t
(string? 'Hi)                   ⇒  #f
```

9.2.2 (string-length *string*)

Returns the length of *string* as an non-negative integer.

```
(string-length "")              ⇒  0
(string-length "The length")    ⇒  10
```

9.2.3 (string-null? *string*)

Returns `#t` if *string* has zero length; otherwise returns `#f`.

```
(string-null? "")               ⇒  #t
(string-null? "Hi")             ⇒  #f
```

9.2.4 (string-ref *string k*)

Returns character *k* of *string*. *k* must be a valid index of *string*.

```
(string-ref "Hello" 1)          ⇒  #\e
(string-ref "Hello" 5)          ERROR 5 not in correct range
```

9.2.5 (string-set! *string k char*)

Stores *char* (a single character string) in element *k* of *string* and returns an unspecified value. *k* must be a valid index of *string*.

```
(define s "Dog")              ⇒  "Dog"
(string-set! s 0 #\L)         ⇒  "Log"
s                             ⇒  "Log"
(string-set! s 3 #\t)         ERROR 3 not in correct range
```

9.3 Comparison of Strings

9.3.1 (string=? *string1 string2*)

9.3.2 (substring=? *string1 start1 end1 string2 start2 end2*)

9.3.3 (string-ci=? *string1 string2*)

9.3.4 (substring-ci=?_ string1_ *start1 end1 string2 start2 end2*)

Returns `#t` if the two strings (substrings) are the same length and contain the same characters in the same (relative) positions; otherwise returns `#f`. `string-ci=?` and `substring-ci=?` don't distinguish uppercase and lowercase letters, but `string=?` and `substring=?` do.

```
(string=? "PIE" "PIE")                  ⇒  #t
(string=? "PIE" "pie")                  ⇒  #f
(string-ci=? "PIE" "pie")               ⇒  #t
(substring=? "Alamo" 1 3 "cola" 2 4)    ⇒  #t ; compares "la"
```

9.3.5 (**string<?** ***string1 string2***)

9.3.6 (**substring<?_ string1_** ***start1 end1 string2 start2 end2***)

9.3.7 (**string-ci<?** ***string1 string2***)

9.3.8 (**substring-ci<?_ string1_** ***start1 end1 string2 start2 end2***)

9.3.9 (**string>?** ***string1 string2***)

9.3.10 (**substring>?_ string1_** ***start1 end1 string2 start2 end2***)

9.3.11 (**string-ci>?** ***string1 string2***)

9.3.12 (**substring-ci>?_ string1_** ***start1 end1 string2 start2 end2***)

9.3.13 (**string<=?** ***string1 string2***)

9.3.14 (**substring<=?_ string1_** ***start1 end1 string2 start2 end2***)

9.3.15 (**string-ci<=?** ***string1 string2***)

9.3.16 (**substring-ci<=?_ string1_** ***start1 end1 string2 start2 end2***)

9.3.17 (**string>=?** ***string1 string2***)

9.3.18 (**substring>=?_ string1_** ***start1 end1 string2 start2 end2***)

9.3.19 (**string-ci>=?** ***string1 string2***)

9.3.20 (**substring-ci>=?_ string1_** ***start1 end1 string2 start2 end2***)

These procedures compare strings (substrings) according to the order of the characters they contain. The arguments are compared using a lexicographic (or dictionary) order. If two strings differ in length but are the same up to the length of the shorter string, the shorter string is considered to be less than the longer string.

```
(string<? "cat" "dog")          ⇒  #t
(string<? "cat" "DOG")          ⇒  #f
(string-ci<? "cat" "DOG")       ⇒  #t
(string>? "catkin" "cat")       ⇒  #t ; shorter is lesser
```

9.3.21 (string-compare *string1 string2 if-eq if-lt if-gt*)

9.3.22 (string-compare-ci *string1 string2 if-eq if-lt if-gt*)

if-eq, *if-lt*, and *if-gt* are procedures of no arguments (thunks). The two strings are compared; if they are equal, *if-eq* is applied, if *string1* is less than *string2*, *if-lt* is applied, else if *string1* is greater than *string2*, *if-gt* is applied. The value of the procedure is the value of the thunk that is applied.

`string-compare` distinguishes uppercase and lowercase letters; `string-compare-ci` does not.

```
(define (cheer) (display "Hooray!"))
(define (boo)   (display "Boo-hiss!"))
(string-compare "a" "b"  cheer  (lambda() 'ignore)  boo)
        -|  Hooray!
        ⇒  unspecified
```

In GoLisp, only `string-compare` and `string-compare-ci` are available by default. If you want the other comparison functions you need to `(load "lisp/strings.scm")`.

9.4 Alphabetic Case in Strings

9.4.1 (string-capitalized? *string*)

9.4.2 (substring-capitalized? *string start end*)

These procedures return `#t` if the first word in the string (substring) is capitalized, and any subsequent words are either lower case or capitalized. Otherwise, they return `#f`. A word is defined as a non-null contiguous sequence of alphabetic characters, delimited by non-alphabetic characters or the limits of the string (substring). A word is capitalized if its first letter is upper case and all its remaining letters are lower case.

```
(map string-capitalized? '(""    "A"    "art"  "Art"  "ART"))
                       ⇒ (#f    #t     #f     #t     #f)
```

9.4.3 (string-upper-case? *string*)

9.4.4 (substring-upper-case? *string start end*)

9.4.5 (string-lower-case? *string*)

9.4.6 (substring-lower-case? *string start end*)

These procedures return `#t` if all the letters in the string (substring) are of the correct case, otherwise they return `#f`. The string (substring) must contain at least one letter or the procedures return `#f`.

```
(map string-upper-case?  '(""    "A"    "art"  "Art"  "ART"))
                       ⇒ (#f    #t     #f     #f     #t)
```

9.4.7 (string-capitalize *string*)

9.4.8 (string-capitalize! *string*)

9.4.9 (substring-capitalize! *string start end*)

`string-capitalize` returns a newly allocated copy of *string* in which the first alphabetic character is uppercase and the remaining alphabetic characters are lowercase. For example, `"abcDEF"` becomes `"Abcdef"`. `string-capitalize!` is the destructive version of `string-capitalize`: it alters *string* and returns an unspecified value. `substring-capitalize!` destructively capitalizes the specified part of *string*.

9.4.10 (string-downcase *string*)

9.4.11 (string-downcase! *string*)

9.4.12 (substring-downcase! *string start end*)

`string-downcase` returns a newly allocated copy of *string* in which all uppercase letters are changed to lowercase. `string-downcase!` is the destructive version of `string-downcase`: it alters *string* and returns an unspecified value. `substring-downcase!` destructively changes the case of the specified part of *string*.

```
(define str "ABCDEFG")          ⇒  unspecified
(substring-downcase! str 3 5)   ⇒  "ABCdeFG"
str                             ⇒  "ABCdeFG"
```

9.4.13 (string-upcase *string*)

9.4.14 (string-upcase! *string*)

9.4.15 (substring-upcase! *string start end*)

`string-upcase` returns a newly allocated copy of *string* in which all lowercase letters are changed to uppercase. `string-upcase!` is the destructive version of `string-upcase`: it alters *string* and returns an unspecified value. `substring-upcase!` destructively changes the case of the specified part of *string*.

9.5 Cutting and Pasting Strings

9.5.1 (string-split *string separator*)

Splits *string* into a list of substrings that are separated by *separator*.

```
(string-split "1-2-3" "-")) ⇒ ("1" "2" "3")
```

9.5.2 (string-join *strings separator*)

Joins the list of *strings* into a single string by interposing *separator*.

```
(string-join '("1" "2" "3") "-") ⇒ "1-2-3"
```

9.5.3 (string-append *string...*)

Returns a newly allocated string made from the concatenation of the given strings.

```
(string-append)                  ⇒  undefined
(string-append "*" "ace" "*")    ⇒  "*ace*"
(string-append "" "" "")         ⇒  ""
(eqv? str (string-append str))   ⇒  #f ; newly allocated
```

9.5.4 (substring *string start end*)

Returns a newly allocated string formed from the characters of *string* beginning with index *start* (inclusive) and ending with *end* (exclusive).

```
(substring "" 0 0)              ⇒ ""
(substring "arduous" 2 5)       ⇒ "duo"
(substring "arduous" 2 8)       ERROR 8 not in correct range

(define (string-copy s)
  (substring s 0 (string-length s)))
```

9.5.5 (string-head *string end*)

Returns a newly allocated copy of the initial substring of *string*, up to but excluding *end*. It could have been defined by:

```
(define (string-head string end)
  (substring string 0 end))

(string-head "uncommon" 2)      ⇒ "un"
```

9.5.6 (string-tail *string start*)

Returns a newly allocated copy of the final substring of *string*, starting at index *start* and going to the end of *string*. It could have been defined by:

```
(define (string-tail string start)
  (substring string start (string-length string)))

(string-tail "uncommon" 2)      ⇒  "common"
```

9.5.7 (string-pad-left *string k* [*char*])

9.5.8 (string-pad-right *string k* [*char*])

These procedures return a newly allocated string created by padding *string* out to length *k*, using *char*. If *char* is not given, it defaults to `#\space`. If *k* is less than the length of *string*, the resulting string is a truncated form of *string*. `string-pad-left` adds padding characters or truncates from the beginning of the string (lowest indices), while `string-pad-right` does so at the end of the string (highest indices).

```
(string-pad-left "hello" 4)             ⇒  "ello"
(string-pad-left "hello" 8)             ⇒  "   hello"
(string-pad-left "hello" 8 #\*)         ⇒  "***hello"
(string-pad-right "hello" 4)            ⇒  "hell"
(string-pad-right "hello" 8)            ⇒  "hello   "
(string-pad-right "hello" 8 #\*)        ⇒  "hello***"
```

9.5.9 (string-trim *string* [*char-set*])

9.5.10 (string-trim-left *string* [*char-set*])

9.5.11 (string-trim-right *string* [*char-set*])

Returns a newly allocated string created by removing all characters that are not in *char-set* from: `string-trim` both ends of *string*; `string-trim-left` the beginning of *string*; or `string-trim-right` the end of *string*. *char-set* defaults to `char-set:not-whitespace`.

```
(string-trim "  in the end  ")          ⇒  "in the end"
(string-trim "              ")          ⇒  ""
(string-trim "100th" char-set:numeric)  ⇒  "100"
(string-trim-left "-.-+-=-" (char-set #\+))
                                        ⇒  "+-=-"
(string-trim "but (+ x y) is" (char-set #\( #\)))
                                        ⇒  "(+ x y)"
```

9.6 Regexp Support

There is some preliminary support for regular expressions.

9.6.1 (re-string-match-go *regexp string*)

This matches regexp against the respective string, returning #f for no match, or a list of strings (see below) if the match succeeds.

When a successful match occurs, the above procedure returns a list of strings. Each string corresponds to an instance of the regular-expression grouping operator '('. Additionally, the first string corresponds to the entire substring matching the regular expression.

Note that this is different from the Scheme matching procedure.

10 Lists

A "pair" (sometimes called a "dotted pair") is a data structure with two fields called the "car" and "cdr" fields (for historical reasons). Pairs are created by the procedure `cons`. The car and cdr fields are accessed by the procedures `car` and `cdr`. The car and cdr fields are assigned by the procedures `set-car!` and `set-cdr!`.

Pairs are used primarily to represent "lists". A list can be defined recursively as either the empty list or a pair whose cdr is a list. More precisely, the set of lists is defined as the smallest set X such that

- The empty list is in X.
- If LIST is in X, then any pair whose cdr field contains LIST is also in X.

The objects in the car fields of successive pairs of a list are the "elements" of the list. For example, a two-element list is a pair whose car is the first element and whose cdr is a pair whose car is the second element and whose cdr is the empty list. The "length" of a list is the number of elements, which is the same as the number of pairs. The "empty list" is a special object of its own type (it is not a pair); it has no elements and its length is zero.

The most general notation (external representation) for GoLisp pairs is the "dotted" notation `(C1 . C2)` where C1 is the value of the car field and C2 is the value of the cdr field. For example, `(4 . 5)` is a pair whose car is `4` and whose cdr is `5`. Note that `(4 . 5)` is the external representation of a pair, not an expression that evaluates to a pair.

A more streamlined notation can be used for lists: the elements of the list are simply enclosed in parentheses and separated by spaces. The empty list is written `()`. For example, the following are equivalent notations for a list of symbols:

```
(a b c d e)
(a . (b . (c . (d . (e . ()))))))
```

Whether a given pair is a list depends upon what is stored in the cdr field. When the **`set-cdr!`** procedure is used, an object can be a list one moment and not the next:

```
(define x (list 'a 'b 'c))
(define y x)
y                                       ⇒ (a b c)
(list? y)                               ⇒ #t
(set-cdr! x 4)                          ⇒ (a . 4)
x                                       ⇒ (a . 4)
(eqv? x y)                              ⇒ #t
y                                       ⇒ (a . 4)
(list? y)                               ⇒ #f
(set-cdr! x x)                          ⇒ <Unprintable looping pair structure>
(list? y)                               ⇒ #f
```

A chain of pairs that doesn't end in the empty list is called an "improper list". Note that an improper list is not a list. The list and dotted notations can be combined to represent improper lists, as the following equivalent notations show:

```
(a b c . d)
(a . (b . (c . d)))
```

Within literal expressions and representations of objects read by the `read` procedure, the forms `'DATUM`, `‘DATUM`, `,DATUM`, and `,@DATUM` denote two-element lists whose first elements are the symbols `quote`, `quasiquote`, `unquote`, and `unquote-splicing`, respectively. The second element in each case is DATUM. This convention is supported so that arbitrary GoLisp programs may be represented as lists. Among other things, this permits the use of the `read` procedure to parse Scheme programs.

10.1 Pairs

This section describes the simple operations that are available for constructing and manipulating arbitrary graphs constructed from pairs.

10.1.1 (pair? *object*)

Returns `#t` if *object* is a pair; otherwise returns `#f`.

```
(pair? '(a . b))                    ⇒ #t
(pair? '(a b c))                    ⇒ #t
(pair? '())                         ⇒ #f
(pair? '#(a b))                     ⇒ #f
```

10.1.2 (cons *obj1 obj2*)

Returns a newly allocated pair whose car is *obj1* and whose cdr is *obj2*. The pair is guaranteed to be different (in the sense of `eqv?`) from every previously existing object.

```
(cons 'a '())                       ⇒ (a)
(cons '(a) '(b c d))                ⇒ ((a) b c d)
(cons "a" '(b c))                   ⇒ ("a" b c)
(cons 'a 3)                         ⇒ (a . 3)
(cons '(a b) 'c)                    ⇒ ((a b) . c)
```

10.1.3 (car *pair*)

Returns the contents of the car field of *pair*. Note that taking the `car` of the empty list results in the empty list.

```
(car '(a b c))                          ⇒ a
(car '((a) b c d))                      ⇒ (a)
(car '(1 . 2))                          ⇒ 1
(car '())                               ⇒ ()
```

10.1.4 (cdr *pair*)

Returns the contents of the cdr field of *pair*. Note that taking the `cdr` of the empty list results in the empty list.

```
(cdr '((a) b c d))                      ⇒ (b c d)
(cdr '(1 . 2))                          ⇒ 2
(cdr '())                               ⇒ ()
```

10.1.5 (set-car! *pair object*)

Stores *object* in the car field of *pair*. The value returned by `set-car!` is unspecified.

```
(define (f) (list 'not-a-constant-list))
(define (g) '(constant-list))
(set-car! (f) 3)                        ⇒ unspecified
(set-car! (g) 3)                        ERROR Illegal datum
```

10.1.6 (set-cdr! *pair object*)

Stores *object* in the cdr field of *pair*. The value returned by `set-cdr!` is unspecified.

10.1.7 (set-nth! *n list new-value*)

Set the `car` pointer of the nth cons cell of *list*. Numbering starts at 1.

```
(define a '(1 2 3 4))
(set-nth! 3 a 0)
a ⇒ (1 2 0 4)
```

10.1.8 (caar *pair*)

10.1.9 (cadr *pair*)

10.1.10 (cdar *pair*)

10.1.11 (cddr *pair*)

10.1.12 (caaar *pair*)

10.1.13 (caadr *pair*)

10.1.14 (cadar *pair*)

10.1.15 (caddr *pair*)

10.1.16 (cdaar *pair*)

10.1.17 (cdadr *pair*)

10.1.18 (cddar *pair*)

10.1.19 (cdddr *pair*)

10.1.20 (caaaar *pair*)

10.1.21 (caaadr *pair*)

10.1.22 (caadar *pair*)

10.1.23 (caaddr *pair*)

10.1.24 (cadaar *pair*)

10.1.25 (cadadr *pair*)

10.1.26 (caddar *pair*)

10.1.27 (cadddr *pair*)

10.1.28 (cdaaar *pair*)

10.1.29 (cdaadr *pair*)

10.1.30 (cdadar *pair*)

10.1.31 (cdaddr *pair*)

10.1.32 (cddaar *pair*)

10.1.33 (cddadr *pair*)

10.1.34 (cdddar *pair*)

10.1.35 (cddddr *pair*)

```
(define caddr (lambda (x) (car (cdr (cdr x)))))
```

10.1.36 (general-car-cdr *object path*)

This procedure is a generalization of `car` and `cdr`. *path* encodes a particular sequence of `car` and `cdr` operations, which `general-car-cdr` executes on *object*. *path* is a non-negative integer that encodes the operations in a bitwise fashion: a zero bit represents a `cdr` operation, and a one bit represents a `car`. The bits are executed LSB to MSB, and the most significant one bit, rather than being interpreted as an operation, signals the end of the sequence.

For example, the following are equivalent:

```
(general-car-cdr OBJECT #b1011)
(cdr (car (car OBJECT)))
```

Here is a partial table of path/operation equivalents:

```
#b10    cdr
#b11    car
#b100   cddr
#b101   cdar
#b110   cadr
#b111   caar
#b1000  cdddr
```

10.1.37 (copy *object*)

This copies an arbitrary *object*, recursively if it is made from pairs.

10.2 Construction of Lists

10.2.1 (list *object...*)

Returns a list of its arguments.

```
(list 'a (+ 3 4) 'c)                    ⇒ (a 7 c)
(list)                                  ⇒ ()
```

These expressions are equivalent:

```
(list OBJ1 OBJ2 ... OBJN)
(cons OBJ1 (cons OBJ2 ... (cons OBJN '()) ...))
```

10.2.2 (make-list k [element])

This procedure returns a newly allocated list of length *k*, whose elements are all *element*. If *element* is not supplied, it defaults to the empty list.

```
(make-list 4 'c)                        ⇒ (c c c c)
```

10.2.3 (cons* object object ...)

`cons*` is similar to `list`, except that `cons*` conses together the last two arguments rather than consing the last argument with the empty list. If the last argument is not a list the result is an improper list. If the last argument is a list, the result is a list consisting of the initial arguments and all of the items in the final argument. If there is only one argument, the result is the argument.

```
(cons* 'a 'b 'c)                        ⇒ (a b . c)
(cons* 'a 'b '(c d))                    ⇒ (a b c d)
(cons* 'a)                              ⇒ a
```

These expressions are equivalent:

```
(cons* OBJ1 OBJ2 ... OBJN-1 OBJN)
(cons OBJ1 (cons OBJ2 ... (cons OBJN-1 OBJN) ...))
```

10.2.4 (make-initialized-list k init-proc)

Returns a *K*-element list. Element I of the list, where 0 <= I < *k*, is produced by `(init-proc I)`. No guarantee is made about the dynamic order in which *init-proc* is applied to these indices.

```
(make-initialized-list 4 (lambda (x) (* x x))) ⇒ (0 1 4 9)
```

10.2.5 (list-copy list)

Returns a newly allocated copy of *list*. This copies each of the pairs comprising *list*. This could have been defined by

```
(define (list-copy list)
  (if (null? list)
      '()
      (cons (car list)
            (list-copy (cdr list)))))
```

10.2.6 (iota count [start [step]])

Returns a list containing the elements

```
(START START+STEP ... START+(COUNT-1)*STEP)
```

count must be a non-negative integer, while *start* and *step* can be any numbers. The *start* and *step* parameters default to 0 and 1, respectively.

```
(iota 5)          ⇒  (0 1 2 3 4)
(iota 5 0 -0.1)   ⇒  (0 -0.1 -0.2 -0.3 -0.4)
```

10.2.7 (interval *hi*)

10.2.8 (interval *lo hi*)

10.2.9 (interval *lo hi step*)

The first form creates a list of numbers from 1 to `hi`, inclusive. `hi` **must** be a positive integer.

```
(interval 5) ⇒ (1 2 3 4 5)
(interval 2) ⇒ (1 2)
```

The second form creates a list of numbers from `lo` to `hi`, inclusive, stepping by 1. If `lo` > `hi`, a step of -1 is used.

```
(interval 1 5) ⇒ (1 2 3 4 5)
(interval -2 2) ⇒ (-2 -1 0 1 2)
(interval 5 1) ⇒ (5 4 3 2 1)
(interval 2 -2) ⇒ (2 1 0 -1 -2)
```

The third form creates a list of numbers from `lo` to `hi`, inclusive (if possible), `step` apart. `step` **must** be non-zero and it's sign must match the ordering of `lo` and `hi`. I.e. if `lo` > `hi`, `step` must be negative, otherwise positive.

```
(interval 1 5 2) ⇒ (1 3 5)
(interval 1 8 2) ⇒ (1 3 5 7)
(interval -2 2 2) ⇒ (-2 0 2)
(interval 2 -2 -2) ⇒ (2 0 -2)
(interval 5 1 -2) ⇒ (5 3 1)
(interval -1 -8 -2) ⇒ (-1 -3 -5 -7)
```

10.2.10 (vector->list *vector*)

10.2.11 (subvector->list *vector start end*)

`vector->list` returns a newly allocated list of the elements of *vector*. `subvector->list` returns a newly allocated list of the elements of the given subvector. The inverse of `vector->list` is `list->vector`.

```
(vector->list '#(dah dah didah))        ⇒ (dah dah didah)
```

10.2.12 (string->list string)

10.2.13 (substring->list string start end)

`string->list` returns a newly allocated list of the character elements of *string*. `substring->list` returns a newly allocated list of the character elements of the given substring. The inverse of `string->list` is `list->string`.

```
(string->list "abcd")                   ⇒ (#\a #\b #\c #\d)
(substring->list "abcdef" 1 3)          ⇒ (#\b #\c)
```

10.3 Selecting List Components

10.3.1 (list? *object*)

Returns `#t` if *object* is a list, otherwise returns `#f`. By definition, all lists have finite length and are terminated by the empty list. This procedure returns an answer even for circular structures.

Any *object* satisfying this predicate will also satisfy exactly one of `pair?` or `null?`.

```
(list? '(a b c))                          ⇒ #t
(list? '())                               ⇒ #t
(list? '(a . b))                          ⇒ #f
(let ((x (list 'a)))
  (set-cdr! x x)
  (list? x))                              ⇒ #f
```

10.3.2 (circular-list? *object*)

Returns `#t` if *object* is a circular list, otherwise returns `#f`.

```
(circular-list? (list 'a 'b 'c))        ⇒ #f
(circular-list? (cons* 'a 'b 'c))       ⇒ #f
(circular-list? (circular-list 'a 'b 'c)) ⇒ #t
```

10.3.3 (dotted-list? *object*)

Returns `#t` if *object* is an improper list, otherwise returns `#f`.

```
(dotted-list? (list 'a 'b 'c))          ⇒ #f
(dotted-list? (cons* 'a 'b 'c))         ⇒ #t
(dotted-list? (circular-list 'a 'b 'c)) ⇒ #f
```

10.3.4 (length *list*)

Returns the length of *list*. Signals an error if *list* isn't a proper list.

```
(length '(a b c))                       ⇒ 3
(length '(a (b) (c d e)))               ⇒ 3
(length '())                            ⇒ 0
(length (circular-list 'a 'b 'c))       ERROR
```

10.3.5 (length+ *clist*)

Returns the length of *clist*, if it is a proper list. Returns `#f` if *clist* is a circular list. Otherwise signals an error.

```
(length+ (list 'a 'b 'c))               ⇒ 3
(length+ (cons* 'a 'b 'c))              ERROR
(length+ (circular-list 'a 'b 'c))      ⇒ #f
```

10.3.6 (null? *object*)

10.3.7 (nil? *object*)

Returns `#t` if *object* is the empty list; otherwise returns `#f`.

```
(null? '(a . b))                        ⇒ #f
(null? '(a b c))                        ⇒ #f
(null? '())                             ⇒ #t
```

10.3.8 (notnull? *object*)

10.3.9 (notnil? *object*)

Returns `#f` if *object* is the empty list; otherwise returns `#t`.

```
(notnull? '(a . b))                        ⇒ #f
(notnull? '(a b c))                        ⇒ #f
(notnull? '())                             ⇒ #t
```

10.3.10 (list-ref *list k*)

10.3.11 (nth *k list*)

Returns the *k_th element of _list*, using zero-origin indexing. The "valid indexes" of a list are the non-negative integers less than the length of the list. The first element of a list has index `0`, the second has index `1`, and so on. `nth` is provided for Common Lisp familiarity.

```
(list-ref '(a b c d) 2)                    ⇒ c
```

10.3.12 (first *list*)

10.3.13 (second *list*)

10.3.14 (third *list*)

10.3.15 (fourth *list*)

10.3.16 (fifth *list*)

10.3.17 (sixth *list*)

10.3.18 (seventh *list*)

10.3.19 (eighth *list*)

10.3.20 (ninth *list*)

10.3.21 (tenth *list*)

Returns the specified element of *list*. It is an error if *list* is not long enough to contain the specified element (for example, if the argument to `seventh` is a list that contains only six elements).

10.3.22 (last *list*)

Returns the last element in the list. An error is raised if *list* is a circular list.

10.4 Cutting and Pasting Lists

10.4.1 (sublist *list start end*)

start and *end* must be integers satisfying

```
0 <= START <= END <= (length LIST)
```

'sublist' returns a newly allocated list formed from the elements of *list* beginning at index *start* (inclusive) and ending at *end* (exclusive).

10.4.2 (list-head *list k*)

10.4.3 (take *k list*)

Returns a newly allocated list consisting of the first K elements of *list*. *k* must not be greater than the length of *list*.

We could have defined `list-head` this way:

```
(define (list-head list k)
  (sublist list 0 k))
```

10.4.4 (list-tail *list k*)

10.4.5 (drop *k list*)

Returns the sublist of *list* obtained by omitting the first *k* elements. The result, if it is not the empty list, shares structure with *list*. *k* must not be greater than the length of *list*.

10.4.6 (append *list...*)

Returns a list consisting of the elements of the first *list* followed by the elements of the other *list* arguments.

```
(append '(x) '(y))                      ⇒ (x y)
(append '(a) '(b c d))                  ⇒ (a b c d)
(append '(a (b)) '((c)))                ⇒ (a (b) (c))
(append)                                ⇒ ()
```

The resulting list is always newly allocated, except that it shares structure with the last *list* argument. The last argument may actually be any object; an improper list results if the last argument is not a proper list.

```
(append '(a b) '(c . d))                ⇒ (a b c . d)
(append '() 'a)                         ⇒ a
```

10.4.7 (append! *list...*)

Returns a list that is the all the *list* arguments concatenated together. The arguments are changed rather than copied. (Compare this with `append`, which copies arguments rather than destroying them.) For example:

```
(define x '(a b c))
(define y '(d e f))
(define z '(g h))
(append! x y z)                         ⇒ (a b c d e f g h)
x                                       ⇒ (a b c d e f g h)
y                                       ⇒ (d e f g h)
z                                       ⇒ (g h)
```

10.4.8 (last-pair *list*)

Returns the last pair in *list*, which may be an improper list. `last-pair` could have been defined this way:

```
(define last-pair
  (lambda (x)
    (if (pair? (cdr x))
        (last-pair (cdr x))
        x)))
```

10.4.9 (except-last-pair *list*)

10.4.10 (except-last-pair! *list*)

These procedures remove the last pair from *list*. *list* may be an improper list, except that it must consist of at least one pair. `except-last-pair` returns a newly allocated copy of *list* that omits the last pair. `except-last-pair!` destructively removes the last pair from *list* and returns *list*. If the cdr of *list* is not a pair, the empty list is returned by either procedure.

10.5 Filtering Lists

10.5.1 (filter *predicate list*)

Returns a newly allocated copy of *list* containing only the elements satisfying *predicate*. *predicate* must be a procedure of one argument.

```
(filter odd? '(1 2 3 4 5)) ⇒ (1 3 5)
```

10.5.2 (remove *predicate list*)

Like `filter`, except that the returned list contains only those elements **not** satisfying *predicate*.

```
(remove odd? '(1 2 3 4 5)) ⇒ (2 4)
```

10.5.3 (partition *predicate list*)

10.5.4 (partition *size step list*)

The first form partitions the elements of *list* with *predicate*, and returns a list of two elements: the list of in-elements and the list of out-elements. The *list* is not disordered–elements occur in the result lists in the same order as they occur in the argument *list*. The dynamic order in which the various applications of *predicate* are made is not specified. One of the returned lists may share a common tail with the argument *list*.

```
(partition symbol? '(one 2 3 four five 6)) ⇒
    ((one four five) (2 3 6))
```

The second form partitions the elements of *list* into lists of length *size*, returning a list of those lists. Only lists of *size* are returned; any at the end that don't fit are discarded. As with the first form, elements occur in the result lists in the same order as they occur in the argument *list*. If the optional *step* argument is ommitted it defaults to size. Each sublist starts at *step* elements from the start of the previous. If *step* $>$ *size* elements will be skipped between sublists. If *step* $<$ *size* the sublists will overlap.

```
(partition 2 '(one 2 3 four five 6))    ⇒ ((one 2) (3 four) (five 6))

(partition 2 '(one 2 3 four five 6 7))  ⇒ ((one 2) (3 four) (five 6) (7))

(partition 2 1 '(1 2 3 4 5 6 7 8 9 0)) ⇒ ((1 2) (2 3) (3 4) (4 5) (5 6) (6 7) (7 8) (8 9))

(partition 2 3 '(1 2 3 4 5 6 7 8 9 0)) ⇒ ((1 2) (4 5) (7 8))
```

10.5.5 (delq element list)

10.5.6 (delv element list)

10.5.7 (delete element list)

Returns a newly allocated copy of *list* with all entries equal to *element* removed. `delq` uses `eq?` to compare *element* with the entries in *list*, `delv` uses `eqv?`, and `delete` uses `equal?`.

10.6 Searching Lists

10.6.1 (find *predicate list*)

Returns the first element in *list* for which *predicate* is true; returns `#f` if it doesn't find such an element. *predicate* must be a procedure of one argument.

```
(find even? '(3 1 4 1 5 9)) ⇒ 4
```

Note that `find` has an ambiguity in its lookup semantics–if `find` returns `#f`, you cannot tell (in general) if it found a `#f` element that satisfied *predicate*, or if it did not find any element at all. In many situations, this ambiguity cannot arise–either the list being searched is known not to contain any `#f` elements, or the list is guaranteed to have an element satisfying *predicate*. However, in cases where this ambiguity can arise, you should use `find-tail` instead of `find` – `find-tail` has no such ambiguity:

```
(cond ((find-tail pred lis)
       => (lambda (pair) ...)) ; Handle (CAR PAIR)
      (else ...)) ; Search failed.
```

10.6.2 (find-tail *predicate list*)

Returns the first pair of *list* whose car satisfies *predicate*; returns `#f` if there's no such pair. `find-tail` can be viewed as a general-predicate variant of `memv`.

10.6.3 (memq *object list*)

10.6.4 (memv *object list*)

10.6.5 (member *object list*)

These procedures return the first pair of *list* whose car is *object*; the returned pair is always one from which *list* is composed. If *object* does not occur in *list*, `#f` (n.b.: not the empty list) is returned. `memq` uses `eq?` to compare *object* with the elements of *list*, while `memv` uses `eqv?` and `member` uses `equal?`.

```
(memq 'a '(a b c))                      ⇒ (a b c)
(memq 'b '(a b c))                      ⇒ (b c)
(memq 'a '(b c d))                      ⇒ #f
(memq (list 'a) '(b (a) c))             ⇒ #f
(member (list 'a) '(b (a) c))           ⇒ ((a) c)
(memq 101 '(100 101 102))               ⇒ (101 102)
(memv 101 '(100 101 102))               ⇒ (101 102)
```

Although they are often used as predicates, `memq`, `memv`, and `member` do not have question marks in their names because they return useful values rather than just `#t` or `#f`.

10.6.6 (memp *predicate list*)

Returns the first pair of *list* for which *predicate* returns `#t` when passed the car; the returned pair is always one from which *list* is composed. If *predicate* never returns `#t`, `#f` (n.b.: not the empty list) is returned.

10.7 Mapping of Lists

10.7.1 (map *procedure list...*)

procedure must be a procedure taking as many arguments as there are *lists*. If more than one *list* is given, then they must all be the same length. `map` applies *procedure* element-wise to the elements of the *lists* and returns a list of the results, in order from left to right. The dynamic order in which *procedure* is applied to the elements of the *lists* is unspecified; use `for-each` to sequence side effects.

```
(map cadr '((a b) (d e) (g h)))           ⇒ (b e h)
(map (lambda (n) (expt n n)) '(1 2 3 4))  ⇒ (1 4 27 256)
(map + '(1 2 3) '(4 5 6))                 ⇒ (5 7 9)
(let ((count 0))
  (map (lambda (ignored)
         (set! count (+ count 1))
         count)
       '(a b c)))                         ⇒ unspecified
```

10.7.2 (for-each *procedure list ...*)

The arguments to `for-each` are like the arguments to `map`, but `for-each` calls *procedure* for its side effects rather than for its values. Unlike `map`, `for-each` is guaranteed to call *procedure* on the elements of the *lists* in order from the first element to the last, and the value returned by `for-each` is unspecified.

```
(let ((v (make-vector 5)))
  (for-each (lambda (i)
              (vector-set! v i (* i i)))
            '(0 1 2 3 4))
  v)                              ⇒ #(0 1 4 9 16)
```

10.8 Reduction of Lists

10.8.1 (reduce *procedure initial list*)

10.8.2 (reduce-left *procedure initial list*)

Combines all the elements of *list* using the binary operation *procedure*. For example, using `+` one can add up all the elements:

```
(reduce-left + 0 list-of-numbers)
```

The argument *initial* is used only if *list* is empty; in this case *initial* is the result of the call to `reduce-left`. If *list* has a single argument, it is returned. Otherwise, the arguments are reduced in a left-associative fashion. For example:

```
(reduce-left + 0 '(1 2 3 4))            ⇒ 10
(reduce-left + 0 '(1 2))                ⇒ 3
(reduce-left + 0 '(1))                  ⇒ 1
(reduce-left + 0 '())                   ⇒ 0
(reduce-left + 0 '(foo))                ⇒ foo
(reduce-left list '() '(1 2 3 4))       ⇒ (((1 2) 3) 4)
```

10.8.3 (reduce-right *procedure initial list*)

Like `reduce-left` except that it is right-associative.

```
(reduce-right list '() '(1 2 3 4))      ⇒ (1 (2 (3 4)))
```

10.8.4 (fold-right *procedure initial list*)

Combines all of the elements of *list* using the binary operation *procedure*. Unlike `reduce-left` and `reduce-right`, *initial* is always used:

```
(fold-right + 0 '(1 2 3 4))             ⇒ 10
(fold-right + 0 '(foo))                 ERROR Illegal datum
(fold-right list '() '(1 2 3 4))        ⇒ (1 (2 (3 (4 ()))))
```

`fold-right` has interesting properties because it establishes a homomorphism between (`cons`, `()`) and (*procedure*, *initial*). It can be thought of as replacing the pairs in the spine of the list with *procedure* and replacing the `()` at the end with *initial*. Many of the classical list-processing procedures can be expressed in terms of `fold-right`, at least for the simple versions that take a fixed number of arguments:

```
(define (copy-list list)
  (fold-right cons '() list))

(define (append list1 list2)
  (fold-right cons list2 list1))

(define (map p list)
  (fold-right (lambda (x r) (cons (p x) r)) '() list))

(define (reverse items)
  (fold-right (lambda (x r) (append r (list x))) '() items))
```

10.8.5 (fold-left *procedure initial list*)

Combines all the elements of *list* using the binary operation *procedure*. Elements are combined starting with *initial* and then the elements of *list* from left to right. Whereas `fold-right` is recursive in nature, capturing the essence of cdr-ing down a list and then computing a result (although all the reduce/fold functions are implemented iteratively in the runtime), `fold-left` is iterative in nature, combining the elements as the list is traversed.

```
(fold-left list '() '(1 2 3 4))          ⇒ (((() 1) 2) 3) 4)

(define (length list)
  (fold-left (lambda (sum element) (+ sum 1)) 0 list))

(define (reverse items)
  (fold-left (lambda (x y) (cons y x)) () items))
```

10.8.6 (any *predicate list...*)

Applies *predicate* across the *lists*, returning true if *predicate* returns true on any application.

If there are n list arguments *list1* ... *listn*, then *predicate* must be a procedure taking n arguments and returning a boolean result.

`any` applies *predicate* to the first elements of the *list* parameters. If this application returns a true value, `any` immediately returns that value. Otherwise, it iterates, applying *predicate* to the second elements of the *list* parameters, then the third, and so forth. The iteration stops when a true value is produced or one of the lists runs out of values; in the latter case, `any` returns `#f`. The application of *predicate* to the last element of the *lists* is a tail call.

Note the difference between `find` and `any` – `find` returns the element that satisfied the predicate; `any` returns the true value that the *predicate* produced.

Like `every`, `any`'s name does not end with a question mark – this is to indicate that it does not return a simple boolean (`#t` or `#f`), but a general value.

```
(any integer? '(a 3 b 2.7))   ⇒ #t
(any integer? '(a 3.1 b 2.7)) ⇒ #f
(any < '(3 1 4 1 5)
       '(2 7 1 8 2)) ⇒ #t
```

10.8.7 (every *predicate list...*)

Applies *predicate* across the *lists*, returning true if *predicate* returns true on every application.

If there are n list arguments *list1* ... *listn*, then *predicate* must be a procedure taking n arguments and returning a boolean result.

`every` applies *predicate* to the first elements of the *list* parameters. If this application returns false, `every` immediately returns false. Otherwise, it iterates, applying *predicate* to the second elements of the *list* parameters, then the third, and so forth. The iteration stops when a false value is produced or one of the *lists* runs out of values. In the latter case, `every` returns the true value produced by its final application of *predicate*. The application of *predicate* to the last element of the *lists* is a tail call.

If one of the *lists* has no elements, `every` simply returns `#t`.

Like `any`, `every`'s name does not end with a question mark – this is to indicate that it does not return a simple boolean (`#t` or `#f`), but a general value.

10.9 Miscellaneous List Operations

10.9.1 (circular-list *object...*)

This procedure is like `list`, except that the returned list is circular.

10.9.2 (reverse *list*)

Returns a newly allocated list consisting of the top-level elements of *list* in reverse order.

```
(reverse '(a b c))                  ⇒ (c b a)
(reverse '(a (b c) d (e (f))))      ⇒ ((e (f)) d (b c) a)
```

10.9.3 (sort *sequence procedure*)

sequence must be either a list or a vector. *procedure* must be a procedure of two arguments that defines a "total ordering" on the elements of *sequence*. In other words, if X and Y are two distinct elements of *sequence*, then it must be the case that

```
(and (PROCEDURE X Y)
     (PROCEDURE Y X))
     ⇒ #f
```

If *sequence* is a list (vector), `sort` returns a newly allocated list (vector) whose elements are those of *sequence*, except that they are rearranged to be sorted in the order defined by *procedure*. So, for example, if the elements of *sequence* are numbers, and *procedure* is `<`, then the resulting elements are sorted in monotonically nondecreasing order. Likewise, if *procedure* is `>`, the resulting elements are sorted in monotonically nonincreasing order. To be precise, if X and Y are any two adjacent elements in the result, where X precedes Y, it is the case that

```
(PROCEDURE Y X)
     ⇒ #f
```

There is also the function `vector-sort` that applies only to vectors, and will raise an erro if applied to a list.

10.9.4 (flatten *list*)

Returns a list with the contents of all top level nested lists placed directly in the result. This is best illustrated with some examples:

```
(flatten '(a b c d)) ⇒ (a b c d)
(flatten '(a (b c) d)) ⇒ (a b c d)
(flatten '(a (b (c d)))) ⇒ (a b (c d))
```

10.9.5 (flatten* *list*)

Returns a list with the contents of all nested lists placed directly in the result. This is also best illustrated with some examples:

```
(flatten* '(a b c d)) ⇒ (a b c d)
(flatten* '(a (b c) d)) ⇒ (a b c d)
(flatten* '(a (b (c d)))) ⇒ (a b c d)
```

10.9.6 (union *list*...)

Returns a list that contains all items in the argument *list*s. Each item appears only once in the result regardless of whether it was repeated in any *list*.

```
(union '(1 2 3) '(4 5))      ⇒ (1 2 3 4 5)
(union '(1 2 3) '(3 4 5))    ⇒ (1 2 3 4 5)
(union '(1 2 3 2) '(4 4 5))  ⇒ (1 2 3 4 5)
```

10.9.7 (intersection *list*...)

Returns a list that contains only items that are in all *list* arguments.

```
(intersection '(1 2 3) '(3 4 5)) ⇒ (3)
(intersection '() '(3 4 5))      ⇒ ()
```

10.9.8 (complement *list*...)

Returns a list that contains only items that were in the first *list* argument, but not in any of the subsequent argument *list*s.

```
(complement '(1 2 3 4 5) '(1 3 5))      ⇒ (2 4)
(complement '() '(1 2))                 ⇒ ()
(complement '(1 2 3 4 5) '(1 2) '(3 4)) ⇒ (5)
```

11 Vectors

Vectors are heterogeneous structures whose elements are indexed by non-negative integers. A vector typically occupies less space than a list of the same length, and the average time required to access a randomly chosen element is typically less for the vector than for the list.

The length of a vector is the number of elements that it contains. This number is a non-negative integer that is fixed when the vector is created. The valid indexes of a vector are the non-negative integers less than the length of the vector. The first element in a vector is indexed by zero, and the last element is indexed by one less than the length of the vector.

Vectors are written using the notation `#(object ...)`. For example, a vector of length 3 containing the number zero in element 0, the list `(2 2 2 2)` in element 1, and the string `"Anna"` in element 2 can be written as

```
#(0 (2 2 2 2) "Anna")
```

Note that this is the external representation of a vector, not an expression evaluating to a vector. Like list constants, vector constants must be quoted:

`'#(0(2222)"Anna")` ⇒ `#(0 (2222) "Anna")`

A number of the vector procedures operate on subvectors. A subvector is a segment of a vector that is specified by two non-negative integers, start and end. Start is the index of the first element that is included in the subvector, and end is one greater than the index of the last element that is included in the subvector. Thus if start and end are the same, they refer to a null subvector, and if start is zero and end is the length of the vector, they refer to the entire vector. The valid indexes of a subvector are the integers between start inclusive and end exclusive.

11.1 Construction of Vectors

11.1.1 (make-vector *k* [*object*])

Returns a newly allocated vector of k elements. If *object* is specified, `make-vector` initializes each element of the vector to the *object*. Otherwise the initial elements of the result are unspecified.

11.1.2 (vector *object...*)

Returns a newly allocated vector whose elements are the given *objects*. `vector` is analogous to `list`.

```
(vector 'a 'b 'c) ⇒ #(a b c)
```

11.1.3 (vector-copy *vector*)

Returns a newly allocated vector that is a copy of *vector*.

11.1.4 (list->vector *list*)

Returns a newly allocated vector initialized to the elements of *list*.

```
(list->vector '(dididit dah)) ⇒ #(dididit dah)
```

11.1.5 (vector->list *vector*)

Returns a newly allocated list initialized to the elements of *vector*.

```
(vector->list '#(didididitdah)) ⇒ (dididit dah)
```

11.1.6 (make-initialized-vector *k initialization*)

Similar to `make-vector`, except that the elements of the result are determined by calling the procedure *initialization* on the indices. For example:

```
(make-initialized-vector 5 (lambda (x) (* x x))) ⇒ #(0 1 4 9 16)
```

11.1.7 (vector-grow *vector k*)

k must be greater than or equal to the length of *vector*. Returns a newly allocated vector of length k. The first `(vector-length vector)` elements of the result are initialized from the corresponding elements of *vector*. The remaining elements of the result are unspecified.

11.2 Enumerating over Vectors

11.2.1 (vector-map *procedure vector...*)

procedure must be a procedure with arity the same as the number or *vector*s. `vector-map` applies *procedure* element-wise to the corresponding elements of each *vector* and returns a newly allocated vector of the results, in order from left to right. The dynamic order in which procedure is applied to the elements of vector is unspecified.

```
(vector-map cadr '#((ab)(de)(gh)))             ⇒ #(b e h)
(vector-map (lambda (n) (* n n)) '#(1 2 3 4)) ⇒ #(1 4 9 16)
(vector-map + '#(1 2 3) '#(4 5 6))             ⇒ #(5 7 9)
```

11.3 Selecting Vector Components

11.3.1 (vector-length *vector*)

Returns the number of elements in *vector*.

11.3.2 (vector-ref *vector k*)

Returns the contents of element k of *vector*. k must be a valid index of *vector*.

```
(vector-ref '#(1 1 2 3 5 8 13 21) 5)    ⇒  8
```

11.3.3 (vector-set! *vector k object*)

Stores *object* in element *k* of *vector* and returns an unspecified value. *K* must be a valid index of *vector*.

```
(let ((vec (vector 0 '(2 2 2 2) "Anna")))
  (vector-set! vec 1 '("Sue" "Sue"))
  vec)
           ⇒  #(0 ("Sue" "Sue") "Anna")
```

11.3.4 (vector-first *vector*)

11.3.5 (vector-second *vector*)

11.3.6 (vector-third *vector*)

11.3.7 (vector-fourth *vector*)

11.3.8 (vector-fifth *vector*)

11.3.9 (vector-sixth *vector*)

11.3.10 (vector-seventh *vector*)

11.3.11 (vector-eighth *vector*)

11.3.12 (vector-ninth *vector*)

11.3.13 (vector-tenth *vector*)

These procedures access the first several elements of *vector* in the obvious way. It is an error if the implicit index of one of these procedures is not a valid index of *vector*.

11.3.14 (vector-last *vector*)

Returns the last element of *vector*.

11.3.15 (vector-binary-search *vector key<? unwrap-key key*)

Searches *vector* for an element with a key matching *key*, returning the element if one is found or *#f* if none. The search operation takes time proportional to the logarithm of the length of *vector*. *unwrap-key* must be a procedure that maps each element of *vector* to a key. *key<?* must be a procedure that implements a total ordering on the keys of the elements.

```
(define (translate number)
  (vector-binary-search '#((1 . i)
                           (2 . ii)
                           (3 . iii)
                           (6 . vi))
                        < car number))
(translate 2)  ⇒  (2 . ii)
(translate 4)  ⇒  #f
```

11.4 Cutting Vectors

11.4.1 (subvector *vector start end*)

Returns a newly allocated vector that contains the elements of *vector* between index *start* (inclusive) and *end* (exclusive).

11.4.2 (vector-head *vector end*)

Equivalent to

```
(subvector vector 0 end)
```

11.4.3 (vector-tail *vector start*)

Equivalent to

```
(subvector vector start (vector-length vector))
```

11.5 Modifying Vectors

11.5.1 (vector-fill! *vector object*)

11.5.2 (subvector-fill! *vector start end object*)

Stores *object* in every element of the vector (subvector) and returns an unspecified value.

11.5.3 (subvector-move-left! *vector1 start1 end1 vector2 start2*)

11.5.4 (subvector-move-right! *vector1 start1 end1 vector2 start2*)

Destructively copies the elements of *vector1*, starting with index *start1* (inclusive) and ending with *end1* (exclusive), into *vector2* starting at index *start2* (inclusive). *vector1*, *start1*, and *end1* must specify a valid subvector, and *start2* must be a valid index for *vector2*. The length of the source subvector must not exceed the length of *vector2* minus the index *start2*.

The elements are copied as follows (note that this is only important when *vector1* and *vector2* are `eqv?`):

`subvector-move-left!`: The copy starts at the left end and moves toward the right (from smaller indices to larger). Thus if *vector1* and *vector2* are the same, this procedure moves the elements toward the left inside the vector.

`subvector-move-right!`: The copy starts at the right end and moves toward the left (from larger indices to smaller). Thus if *vector1* and *vector2* are the same, this procedure moves the elements toward the right inside the vector.

11.5.5 (vector-sort! *vector procedure*)

procedure must be a procedure of two arguments that defines a *total ordering* on the elements of *vector*. The elements of *vector* are rearranged so that they are sorted in the order defined by *procedure*. The elements are rearranged in place, that is, VECTOR is destructively modified so that its elements are in the new order.

`sort!` returns *vector* as its value.

See also the definition of `sort`.

12 Associations

12.1 Association Lists

"Association lists" are one of Lisp's oldest association mechanisms. Because they are made from ordinary pairs, they are easy to build and manipulate, and very flexible in use. However, the average lookup time for an association list is linear in the number of associations. Frames are a more efficient

An "association list", or "alist", is a data structure used very frequently in Scheme. An alist is a list of pairs, each of which is called an "association". The car of an association is called the "key", and the cdr is called the "value". Having lists as pair values can cause confusion becase the pair in the alist look like proper lists and not dotted pairs. Functions that look specifically for dotted pairs will not consider it an association list (e.g. `alist?`) while those that don't will work fine (e.g. the `assoc` & `dissoc` functions). The latter simply look at the car and cdr of the pairs. not whether they are canonical dotted pairs (i.e. their cdr is not a pair).

```
'((a . (1 2)) (b . (3 4)))                    ⇒ ((a 1 2) (b 3 4))
(alist? '((a . (1 2)) (b . (3 4))))           ⇒ #f
(assoc 'b '((a . (1 2)) (b . (3 4))))         ⇒ (b 3 4)
(cdr (assoc 'b '((a . (1 2)) (b . (3 4)))))   ⇒ (3 4)
```

An advantage of the alist representation is that an alist can be incrementally augmented simply by adding new entries to the front. Moreover, because the searching procedures 'assv' et al. search the alist in order, new entries can "shadow" old entries. If an alist is viewed as a mapping from keys to data, then the mapping can be not only augmented but also altered in a non-destructive manner by adding new entries to the front of the alist.

12.1.1 (alist? *object*)

Returns `#t` if *object* is an association list (including the empty list); otherwise returns `#f`. Any *object* satisfying this predicate also satisfies `list?`.

12.1.2 (acons *key value* [*alist*])

Returns the result of consing a pair `(key . value)` to *alist*. If *alist* is omitted, it defaults to the empty list.

```
(acons 'a 1)            ⇒ ((a . 1))
(acons 'a 1 '((b . 2))) ⇒ ((a . 1) (b . 2))
(acons 'b 1 '((b . 2))) ⇒ ((b . 1) (b . 2))
```

12.1.3 (pairlis *keys values* [*alist*])

Creates an association list from lists of *keys* and *values* by acons-ing onto *alist*. If *alist* is omitted, it defaults to the empty list. Note that the key and value lists are paired up in left to right order, but the order they are consed onto *alist* is unspecified.

```
(pairlis '(a b) '(1 2))                    ⇒ ((b . 2) (a . 1)))
(pairlis '(a b) '(1 2) '((c . 3) (d . 4))) ⇒ ((b . 2) (a . 1) (c . 3) (d . 4))))
```

12.1.4 (assq *object alist*)

12.1.5 (assv *object alist*)

12.1.6 (assoc *object alist*)

These procedures find the first pair in *alist* whose car field is *object*, and return that pair; the returned pair is always an **element** of *alist*, **not** one of the pairs from which *alist* is composed. If no pair in *alist* has *object* as its car, `#f` (n.b.: not the empty list) is returned. `assq` uses `eq?` to compare *object* with the car fields of the pairs in *alist*, while `assv` uses `eqv?` and `assoc` uses `equal?`.

```
(define e '((a . 1) (b . 2) (c . 3)))
(assq 'a e)                              ⇒  (a . 1)
(assq 'b e)                              ⇒  (b . 2)
(assq 'd e)                              ⇒  #f
(assq (list 'a) '(((a)) ((b)) ((c))))   ⇒  #f
(assoc (list 'a) '(((a)) ((b)) ((c))))  ⇒  ((a))
(assv 5 '((2 . 3) (5 . 7) (11 . 13)))   ⇒  (5 . 7)
```

12.1.7 (rassoc *value alist*)

Return the pair from *alist* whose cdr is equal to *value*. `#f` is returned is *value* isn't found.

```
(rassoc 1 '((a . 1) (b . 2) (c . 3))) ⇒ (a . 1)
(rassoc 3 '((a . 1) (b . 2)))         ⇒ #f
```

12.1.8 (del-assq *object alist*)

12.1.9 (dissq *object alist*)

12.1.10 (del-assv *object alist*)

12.1.11 (dissv *object alist*)

12.1.12 (del-assoc *object alist*)

12.1.13 (dissoc *object alist*)

These procedures return a newly allocated copy of *alist* in which all associations with keys equal to *object* have been removed. Note that while the returned copy is a newly allocated list, the association pairs that are the elements of the list are shared with *alist*, not copied. `del-assq`/`dissq` use `eq?` to compare *object* with the keys, while `del-assv`/`dissv` use `eqv?` and `del-assoc`/`dissoc` use `equal?`.

```
(define a
  '((butcher . "231 e22nd St.")
    (baker . "515 w23rd St.")
    (hardware . "988 Lexington Ave.")))

(del-assq 'baker a)
     ⇒
     ((butcher . "231 e22nd St.")
      (hardware . "988 Lexington Ave."))
```

13 Frames

GoLisp contains a frame system inspired by Self [4] and NewtonScript [5].

A frame is a set of named slots that hold arbitrary values. Slot names must be symbols that end with a colon. For example: `color:`, or `height:`. When evaluated normally, these special symbols don't get looked up in the environment, they simply evaluate to themselves.

```
(define a a:)

'a ⇒ a
a  ⇒ a:

'a: ⇒ a:
a:  ⇒ a:
```

13.0.1 (make-slotname *symbol*)

This function takes a symbol or string and returns an interned slotname based on it, doing what is required.

```
(make-slotname 'name) ⇒ name:
(make-slotname "name") ⇒ name:
(make-slotname name:) ⇒ name:
```

13.1 Basic functions

13.1.1 (make-frame *slot-name slot-value* ...)

Frames can be created using the `make-frame` function, passing it an alternating sequence of slot names and values:

```
(make-frame a: 1 b: 2)
```

This results in a frame with two slots, named `a:` and `b:` with values `1` and `2`, respectively.

13.1.2 { *slot-name slot-value ...* }

This is an alternative syntax for defining frame literals:

```
{a: 1 b: 2}
```

Both are equivalent. Slot names and values in both cases are evaluated (this is one reason for the non-evaluating symbols: it avoiding having to quote literal slot names).

13.1.3 (clone *frame*)

Frames represent things. For example, you could use a frame that looks like `{x: 1 y: 10}` to represent a point. A system that would use point frames will typically need many independant points. The approach to this is to create a prototypical point data frame, and use the `clone` function to create individual, independant frames:

```
(define point {x: 1 y: 1})
(define p1 (clone point))
(set-slot! p1 x: 5)
(get-slot p1 x:)    ⇒ 5
(get-slot point x:) ⇒ 1
```

13.1.4 (has-slot? *frame slot-name*)

13.1.5 (*slot-name*? *frame*)

The `has-slot?` function is used to query whether a frame contains (directly or in an ancestor) the particular slot:

```
(define f {a: 1 b: 2})
(has-slot? f a:)      ⇒ #t
(a:? f)               ⇒ #t
(has-slot? f c:)      ⇒ #f
(c:? f)               ⇒ #f
```

13.1.6 (get-slot *frame slot-name*)

13.1.7 (*slot-name frame*)

The `get-slot` function is used to retrieve values from frame slots:

```
(define f {a: 1 b: 2})
(get-slot f a:)       ⇒ 1
(a: f)                ⇒ 1
(get-slot f b:)       ⇒ 2
(b: f)                ⇒ 2
```

If the frame passed to `get-slot` contains a slot with the specified name, it's value is returned. If not, then parent frames are searched in a nondeterministic order until a slot with the specified name is found. If a matching slot is found, it's value is returned. If none is found an error is raised.

```
(define f {a: 1 b: 2})
(define g {parent*: f c: 3})

(get-slot g c:) ⇒ 3
(get-slot g a:) ⇒ 1  ; from the frame f
```

13.1.8 (get-slot-or-nil *frame slot-name*)

The same as above, except that if a matching slot is not found, `nil` is returned instead of raising an error.

13.1.9 (set-slot! *frame slot-name new-value*)

13.1.10 (*slot-name*! *frame new-value*)

The `set-slot!` function is used to change values in frame slots:

```
(define f {a: 1 b: 2})
(get-slot f a:)    ⇒ 1
(set-slot! f a: 5) ⇒ 5
(a:! f 5) ⇒ 5
(get-slot f a:)    ⇒ 5
```

Trying to set a slot that doesn't exist in the frame will result in a corresponding slot being created.

```
(define f {a: 1 b: 2})
(set-slot! f c: 5) ⇒ 5
f                  ⇒ {a: 1 b: 2 c: 5}
```

13.1.11 (remove-slot! *frame slot-name*)

The `remove-slot!` function is used to function is used to remove a slot from a frame. It only removes slots from the frame itself. not any of it's parents. `remove-slot!` return `#t` if the slot was removed, `#f` otherwise.

```
(define f {a: 1 b: 2})
(remove-slot! f a:) ⇒ #t
f                   ⇒ {b: 2}
(remove-slot! f a:) ⇒ #f
```

13.1.12 (frame-keys *frame*)

Returns a list of the slot names in *frame*. Note that the order of the result is nondeterministic.

```
(frame-keys {a: 1 b: 2}) ⇒ (a: b:)
```

13.1.13 (frame-values *frame*)

Returns a list of the slot values in *frame*. Note that the order of the result is nondeterministic.

```
(frame-values {a: 1 b: 2}) ⇒ (1 2)
```

13.2 Parent slots

Frames can have slots that refer to other slots to provide prototype inheritance. These slots have names that have a `*` immediately preceeding the trailing `:`, for example `proto*:`. The names of the parent slots don't matter; it is the trailing `*:` in the name that marks them as parent slots. A frame can have any number of parent slots.

When a slot is being searched for, if it isn't found in the specified slot, these *parent* slots are recursively searched until the requested slot is found or the entire graph has been examined.

```
> (define y {a: 1})
=⇒ {a: 1}
> (define x {b: 2 p*: y})
=⇒ {b: 2 p*: {...}}

> (a: x)
=⇒ 1
```

Note: Parent slots are searched in arbitrary order.

13.3 Function slots

Now things get interesting. Slot values can be functions (typically `lambda` expressions) as well as data. Function slots can be executed by using the `send` function

13.3.1 (send *frame slot-name arg...*)

13.3.2 (*slot-name*> *frame arg...*)

```
(define f {
  add: (lambda () (+ 1 2))
})
(send f add:) ⇒ 3
(add:> f) ⇒ 3
```

As expected, parameters are supported:

```
(define f {
  add: (lambda (x) (+ 1 x))
})
(send f add: 2) ⇒ 3
```

In the body of a function, slots can be refrenced like normal variables. To do so, simply omit the trailing colon:

```
(define f {
  a: 3
  add: (lambda (x) (+ a x))
})
(send f add: 2) ⇒ 5
```

Likewise, functions in the frame (or parent frames) can be referred to directly by name.

```
(define f {
  a: 5
  b: 2
  foo: (lambda (x) (+ x a))
  bar: (lambda () (foo b))
})
(send f bar:) ⇒ 7
```

Bindings defined in the local environment (e.g. by a `let` form) hide frame slots of the same name. In the following, `let` overrides the `a:` slot by introducing a local binding for `a`.

```
(let ((f {a: 42})
      (g {
         parent*: f
         foo:
           (lambda ()
             (let ((a 10))
             (+ 1 a)))
      }))
(send g foo:) ⇒ 11
```

Of course the end game of all this is to be able to inherit functions from parent frames:

```
(define f {
  a: 5
  foo: (lambda (x) (+ x a))
})
(define g {
  parent*: f
  b: 2
  bar: (lambda () (foo b))
})
(send g bar:) ⇒ 7
```

Notice that we've been saying parent **frames**, i.e. plural. Also note that parent slot names are arbitrary and for documentation purposes only. A frame can have any number of parents. When a slot is looked for, the explictily specified frame is searched first, recursively followed by parent frames in a nondeterministic order until a matching slot is found. If none are found, the result is nil.

```
(define e {a: 5})
(define f {b: 2})
(define g {
  parent-e*: e
  parent-f*: f
  foo: (lambda (x) (+ x a))
  bar: (lambda () (foo b))}))
(send g bar:)       ⇒ 7
(set-slot! g a: 10)
(get-slot g a:)     ⇒ 10
(get-slot e a:)     ⇒ 5
```

When you set a slot with parent frames involved, if the slot is found in the explicit frame it's value is set and the new value is returned. If it doesn't exist in the explicit frame, it gets created there. This new slot now hides any slots in a parent with the same name.

13.3.3 (send-super *slot-name arg...*)

13.3.4 (*slot-name^ arg...*)

Like `send`, but sends to the first parent that has the named slot. `send-super` **can only be used from within a function slot.**

13.3.5 (apply-slot *frame slot-name sexpr...*)

Apply the function that results from evaluating the function in slot *slot-name* of *frame* to the argument list resulting from evaluating each *sexpr*.

Each initial *sexpr* can evaluate to any type of object, but the final one (and there must be at least one *sexpr*) must evaluate to a list.

```
(define f {
  foo:
    (lambda (x y z)
      (+ 1 x y z))
})
(apply-slot f foo: 2 '(3 4)) ⇒ 10
(apply-slot f foo: '(2 3 4)) ⇒ 10
```

13.3.6 (apply-slot-super *slot-name sexpr...*)

Like `apply-slot`, but sends to the first parent that has the named slot. `apply-slot-super` **can only be used from within a function slot.**

13.4 Dynamic inheritence

Parent slots are slots like any other and can have their values changed at any time. This ability is somewhat unusual for those with a heavy OO background but can be very useful for changing behavior on the fly. A prime example of this is the implimentation of a state machine. The functions for each state can be placed in different frames and transitions can modify the slot contaiing that state behavior.

Here's an example of this.

```
(define state {
  name: ""
  enter: (lambda ())
  halt: (lambda ())
  set-speed: (lambda (s))
  halt: (lambda ())
  transition-to:
```

```
    (lambda (s)
      (set! state* s)
      (enter))
})

(define stop-state {
  name: "stop"
  parent*: state
  enter:
    (lambda ()
      (set! speed 0)
      (transition-to idle-state))
})

(define idle-state {
  name: "idle"
  parent*: state
  set-speed:
    (lambda (s)
      (set! speed s)
      (transition-to start-state))
})

(define start-state {
  name: "start"
  parent*: state
  halt:
    (lambda ()
      (transition-tostop-state))
  set-speed:
    (lambda (s)
      (set! speed s)
      (transition-to change-speed-state))
})

(define change-speed-state {
  name: "change-speed"
  parent*: state
  halt:
    (lambda ()
      (transition-to stop-state))
  set-speed:
    (lambda (s)
      (set! speed s))
})
```

```
(define motor {
  speed: 0
  state*: state
  start:
    (lambda ()
      (transition-to stop-state))
})
```

Now you can do things like the following:

```
(send motor start:)
motor ⇒ {speed: 0 state*: {name: "idle" ...}}
(send motor set-speed: 10)
motor ⇒ {speed: 10 state*: {name: "start" ...}}
(send motor set-speed: 20)
motor ⇒ {speed: 20 state*: {name: "change-speed" ...}}
(send motor set-speed: 15)
motor ⇒ {speed: 15 state*: {name: "change-speed" ...}}
(send motor halt:)
motor ⇒ {speed: 0 state*: {name: "idle" ...}}
```

13.5 Json support

GoLisp has built-in support for converting between stringified Json and frames, according to the following rules:

- numbers and strings map directly in both directions
- frames recursively map to objects, and the reverse
- lists recursively map to arrays, and the reverse
- frame slots names map to string field names, and the reverse
- function slots **do not** get mapped to json
- parent slots **DO not** get mapped to json

13.5.1 (json->lisp *string*)

```
(json->lisp "{'key': [1, 2, 3]}") ⇒ {key: (1 2 3)}
```

13.5.2 (lisp->json *frame*)

```
(lisp->json {key: (1 2 3)}) ⇒ "{'key': [1, 2, 3]}"
```

14 Miscellaneous Datatypes

14.1 Booleans

The "boolean objects" are "true" and "false". The boolean constant true is written as `#t`, and the boolean constant false is written as `#f`.

The primary use for boolean objects is in the conditional expressions `if`, `cond`, `and`, and `or`; the behavior of these expressions is determined by whether objects are true or false. These expressions count only `#f` as false. They count everything else, including `#t`, pairs, symbols, numbers, strings, vectors, and procedures as true.

Boolean constants evaluate to themselves, so you don't need to quote them.

```
#t                                    ⇒  #t
#f                                    ⇒  #f
'#f                                   ⇒  #f
t                                     ERROR Unbound variable
```

14.1.1 false

14.1.2 true

These variables are bound to the objects `#f` and `#t` respectively.

Note that the symbol `true` is not equivalent to `#t`, and the symbol `false` is not equivalent to `#f`.

14.1.3 (boolean? *object*)

Returns `#t` if *object* is either `#t` or `#f`; otherwise returns `#f`.

```
(boolean? #f)                           ⇒  #t
(boolean? 0)                            ⇒  #f
```

14.1.4 (not *object*)

14.1.5 (false? *object*)

These procedures return `#t` if *object* is false; otherwise they return `#f`. In other words they *invert* boolean values. These two procedures have identical semantics; their names are different to give different connotations to the test.

```
(not #t)                                ⇒  #f
(not 3)                                 ⇒  #f
(not (list 3))                          ⇒  #f
(not #f)                                ⇒  #t
```

14.1.6 (boolean=? *obj1 obj2*)

This predicate is true iff *obj1* and *obj2* are either both true or both false.

14.1.7 (boolean/and *object...*)

This procedure returns `#t` if none of its arguments are `#f`. Otherwise it returns `#f`.

14.1.8 (boolean/or *object...*)

This procedure returns `#f` if all of its arguments are `#f`. Otherwise it returns `#t`.

14.2 Symbols

Unlike MIT/GNU Scheme, GoLisp only provides one type of symbol: "interned". Interned symbols are far more common than uninterned symbols, and there are more ways to create them. We decided that uninterned symbols were not necessary for our uses. Throughtout this document "symbol" means "interned symbol"

Symbols have an extremely useful property: any two symbols whose names are the same, in the sense of `string=?`, are the same object (i.e. they are `eq?` to one another). The term "interned" refers to the process of "interning" by which this is accomplished.

The rules for writing an symbol are the same as the rules for writing an identifier. Any symbol that has been returned as part of a literal expression, or read using the `read` procedure and subsequently written out using the `write` procedure, will read back in as the identical symbol (in the sense of `eq?`).

Usually it is also true that reading in an symbol that was previously written out produces the same symbol. An exception are symbols created by the procedures `string->symbol` and `intern`; they can create symbols for which this write/read invariance may not hold because the symbols' names contain special characters.

14.2.1 (symbol? *object*)

Returns `#t` if *object* is a symbol, otherwise returns `#f`.

```
(symbol? 'foo)                                  ⇒  #t
(symbol? (car '(a b)))                          ⇒  #t
(symbol? "bar")                                 ⇒  #f
```

14.2.2 (symbol->string *symbol*)

Returns the name of *symbol* as a string. If *symbol* was returned by `string->symbol`, the value of this procedure will be identical (in the sense of `string=?`) to the string that was passed to `string->symbol`. Unlike MIT/GNU Scheme, the result of `symbol->string` is not converted to lower case.

```
(symbol->string 'flying-fish)           ⇒  "flying-fish"
(symbol->string 'Martin)                ⇒  "Martin"
(symbol->string (string->symbol "Malvina"))
                                        ⇒  "Malvina"
```

14.2.3 (intern *string*)

Returns the symbol whose name is *string*. This is the preferred way to create symbols, as it guarantees the following independent of which case the implementation uses for symbols' names:

```
(eq? 'bitBlt (intern "bitBlt")) ⇒     #t
```

The user should take care that *string* obeys the rules for identifiers, otherwise the resulting symbol cannot be read as itself.

14.2.4 (string->symbol *string*)

Returns the interned symbol whose name is *string*. Although you can use this procedure to create symbols with names containing special characters, it's usually a bad idea to create such symbols because they cannot be read as themselves. See `symbol->string`.

```
(eq? 'mISSISSIppi 'mississippi)         ⇒  #t
(string->symbol "mISSISSIppi")
     ⇒  the symbol with the name "mISSISSIppi"
(eq? 'bitBlt (string->symbol "bitBlt")) ⇒  #t
(eq? 'JollyWog
      (string->symbol
        (symbol->string 'JollyWog)))    ⇒  #t
(string=? "K. Harper, M.D."
           (symbol->string
             (string->symbol
               "K. Harper, M.D.")))     ⇒  #t
```

14.2.5 (gensym [*prefix*])

Create a new, unique symbol made from the *prefix* (or `GENSYM` if a prefix is omitted) and an increasing integer. This is useful when you are generating code and need a unique name (in a macro, for example).

```
(gensym)       ⇒ GENSYM1
(gensym)       ⇒ GENSYM2
(gensym)       ⇒ GENSYM3

(gensym "hi") ⇒ hi1
(gensym "ho") ⇒ ho1
(gensym "hi") ⇒ hi2
(gensym "ho") ⇒ ho2
(gensym "ho") ⇒ ho3
(gensym "hi") ⇒ hi3

(gensym)       ⇒ GENSYM4
```

14.2.6 (symbol<? *symbol1 symbol2*)

This procedure computes a total order on symbols. It is equivalent to

```
(string<? (symbol->string symbol1)
          (symbol->string symbol2))
```

14.3 Bytearrays

Bytearrays are an extension that GoLisp makes to Scheme arising from the need to implement byte level communication protocols for SteelSeries Engine 3.

14.3.1 (list->bytearray *list of bytes and/or bytearrays*)

The list must be comprised of elements that are either numbers between 0 and 255, inclusive, or existing bytearray objects. The result is an *object* containing a `[]byte`.

```
(list->bytearray '(1 2 3 4))     ⇒ [1 2 3 4]
(list->bytearray '(1 [2 3] 4))   ⇒ [1 2 3 4]
(list->bytearray '([1 2] [3 4])) ⇒ [1 2 3 4]
```

14.3.2 (bytearray->list *bytearray*)

This is the opposite of the previous function. The result is a list containing the numbers in the bytearray.

```
(bytearray->list [1 2 3 4]) ⇒ (1 2 3 4)
```

14.3.3 (replace-byte *bytearray index value*)

Makes a copy of *bytearray* and replaces the byte at *index* with *value*. The new bytearray with the replaced byte is returned. *index* must be a valid index into the byte array (zero based), and *value* must be a valid byte value, i.e. between 0 and 255, inclusive.

```
(define a [1 2 3 4])     ⇒ [1 2 3 4]
(replace-byte a 2 100)   ⇒ [1 2 100 4]
a                        ⇒ [1 2 3 4]
```

14.3.4 (replace-byte! *bytearray index value*)

Replaces the byte at *index* with *value*. *index* must be a valid index into the byte array (zero based), and *value* must be a valid byte value, i.e. between 0 and 255, inclusive. The original byte array is modified and the returned bytearray object is the one that is passed to the function.

```
(define a [1 2 3 4])     ⇒ [1 2 3 4]
(replace-byte! a 2 100)  ⇒ [1 2 100 4]
a                        ⇒ [1 2 100 4]
```

14.3.5 (extract-byte *bytearray index*)

Fetch and return the byte at *index*. *index* must be a valid index into the byte array (zero based).

```
(extract-byte [1 2 3 4] 2) ⇒ 3
```

14.3.6 (append-bytes *bytearray byte...*)

14.3.7 (append-bytes *bytearray list of bytes*)

14.3.8 (append-bytes *bytearray bytearray...*)

Appends the rest of the arguments to a copy of the bytearray that is the first arg. The copy is returned. Things that can be appended are: a single byte, a sequence of bytes (as a sequence of separate arguments), a list of bytes, a bytearray object, a sequence of bytearray objects (as a sequence of separate arguments), and code that evaluates to a byte, list of bytes, or bytearray.

```
(append-bytes [1 2 3] 4)            ⇒ [1 2 3 4]
(append-bytes [1 2 3] 4 5 6)        ⇒ [1 2 3 4 5 6]
(append-bytes [1 2 3] '(4 5 6))     ⇒ [1 2 3 4 5 6]
(append-bytes [1 2 3] [4 5 6])        ⇒ [1 2 3 4 5 6]
(append-bytes [1 2 3] [4 5] [6])      ⇒ [1 2 3 4 5 6]
(append-bytes [1 2 3] (list 4 5 6))  ⇒ [1 2 3 4 5 6]
```

14.3.9 (append-bytes! *bytearray byte...*)

14.3.10 (append-bytes! *bytearray list of bytes*)

14.3.11 (append-bytes! *bytearray bytearray...*)

As with `append-bytes`, but modifies and returns *bytearray* rather than making a copy.

```
(define a [1 2 3])  ⇒ [1 2 3]
(append-bytes a 4)  ⇒ [1 2 3 4]
a                   ⇒ [1 2 3]
(append-bytes! a 4) ⇒ [1 2 3 4]
a                   ⇒ [1 2 3 4]
```

14.3.12 (take *k bytearray*)

As with the list implementation of `take`, fetches and returns a new bytearray consisting of the bytes from *bytearray* starting at index *k*.

```
(take 1 [1 2 3] ⇒ [1])
(take 3 [1 2 3] ⇒ [1 2 3])
```

14.3.13 (drop *k bytearray*)

```
(drop 1 [1 2 3] ⇒ [2 3])
(drop 2 [1 2 3] ⇒ [3])
```

As with the list implementation of `drop`, fetches and returns a new bytearray consisting of the bytes from *bytearray* prior to index *k*.

14.3.14 (extract-bytes *bytearray index length*)

Returns a new bytearray consisting of *length* bytes from *bytearray*, starting at index *index*. This is functionally equivalent to `(take length (drop index bytearray))` with bounds checking added.

```
(extract-bytes [1 2 3 4 5] 0 1) ⇒ [1]
(extract-bytes [1 2 3 4 5] 0 3) ⇒ [1 2 3]
(extract-bytes [1 2 3 4 5] 2 1) ⇒ [3]
(extract-bytes [1 2 3 4 5] 2 3) ⇒ [3 4 5]
```

15 Environments

Scheme (and thus GoLisp) is lexically scoped. This is implemented by the creation of a lexical environment (aka symbol table) for each lexical scope:

- function/lambda/macro invocations, which holds parameters and any local definitions
- `let` structures, which hold the `let` bindings
- `do` structures, which hold the `do` bindings

Functions and lambdas capture a reference to the environment in which they were defined, so they always have access to it's bindings (that's a closure, btw).

Each environment has a connection to it's containing environment, and can override/hide bindings in outer scopes. When a symbol is evaluated, the most local environment is searched first. If a binding for the system isn't found there, the containing environment is searched. This continues until a binding for the sybol is found or we go all the way to the global environment and still can't find a binding.

Section 3.2 of [1] does a great job of explaining environments in Scheme, which is the basis for environments in GoLisp.

In Scheme, some environments are more important than others, mainly as they tend to be larger, long lived, and serve as the root of many other environments as a program runs. These are known as *top level environments*. Specifially, these are the global environment (the only environment that is contained by nothing), and any environments directly below it in the environment tree. The REPL runs in one such environment, which effectively sandboxes it, protecting the bindings in the global environment from corruption.

15.0.1 (environment? *object*)

Returns `#t` if *object* is an environment; otherwise returns `#f`.

15.0.2 (environment-has-parent? *environment*)

Returns `#t` if *environment* has a parent environment; otherwise returns `#f`.

15.0.3 (environment-parent *environment*)

Returns the parent environment of *environment*. It is an error if *environment* has no parent.

15.0.4 (environment-bound-names *environment*)

Returns a newly allocated list of the names (symbols) that are bound by *environment*. This does not include the names that are bound by the parent environment of *environment*. It does include names that are unassigned or keywords in *environment*.

15.0.5 (environment-macro-names *environment*)

Returns a newly allocated list of the names (symbols) that are bound to syntactic keywords in *environment*.

15.0.6 (environment-bindings *environment*)

Returns a newly allocated list of the bindings of *environment*; does not include the bindings of the parent environment. Each element of this list takes one of two forms: `(symbol)` indicates that *symbol* is bound but unassigned, while `(symbol object)` indicates that *symbol* is bound, and its value is `object`.

15.0.7 (environment-reference-type *environment symbol*)

Returns a symbol describing the reference type of *symbol* in *environment* or one of its ancestor environments. The result is one of the following:

- `normal` means *symbol* is a variable binding with a normal value.
- `unassigned` means *symbol* is a variable binding with no value.
- `macro` means *symbol* is a keyword binding.
- `unbound` means *symbol* has no associated binding.

15.0.8 (environment-bound? *environment symbol*)

Returns `#t` if *symbol* is bound in *environment* or one of its ancestor environments; otherwise returns `#f`. This is equivalent to

```
(not (eq? 'unbound
          (environment-reference-type environment symbol)))
```

15.0.9 (environment-assigned? *environment symbol*)

Returns `#t` if *symbol* is bound in *environment* or one of its ancestor environments, and has a normal value. Returns `#f` if it is bound but unassigned. Signals an error if it is unbound or is bound to a keyword.

15.0.10 (environment-lookup *environment symbol*)

symbol must be bound to a normal value in *environment* or one of its ancestor environments. Returns the value to which it is bound. Signals an error if unbound, unassigned, or a keyword.

15.0.11 (environment-lookup-macro *environment symbol*)

If *symbol* is a keyword binding in *environment* or one of its ancestor environments, returns the value of the binding. Otherwise, returns `#f`. Does not signal any errors other than argument-type errors.

15.0.12 (environment-assignable? *environment symbol*)

symbol must be bound in *environment* or one of its ancestor environments. Returns `#t` if the binding may be modified by side effect.

15.0.13 (environment-assign! *environment symbol value*)

symbol must be bound in *environment* or one of its ancestor environments, and must be assignable. Modifies the binding to have *value* as its value, and returns an unspecified result.

15.0.14 (environment-definable? *environment symbol*)

Returns `#t` if *symbol* is definable in *environment*, and `#f` otherwise.

15.0.15 (environment-define *environment symbol value*)

Defines *symbol* to be bound to object in *environment*, and returns an unspecified value. Signals an error if *symbol* isn't definable in *environment*.

15.0.16 (eval *sexpr environment*)

Evaluates *sexpr* in *environment*. You rarely need eval in ordinary programs; it is useful mostly for evaluating expressions that have been created "on the fly" by a program.

15.0.17 (system-global-environment)

The function `system-global-environment` is returns the distinguished environment that's the highest level ancestor of all other environments. It is the parent environment of all other top-level environments. Primitives, system procedures, and most syntactic keywords are bound in this environment.

15.0.18 (the-environment)

Returns the current environment. This form may only be evaluated in a top-level environment. An error is signalled if it appears elsewhere.

15.0.19 (procedure-environment *procedure*)

Returns the closing environment of *procedure*. Signals an error if *procedure* is a primitive procedure.

15.0.20 (make-top-level-environment [*names* [*values*]])

Returns a newly allocated top-level environment. extend-top-level-environment creates an environment that has parent environment, make-top-level-environment creates an environment that has parent system-global-environment, and make-root-top-level-environment creates an environment that has no parent.

The optional arguments *names* and *values* are used to specify initial bindings in the new environment. If specified, *names* must be a list of symbols, and *values* must be a list of objects. If only *names* is specified, each name in *names* will be bound in the environment, but unassigned. If *names* and *values* are both specified, they must be the same length, and each name in *names* will be bound to the corresponding value in *values*. If neither *names* nor *values* is specified, the environment will have no initial bindings.

Environments in GoLisp differ slightly from standard Scheme in that they have a name attached. For the various forms of `let` and `do` this is simply `"let"` and `"do"`, respectively. Not of much use, but then these are just a byproduct of having lexical scopes. What's more useful is the higher level environments. This brings us to the real reason for adding environment support: game integration sandboxes. When we were writing the game integration functionallity for Engine3, we wanted each game's event handling to live in a separate sandbox. This is implemented buy creating a new top level environment under the global environment. The problem here is that it's off in it's own world, separate from the repl. By naming environments (in this case by the name of the game), we can add a function to return an environment given it's name. That allows us to peek inside the sandbox from the repl, examining and manipulating the bindings there. And so we added a function to let us do that:

15.0.21 (find-top-level-environment *name*)

Returns the top level environment with the given name.

16 Utility

GoLisp provides a handful of utility functions.

16.0.1 (random-byte)

Returns a psuedo-random unsigned integer between 0 and 255, inclusive.

```
(random-byte) ⇒ 13
(random-byte) ⇒ 207
```

16.0.2 (random)

The return value is a pseudorandom integer in the range [0, 2,147,483,647]

16.0.3 (random *modulus*)

Modulus must be a positive real number. If *modulus* is an integer, `random` returns a pseudo-random number between zero (inclusive) and *modulus* (exclusive). If *modulus* is the float 1.0, the returned number is a float in the range [0.0, 1.0). Other float values of *modulus* are rejected.

16.0.4 (sleep *millis*)

Sleep for *millis* milliseconds.

```
(sleep 1000)  ;; resumes execution 1 second later
```

16.0.5 (time *expression...*)

Evaluates each *expression* and reports the number of milliseconds taken to do so.

16.0.6 (write-line *object...*)

Writes the concatenation of the string forms of *objects* followed by a newline.

```
> (write-line "Hello, " 42 " world")
Hello, 42 world
=⇒ ()
```

16.0.7 (str *object...*)

If you provide multiple arguments to `str` it creates a string from concatenating the string forms of all the *objects*.

```
(str 1 "." 2) ⇒ "1.2"
```

16.0.8 (copy *object*)

Make a copy of the result of evaluating *object*, IFF it's mutable. This is limited to lists and association lists. All other values are immutable. Copying an immutable item will return the item, whereas copying a list or association list will make a deep copy of the structure, and return it.

16.0.9 (exec *command arg...*)

Makes an operating system call. `command` is the command to execute and the `arg`s are the arguments passed on the command line to `command`. `command` must be a string, and the `arg`s can be anything.

17 Concurrency

GoLisp has limited concurrency support that is built on top of goroutines and channels.

17.1 Process management

These functions make use of a *process* object. This is an opaque piece of data that wraps a custom structure used by the concurrency code; it is returned from `fork` and `schedule` and is used by `proc-sleep`, `wake`, and `abandon` to interact with the underlying goroutine.

17.1.1 (fork *function*)

Executes *function* in a separate goroutine. When *function* ends, the goroutine terminates. *function* takes a single argument which is the process object that is returned.

```
(define (run-once proc)
  (write-line "start")
  (sleep 1000)
  (write-line "stop"))

(fork run-once)

> start
[a second goes by]
stop
[run-once completes and the goroutine terminates]
```

17.1.2 (proc-sleep *process millis*)

Use `proc-sleep` in a forked function to sleep for *millis* milliseconds. Using `proc-sleep` rather than `sleep` (which can be used) allows code in another process (that has a reference to the process object of the forked code) to preemptively terminate the sleep using the `wake` function.

`proc-sleep` returns a boolean that indicates whether the sleep was terminated using `wake`.

17.1.3 (wake *process*)

Preemptively terminate a `proc-sleep` in the code associated with *process*.

```
> (define (run proc)
    (do ((woken #f woken))
        (woken (write-line "woken"))
      (write-line "tick")
      (set! woken (proc-sleep proc 10000))))

> (define p (fork run))
```

```
tick
tick
[times goes by, tick is printed every 10 seconds]
> (wake p)
woken
[run completes and the goroutine terminates]
```

17.1.4 (join *process*)

Blocks the calling function until the process completes or aborts with an error. The return value of the process is returned, or nil if the process ran into an error and aborted. Attempting to call `join` on a process twice raises an error.

```
(define (run proc) '(1 2 3))
(define p (fork run))
(join p) ⇒ (1 2 3)
```

17.1.5 (schedule *millis function*)

Schedule *function* to be evaluated in a separate goroutine *millis* milliseconds from now. *function* takes a single argument which is the process object that is returned. The process object associated with that goroutine is returned immediately.

```
> (define (run-delayed proc)
    (write-line "running"))

> (schedule 10000 run-delayed)
[10 seconds pass]
running
[run-delayed completes and the goroutine terminates]
```

17.1.6 (abandon *process*)

Cancels the scheduled evaluation associated with *process*.

```
> (define (run-delayed proc)
    (write-line "running"))

> (define p (schedule 10000 run-delayed))
[5 seconds pass]
> (abandon p)
[the delay is cancelled and the goroutine terminates]
```

17.1.7 (reset-timeout *process*)

Resets the timer on a scheduled process. Causing it to start over. You can use this function to postpone the evaluation of scheduled code.

```
> (define (run-delayed proc)
    (write-line "running"))

> (define p (schedule 10000 run-delayed))
[less than 10 seconds pass]
> (reset-timeout p)
[10 seconds pass]
running
[run-delayed completes and the goroutine terminates]
```

17.2 Atomic Operations

GoLisp has support for several kinds of atomic operations. These can be useful for protecting memory when working with GoLisp code with concurrent processes, or just Go code with multiple goroutines.

These functions make use of a *atomic* object. This is an opaque piece of data that wraps an integer; it is returned from `atomic` and is used by all the `atomic-*` primitives to interact with the underlying integer using only atomic operations.

17.2.1 (atomic [*value*])

Creates a new *atomic* object and returns it. It can optionally be passed a starting value to initialize to. Otherwise, the starting value is 0.

```
> (atomic)   ⇒ <atomic object with value 0>
> (atomic 5) ⇒ <atomic object with value 5>
```

17.2.2 (atomic-load *atomic*)

Loads the current integer value of the *atomic* object and returns it as an integer.

```
> (define a (atomic 5))
> (atomic-load a) ⇒ 5
```

17.2.3 (atomic-store! *atomic new*)

Stores a new integer value in a *atomic* object.

```
> (define a (atomic 5))
> (atomic-store! a 8)
> (atomic-load a) ⇒ 8
```

17.2.4 (atomic-add! *atomic delta*)

Adds the *delta* value to the one stored in the *atomic* object. The new sum is also returned.

```
> (define a (atomic 5))
> (atomic-add! a 4) ⇒ 9
> (atomic-load a)   ⇒ 9
```

17.2.5 (atomic-swap! *atomic new*)

Swaps the value currently in the *atomic* object with a new value. The old value is returned.

```
> (define a (atomic 5))
> (atomic-swap! a 4) ⇒ 5
> (atomic-load a)    ⇒ 4
```

17.2.6 (atomic-compare-and-swap! *atomic old new*)

The value in the *atomic* object is compared to *old*. If the value matches, the value in the *atomic* object is swapped with the value in *new* and true is returned. Otherwise, the values are not swapped and false is returned.

```
> (define a (atomic 5))
> (atomic-compare-and-swap! a 5 4) ⇒ #t
> (atomic-load a)                  ⇒ 4

> (define b (atomic 5))
> (atomic-compare-and-swap! b 9 4) ⇒ #f
> (atomic-load b)                  ⇒ 5
```

17.3 Channels

Channels are the main way you communicate between goroutines in Go. GoLisp has full support of channels.

17.3.1 (make-channel [*buffer-size*])

Creates a new channel object with an optional buffer size. If *buffer-size* is omitted or 0, the channel is unbuffered.

17.3.2 (channel-write *channel value*)

17.3.3 (*channel*<- *value*)

Writes a value to a channel. If the channel is unbuffered or has a full buffer, this call locks until there either another process tries to read from the channel or room is made in the buffer.

17.3.4 (channel-read *channel*)

17.3.5 (<-*channel*)

Reads a value from a channel. If the channel is unbuffered or has no buffered data, this call locks until there is data in the channel. `<-channel` returns two values. The first value is the data read from the channel. The second value is a boolean flag stating whether there is more data in the channel. If the channel is closed and there are no more items left in the buffer, a false flag is returned. Otherwise, a true flag is returned. If a flag of false is returned, the first value will also be nil.

```
> (define c (make-channel 1))
> (channel-write c 1)
> (c<- 1) ; alternate syntax for the previous line
> (channel-read c) ⇒ (1 #t)
> (<-c)            ⇒ (1 #t) ; alternate syntax for the previous line
> (channel-read c) ; blocks until another process writes to c
```

17.3.6 (channel-try-write *channel value*)

Tries to write a value to a channel. If the channel is unbuffered with nobody waiting for a write or has a full buffer, it returns immediately a false value. Otherwise, it writes the value to the channel and returns a true value.

(define c (make-channel 1) (channel-try-write c 1) ⇒ #t

(define c (make-channel)) (channel-try-write c 1) ⇒ #f

17.3.7 (channel-try-read *channel*)

Tries to reads a value from a channel. This call returns three values. The first is whether data could be read or not. The second is the data that is read, or nil if none was. The last value is whether the channel has more data in it.

```
> (define c (make-channel 1))
> (c<- 1)
> (channel-try-read c) ⇒ (#t 1 #t)
> (channel-try-read c) ⇒ (#f () #t)
```

17.3.8 (close-channel *channel*)

Closes the specified channel. The channel's buffered is cleared by any other goroutines trying to read from it then all other reads immediately return with the more flag set to false. Trying to write to a closed channel or trying to close a channel twice results in an error.

```
> (define c (make-channel 1))
> (c<- 1)
> (close-channel c)
> (<-c) ⇒ (1 #t)
> (<-c) ⇒ (() #f)
> (<-c) ⇒ (() #f) ; repeats on subsequent calls
> (channel-try-read c) ⇒ (#t () #f)
```

18 Time and Date

All of these date and time function use the local time.

18.0.1 (time-now)

Return the current local time as a list of the form `(hour minute second)`.

```
(time-now) ⇒ (19 53 50)
```

18.0.2 (seconds)

Returns the number of seconds since midnight on the current day.

```
(seconds) ⇒ 71644
```

18.0.3 (millis)

Returns the number of milliseconds since midnight on the current day.

```
(millis) ⇒ 71646935
```

18.0.4 (date-today)

Return today's date as a list of the form `(year month day)`.

```
(date-today) ⇒ (2016 12 19)
```

18.0.5 (date-in-days *days*)

Return the date in *days* days from today as a list of the form `(year month day)`.

```
(date-in-days 3) ⇒ (2016 12 22)
```

18.0.6 (day-of-week)

Returns a symbol representing the current day of the week.

```
(day-of-week) ⇒ monday
```

19 Timers

19.0.1 (timer *millis func*)

Schedules *func* (a function of zero arguments) to be evaluated after *millis* milliseconds. Returns a timer object.

19.0.2 (stop-timer *timer*)

Stop and cancel the timer *timer*.

19.0.3 (ticker *millis func*)

Schedules *func* (a function of zero arguments) to be evaluated after *millis* milliseconds and every *millis* milliseconds thereafter until stopped. Returns a ticker object.

19.0.4 (stop-ticker *ticker*)

Stop and cancel the ticker *ticker*.

20 Input/Output

20.0.1 (open-input-file *filename*)

Takes a *filename* referring to an existing file and returns an input port capable of delivering characters from the file.

20.0.2 (open-output-file *filename* [*append?*])

Takes a *filename* referring to an output file to be created and returns an output port capable of writing characters to a new file by that name.

If *append?* is given and not `#f`, the file is opened in append mode. In this mode, the contents of the file are not overwritten; instead any characters written to the file are appended to the end of the existing contents. If the file does not exist, append mode creates the file and writes to it in the normal way.

20.0.3 (close-port *port*)

Closes *port* and returns an unspecified value. The associated file is also closed.

20.0.4 (write-bytes *byte-array output-port*)

Writes *byte-array* to *output-port* as a stream of raw bytes. Most usefull for interacting with external devices via serial/usb ports.

20.0.5 (write-string *string* [*output-port*])

Writes *string* to *output-port*, performs discretionary output flushing, and returns an unspecified value.

20.0.6 (newline [*output-port*])

Writes an end-of-line to *output-port*, performs discretionary output flushing, and returns an unspecified value.

20.0.7 (write *object* [*output-port*])

Writes a written representation of *object* to *output-port*, and returns an unspecified value. If *object* has a standard external representation, then the written representation generated by `write` shall be parsable by `read` into an equivalent object. Thus strings that appear in the written representation are enclosed in doublequotes, and within those strings backslash and doublequote are escaped by backslashes. `write` performs discretionary output flushing and returns an unspecified value.

20.0.8 (read-string [*input-port*])

Reads characters from *input-port* until it finds a terminating character or encounters end of line. The port is updated to point to the terminating character, or to end of line if no terminating character was found. `read-string` returns the characters, up to but excluding the terminating character, as a newly allocated string.

20.0.9 (read [*input-port*])

Converts external representations of Scheme objects into the objects themselves. read returns the next object parsable from *input-port*, updating *input-port* to point to the first character past the end of the written representation of the object. If an end of file is encountered in the input before any characters are found that can begin an object, read returns an end-of-file object. The *input-port* remains open, and further attempts to read will also return an end-of-file object. If an end of file is encountered after the beginning of an object's written representation, but the written representation is incomplete and therefore not parsable, an error is signalled.

20.0.10 (eof-object? *object*)

Returns `#t` if *object* is an end-of-file object; otherwise returns `#f`.

20.0.11 (format *destination control-string argument...*)

Writes the characters of *control-string* to *destination*, except that a tilde (`~`) introduces a format directive. The character after the tilde, possibly preceded by prefix parameters and modifiers, specifies what kind of formatting is desired. Some directives use an *argument* to create their output; the typical directive puts the next *argument* into the output, formatted in some special way. It is an error if no *argument* remains for a directive requiring an *argument*.

The output is sent to *destination*. If *destination* is `#f`, a string is created that contains the output; this string is returned as the value of the call to `format`. If *destination* is `#t`, the output is sent to `stdout`. In all other cases `format` returns an unspecified value. Otherwise, destination must be an output port, and the output is sent there.

A format directive consists of a tilde (∼), an optional prefix parameter, an optional at-sign (`@`) modifier, and a single character indicating what kind of directive this is. The alphabetic case of the directive character is ignored. The prefix parameters are generally integers, notated as optionally signed decimal numbers.

In place of a prefix parameter to a directive, you can put the letter `V` (or `v`), which takes an argument for use as a parameter to the directive. Normally this should be an integer. This feature allows variable-width fields and the like. You can also use the character `#` in place of a parameter; it represents the number of arguments remaining to be processed.

`~A`: The next argument, which may be any object, is printed as if by write-line. `~<mincol>A` inserts spaces on the right, if necessary, to make the width at least `mincol` columns. The `@` modifier causes the spaces to be inserted on the left rather than the right.

`~S`: The next argument, which may be any object, is printed as if by write (in as read-able format as possible). `~mincolS` inserts spaces on the right, if necessary, to make the width at least mincol columns. The `@` modifier causes the spaces to be inserted on the left rather than the right.

`~%`: This outputs a newline character. This outputs a `\#newline` character. `~n%` outputs n newlines. No argument is used. Simply putting a newline in control-string would work, but `~%` is often used because it makes the control string look nicer in the middle of a program.

`~~`: This outputs a tilde; `~n~`: outputs n tildes.

`~newline`: Tilde immediately followed by a newline ignores the newline and any following whitespace characters. With an `@`, the newline is left in place, but any following whitespace is ignored. This directive is typically used when control-string is too long to fit nicely into one line of the program:

```
(define (type-clash-error procedure arg spec actual)
   (format
    #t
    "~%Procedure ~S~%requires its %A argument ~
     to be of type ~S,~%but it was called with ~
     an argument of type ~S.~%"
    procedure arg spec actual))
(type-clash-error 'vector-ref
                  "first"
                  'integer
                  'vector)
```

prints

```
Procedure vector-ref
requires its first argument to be of type integer,
but it was called with an argument of type vector.
```

Note that in this example newlines appear in the output only as specified by the ~% directives; the actual newline characters in the control string are suppressed because each is preceded by a tilde.

21 Testing

Golisp has a builtin testing framework, completely written in GoLisp.

21.1 Structure

Contests and it-clauses divide up the testing of a system under test into fixtures and focused sets of assertions.

21.1.1 (context *tag-string fixture it...*)

tag-string is a string used to identify the context in the test run's output. This should describe what the context focusses on. *fixture* is a sequence of expressions that typically add definitions (symbol bindings) to the environment created by `context`. Each *it* expression (defined using `it`) performs speific tests relevant to the context.

21.1.2 (it *tag assertion...*)

This defines a cohesive block of assertions. The context's fixture code will be run in a new environment for each *it* block, thus isolating each it.

21.2 Assertions

Assertions are used to make provable (via execution) statements about the system under test.

21.2.1 (assert-true *expression*)

Passes if *expression* evaluates to a truthy value, fails otherwise.

21.2.2 (assert-false *expression*)

Passes if *expression* evaluates to a falsy value, fails otherwise.

21.2.3 (assert-eq *actual expected*)

Passes if the result of evaluating *actual* is equal (using `(equal? actual expected)`) to the result of evaluating *expected*, fails otherwise.

21.2.4 (assert-neq *actual expected*)

Passes if the result of evaluating *actual* is **not** equal (using `(not (equal? actual expected))`to the result of evaluating *expected*, fails otherwise.

21.2.5 (assert-nil *expression*)

Passes if *expression* evaluates to nil, fails otherwise.

21.2.6 (assert-not-nil *expression*)

Passes if *expression* evaluates to anything **other than** nil, fails otherwise.

21.2.7 (assert-error *expression*)

Passes if evaluating *expression* results in an error being signalled, fails if it evaluates without problems.

21.2.8 (assert-nerror *expression*)

Passes if evaluating *expression* does not result in an error being signalled, fails if it the evaluation signals an error.

21.3 Usage

Generally you should create a test file for each feature you are testing. The file is a plain lisp file and can contain any lisp code, including global variable and function definitions.

For example, here is the test file for scoping:

```
(context "environments"

  ((define a 5)
   (define (foo a)
     (lambda (x)
      (+ a x))))

  (it "can access a in the global env"
      (assert-eq a 5))

  (it "gets a from the function's local env"
      (assert-eq ((foo 1) 5) 6)
      (assert-eq ((foo 2) 5) 7)
      (assert-eq ((foo 10) 7) 17)))
```

Running a test results in a stream of status output for each test, followed at the very end by a summary. Running the above results in the following:

```
environments

  can access a in the global env
    (assert-eq a 5)

  gets a from the function's local env
    (assert-eq ((foo 1) 5) 6)
    (assert-eq ((foo 2) 5) 7)
    (assert-eq ((foo 10) 7) 17)

Ran 4 tests in 0.003 seconds
4 passes, 0 failures, 0 errors
```

If we introduce a failure, the output would be:

```
environments

  can access a in the global env
    (assert-eq a 5)

  gets a from the function's local env
    (assert-eq ((foo 1) 5) 6)
    (assert-eq ((foo 2) 5) 8)
      - expected 8, but was 7
    (assert-eq ((foo 10) 7) 17)

Ran 4 tests in 0.002 seconds
3 passes, 1 failures, 0 errors

Failures:
  environments gets a from the function's local env:
    (assert-eq ((foo 2) 5) 8)
      - expected 8, but was 7
```

Errors are also reported. Errors are problems that occur while evaluating the clauses, that aren't failures. Essentially they indicate bugs of some sort.

```
environments

  can access a in the global env
    (assert-eq a 5)
```

```
  gets a from the function's local env
    (assert-eq ((foo 1) 5) 6)
    ERROR: Quotent: (7 0) -> Divide by zero.

Ran 3 tests in 0.002 seconds
2 passes, 0 failures, 1 errors

Errors:
  environments gets a from the function's local env:
    ERROR: Quotent: (7 0) -> Divide by zero.
```

The above output was generated by the testing framwork running in verbose mode. You can also run in quiet mode which only outputs the summary:

```
Ran 4 tests in 0.003 seconds
4 passes, 0 failures, 0 errors
```

You run tests by running the golisp repl in test mode, providing either a directory or filename. If you provide a directory all files in it that match `*_test.scm` will be run. If you provide a filename, only that file will be run.

```
$golisp -t tests/scope_test.scm

Ran 4 tests in 0.002 seconds
4 passes, 0 failures, 0 errors
```

```
$golisp -t tests

Ran 935 tests in 0.273 seconds
935 passes, 0 failures, 0 errors
```

Adding the `-v` flag will produce the detailed output above.

22 Extending GoLisp

22.1 Defining primitives

The Go function `MakePrimitiveFunction` allows you to create primitive functions.

```
MakePrimitiveFunction(name string, argCount string,
                      function func(*Data, *SymbolTableFrame)(*Data, error))
```

The arguments are:

1. The function name. This is the name of a symbol which will be used to reference the function.
2. An argument count expectation. This is a string that specifies how many arguments the primitive expects. It can take several forms:
   - A single, specific number. E.g. exactly two: `"2"`
   - A minimum number. E.g. at least two: `">=2"`
   - A range of values. E.g. between two and five, inclusive: `"(2,5)"`
   - One of a selection of the above: E.g. `"2|3|>=5"`
   - An unspecified number, any checking must be done in the primitive definition: `"*"`
3. The Go function which implements the primitive. This function **must** have the signature

   ```
   func <Name>(args *Data, env *SymbolTableFrame) (*Data, error)
   ```

The implementing function takes two parameters as seen above:

1. A Lisp list containing the arguments
2. The environment in which the primitive is being evaluated. This is used when calling `Eval` or `Apply`, as well as for any symbol lookups or bindings.

Primitives use, like functions defined in LISP, applicative evaluation order. That means that all arguments are evaluated and the resulting values passed to the function. This frees you from having to evaluate the arguments and handle errors. You still have to verify the number of arguments (only if you used -1 as trhe argument cound in the `MakePrimitiveFunction` call) and their type, if applicable.

```
MakeSpecialForm(name string,
                argCount int,
                function func(*Data, *SymbolTableFrame) (*Data, error))
```

An example:

```
MakePrimitiveFunction("!", "1", BooleanNot)

func BooleanNot(args *Data, env *SymbolTableFrame) (result *Data, err error) {
    val := BooleanValue(First(args))
    return BooleanWithValue(!val), nil
 }
```

You can extend the goLisp runtime without changing any of it's code. You simply import the golisp package (typically aliased to `.` to make the code less noisy) and place calls to `MakePrimitiveFunction` in your package's `init` block.

22.2 Defining primitives with argument type checking

There is also the `MakeTypedPrimitiveFunction` function that takes an additional argument which is an array of `uint32`s, one element for each argument. If the defined function accepts an arbitrary number of arguments, the final type specification is used for the remainder. For example, if there are 3 argument type specifications and the function is passed 5 arguments, the final specification is used for the 3rd, 4th, and 5th arguments.

```
MakeTypedPrimitiveFunction("mqtt/publish", "3", mqttPublishImpl,
                           []uint32\{StringType, IntegerType, StringType\})
```

There is not currently a way to provide a type specification for a primitive's return value.

22.3 Defining special forms

There is another, very similar function that you will typically not need unless you are hacking on the language itself (as opposed to adding builting functions):

```
MakeSpecialForm(name string,
                argCount int,
                function func(*Data, *SymbolTableFrame) (*Data, error))
```

Arguments and the signature of the implementing function are identical to `MakePrimitiveFunction`. The only difference is that this defines a *special form* which uses normal evaluation order. I.e. arguments are not evaluated before calling the function; the raw sexpressions are passed in. Thus the implementing function has full control over what gets evaluated and when. For example:

```
MakeSpecialForm("if", "2|3", IfImpl)

func IfImpl(args *Data, env *SymbolTableFrame) (result *Data, err error) {
    c, err := Eval(First(args), env)
    if err != nil {
        return
    }

    if BooleanValue(c) {
        return Eval(Second(args), env)
    } else {
        return Eval(Third(args), env)
    }
}
```

22.4 Data

The core lisp data element is the data type which logically contains a type tag and a value. The type tags are defined by the constants: `ConsCellType`, `NumberType`, `BooleanType`, `StringType`, `SymbolType`, `FunctionType`, `PrimitiveType`, `ObjectType`. As the language evolves this list (and the associated functions) will change. Refer to the file `data.go` for the definitive information.

The types are described earlier. If you need to check the type of a piece of data you can fetch it's type using the `TypeOf(*Data) uint32` function and then compare it to a type tag constant. Additionally there are predicate functions for the most common types that have the general form: