SLIB

The Portable Scheme Library
Version 2d3

by Aubrey Jaffer

SLIB is a portable library for the programming language *Scheme*. It provides a platform independent framework for using *packages* of Scheme procedures and syntax. As distributed, SLIB contains useful packages for all Scheme implementations. Its catalog can be transparently extended to accomodate packages specific to a site, implementation, user, or directory.

More people than I can name have contributed to SLIB. Thanks to all of you!

> SLIB 2d3, released February 2002.
> Aubrey Jaffer <agj @ alum.mit.edu>
> http://swissnet.ai.mit.edu/~jaffer/SLIB.html

1 The Library System

1.1 Feature

SLIB denotes *features* by symbols. SLIB maintains a list of features supported by the Scheme *session*. The set of features provided by a session may change over time. Some features are properties of the Scheme implementation being used. The following features detail what sort of numbers are available from an implementation.

- 'inexact
- 'rational
- 'real
- 'complex
- 'bignum

Other features correspond to the presence of sets of Scheme procedures or syntax (macros).

provided? *feature* Function

Returns `#t` if *feature* is supported by the current Scheme session.

provide *feature* Procedure

Informs SLIB that *feature* is supported. Henceforth `(provided?` *feature*`)` will return `#t`.

```
(provided? 'foo)    ⇒ #f
(provide 'foo)
(provided? 'foo)    ⇒ #t
```

1.2 Requesting Features

SLIB creates and maintains a *catalog* mapping features to locations of files introducing procedures and syntax denoted by those features.

At the beginning of each section of this manual, there is a line like `(require '`*feature*`)`. The Scheme files comprising SLIB are cataloged so that these feature names map to the corresponding files.

SLIB provides a form, `require`, which loads the files providing the requested feature.

require *feature* Procedure

- If `(provided?` *feature*`)` is true, then `require` just returns an unspecified value.
- Otherwise, if *feature* is found in the catalog, then the corresponding files will be loaded and an unspecified value returned.

  Subsequently `(provided?` *feature*`)` will return `#t`.

- Otherwise (*feature* not found in the catalog), an error is signaled.

The catalog can also be queried using `require:feature->path`.

require:feature->path *feature* Function

- If *feature* is already provided, then returns `#t`.
- Otherwise, if *feature* is in the catalog, the path or list of paths associated with *feature* is returned.
- Otherwise, returns `#f`.

1.3 Library Catalogs

At the start of a session no catalog is present, but is created with the first catalog inquiry (such as `(require 'random)`). Several sources of catalog information are combined to produce the catalog:

- standard SLIB packages.
- additional packages of interest to this site.
- packages specifically for the variety of Scheme which this session is running.
- packages this user wants to always have available. This catalog is the file '`homecat`' in the user's *HOME* directory.
- packages germane to working in this (current working) directory. This catalog is the file '`usercat`' in the directory to which it applies. One would typically `cd` to this directory before starting the Scheme session.

Catalog files consist of one or more *association lists*. In the circumstance where a feature symbol appears in more than one list, the latter list's association is retrieved. Here are the supported formats for elements of catalog lists:

`(`*feature* `. <`*symbol*`>)`
: Redirects to the feature named `<`*symbol*`>`.

`(`*feature* `. "<`*path*`>")`
: Loads file `<`*path*`>`.

`(`*feature* `source "<`*path*`>")`
: `slib:load`s the Scheme source file `<`*path*`>`.

`(`*feature* `compiled "<`*path*`>" ...)`
: `slib:load-compiled`s the files `<`*path*`>` . . . .

`(`*feature* `aggregate <`*symbol*`> ...)`
: `require:require`s the features `<`*symbol*`>` . . . .

The various macro styles first `require` the named macro package, then just load `<`*path*`>` or load-and-macro-expand `<`*path*`>` as appropriate for the implementation.

`(`*feature* `defmacro "<`*path*`>")`
: `defmacro:load`s the Scheme source file `<`*path*`>`.

(*feature* `macro-by-example` "<*path*>")
: `defmacro:load`s the Scheme source file <*path*>.

(*feature* `macro` "<*path*>")
: `macro:load`s the Scheme source file <*path*>.

(*feature* `macros-that-work` "<*path*>")
: `macro:load`s the Scheme source file <*path*>.

(*feature* `syntax-case` "<*path*>")
: `macro:load`s the Scheme source file <*path*>.

(*feature* `syntactic-closures` "<*path*>")
: `macro:load`s the Scheme source file <*path*>.

Here is an example of a '`usercat`' catalog. A Program in this directory can invoke the '`run`' feature with `(require 'run)`.

```
;;; "usercat": SLIB catalog additions for SIMSYNCH.     -*-scheme-*-

(
 (simsynch      . "../synch/simsynch.scm")
 (run           . "../synch/run.scm")
 (schlep        . "schlep.scm")
)
```

1.4 Catalog Compilation

SLIB combines the catalog information which doesn't vary per user into the file '`slibcat`' in the implementation-vicinity. Therefore '`slibcat`' needs change only when new software is installed or compiled. Because the actual pathnames of files can differ from installation to installation, SLIB builds a separate catalog for each implementation it is used with.

The definition of `*SLIB-VERSION*` in SLIB file '`require.scm`' is checked against the catalog association of `*SLIB-VERSION*` to ascertain when versions have changed. I recommend that the definition of `*SLIB-VERSION*` be changed whenever the library is changed. If multiple implementations of Scheme use SLIB, remember that recompiling one '`slibcat`' will fix only that implementation's catalog.

The compilation scripts of Scheme implementations which work with SLIB can automatically trigger catalog compilation by deleting '`slibcat`' or by invoking a special form of `require`:

require 'new-catalog Procedure

This will load '`mklibcat`', which compiles and writes a new '`slibcat`'.

Another special form of `require` erases SLIB's catalog, forcing it to be reloaded the next time the catalog is queried.

require #f — Procedure

Removes SLIB's catalog information. This should be done before saving an executable image so that, when restored, its catalog will be loaded afresh.

Each file in the table below is descibed in terms of its file-system independent *vicinity* (see Section 1.5.2 [Vicinity], page 5). The entries of a catalog in the table override those of catalogs above it in the table.

`implementation-vicinity` '`slibcat`'
: This file contains the associations for the packages comprising SLIB, the '`implcat`' and the '`sitecat`'s. The associations in the other catalogs override those of the standard catalog.

`library-vicinity` '`mklibcat.scm`'
: creates '`slibcat`'.

`library-vicinity` '`sitecat`'
: This file contains the associations specific to an SLIB installation.

`implementation-vicinity` '`implcat`'
: This file contains the associations specific to an implementation of Scheme. Different implementations of Scheme should have different `implementation-vicinity`.

`implementation-vicinity` '`mkimpcat.scm`'
: if present, creates '`implcat`'.

`implementation-vicinity` '`sitecat`'
: This file contains the associations specific to a Scheme implementation installation.

`home-vicinity` '`homecat`'
: This file contains the associations specific to an SLIB user.

`user-vicinity` '`usercat`'
: This file contains associations effecting only those sessions whose *working directory* is `user-vicinity`.

1.5 Built-in Support

The procedures described in these sections are supported by all implementations as part of the '`*.init`' files or by '`require.scm`'.

1.5.1 Require

features — Variable

Is a list of symbols denoting features supported in this implementation. **features** can grow as modules are `required`. **features** must be defined by all implementations (see Section 7.2 [Porting], page 218).

Here are features which SLIB (‘`require.scm`’) adds to **features** when appropriate.

- ’inexact
- ’rational
- ’real
- ’complex
- ’bignum

For each item, `(provided? '`*feature*`)` will return `#t` if that feature is available, and `#f` if not.

modules Variable

Is a list of pathnames denoting files which have been loaded.

catalog Variable

Is an association list of features (symbols) and pathnames which will supply those features. The pathname can be either a string or a pair. If pathname is a pair then the first element should be a macro feature symbol, `source`, or `compiled`. The cdr of the pathname should be either a string or a list.

In the following functions if the argument *feature* is not a symbol it is assumed to be a pathname.

provided? *feature* Function

Returns `#t` if *feature* is a member of `*features*` or `*modules*` or if *feature* is supported by a file already loaded and `#f` otherwise.

require *feature* Procedure

feature is a symbol. If `(provided?` *feature*`)` is true `require` returns. Otherwise, if `(assq` *feature* `*catalog*)` is not `#f`, the associated files will be loaded and `(provided?` *feature*`)` will henceforth return `#t`. An unspecified value is returned. If *feature* is not found in `*catalog*`, then an error is signaled.

require *pathname* Procedure

pathname is a string. If *pathname* has not already been given as an argument to `require`, *pathname* is loaded. An unspecified value is returned.

provide *feature* Procedure

Assures that *feature* is contained in `*features*` if *feature* is a symbol and `*modules*` otherwise.

require:feature->path *feature* Function

Returns `#t` if *feature* is a member of `*features*` or `*modules*` or if *feature* is supported by a file already loaded. Returns a path if one was found in `*catalog*` under the feature name, and `#f` otherwise. The path can either be a string suitable as an argument to load or a pair as described above for *catalog*.

1.5.2 Vicinity

A vicinity is a descriptor for a place in the file system. Vicinities hide from the programmer the concepts of host, volume, directory, and version. Vicinities express only the concept of a file environment where a file name can be resolved to a file in a system independent manner. Vicinities can even be used on *flat* file systems (which have no directory structure) by having the vicinity express constraints on the file name. On most systems a vicinity would be a string. All of these procedures are file system dependent.

These procedures are provided by all implementations.

make-vicinity *path* Function

Returns the vicinity of *path* for use by `in-vicinity`.

program-vicinity Function

Returns the vicinity of the currently loading Scheme code. For an interpreter this would be the directory containing source code. For a compiled system (with multiple files) this would be the directory where the object or executable files are. If no file is currently loading it the result is undefined. **Warning:** `program-vicinity` can return incorrect values if your program escapes back into a `load`.

library-vicinity Function

Returns the vicinity of the shared Scheme library.

implementation-vicinity Function

Returns the vicinity of the underlying Scheme implementation. This vicinity will likely contain startup code and messages and a compiler.

user-vicinity Function

Returns the vicinity of the current directory of the user. On most systems this is ‘`""`’ (the empty string).

home-vicinity Function

Returns the vicinity of the user’s *HOME* directory, the directory which typically contains files which customize a computer environment for a user. If scheme is running without a user (eg. a daemon) or if this concept is meaningless for the platform, then `home-vicinity` returns `#f`.

in-vicinity *vicinity filename* Function

Returns a filename suitable for use by `slib:load`, `slib:load-source`, `slib:load-compiled`, `open-input-file`, `open-output-file`, etc. The returned filename is *filename* in *vicinity*. `in-vicinity` should allow *filename* to override *vicinity* when *filename* is an absolute pathname and *vicinity* is equal to the value of `(user-vicinity)`. The behavior of `in-vicinity` when *filename* is absolute and *vicinity* is not equal to the value of `(user-vicinity)` is unspecified. For most systems `in-vicinity` can be `string-append`.

sub-vicinity *vicinity name* Function

Returns the vicinity of *vicinity* restricted to *name*. This is used for large systems where names of files in subsystems could conflict. On systems with directory structure `sub-vicinity` will return a pathname of the subdirectory *name* of *vicinity*.

1.5.3 Configuration

These constants and procedures describe characteristics of the Scheme and underlying operating system. They are provided by all implementations.

char-code-limit Constant

An integer 1 larger that the largest value which can be returned by `char->integer`.

most-positive-fixnum Constant

In implementations which support integers of practically unlimited size, *most-positive-fixnum* is a large exact integer within the range of exact integers that may result from computing the length of a list, vector, or string.

In implementations which do not support integers of practically unlimited size, *most-positive-fixnum* is the largest exact integer that may result from computing the length of a list, vector, or string.

slib:tab Constant

The tab character.

slib:form-feed Constant

The form-feed character.

software-type Function

Returns a symbol denoting the generic operating system type. For instance, `unix`, `vms`, `macos`, `amiga`, or `ms-dos`.

slib:report-version Function

Displays the versions of SLIB and the underlying Scheme implementation and the name of the operating system. An unspecified value is returned.

```
(slib:report-version) ⇒ slib "2d3" on scm "5b1" on unix
```

slib:report Function

Displays the information of `(slib:report-version)` followed by almost all the information neccessary for submitting a problem report. An unspecified value is returned.

slib:report *#t* Function

provides a more verbose listing.

slib:report *filename* Function

Writes the report to file ‘`filename`’.

```
(slib:report)
⇒
slib "2d3" on scm "5b1" on unix
(implementation-vicinity) is "/home/jaffer/scm/"
(library-vicinity) is "/home/jaffer/slib/"
(scheme-file-suffix) is ".scm"
loaded *features* :
        trace alist qp sort
        common-list-functions macro values getopt
        compiled
implementation *features* :
        bignum complex real rational
        inexact vicinity ed getenv
        tmpnam abort transcript with-file
        ieee-p1178 rev4-report rev4-optional-procedures hash
        object-hash delay eval dynamic-wind
        multiarg-apply multiarg/and- logical defmacro
        string-port source current-time record
        rev3-procedures rev2-procedures sun-dl string-case
        array dump char-ready? full-continuation
        system
implementation *catalog* :
        (i/o-extensions compiled "/home/jaffer/scm/ioext.so")
        ...
```

1.5.4 Input/Output

These procedures are provided by all implementations.

file-exists? *filename* Function

Returns `#t` if the specified file exists. Otherwise, returns `#f`. If the underlying implementation does not support this feature then `#f` is always returned.

delete-file *filename* Function

Deletes the file specified by *filename*. If *filename* can not be deleted, `#f` is returned. Otherwise, `#t` is returned.

open-file *filename modes* Function

filename should be a string naming a file. `open-file` returns a port depending on the symbol *modes*:

r	an input port capable of delivering characters from the file.
rb	a *binary* input port capable of delivering characters from the file.
w	an output port capable of writing characters to a new file by that name.
wb	a *binary* output port capable of writing characters to a new file by that name.

If an implementation does not distinguish between binary and non-binary files, then it must treat rb as r and wb as w.

If the file cannot be opened, either #f is returned or an error is signalled. For output, if a file with the given name already exists, the effect is unspecified.

port? *obj* Function

Returns `#t` if *obj* is an input or output port, otherwise returns `#f`.

close-port *port* Procedure

Closes the file associated with *port*, rendering the *port* incapable of delivering or accepting characters.

`close-file` has no effect if the file has already been closed. The value returned is unspecified.

call-with-open-ports *proc ports . . .* Function
call-with-open-ports *ports . . . proc* Function

Proc should be a procedure that accepts as many arguments as there are *ports* passed to `call-with-open-ports`. `call-with-open-ports` calls *proc* with *ports* . . . . If *proc* returns, then the ports are closed automatically and the value yielded by the *proc* is returned. If *proc* does not return, then the ports will not be closed automatically unless it is possible to prove that the ports will never again be used for a read or write operation.

tmpnam Function

Returns a pathname for a file which will likely not be used by any other process. Successive calls to `(tmpnam)` will return different pathnames.

current-error-port Function

Returns the current port to which diagnostic and error output is directed.

force-output Procedure
force-output *port* Procedure

Forces any pending output on *port* to be delivered to the output device and returns an unspecified value. The *port* argument may be omitted, in which case it defaults to the value returned by `(current-output-port)`.

output-port-width Function
output-port-width *port* Function

Returns the width of *port*, which defaults to `(current-output-port)` if absent. If the width cannot be determined 79 is returned.

output-port-height Function
output-port-height *port* Function

Returns the height of *port*, which defaults to `(current-output-port)` if absent. If the height cannot be determined 24 is returned.

1.5.5 System

These procedures are provided by all implementations.

slib:load-source *name* Procedure

Loads a file of Scheme source code from *name* with the default filename extension used in SLIB. For instance if the filename extension used in SLIB is '`.scm`' then `(slib:load-source "foo")` will load from file '`foo.scm`'.

slib:load-compiled *name* Procedure

On implementations which support separtely loadable compiled modules, loads a file of compiled code from *name* with the implementation's filename extension for compiled code appended.

slib:load *name* Procedure

Loads a file of Scheme source or compiled code from *name* with the appropriate suffixes appended. If both source and compiled code are present with the appropriate names then the implementation will load just one. It is up to the implementation to choose which one will be loaded.

If an implementation does not support compiled code then `slib:load` will be identical to `slib:load-source`.

slib:eval *obj* Procedure

`eval` returns the value of *obj* evaluated in the current top level environment. Section 6.4.11 [Eval], page 204 provides a more general evaluation facility.

slib:eval-load *filename eval* Procedure

filename should be a string. If filename names an existing file, the Scheme source code expressions and definitions are read from the file and *eval* called with them sequentially. The `slib:eval-load` procedure does not affect the values returned by `current-input-port` and `current-output-port`.

slib:warn *arg1 arg2 . . .* Procedure

Outputs a warning message containing the arguments.

slib:error *arg1 arg2 . . .* Procedure

Outputs an error message containing the arguments, aborts evaluation of the current form and responds in a system dependent way to the error. Typical responses are to abort the program or to enter a read-eval-print loop.

slib:exit *n* Procedure
slib:exit Procedure

Exits from the Scheme session returning status *n* to the system. If *n* is omitted or `#t`, a success status is returned to the system (if possible). If *n* is `#f` a failure is returned to the system (if possible). If *n* is an integer, then *n* is returned to the system (if possible). If the Scheme session cannot exit an unspecified value is returned from `slib:exit`.

1.5.6 Miscellany

These procedures are provided by all implementations.

identity *x* Function

identity returns its argument.

Example:

```
(identity 3)
   ⇒ 3
(identity '(foo bar))
   ⇒ (foo bar)
(map identity lst)
   ≡ (copy-list lst)
```

1.5.6.1 Mutual Exclusion

An *exchanger* is a procedure of one argument regulating mutually exclusive access to a resource. When a exchanger is called, its current content is returned, while being replaced by its argument in an atomic operation.

make-exchanger *obj* Function

Returns a new exchanger with the argument *obj* as its initial content.

```
(define queue (make-exchanger (list a)))
```

A queue implemented as an exchanger holding a list can be protected from reentrant execution thus:

```
(define (pop queue)
  (let ((lst #f))
    (dynamic-wind
        (lambda () (set! lst (queue #f)))
        (lambda () (and lst (not (null? lst))
                        (let ((ret (car lst)))
                          (set! lst (cdr lst))
                          ret)))
        (lambda () (and lst (queue lst))))))

(pop queue)         ⇒ a

(pop queue)         ⇒ #f
```

1.5.6.2 Legacy

The following procedures were present in Scheme until R4RS (see section "Language changes " in *Revised(4) Scheme*). They are provided by all SLIB implementations.

t Constant

Derfined as `#t`.

nil Constant

Defined as `#f`.

last-pair *l* Function

Returns the last pair in the list *l*. Example:

```
(last-pair (cons 1 2))
   ⇒ (1 . 2)
(last-pair '(1 2))
   ⇒ (2)
    ≡ (cons 2 '())
```

1.6 About this manual

- Entries that are labeled as Functions are called for their return values. Entries that are labeled as Procedures are called primarily for their side effects.
- Examples in this text were produced using the `scm` Scheme implementation.
- At the beginning of each section, there is a line that looks like `(require 'feature)`. Include this line in your code prior to using the package.

2 Scheme Syntax Extension Packages

2.1 Defmacro

Defmacros are supported by all implementations.

gentemp Function

Returns a new (interned) symbol each time it is called. The symbol names are implementation-dependent

```
(gentemp) ⇒ scm:G0
(gentemp) ⇒ scm:G1
```

defmacro:eval *e* Function

Returns the `slib:eval` of expanding all defmacros in scheme expression *e*.

defmacro:load *filename* Function

filename should be a string. If filename names an existing file, the `defmacro:load` procedure reads Scheme source code expressions and definitions from the file and evaluates them sequentially. These source code expressions and definitions may contain defmacro definitions. The `macro:load` procedure does not affect the values returned by `current-input-port` and `current-output-port`.

defmacro? *sym* Function

Returns `#t` if *sym* has been defined by `defmacro`, `#f` otherwise.

macroexpand-1 *form* Function
macroexpand *form* Function

If *form* is a macro call, `macroexpand-1` will expand the macro call once and return it. A *form* is considered to be a macro call only if it is a cons whose `car` is a symbol for which a `defmacro` has been defined.

`macroexpand` is similar to `macroexpand-1`, but repeatedly expands *form* until it is no longer a macro call.

defmacro *name lambda-list form . . .* Macro

When encountered by `defmacro:eval`, `defmacro:macroexpand*`, or `defmacro:load` defines a new macro which will henceforth be expanded when encountered by `defmacro:eval`, `defmacro:macroexpand*`, or `defmacro:load`.

2.1.1 Defmacroexpand

```
(require 'defmacroexpand)
```

defmacro:expand* *e* Function

Returns the result of expanding all defmacros in scheme expression *e*.

2.2 R4RS Macros

`(require 'macro)` is the appropriate call if you want R4RS high-level macros but don't care about the low level implementation. If an SLIB R4RS macro implementation is already loaded it will be used. Otherwise, one of the R4RS macros implemetations is loaded.

The SLIB R4RS macro implementations support the following uniform interface:

macro:expand *sexpression* Function

Takes an R4RS expression, macro-expands it, and returns the result of the macro expansion.

macro:eval *sexpression* Function

Takes an R4RS expression, macro-expands it, evals the result of the macro expansion, and returns the result of the evaluation.

macro:load *filename* Procedure

filename should be a string. If filename names an existing file, the `macro:load` procedure reads Scheme source code expressions and definitions from the file and evaluates them sequentially. These source code expressions and definitions may contain macro definitions. The `macro:load` procedure does not affect the values returned by `current-input-port` and `current-output-port`.

2.3 Macro by Example

`(require 'macro-by-example)`

A vanilla implementation of *Macro by Example* (Eugene Kohlbecker, R4RS) by Dorai Sitaram, (dorai @ cs.rice.edu) using `defmacro`.

- generating hygienic global `define-syntax` Macro-by-Example macros **cheaply**.
- can define macros which use `...`.
- needn't worry about a lexical variable in a macro definition clashing with a variable from the macro use context
- don't suffer the overhead of redefining the repl if `defmacro` natively supported (most implementations)

2.3.1 Caveat

These macros are not referentially transparent (see section "Macros" in *Revised(4) Scheme*). Lexically scoped macros (i.e., `let-syntax` and `letrec-syntax`) are not supported. In any case, the problem of referential transparency gains poignancy only when `let-syntax` and `letrec-syntax` are used. So you will not be courting large-scale disaster unless you're using system-function names as local variables with unintuitive bindings that the macro can't use. However, if you must have the full *r4rs* macro functionality, look to the more featureful (but also more expensive) versions of syntax-rules available in slib

Section 2.4 [Macros That Work], page 15, Section 2.5 [Syntactic Closures], page 18, and Section 2.6 [Syntax-Case Macros], page 25.

define-syntax *keyword transformer-spec* Macro

The *keyword* is an identifier, and the *transformer-spec* should be an instance of `syntax-rules`.

The top-level syntactic environment is extended by binding the *keyword* to the specified transformer.

```
(define-syntax let*
  (syntax-rules ()
    ((let* () body1 body2 ...)
     (let () body1 body2 ...))
    ((let* ((name1 val1) (name2 val2) ...)
       body1 body2 ...)
     (let ((name1 val1))
       (let* (( name2 val2) ...)
         body1 body2 ...)))))
```

syntax-rules *literals syntax-rule . . .* Macro

literals is a list of identifiers, and each *syntax-rule* should be of the form

(*pattern template*)

where the *pattern* and *template* are as in the grammar above.

An instance of `syntax-rules` produces a new macro transformer by specifying a sequence of hygienic rewrite rules. A use of a macro whose keyword is associated with a transformer specified by `syntax-rules` is matched against the patterns contained in the *syntax-rule*s, beginning with the leftmost *syntax-rule*. When a match is found, the macro use is trancribed hygienically according to the template.

Each pattern begins with the keyword for the macro. This keyword is not involved in the matching and is not considered a pattern variable or literal identifier.

2.4 Macros That Work

```
(require 'macros-that-work)
```

Macros That Work differs from the other R4RS macro implementations in that it does not expand derived expression types to primitive expression types.

macro:expand *expression* Function
macwork:expand *expression* Function

Takes an R4RS expression, macro-expands it, and returns the result of the macro expansion.

macro:eval *expression* Function
macwork:eval *expression* Function

macro:eval returns the value of *expression* in the current top level environment. *expression* can contain macro definitions. Side effects of *expression* will affect the top level environment.

macro:load *filename* Procedure
macwork:load *filename* Procedure

filename should be a string. If filename names an existing file, the macro:load procedure reads Scheme source code expressions and definitions from the file and evaluates them sequentially. These source code expressions and definitions may contain macro definitions. The macro:load procedure does not affect the values returned by current-input-port and current-output-port.

References:

The *Revised^4 Report on the Algorithmic Language Scheme* Clinger and Rees [editors]. To appear in LISP Pointers. Also available as a technical report from the University of Oregon, MIT AI Lab, and Cornell.

Macros That Work. Clinger and Rees. POPL '91.

The supported syntax differs from the R4RS in that vectors are allowed as patterns and as templates and are not allowed as pattern or template data.

```
transformer spec  ↦  (syntax-rules literals rules)

rules  ↦  ()
         |  (rule . rules)

rule  ↦  (pattern template)

pattern  ↦  pattern_var      ; a symbol not in literals
           |  symbol           ; a symbol in literals
           |  ()
           |  (pattern . pattern)
           |  (ellipsis_pattern)
           |  #(pattern*)                     ; extends R4RS
           |  #(pattern* ellipsis_pattern)    ; extends R4RS
           |  pattern_datum

template  ↦  pattern_var
            |  symbol
            |  ()
            |  (template2 . template2)
            |  #(template*)                   ; extends R4RS
            |  pattern_datum

template2  ↦  template
             |  ellipsis_template
```

```
pattern_datum  ↦  string                    ; no vector
                |  character
                |  boolean
                |  number

ellipsis_pattern  ↦ pattern ...

ellipsis_template  ↦  template ...

pattern_var  ↦  symbol   ; not in literals

literals  ↦  ()
           |  (symbol . literals)
```

2.4.1 Definitions

Scope of an ellipsis
: Within a pattern or template, the scope of an ellipsis (...) is the pattern or template that appears to its left.

Rank of a pattern variable
: The rank of a pattern variable is the number of ellipses within whose scope it appears in the pattern.

Rank of a subtemplate
: The rank of a subtemplate is the number of ellipses within whose scope it appears in the template.

Template rank of an occurrence of a pattern variable
: The template rank of an occurrence of a pattern variable within a template is the rank of that occurrence, viewed as a subtemplate.

Variables bound by a pattern
: The variables bound by a pattern are the pattern variables that appear within it.

Referenced variables of a subtemplate
: The referenced variables of a subtemplate are the pattern variables that appear within it.

Variables opened by an ellipsis template
: The variables opened by an ellipsis template are the referenced pattern variables whose rank is greater than the rank of the ellipsis template.

2.4.2 Restrictions

No pattern variable appears more than once within a pattern.

For every occurrence of a pattern variable within a template, the template rank of the occurrence must be greater than or equal to the pattern variable's rank.

Every ellipsis template must open at least one variable.

For every ellipsis template, the variables opened by an ellipsis template must all be bound to sequences of the same length.

The compiled form of a *rule* is

```
rule  ↦  (pattern template inserted)

pattern  ↦  pattern_var
          |  symbol
          |  ()
          |  (pattern . pattern)
          |  ellipsis_pattern
          |  #(pattern)
          |  pattern_datum

template  ↦  pattern_var
           |  symbol
           |  ()
           |  (template2 . template2)
           |  #(pattern)
           |  pattern_datum

template2  ↦  template
            |  ellipsis_template

pattern_datum  ↦  string
                |  character
                |  boolean
                |  number

pattern_var  ↦  #(V symbol rank)

ellipsis_pattern  ↦  #(E pattern pattern_vars)

ellipsis_template  ↦  #(E template pattern_vars)

inserted  ↦  ()
           |  (symbol . inserted)

pattern_vars  ↦  ()
               |  (pattern_var . pattern_vars)

rank  ↦  exact non-negative integer
```

where V and E are unforgeable values.

The pattern variables associated with an ellipsis pattern are the variables bound by the pattern, and the pattern variables associated with an ellipsis template are the variables opened by the ellipsis template.

If the template contains a big chunk that contains no pattern variables or inserted identifiers, then the big chunk will be copied unnecessarily. That shouldn't matter very often.

2.5 Syntactic Closures

```
(require 'syntactic-closures)
```

macro:expand *expression* Function
synclo:expand *expression* Function

Returns scheme code with the macros and derived expression types of *expression* expanded to primitive expression types.

macro:eval *expression* Function
synclo:eval *expression* Function

`macro:eval` returns the value of *expression* in the current top level environment. *expression* can contain macro definitions. Side effects of *expression* will affect the top level environment.

macro:load *filename* Procedure
synclo:load *filename* Procedure

filename should be a string. If filename names an existing file, the `macro:load` procedure reads Scheme source code expressions and definitions from the file and evaluates them sequentially. These source code expressions and definitions may contain macro definitions. The `macro:load` procedure does not affect the values returned by `current-input-port` and `current-output-port`.

2.5.1 Syntactic Closure Macro Facility

A Syntactic Closures Macro Facility
by Chris Hanson
9 November 1991

This document describes *syntactic closures*, a low-level macro facility for the Scheme programming language. The facility is an alternative to the low-level macro facility described in the *Revised^4 Report on Scheme.* This document is an addendum to that report.

The syntactic closures facility extends the BNF rule for *transformer spec* to allow a new keyword that introduces a low-level macro transformer:

transformer spec `:= (transformer` *expression*`)`

Additionally, the following procedures are added:

```
make-syntactic-closure
capture-syntactic-environment
identifier?
identifier=?
```

The description of the facility is divided into three parts. The first part defines basic terminology. The second part describes how macro transformers are defined. The third part describes the use of *identifiers*, which extend the syntactic closure mechanism to be compatible with `syntax-rules`.

2.5.1.1 Terminology

This section defines the concepts and data types used by the syntactic closures facility.

- *Forms* are the syntactic entities out of which programs are recursively constructed. A form is any expression, any definition, any syntactic keyword, or any syntactic closure. The variable name that appears in a `set!` special form is also a form. Examples of forms:

  ```
  17
  #t
  car
  (+ x 4)
  (lambda (x) x)
  (define pi 3.14159)
  if
  define
  ```
- An *alias* is an alternate name for a given symbol. It can appear anywhere in a form that the symbol could be used, and when quoted it is replaced by the symbol; however, it does not satisfy the predicate `symbol?`. Macro transformers rarely distinguish symbols from aliases, referring to both as identifiers.
- A *syntactic* environment maps identifiers to their meanings. More precisely, it determines whether an identifier is a syntactic keyword or a variable. If it is a keyword, the meaning is an interpretation for the form in which that keyword appears. If it is a variable, the meaning identifies which binding of that variable is referenced. In short, syntactic environments contain all of the contextual information necessary for interpreting the meaning of a particular form.
- A *syntactic closure* consists of a form, a syntactic environment, and a list of identifiers. All identifiers in the form take their meaning from the syntactic environment, except those in the given list. The identifiers in the list are to have their meanings determined later. A syntactic closure may be used in any context in which its form could have been used. Since a syntactic closure is also a form, it may not be used in contexts where a form would be illegal. For example, a form may not appear as a clause in the cond special form. A syntactic closure appearing in a quoted structure is replaced by its form.

2.5.1.2 Transformer Definition

This section describes the `transformer` special form and the procedures `make-syntactic-closure` and `capture-syntactic-environment`.

transformer *expression* Syntax

Syntax: It is an error if this syntax occurs except as a *transformer spec*.

Semantics: The *expression* is evaluated in the standard transformer environment to yield a macro transformer as described below. This macro transformer is bound to a macro keyword by the special form in which the `transformer` expression appears (for example, `let-syntax`).

A *macro transformer* is a procedure that takes two arguments, a form and a syntactic environment, and returns a new form. The first argument, the *input form*, is the form in which the macro keyword occurred. The second argument, the *usage environment*, is the syntactic environment in which the input form occurred. The result of the transformer, the *output form*, is automatically closed in the *transformer environment*, which is the syntactic environment in which the `transformer` expression occurred.

For example, here is a definition of a push macro using `syntax-rules`:

```
(define-syntax  push
  (syntax-rules ()
    ((push item list)
     (set! list (cons item list)))))
```

Here is an equivalent definition using `transformer`:

```
(define-syntax push
  (transformer
   (lambda (exp env)
     (let ((item
            (make-syntactic-closure env '() (cadr exp)))
           (list
            (make-syntactic-closure env '() (caddr exp))))
       `(set! ,list (cons ,item ,list))))))
```

In this example, the identifiers `set!` and `cons` are closed in the transformer environment, and thus will not be affected by the meanings of those identifiers in the usage environment `env`.

Some macros may be non-hygienic by design. For example, the following defines a loop macro that implicitly binds `exit` to an escape procedure. The binding of `exit` is intended to capture free references to `exit` in the body of the loop, so `exit` must be left free when the body is closed:

```
(define-syntax loop
  (transformer
   (lambda (exp env)
     (let ((body (cdr exp)))
       `(call-with-current-continuation
         (lambda (exit)
           (let f ()
             ,@(map (lambda  (exp)
                      (make-syntactic-closure env '(exit)
                                              exp))
                    body)
             (f))))))))
```

To assign meanings to the identifiers in a form, use `make-syntactic-closure` to close the form in a syntactic environment.

make-syntactic-closure *environment free-names form* Function

environment must be a syntactic environment, *free-names* must be a list of identifiers, and *form* must be a form. `make-syntactic-closure` constructs and returns a syntactic closure of *form* in *environment*, which can be used anywhere that *form* could have been used. All the identifiers used in *form*, except those explicitly excepted by *free-names*, obtain their meanings from *environment*.

Here is an example where *free-names* is something other than the empty list. It is instructive to compare the use of *free-names* in this example with its use in the `loop` example above: the examples are similar except for the source of the identifier being left free.

```
(define-syntax let1
  (transformer
   (lambda (exp env)
     (let ((id (cadr exp))
           (init (caddr exp))
           (exp (cadddr exp)))
       `((lambda (,id)
           ,(make-syntactic-closure env (list id) exp))
         ,(make-syntactic-closure env '() init))))))
```

`let1` is a simplified version of `let` that only binds a single identifier, and whose body consists of a single expression. When the body expression is syntactically closed in its original syntactic environment, the identifier that is to be bound by `let1` must be left free, so that it can be properly captured by the `lambda` in the output form.

To obtain a syntactic environment other than the usage environment, use `capture-syntactic-environment`.

capture-syntactic-environment *procedure* Function

`capture-syntactic-environment` returns a form that will, when transformed, call *procedure* on the current syntactic environment. *procedure* should compute and return a new form to be transformed, in that same syntactic environment, in place of the form.

An example will make this clear. Suppose we wanted to define a simple `loop-until` keyword equivalent to

```
(define-syntax loop-until
  (syntax-rules ()
    ((loop-until id init test return step)
     (letrec ((loop
               (lambda (id)
                 (if test return (loop step)))))
       (loop init)))))
```

The following attempt at defining `loop-until` has a subtle bug:

```
(define-syntax loop-until
  (transformer
   (lambda (exp env)
     (let ((id (cadr exp))
           (init (caddr exp))
           (test (cadddr exp))
           (return (cadddr (cdr exp)))
           (step (cadddr (cddr exp)))
           (close
            (lambda (exp free)
              (make-syntactic-closure env free exp))))
       `(letrec ((loop
                  (lambda (,id)
                    (if ,(close test (list id))
                        ,(close return (list id))
                        (loop ,(close step (list id)))))))
          (loop ,(close init '())))))))
```

This definition appears to take all of the proper precautions to prevent unintended captures. It carefully closes the subexpressions in their original syntactic environment and it leaves the `id` identifier free in the `test`, `return`, and `step` expressions, so that it will be captured by the binding introduced by the `lambda` expression. Unfortunately it uses the identifiers `if` and `loop` within that `lambda` expression, so if the user of `loop-until` just happens to use, say, `if` for the identifier, it will be inadvertently captured.

The syntactic environment that `if` and `loop` want to be exposed to is the one just outside the `lambda` expression: before the user's identifier is added to the syntactic environment, but after the identifier loop has been added. `capture-syntactic-environment` captures exactly that environment as follows:

```
(define-syntax loop-until
  (transformer
   (lambda (exp env)
     (let ((id (cadr exp))
           (init (caddr exp))
           (test (cadddr exp))
           (return (cadddr (cdr exp)))
           (step (cadddr (cddr exp)))
           (close
            (lambda (exp free)
              (make-syntactic-closure env free exp))))
       `(letrec ((loop
                  ,(capture-syntactic-environment
                    (lambda (env)
                      `(lambda (,id)
                         (,(make-syntactic-closure env '() `if)
                          ,(close test (list id))
                          ,(close return (list id))
                          (,(make-syntactic-closure env '()
                                                     `loop)
                           ,(close step (list id)))))))))
          (loop ,(close init '())))))))
```

In this case, having captured the desired syntactic environment, it is convenient to construct syntactic closures of the identifiers `if` and the `loop` and use them in the body of the `lambda`.

A common use of `capture-syntactic-environment` is to get the transformer environment of a macro transformer:

```
(transformer
 (lambda (exp env)
   (capture-syntactic-environment
    (lambda (transformer-env)
      ...))))
```

2.5.1.3 Identifiers

This section describes the procedures that create and manipulate identifiers. Previous syntactic closure proposals did not have an identifier data type – they just used symbols. The identifier data type extends the syntactic closures facility to be compatible with the high-level `syntax-rules` facility.

As discussed earlier, an identifier is either a symbol or an *alias*. An alias is implemented as a syntactic closure whose *form* is an identifier:

```
(make-syntactic-closure env '() 'a)
   ⇒ an alias
```

Aliases are implemented as syntactic closures because they behave just like syntactic closures most of the time. The difference is that an alias may be bound to a new value (for example by `lambda` or `let-syntax`); other syntactic closures may not be used this way. If an alias is bound, then within the scope of that binding it is looked up in the syntactic environment just like any other identifier.

Aliases are used in the implementation of the high-level facility `syntax-rules`. A macro transformer created by `syntax-rules` uses a template to generate its output form, substituting subforms of the input form into the template. In a syntactic closures implementation, all of the symbols in the template are replaced by aliases closed in the transformer environment, while the output form itself is closed in the usage environment. This guarantees that the macro transformation is hygienic, without requiring the transformer to know the syntactic roles of the substituted input subforms.

identifier? *object* Function

Returns `#t` if *object* is an identifier, otherwise returns `#f`. Examples:

```
(identifier? 'a)
   ⇒ #t
(identifier? (make-syntactic-closure env '() 'a))
   ⇒ #t
(identifier? "a")
   ⇒ #f
(identifier? #\a)
   ⇒ #f
(identifier? 97)
```

```
   ⇒ #f
(identifier? #f)
   ⇒ #f
(identifier? '(a))
   ⇒ #f
(identifier? '#(a))
   ⇒ #f
```

The predicate `eq?` is used to determine if two identifers are "the same". Thus `eq?` can be used to compare identifiers exactly as it would be used to compare symbols. Often, though, it is useful to know whether two identifiers "mean the same thing". For example, the `cond` macro uses the symbol `else` to identify the final clause in the conditional. A macro transformer for `cond` cannot just look for the symbol `else`, because the `cond` form might be the output of another macro transformer that replaced the symbol `else` with an alias. Instead the transformer must look for an identifier that "means the same thing" in the usage environment as the symbol `else` means in the transformer environment.

identifier=? *environment1 identifier1 environment2 identifier2* Function

environment1 and *environment2* must be syntactic environments, and *identifier1* and *identifier2* must be identifiers. `identifier=?` returns `#t` if the meaning of *identifier1* in *environment1* is the same as that of *identifier2* in *environment2*, otherwise it returns `#f`. Examples:

```
(let-syntax
    ((foo
      (transformer
       (lambda (form env)
         (capture-syntactic-environment
          (lambda (transformer-env)
            (identifier=? transformer-env 'x env 'x)))))))
  (list (foo)
        (let ((x 3))
          (foo))))
   ⇒ (#t #f)

(let-syntax ((bar foo))
  (let-syntax
      ((foo
        (transformer
         (lambda (form env)
           (capture-syntactic-environment
            (lambda (transformer-env)
              (identifier=? transformer-env 'foo
                            env (cadr form))))))))
    (list (foo foo)
          (foobar))))
   ⇒ (#f #t)
```

2.5.1.4 Acknowledgements

The syntactic closures facility was invented by Alan Bawden and Jonathan Rees. The use of aliases to implement `syntax-rules` was invented by Alan Bawden (who prefers to call them *synthetic names*). Much of this proposal is derived from an earlier proposal by Alan Bawden.

2.6 Syntax-Case Macros

```
(require 'syntax-case)
```

macro:expand *expression* Function
syncase:expand *expression* Function

Returns scheme code with the macros and derived expression types of *expression* expanded to primitive expression types.

macro:eval *expression* Function
syncase:eval *expression* Function

`macro:eval` returns the value of *expression* in the current top level environment. *expression* can contain macro definitions. Side effects of *expression* will affect the top level environment.

macro:load *filename* Procedure
syncase:load *filename* Procedure

filename should be a string. If filename names an existing file, the `macro:load` procedure reads Scheme source code expressions and definitions from the file and evaluates them sequentially. These source code expressions and definitions may contain macro definitions. The `macro:load` procedure does not affect the values returned by `current-input-port` and `current-output-port`.

This is version 2.1 of `syntax-case`, the low-level macro facility proposed and implemented by Robert Hieb and R. Kent Dybvig.

This version is further adapted by Harald Hanche-Olsen <hanche @ imf.unit.no> to make it compatible with, and easily usable with, SLIB. Mainly, these adaptations consisted of:

- Removing white space from '`expand.pp`' to save space in the distribution. This file is not meant for human readers anyway. . .
- Removed a couple of Chez scheme dependencies.
- Renamed global variables used to minimize the possibility of name conflicts.
- Adding an SLIB-specific initialization file.
- Removing a couple extra files, most notably the documentation (but see below).

If you wish, you can see exactly what changes were done by reading the shell script in the file '`syncase.sh`'.

The two PostScript files were omitted in order to not burden the SLIB distribution with them. If you do intend to use `syntax-case`, however, you should get these files and print them out on a PostScript printer. They are available with the original `syntax-case` distribution by anonymous FTP in '`cs.indiana.edu:/pub/scheme/syntax-case`'.

In order to use syntax-case from an interactive top level, execute:

```
(require 'syntax-case)
(require 'repl)
(repl:top-level macro:eval)
```

See the section Repl (see Section 6.5.1 [Repl], page 209) for more information.

To check operation of syntax-case get '`cs.indiana.edu:/pub/scheme/syntax-case`', and type

```
(require 'syntax-case)
(syncase:sanity-check)
```

Beware that `syntax-case` takes a long time to load – about 20s on a SPARCstation SLC (with SCM) and about 90s on a Macintosh SE/30 (with Gambit).

2.6.1 Notes

All R4RS syntactic forms are defined, including `delay`. Along with `delay` are simple definitions for `make-promise` (into which `delay` expressions expand) and `force`.

`syntax-rules` and `with-syntax` (described in *TR356*) are defined.

`syntax-case` is actually defined as a macro that expands into calls to the procedure `syntax-dispatch` and the core form `syntax-lambda`; do not redefine these names.

Several other top-level bindings not documented in TR356 are created:

- the "hooks" in '`hooks.ss`'
- the `build-` procedures in '`output.ss`'
- `expand-syntax` (the expander)

The syntax of define has been extended to allow `(define` *id*`)`, which assigns *id* to some unspecified value.

We have attempted to maintain R4RS compatibility where possible. The incompatibilities should be confined to '`hooks.ss`'. Please let us know if there is some incompatibility that is not flagged as such.

Send bug reports, comments, suggestions, and questions to Kent Dybvig (dyb `@` iuvax.cs.indiana.edu).

2.6.2 Note from maintainer

Included with the `syntax-case` files was '`structure.scm`' which defines a macro `define-structure`. There is no documentation for this macro and it is not used by any code in SLIB.

2.7 Fluid-Let

```
(require 'fluid-let)
```

fluid-let (*bindings* ...) *forms*... Syntax

```
(fluid-let ((variable init) ...)
   expression expression ...)
```

The *init*s are evaluated in the current environment (in some unspecified order), the current values of the *variable*s are saved, the results are assigned to the *variable*s, the *expression*s are evaluated sequentially in the current environment, the *variable*s are restored to their original values, and the value of the last *expression* is returned.

The syntax of this special form is similar to that of `let`, but `fluid-let` temporarily rebinds existing *variable*s. Unlike `let`, `fluid-let` creates no new bindings; instead it *assigns* the values of each *init* to the binding (determined by the rules of lexical scoping) of its corresponding *variable*.

2.8 Yasos

`(require 'oop)` or `(require 'yasos)`

'Yet Another Scheme Object System' is a simple object system for Scheme based on the paper by Norman Adams and Jonathan Rees: *Object Oriented Programming in Scheme*, Proceedings of the 1988 ACM Conference on LISP and Functional Programming, July 1988 [ACM #552880].

Another reference is:

Ken Dickey. Scheming with Objects *AI Expert* Volume 7, Number 10 (October 1992), pp. 24-33.

2.8.1 Terms

Object	Any Scheme data object.
Instance	An instance of the OO system; an *object*.
Operation	A *method*.
Notes:	The object system supports multiple inheritance. An instance can inherit from 0 or more ancestors. In the case of multiple inherited operations with the same identity, the operation used is that from the first ancestor which contains it (in the ancestor `let`). An operation may be applied to any Scheme data object—not just instances. As code which creates instances is just code, there are no *classes* and no meta-*anything*. Method dispatch is by a procedure call a la CLOS rather than by `send` syntax a la Smalltalk.

Disclaimer:

There are a number of optimizations which can be made. This implementation is expository (although performance should be quite reasonable). See the L&FP paper for some suggestions.

2.8.2 Interface

define-operation (*opname self arg ...*) *default-body* Syntax

Defines a default behavior for data objects which don't handle the operation *opname*. The default behavior (for an empty *default-body*) is to generate an error.

define-predicate *opname?* Syntax

Defines a predicate *opname?*, usually used for determining the *type* of an object, such that `(`*opname? object*`)` returns `#t` if *object* has an operation *opname?* and `#f` otherwise.

object ((*name self arg* ...) *body*) ... Syntax

Returns an object (an instance of the object system) with operations. Invoking `(`*name object arg* ... executes the *body* of the *object* with *self* bound to *object* and with argument(s) *arg*....

object-with-ancestors ((*ancestor1 init1*) ...) *operation* ... Syntax

A `let`-like form of `object` for multiple inheritance. It returns an object inheriting the behaviour of *ancestor1* etc. An operation will be invoked in an ancestor if the object itself does not provide such a method. In the case of multiple inherited operations with the same identity, the operation used is the one found in the first ancestor in the ancestor list.

operate-as *component operation self arg ...* Syntax

Used in an operation definition (of *self*) to invoke the *operation* in an ancestor *component* but maintain the object's identity. Also known as "send-to-super".

print *obj port* Procedure

A default `print` operation is provided which is just `(format` *port obj*`)` (see Section 3.2 [Format], page 39) for non-instances and prints *obj* preceded by '`#<INSTANCE>`' for instances.

size *obj* Function

The default method returns the number of elements in *obj* if it is a vector, string or list, `2` for a pair, `1` for a character and by default id an error otherwise. Objects such as collections (see Section 6.1.8 [Collections], page 163) may override the default in an obvious way.

2.8.3 Setters

Setters implement *generalized locations* for objects associated with some sort of mutable state. A *getter* operation retrieves a value from a generalized location and the corresponding setter operation stores a value into the location. Only the getter is named – the setter is specified by a procedure call as below. (Dylan uses special syntax.) Typically, but not necessarily, getters are access operations to extract values from Yasos objects (see Section 2.8 [Yasos], page 28). Several setters are predefined, corresponding to getters `car`, `cdr`, `string-ref` and `vector-ref` e.g., `(setter car)` is equivalent to `set-car!`.

This implementation of setters is similar to that in Dylan(TM) (*Dylan: An object-oriented dynamic language*, Apple Computer Eastern Research and Technology). Common LISP provides similar facilities through `setf`.

setter *getter* Function

Returns the setter for the procedure *getter*. E.g., since `string-ref` is the getter corresponding to a setter which is actually `string-set!`:

```
(define foo "foo")
((setter string-ref) foo 0 #\F) ; set element 0 of foo
foo ⇒ "Foo"
```

set *place new-value* Syntax

If *place* is a variable name, `set` is equivalent to `set!`. Otherwise, *place* must have the form of a procedure call, where the procedure name refers to a getter and the call indicates an accessible generalized location, i.e., the call would return a value. The return value of `set` is usually unspecified unless used with a setter whose definition guarantees to return a useful value.

```
(set (string-ref foo 2) #\O)  ; generalized location with getter
foo ⇒ "FoO"
(set foo "foo")               ; like set!
foo ⇒ "foo"
```

add-setter *getter setter* Procedure

Add procedures *getter* and *setter* to the (inaccessible) list of valid setter/getter pairs. *setter* implements the store operation corresponding to the *getter* access operation for the relevant state. The return value is unspecified.

remove-setter-for *getter* Procedure

Removes the setter corresponding to the specified *getter* from the list of valid setters. The return value is unspecified.

define-access-operation *getter-name* Syntax

Shorthand for a Yasos `define-operation` defining an operation *getter-name* that objects may support to return the value of some mutable state. The default operation is to signal an error. The return value is unspecified.

2.8.4 Examples

```
;;; These definitions for PRINT and SIZE are
;;; already supplied by
(require 'yasos)

(define-operation (print obj port)
  (format port
          (if (instance? obj) "#<instance>" "~s")
          obj))

(define-operation (size obj)
  (cond
   ((vector? obj) (vector-length obj))
   ((list?   obj) (length obj))
   ((pair?   obj) 2)
   ((string? obj) (string-length obj))
   ((char?   obj) 1)
   (else
    (error "Operation not supported: size" obj))))

(define-predicate cell?)
(define-operation (fetch obj))
(define-operation (store! obj newValue))

(define (make-cell value)
  (object
   ((cell? self) #t)
   ((fetch self) value)
   ((store! self newValue)
    (set! value newValue)
    newValue)
   ((size self) 1)
   ((print self port)
    (format port "#<Cell: ~s>" (fetch self)))))

(define-operation (discard obj value)
  (format #t "Discarding ~s~%" value))

(define (make-filtered-cell value filter)
  (object-with-ancestors
   ((cell (make-cell value)))
   ((store! self newValue)
   (if (filter newValue)
       (store! cell newValue)
       (discard self newValue)))))

(define-predicate array?)
```

```
(define-operation (array-ref array index))
(define-operation (array-set! array index value))

(define (make-array num-slots)
  (let ((anArray (make-vector num-slots)))
    (object
     ((array? self) #t)
     ((size self) num-slots)
     ((array-ref self index)
      (vector-ref  anArray index))
     ((array-set! self index newValue)
      (vector-set! anArray index newValue))
     ((print self port)
      (format port "#<Array ~s>" (size self))))))

(define-operation (position obj))
(define-operation (discarded-value obj))

(define (make-cell-with-history value filter size)
  (let ((pos 0) (most-recent-discard #f))
    (object-with-ancestors
     ((cell (make-filtered-call value filter))
      (sequence (make-array size)))
     ((array? self) #f)
     ((position self) pos)
     ((store! self newValue)
      (operate-as cell store! self newValue)
      (array-set! self pos newValue)
      (set! pos (+ pos 1)))
     ((discard self value)
      (set! most-recent-discard value))
     ((discarded-value self) most-recent-discard)
     ((print self port)
      (format port "#<Cell-with-history ~s>"
              (fetch self))))))

(define-access-operation fetch)
(add-setter fetch store!)
(define foo (make-cell 1))
(print foo #f)
⇒ "#<Cell: 1>"
(set (fetch foo) 2)
⇒
(print foo #f)
⇒ "#<Cell: 2>"
(fetch foo)
⇒ 2
```

3 Textual Conversion Packages

3.1 Precedence Parsing

`(require 'precedence-parse)` or `(require 'parse)`

This package implements:

- a Pratt style precedence parser;
- a *tokenizer* which congeals tokens according to assigned classes of constituent characters;
- procedures giving direct control of parser rulesets;
- procedures for higher level specification of rulesets.

3.1.1 Precedence Parsing Overview

This package offers improvements over previous parsers.

- Common computer language constructs are concisely specified.
- Grammars can be changed dynamically. Operators can be assigned different meanings within a lexical context.
- Rulesets don't need compilation. Grammars can be changed incrementally.
- Operator precedence is specified by integers.
- All possibilities of bad input are handled[1] and return as much structure as was parsed when the error occured; The symbol `?` is substituted for missing input.

Here are the higher-level syntax types and an example of each. Precedence considerations are omitted for clarity. See Section 3.1.5 [Grammar Rule Definition], page 37 for full details.

nofix bye exit Grammar

```
bye
```

calls the function `exit` with no arguments.

prefix - negate Grammar

```
- 42
```

Calls the function `negate` with the argument `42`.

infix - difference Grammar

```
x - y
```

Calls the function `difference` with arguments `x` and `y`.

[1] How do I know this? I parsed 250kbyte of random input (an e-mail file) with a non-trivial grammar utilizing all constructs.

nary + sum Grammar

```
x + y + z
```

Calls the function `sum` with arguments `x`, `y`, and `y`.

postfix ! factorial Grammar

```
5 !
```

Calls the function `factorial` with the argument `5`.

prestfix set set! Grammar

```
set foo bar
```

Calls the function `set!` with the arguments `foo` and `bar`.

commentfix /* */ Grammar

```
/* almost any text here */
```

Ignores the comment delimited by `/*` and `*/`.

matchfix { list } Grammar

```
{0, 1, 2}
```

Calls the function `list` with the arguments `0`, `1`, and `2`.

inmatchfix (funcall) Grammar

```
f(x, y)
```

Calls the function `funcall` with the arguments `f`, `x`, and `y`.

delim ; Grammar

```
set foo bar;
```

delimits the extent of the restfix operator `set`.

3.1.2 Ruleset Definition and Use

syn-defs Variable

A grammar is built by one or more calls to `prec:define-grammar`. The rules are appended to **syn-defs**. The value of **syn-defs** is the grammar suitable for passing as an argument to `prec:parse`.

syn-ignore-whitespace Constant

Is a nearly empty grammar with whitespace characters set to group 0, which means they will not be made into tokens. Most rulesets will want to start with `*syn-ignore-whitespace*`

In order to start defining a grammar, either

```
(set! *syn-defs* '())
```

or

```
(set! *syn-defs* *syn-ignore-whitespace*)
```

prec:define-grammar *rule1 ...* Function

Appends *rule1 ...* to **syn-defs**. `prec:define-grammar` is used to define both the character classes and rules for tokens.

Once your grammar is defined, save the value of `*syn-defs*` in a variable (for use when calling `prec:parse`).

```
(define my-ruleset *syn-defs*)
```

prec:parse *ruleset delim* Function
prec:parse *ruleset delim port* Function

The *ruleset* argument must be a list of rules as constructed by `prec:define-grammar` and extracted from **syn-defs**.

The token *delim* may be a character, symbol, or string. A character *delim* argument will match only a character token; i.e. a character for which no token-group is assigned. A symbols or string will match only a token string; i.e. a token resulting from a token group.

`prec:parse` reads a *ruleset* grammar expression delimited by *delim* from the given input *port*. `prec:parse` returns the next object parsable from the given input *port*, updating *port* to point to the first character past the end of the external representation of the object.

If an end of file is encountered in the input before any characters are found that can begin an object, then an end of file object is returned. If a delimiter (such as *delim*) is found before any characters are found that can begin an object, then `#f` is returned.

The *port* argument may be omitted, in which case it defaults to the value returned by `current-input-port`. It is an error to parse from a closed port.

3.1.3 Token definition

tok:char-group *group chars chars-proc* Function

The argument *chars* may be a single character, a list of characters, or a string. Each character in *chars* is treated as though `tok:char-group` was called with that character alone.

The argument *chars-proc* must be a procedure of one argument, a list of characters. After `tokenize` has finished accumulating the characters for a token, it calls *chars-proc* with the list of characters. The value returned is the token which `tokenize` returns.

The argument *group* may be an exact integer or a procedure of one character argument. The following discussion concerns the treatment which the tokenizing routine, `tokenize`, will accord to characters on the basis of their groups.

When *group* is a non-zero integer, characters whose group number is equal to or exactly one less than *group* will continue to accumulate. Any other character causes the accumulation to stop (until a new token is to be read).

The *group* of zero is special. These characters are ignored when parsed pending a token, and stop the accumulation of token characters when the accumulation has already begun. Whitespace characters are usually put in group 0.

If *group* is a procedure, then, when triggerd by the occurence of an initial (no accumulation) *chars* character, this procedure will be repeatedly called with each successive character from the input stream until the *group* procedure returns a non-false value.

The following convenient constants are provided for use with `tok:char-group`.

tok:decimal-digits Constant

Is the string `"0123456789"`.

tok:upper-case Constant

Is the string consisting of all upper-case letters ("ABCDEFGHIJKLMNOPQRSTUVWXYZ").

tok:lower-case Constant

Is the string consisting of all lower-case letters ("abcdefghijklmnopqrstuvwxyz").

tok:whitespaces Constant

Is the string consisting of all characters between 0 and 255 for which `char-whitespace?` returns true.

3.1.4 Nud and Led Definition

This section describes advanced features. You can skip this section on first reading.

The *Null Denotation* (or *nud*) of a token is the procedure and arguments applying for that token when *Left*, an unclaimed parsed expression is not extant.

The *Left Denotation* (or *led*) of a token is the procedure, arguments, and lbp applying for that token when there is a *Left*, an unclaimed parsed expression.

In his paper,

> Pratt, V. R. Top Down Operator Precendence. *SIGACT/SIGPLAN Symposium on Principles of Programming Languages*, Boston, 1973, pages 41-51

the *left binding power* (or *lbp*) was an independent property of tokens. I think this was done in order to allow tokens with NUDs but not LEDs to also be used as delimiters, which was a problem for statically defined syntaxes. It turns out that *dynamically binding* NUDs and LEDs allows them independence.

For the rule-defining procedures that follow, the variable *tk* may be a character, string, or symbol, or a list composed of characters, strings, and symbols. Each element of *tk* is treated as though the procedure were called for each element.

Character *tk* arguments will match only character tokens; i.e. characters for which no token-group is assigned. Symbols and strings will both match token strings; i.e. tokens resulting from token groups.

prec:make-nud *tk sop arg1 . . .* Function

Returns a rule specifying that *sop* be called when *tk* is parsed. If *sop* is a procedure, it is called with *tk* and *arg1 . . .* as its arguments; the resulting value is incorporated into the expression being built. Otherwise, `(list` *sop arg1* `...)` is incorporated.

If no NUD has been defined for a token; then if that token is a string, it is converted to a symbol and returned; if not a string, the token is returned.

prec:make-led *tk sop arg1 . . .* Function

Returns a rule specifying that *sop* be called when *tk* is parsed and *left* has an unclaimed parsed expression. If *sop* is a procedure, it is called with *left*, *tk*, and *arg1 . . .* as its arguments; the resulting value is incorporated into the expression being built. Otherwise, *left* is incorporated.

If no LED has been defined for a token, and *left* is set, the parser issues a warning.

3.1.5 Grammar Rule Definition

Here are procedures for defining rules for the syntax types introduced in Section 3.1.1 [Precedence Parsing Overview], page 33.

For the rule-defining procedures that follow, the variable *tk* may be a character, string, or symbol, or a list composed of characters, strings, and symbols. Each element of *tk* is treated as though the procedure were called for each element.

For procedures prec:delim, . . ., prec:prestfix, if the *sop* argument is `#f`, then the token which triggered this rule is converted to a symbol and returned. A false *sop* argument to the procedures prec:commentfix, prec:matchfix, or prec:inmatchfix has a different meaning.

Character *tk* arguments will match only character tokens; i.e. characters for which no token-group is assigned. Symbols and strings will both match token strings; i.e. tokens resulting from token groups.

prec:delim *tk* Function

Returns a rule specifying that *tk* should not be returned from parsing; i.e. *tk*'s function is purely syntactic. The end-of-file is always treated as a delimiter.

prec:nofix *tk sop* Function

Returns a rule specifying the following actions take place when *tk* is parsed:

- If *sop* is a procedure, it is called with no arguments; the resulting value is incorporated into the expression being built. Otherwise, the list of *sop* is incorporated.

prec:prefix *tk sop bp rule1 . . .* Function

Returns a rule specifying the following actions take place when *tk* is parsed:

- The rules *rule1* . . . augment and, in case of conflict, override rules currently in effect.
- `prec:parse1` is called with binding-power *bp*.
- If *sop* is a procedure, it is called with the expression returned from `prec:parse1`; the resulting value is incorporated into the expression being built. Otherwise, the list of *sop* and the expression returned from `prec:parse1` is incorporated.
- The ruleset in effect before *tk* was parsed is restored; *rule1* . . . are forgotten.

prec:infix *tk sop lbp bp rule1 . . .* Function

Returns a rule declaring the left-binding-precedence of the token *tk* is *lbp* and specifying the following actions take place when *tk* is parsed:

- The rules *rule1* . . . augment and, in case of conflict, override rules currently in effect.
- One expression is parsed with binding-power *lbp*. If instead a delimiter is encountered, a warning is issued.
- If *sop* is a procedure, it is applied to the list of *left* and the parsed expression; the resulting value is incorporated into the expression being built. Otherwise, the list of *sop*, the *left* expression, and the parsed expression is incorporated.
- The ruleset in effect before *tk* was parsed is restored; *rule1* . . . are forgotten.

prec:nary *tk sop bp* Function

Returns a rule declaring the left-binding-precedence of the token *tk* is *bp* and specifying the following actions take place when *tk* is parsed:

- Expressions are parsed with binding-power *bp* as far as they are interleaved with the token *tk*.
- If *sop* is a procedure, it is applied to the list of *left* and the parsed expressions; the resulting value is incorporated into the expression being built. Otherwise, the list of *sop*, the *left* expression, and the parsed expressions is incorporated.

prec:postfix *tk sop lbp* Function

Returns a rule declaring the left-binding-precedence of the token *tk* is *lbp* and specifying the following actions take place when *tk* is parsed:

- If *sop* is a procedure, it is called with the *left* expression; the resulting value is incorporated into the expression being built. Otherwise, the list of *sop* and the *left* expression is incorporated.

prec:prestfix *tk sop bp rule1 . . .* Function

Returns a rule specifying the following actions take place when *tk* is parsed:

- The rules *rule1* . . . augment and, in case of conflict, override rules currently in effect.
- Expressions are parsed with binding-power *bp* until a delimiter is reached.

- If *sop* is a procedure, it is applied to the list of parsed expressions; the resulting value is incorporated into the expression being built. Otherwise, the list of *sop* and the parsed expressions is incorporated.
- The ruleset in effect before *tk* was parsed is restored; *rule1 ...* are forgotten.

prec:commentfix *tk stp match rule1 ...* Function

Returns rules specifying the following actions take place when *tk* is parsed:

- The rules *rule1 ...* augment and, in case of conflict, override rules currently in effect.
- Characters are read until and end-of-file or a sequence of characters is read which matches the *string match*.
- If *stp* is a procedure, it is called with the string of all that was read between the *tk* and *match* (exclusive).
- The ruleset in effect before *tk* was parsed is restored; *rule1 ...* are forgotten.

Parsing of commentfix syntax differs from the others in several ways. It reads directly from input without tokenizing; It calls *stp* but does not return its value; nay any value. I added the *stp* argument so that comment text could be echoed.

prec:matchfix *tk sop sep match rule1 ...* Function

Returns a rule specifying the following actions take place when *tk* is parsed:

- The rules *rule1 ...* augment and, in case of conflict, override rules currently in effect.
- A rule declaring the token *match* a delimiter takes effect.
- Expressions are parsed with binding-power `0` until the token *match* is reached. If the token *sep* does not appear between each pair of expressions parsed, a warning is issued.
- If *sop* is a procedure, it is applied to the list of parsed expressions; the resulting value is incorporated into the expression being built. Otherwise, the list of *sop* and the parsed expressions is incorporated.
- The ruleset in effect before *tk* was parsed is restored; *rule1 ...* are forgotten.

prec:inmatchfix *tk sop sep match lbp rule1 ...* Function

Returns a rule declaring the left-binding-precedence of the token *tk* is *lbp* and specifying the following actions take place when *tk* is parsed:

- The rules *rule1 ...* augment and, in case of conflict, override rules currently in effect.
- A rule declaring the token *match* a delimiter takes effect.
- Expressions are parsed with binding-power `0` until the token *match* is reached. If the token *sep* does not appear between each pair of expressions parsed, a warning is issued.
- If *sop* is a procedure, it is applied to the list of *left* and the parsed expressions; the resulting value is incorporated into the expression being built. Otherwise, the list of *sop*, the *left* expression, and the parsed expressions is incorporated.

- The ruleset in effect before *tk* was parsed is restored; *rule1* ... are forgotten.

3.2 Format (version 3.0)

```
(require 'format)
```

3.2.1 Format Interface

format *destination format-string . arguments* Function

An almost complete implementation of Common LISP format description according to the CL reference book *Common LISP* from Guy L. Steele, Digital Press. Backward compatible to most of the available Scheme format implementations.

Returns `#t`, `#f` or a string; has side effect of printing according to *format-string*. If *destination* is `#t`, the output is to the current output port and `#t` is returned. If *destination* is `#f`, a formatted string is returned as the result of the call. NEW: If *destination* is a string, *destination* is regarded as the format string; *format-string* is then the first argument and the output is returned as a string. If *destination* is a number, the output is to the current error port if available by the implementation. Otherwise *destination* must be an output port and `#t` is returned.

format-string must be a string. In case of a formatting error format returns `#f` and prints a message on the current output or error port. Characters are output as if the string were output by the `display` function with the exception of those prefixed by a tilde (`~`). For a detailed description of the *format-string* syntax please consult a Common LISP format reference manual. For a test suite to verify this format implementation load '`formatst.scm`'. Please send bug reports to `lutzeb@cs.tu-berlin.de`.

Note: `format` is not reentrant, i.e. only one `format`-call may be executed at a time.

3.2.2 Format Specification (Format version 3.0)

Please consult a Common LISP format reference manual for a detailed description of the format string syntax. For a demonstration of the implemented directives see '`formatst.scm`'.

This implementation supports directive parameters and modifiers (`:` and `@` characters). Multiple parameters must be separated by a comma (`,`). Parameters can be numerical parameters (positive or negative), character parameters (prefixed by a quote character (`'`), variable parameters (`v`), number of rest arguments parameter (`#`), empty and default parameters. Directive characters are case independent. The general form of a directive is:

directive ::= `~{`*directive-parameter*,`}`[:][`@`]*directive-character*

directive-parameter ::= [[-`|`+]`{`0-9`}+` `|` '*character* `|` v `|` #]

3.2.2.1 Implemented CL Format Control Directives

Documentation syntax: Uppercase characters represent the corresponding control directive characters. Lowercase characters represent control directive parameter descriptions.

`~A` Any (print as `display` does).

- `~@A` left pad.
- `~`*mincol*`,`*colinc*`,`*minpad*`,`*padchar*`A` full padding.

`~S` S-expression (print as `write` does).

- `~@S` left pad.
- `~`*mincol*`,`*colinc*`,`*minpad*`,`*padchar*`S` full padding.

`~D` Decimal.

- `~@D` print number sign always.
- `~:D` print comma separated.
- `~`*mincol*`,`*padchar*`,`*commachar*`D` padding.

`~X` Hexadecimal.

- `~@X` print number sign always.
- `~:X` print comma separated.
- `~`*mincol*`,`*padchar*`,`*commachar*`X` padding.

`~O` Octal.

- `~@O` print number sign always.
- `~:O` print comma separated.
- `~`*mincol*`,`*padchar*`,`*commachar*`O` padding.

`~B` Binary.

- `~@B` print number sign always.
- `~:B` print comma separated.
- `~`*mincol*`,`*padchar*`,`*commachar*`B` padding.

`~`*n*`R` Radix *n*.

- `~`*n*`,`*mincol*`,`*padchar*`,`*commachar*`R` padding.

`~@R` print a number as a Roman numeral.

`~:@R` print a number as an “old fashioned” Roman numeral.

`~:R` print a number as an ordinal English number.

`~R` print a number as a cardinal English number.

`~P` Plural.

- `~@P` prints `y` and `ies`.
- `~:P` as `~P but jumps 1 argument backward.`
- `~:@P` as `~@P but jumps 1 argument backward.`

`~C` Character.

- `~@C` prints a character as the reader can understand it (i.e. `#\` prefixing).
- `~:C` prints a character as emacs does (eg. `^C` for ASCII 03).

`~F` Fixed-format floating-point (prints a flonum like *mmm.nnn*).

`~`*width*`,`*digits*`,`*scale*`,`*overflowchar*`,`*padchar*`F`

- `~@F` If the number is positive a plus sign is printed.

`~E` Exponential floating-point (prints a flonum like *mmm.nnn*`E`*ee*).

`~`*width*`,`*digits*`,`*exponentdigits*`,`*scale*`,`*overflowchar*`,`*padchar*`,`*exponentchar*`E`

- `~@E` If the number is positive a plus sign is printed.

`~G` General floating-point (prints a flonum either fixed or exponential).

`~`*width*`,`*digits*`,`*exponentdigits*`,`*scale*`,`*overflowchar*`,`*padchar*`,`*exponentchar*`G`

- `~@G` If the number is positive a plus sign is printed.

`~$` Dollars floating-point (prints a flonum in fixed with signs separated).

`~`*digits*`,`*scale*`,`*width*`,`*padchar*`$`

- `~@$` If the number is positive a plus sign is printed.
- `~:@$` A sign is always printed and appears before the padding.
- `~:$` The sign appears before the padding.

`~%` Newline.

- `~`*n*`%` print *n* newlines.

`~&` print newline if not at the beginning of the output line.

- `~`*n*`&` prints `~&` and then *n-1* newlines.

`~|` Page Separator.

- `~`*n*`|` print *n* page separators.

`~~` Tilde.

- `~`*n*`~` print *n* tildes.

`~`<newline>

Continuation Line.

	`~:<newline>`	newline is ignored, white space left.
	`~@<newline>`	newline is left, white space ignored.
`~T`	Tabulation.	
	`~@T`	relative tabulation.
	`~`*colnum,colinc*`T`	full tabulation.
`~?`	Indirection (expects indirect arguments as a list).	
	`~@?`	extracts indirect arguments from format arguments.
`~(`*str*`~)`	Case conversion (converts by `string-downcase`).	
	`~:(`*str*`~)`	converts by `string-capitalize`.
	`~@(`*str*`~)`	converts by `string-capitalize-first`.
	`~:@(`*str*`~)`	converts by `string-upcase`.
`~*`	Argument Jumping (jumps 1 argument forward).	
	`~`*n*`*`	jumps *n* arguments forward.
	`~:*`	jumps 1 argument backward.
	`~`*n*`:*`	jumps *n* arguments backward.
	`~@*`	jumps to the 0th argument.
	`~`*n*`@*`	jumps to the *n*th argument (beginning from 0)
`~[`*str0*`~;`*str1*`~;...~;`*strn*`~]`	Conditional Expression (numerical clause conditional).	
	`~`*n*`[`	take argument from *n*.
	`~@[`	true test conditional.
	`~:[`	if-else-then conditional.
	`~;`	clause separator.
	`~:;`	default clause follows.
`~{`*str*`~}`	Iteration (args come from the next argument (a list)).	
	`~`*n*`{`	at most *n* iterations.
	`~:{`	args from next arg (a list of lists).
	`~@{`	args from the rest of arguments.
	`~:@{`	args from the rest args (lists).
`~^`	Up and out.	
	`~`*n*`^`	aborts if $n = 0$
	`~`*n*`,`*m*`^`	aborts if $n = m$
	`~`*n*`,`*m*`,`*k*`^`	aborts if $n <= m <= k$

3.2.2.2 Not Implemented CL Format Control Directives

`~:A` print `#f` as an empty list (see below).

`~:S` print `#f` as an empty list (see below).

`~<~>` Justification.

`~:^` (sorry I don't understand its semantics completely)

3.2.2.3 Extended, Replaced and Additional Control Directives

`~`*mincol*`,`*padchar*`,`*commachar*`,`*commawidth*`D`
`~`*mincol*`,`*padchar*`,`*commachar*`,`*commawidth*`X`
`~`*mincol*`,`*padchar*`,`*commachar*`,`*commawidth*`O`
`~`*mincol*`,`*padchar*`,`*commachar*`,`*commawidth*`B`
`~`*n*`,`*mincol*`,`*padchar*`,`*commachar*`,`*commawidth*`R`
commawidth is the number of characters between two comma characters.

`~I` print a R4RS complex number as `~F~@Fi` with passed parameters for `~F`.

`~Y` Pretty print formatting of an argument for scheme code lists.

`~K` Same as `~?`.

`~!` Flushes the output if format *destination* is a port.

`~_` Print a `#\space` character

  `~`*n*`_` print *n* `#\space` characters.

`~/` Print a `#\tab` character

  `~`*n*`/` print *n* `#\tab` characters.

`~`*n*`C` Takes *n* as an integer representation for a character. No arguments are consumed. *n* is converted to a character by `integer->char`. *n* must be a positive decimal number.

`~:S` Print out readproof. Prints out internal objects represented as `#<...>` as strings `"#<...>"` so that the format output can always be processed by `read`.

`~:A` Print out readproof. Prints out internal objects represented as `#<...>` as strings `"#<...>"` so that the format output can always be processed by `read`.

`~Q` Prints information and a copyright notice on the format implementation.

  `~:Q` prints format version.

`~F, ~E, ~G, ~$`
may also print number strings, i.e. passing a number as a string and format it accordingly.

3.2.2.4 Configuration Variables

Format has some configuration variables at the beginning of '`format.scm`' to suit the systems and users needs. There should be no modification necessary for the configuration that comes with SLIB. If modification is desired the variable should be set after the format code is loaded. Format detects automatically if the running scheme system implements floating point numbers and complex numbers.

format:symbol-case-conv
: Symbols are converted by `symbol->string` so the case type of the printed symbols is implementation dependent. `format:symbol-case-conv` is a one arg closure which is either `#f` (no conversion), `string-upcase`, `string-downcase` or `string-capitalize`. (default `#f`)

format:iobj-case-conv
: As *format:symbol-case-conv* but applies for the representation of implementation internal objects. (default `#f`)

format:expch
: The character prefixing the exponent value in `~E` printing. (default `#\E`)

3.2.2.5 Compatibility With Other Format Implementations

SLIB format 2.x:
: See '`format.doc`'.

SLIB format 1.4:
: Downward compatible except for padding support and `~A`, `~S`, `~P`, `~X` uppercase printing. SLIB format 1.4 uses C-style `printf` padding support which is completely replaced by the CL `format` padding style.

MIT C-Scheme 7.1:
: Downward compatible except for `~`, which is not documented (ignores all characters inside the format string up to a newline character). (7.1 implements `~a`, `~s`, `~`*newline*, `~~`, `~%`, numerical and variable parameters and `:/@` modifiers in the CL sense).

Elk 1.5/2.0:
: Downward compatible except for `~A` and `~S` which print in uppercase. (Elk implements `~a`, `~s`, `~~`, and `~%` (no directive parameters or modifiers)).

Scheme->C 01nov91:
: Downward compatible except for an optional destination parameter: S2C accepts a format call without a destination which returns a formatted string. This is equivalent to a #f destination in S2C. (S2C implements `~a`, `~s`, `~c`, `~%`, and `~~` (no directive parameters or modifiers)).

This implementation of format is solely useful in the SLIB context because it requires other components provided by SLIB.

3.3 Standard Formatted I/O

3.3.1 stdio

```
(require 'stdio)
```

`requires` `printf` and `scanf` and additionally defines the symbols:

stdin Variable

Defined to be `(current-input-port)`.

stdout Variable

Defined to be `(current-output-port)`.

stderr Variable

Defined to be `(current-error-port)`.

3.3.2 Standard Formatted Output

```
(require 'printf)
```

printf *format arg1 . . .* Procedure
fprintf *port format arg1 . . .* Procedure
sprintf *str format arg1 . . .* Procedure
sprintf *#f format arg1 . . .* Procedure
sprintf *k format arg1 . . .* Procedure

Each function converts, formats, and outputs its *arg1* . . . arguments according to the control string *format* argument and returns the number of characters output.

`printf` sends its output to the port `(current-output-port)`. `fprintf` sends its output to the port *port*. `sprintf` `string-set!`s locations of the non-constant string argument *str* to the output characters.

Two extensions of `sprintf` return new strings. If the first argument is `#f`, then the returned string's length is as many characters as specified by the *format* and data; if the first argument is a non-negative integer *k*, then the length of the returned string is also bounded by *k*.

The string *format* contains plain characters which are copied to the output stream, and conversion specifications, each of which results in fetching zero or more of the arguments *arg1* . . . . The results are undefined if there are an insufficient number of arguments for the format. If *format* is exhausted while some of the *arg1* . . . arguments remain unused, the excess *arg1* . . . arguments are ignored.

The conversion specifications in a format string have the form:

`%` [*flags*] [*width*] [. *precision*] [*type*] *conversion*

An output conversion specifications consist of an initial '`%`' character followed in sequence by:

- Zero or more *flag characters* that modify the normal behavior of the conversion specification.

  '`-`' Left-justify the result in the field. Normally the result is right-justified.

  '`+`' For the signed '`%d`' and '`%i`' conversions and all inexact conversions, prefix a plus sign if the value is positive.

  '` `' For the signed '`%d`' and '`%i`' conversions, if the result doesn't start with a plus or minus sign, prefix it with a space character instead. Since the '`+`' flag ensures that the result includes a sign, this flag is ignored if both are specified.

  '`#`' For inexact conversions, '`#`' specifies that the result should always include a decimal point, even if no digits follow it. For the '`%g`' and '`%G`' conversions, this also forces trailing zeros after the decimal point to be printed where they would otherwise be elided.

  For the '`%o`' conversion, force the leading digit to be '`0`', as if by increasing the precision. For '`%x`' or '`%X`', prefix a leading '`0x`' or '`0X`' (respectively) to the result. This doesn't do anything useful for the '`%d`', '`%i`', or '`%u`' conversions. Using this flag produces output which can be parsed by the `scanf` functions with the '`%i`' conversion (see Section 3.3.3 [Standard Formatted Input], page 49).

  '`0`' Pad the field with zeros instead of spaces. The zeros are placed after any indication of sign or base. This flag is ignored if the '`-`' flag is also specified, or if a precision is specified for an exact converson.

- An optional decimal integer specifying the *minimum field width*. If the normal conversion produces fewer characters than this, the field is padded (with spaces or zeros per the '`0`' flag) to the specified width. This is a *minimum* width; if the normal conversion produces more characters than this, the field is *not* truncated.

  Alternatively, if the field width is '`*`', the next argument in the argument list (before the actual value to be printed) is used as the field width. The width value must be an integer. If the value is negative it is as though the '`-`' flag is set (see above) and the absolute value is used as the field width.

- An optional *precision* to specify the number of digits to be written for numeric conversions and the maximum field width for string conversions. The precision is specified by a period ('`.`') followed optionally by a decimal integer (which defaults to zero if omitted).

  Alternatively, if the precision is '`.*`', the next argument in the argument list (before the actual value to be printed) is used as the precision. The value must be an integer, and is ignored if negative. If you specify '`*`' for both the field width and precision, the field width argument precedes the precision argument. The '`.*`' precision is an enhancement. C library versions may not accept this syntax.

For the '`%f`', '`%e`', and '`%E`' conversions, the precision specifies how many digits follow the decimal-point character. The default precision is `6`. If the precision is explicitly `0`, the decimal point character is suppressed.

For the '`%g`' and '`%G`' conversions, the precision specifies how many significant digits to print. Significant digits are the first digit before the decimal point, and all the digits after it. If the precision is `0` or not specified for '`%g`' or '`%G`', it is treated like a value of `1`. If the value being printed cannot be expressed accurately in the specified number of digits, the value is rounded to the nearest number that fits.

For exact conversions, if a precision is supplied it specifies the minimum number of digits to appear; leading zeros are produced if necessary. If a precision is not supplied, the number is printed with as many digits as necessary. Converting an exact '`0`' with an explicit precision of zero produces no characters.

- An optional one of '`l`', '`h`' or '`L`', which is ignored for numeric conversions. It is an error to specify these modifiers for non-numeric conversions.
- A character that specifies the conversion to be applied.

3.3.2.1 Exact Conversions

'`b`', '`B`'
: Print an integer as an unsigned binary number.

  Note: '`%b`' and '`%B`' are SLIB extensions.

'`d`', '`i`'
: Print an integer as a signed decimal number. '`%d`' and '`%i`' are synonymous for output, but are different when used with `scanf` for input (see Section 3.3.3 [Standard Formatted Input], page 49).

'`o`'
: Print an integer as an unsigned octal number.

'`u`'
: Print an integer as an unsigned decimal number.

'`x`', '`X`'
: Print an integer as an unsigned hexadecimal number. '`%x`' prints using the digits '`0123456789abcdef`'. '`%X`' prints using the digits '`0123456789ABCDEF`'.

3.3.2.2 Inexact Conversions

'`f`'
: Print a floating-point number in fixed-point notation.

'`e`', '`E`'
: Print a floating-point number in exponential notation. '`%e`' prints '`e`' between mantissa and exponont. '`%E`' prints '`E`' between mantissa and exponont.

'`g`', '`G`'
: Print a floating-point number in either fixed or exponential notation, whichever is more appropriate for its magnitude. Unless an '`#`' flag has been supplied, trailing zeros after a decimal point will be stripped off. '`%g`' prints '`e`' between mantissa and exponont. '`%G`' prints '`E`' between mantissa and exponent.

‘`k`’, ‘`K`’
: Print a number like ‘`%g`’, except that an SI prefix is output after the number, which is scaled accordingly. ‘`%K`’ outputs a space between number and prefix, ‘`%k`’ does not.

3.3.2.3 Other Conversions

‘`c`’
: Print a single character. The ‘`-`’ flag is the only one which can be specified. It is an error to specify a precision.

‘`s`’
: Print a string. The ‘`-`’ flag is the only one which can be specified. A precision specifies the maximum number of characters to output; otherwise all characters in the string are output.

‘`a`’, ‘`A`’
: Print a scheme expression. The ‘`-`’ flag left-justifies the output. The ‘`#`’ flag specifies that strings and characters should be quoted as by `write` (which can be read using `read`); otherwise, output is as `display` prints. A precision specifies the maximum number of characters to output; otherwise as many characters as needed are output.

: *Note:* ‘`%a`’ and ‘`%A`’ are SLIB extensions.

‘`%`’
: Print a literal ‘`%`’ character. No argument is consumed. It is an error to specify flags, field width, precision, or type modifiers with ‘`%%`’.

3.3.3 Standard Formatted Input

```
(require 'scanf)
```

scanf-read-list *format* — Function
scanf-read-list *format port* — Function
scanf-read-list *format string* — Function

scanf *format arg1 . . .* — Macro
fscanf *port format arg1 . . .* — Macro
sscanf *str format arg1 . . .* — Macro

Each function reads characters, interpreting them according to the control string *format* argument.

`scanf-read-list` returns a list of the items specified as far as the input matches *format*. `scanf`, `fscanf`, and `sscanf` return the number of items successfully matched and stored. `scanf`, `fscanf`, and `sscanf` also set the location corresponding to *arg1* . . . using the methods:

symbol
: `set!`

car expression
: `set-car!`

cdr expression
: `set-cdr!`

vector-ref expression
: `vector-set!`

substring expression
: `substring-move-left!`

The argument to a `substring` expression in *arg1* . . . must be a non-constant string. Characters will be stored starting at the position specified by the second argument to `substring`. The number of characters stored will be limited by either the position specified by the third argument to `substring` or the length of the matched string, whichever is less.

The control string, *format*, contains conversion specifications and other characters used to direct interpretation of input sequences. The control string contains:

- White-space characters (blanks, tabs, newlines, or formfeeds) that cause input to be read (and discarded) up to the next non-white-space character.
- An ordinary character (not '`%`') that must match the next character of the input stream.
- Conversion specifications, consisting of the character '`%`', an optional assignment suppressing character '`*`', an optional numerical maximum-field width, an optional '`l`', '`h`' or '`L`' which is ignored, and a conversion code.

Unless the specification contains the '`n`' conversion character (described below), a conversion specification directs the conversion of the next input field. The result of a conversion specification is returned in the position of the corresponding argument points, unless '`*`' indicates assignment suppression. Assignment suppression provides a way to describe an input field to be skipped. An input field is defined as a string of characters; it extends to the next inappropriate character or until the field width, if specified, is exhausted.

> *Note:* This specification of format strings differs from the *ANSI C* and *POSIX* specifications. In SLIB, white space before an input field is not skipped unless white space appears before the conversion specification in the format string. In order to write format strings which work identically with *ANSI C* and SLIB, prepend whitespace to all conversion specifications except '`[`' and '`c`'.

The conversion code indicates the interpretation of the input field; For a suppressed field, no value is returned. The following conversion codes are legal:

'`%`'	A single % is expected in the input at this point; no value is returned.
'`d`', '`D`'	A decimal integer is expected.
'`u`', '`U`'	An unsigned decimal integer is expected.
'`o`', '`O`'	An octal integer is expected.
'`x`', '`X`'	A hexadecimal integer is expected.
'`i`'	An integer is expected. Returns the value of the next input item, interpreted according to C conventions; a leading '`0`' implies octal, a leading '`0x`' implies hexadecimal; otherwise, decimal is assumed.

‘n’ Returns the total number of bytes (including white space) read by `scanf`. No input is consumed by `%n`.

‘f’, ‘F’, ‘e’, ‘E’, ‘g’, ‘G’
A floating-point number is expected. The input format for floating-point numbers is an optionally signed string of digits, possibly containing a radix character ‘`.`’, followed by an optional exponent field consisting of an ‘`E`’ or an ‘`e`’, followed by an optional ‘`+`’, ‘`-`’, or space, followed by an integer.

‘c’, ‘C’ *Width* characters are expected. The normal skip-over-white-space is suppressed in this case; to read the next non-space character, use ‘`%1s`’. If a field width is given, a string is returned; up to the indicated number of characters is read.

‘s’, ‘S’ A character string is expected The input field is terminated by a white-space character. `scanf` cannot read a null string.

‘[’ Indicates string data and the normal skip-over-leading-white-space is suppressed. The left bracket is followed by a set of characters, called the scanset, and a right bracket; the input field is the maximal sequence of input characters consisting entirely of characters in the scanset. ‘`^`’, when it appears as the first character in the scanset, serves as a complement operator and redefines the scanset as the set of all characters not contained in the remainder of the scanset string. Construction of the scanset follows certain conventions. A range of characters may be represented by the construct first-last, enabling ‘`[0123456789]`’ to be expressed ‘`[0-9]`’. Using this convention, first must be lexically less than or equal to last; otherwise, the dash stands for itself. The dash also stands for itself when it is the first or the last character in the scanset. To include the right square bracket as an element of the scanset, it must appear as the first character (possibly preceded by a ‘`^`’) of the scanset, in which case it will not be interpreted syntactically as the closing bracket. At least one character must match for this conversion to succeed.

The `scanf` functions terminate their conversions at end-of-file, at the end of the control string, or when an input character conflicts with the control string. In the latter case, the offending character is left unread in the input stream.

3.4 Program and Arguments

3.4.1 Getopt

```
(require 'getopt)
```

This routine implements Posix command line argument parsing. Notice that returning values through global variables means that `getopt` is *not* reentrant.

optind Variable

Is the index of the current element of the command line. It is initially one. In order to parse a new command line or reparse an old one, **opting** must be reset.

optarg Variable

Is set by getopt to the (string) option-argument of the current option.

getopt *argc argv optstring* Procedure

Returns the next option letter in *argv* (starting from `(vector-ref argv *optind*)`) that matches a letter in *optstring*. *argv* is a vector or list of strings, the 0th of which getopt usually ignores. *argc* is the argument count, usually the length of *argv*. *optstring* is a string of recognized option characters; if a character is followed by a colon, the option takes an argument which may be immediately following it in the string or in the next element of *argv*.

optind is the index of the next element of the *argv* vector to be processed. It is initialized to 1 by '`getopt.scm`', and `getopt` updates it when it finishes with each element of *argv*.

`getopt` returns the next option character from *argv* that matches a character in *optstring*, if there is one that matches. If the option takes an argument, `getopt` sets the variable **optarg** to the option-argument as follows:

- If the option was the last character in the string pointed to by an element of *argv*, then **optarg** contains the next element of *argv*, and **optind** is incremented by 2. If the resulting value of **optind** is greater than or equal to *argc*, this indicates a missing option argument, and `getopt` returns an error indication.
- Otherwise, **optarg** is set to the string following the option character in that element of *argv*, and **optind** is incremented by 1.

If, when `getopt` is called, the string `(vector-ref argv *optind*)` either does not begin with the character `#\-` or is just `"-"`, `getopt` returns `#f` without changing **optind**. If `(vector-ref argv *optind*)` is the string `"--"`, `getopt` returns `#f` after incrementing **optind**.

If `getopt` encounters an option character that is not contained in *optstring*, it returns the question-mark `#\?` character. If it detects a missing option argument, it returns the colon character `#\:` if the first character of *optstring* was a colon, or a question-mark character otherwise. In either case, `getopt` sets the variable *getopt:opt* to the option character that caused the error.

The special option `"--"` can be used to delimit the end of the options; `#f` is returned, and `"--"` is skipped.

RETURN VALUE

`getopt` returns the next option character specified on the command line. A colon `#\:` is returned if `getopt` detects a missing argument and the first character of *optstring* was a colon `#\:`.

A question-mark `#\?` is returned if `getopt` encounters an option character not in *optstring* or detects a missing argument and the first character of *optstring* was not a colon `#\:`.

Otherwise, `getopt` returns `#f` when all command line options have been parsed.

Example:

```
#! /usr/local/bin/scm
;;;This code is SCM specific.
(define argv (program-arguments))
(require 'getopt)

(define opts ":a:b:cd")
(let loop ((opt (getopt (length argv) argv opts)))
  (case opt
    ((#\a) (print "option a: " *optarg*))
    ((#\b) (print "option b: " *optarg*))
    ((#\c) (print "option c"))
    ((#\d) (print "option d"))
    ((#\?) (print "error" getopt:opt))
    ((#\:) (print "missing arg" getopt:opt))
    ((#f) (if (< *optind* (length argv))
              (print "argv[" *optind* "]="
                     (list-ref argv *optind*)))
          (set! *optind* (+ *optind* 1))))
  (if (< *optind* (length argv))
      (loop (getopt (length argv) argv opts))))

(slib:exit)
```

3.4.2 Getopt–

getopt– *argc argv optstring* Function

The procedure `getopt--` is an extended version of `getopt` which parses *long option names* of the form '`--hold-the-onions`' and '`--verbosity-level=extreme`'. `Getopt--` behaves as `getopt` except for non-empty options beginning with '`--`'.

Options beginning with '`--`' are returned as strings rather than characters. If a value is assigned (using '`=`') to a long option, `*optarg*` is set to the value. The '`=`' and value are not returned as part of the option string.

No information is passed to `getopt--` concerning which long options should be accepted or whether such options can take arguments. If a long option did not have an argument, `*optarg` will be set to `#f`. The caller is responsible for detecting and reporting errors.

```
(define opts ":-:b:")
(define argc 5)
(define argv '("foo" "-b9" "--f1" "--2=" "--g3=35234.342" "--"))
(define *optind* 1)
(define *optarg* #f)
(require 'qp)
(do ((i 5 (+ -1 i)))
    ((zero? i))
    (define opt (getopt-- argc argv opts))
    (print *optind* opt *optarg*)))
  ⊣
 2 #\b "9"
 3 "f1" #f
 4 "2" ""
 5 "g3" "35234.342"
 5 #f "35234.342"
```

3.4.3 Command Line

```
(require 'read-command)
```

read-command *port* — Function
read-command — Function

`read-command` converts a *command line* into a list of strings suitable for parsing by `getopt`. The syntax of command lines supported resembles that of popular *shells*. `read-command` updates *port* to point to the first character past the command delimiter.

If an end of file is encountered in the input before any characters are found that can begin an object or comment, then an end of file object is returned.

The *port* argument may be omitted, in which case it defaults to the value returned by `current-input-port`.

The fields into which the command line is split are delimited by whitespace as defined by `char-whitespace?`. The end of a command is delimited by end-of-file or unescaped semicolon (⟨;⟩) or ⟨newline⟩. Any character can be literally included in a field by escaping it with a backslach (⟨\⟩).

The initial character and types of fields recognized are:

‘\’
: The next character has is taken literally and not interpreted as a field delimiter. If ⟨\⟩ is the last character before a ⟨newline⟩, that ⟨newline⟩ is just ignored. Processing continues from the characters after the ⟨newline⟩ as though the backslash and ⟨newline⟩ were not there.

‘"’
: The characters up to the next unescaped ⟨"⟩ are taken literally, according to [R4RS] rules for literal strings (see section “Strings” in *Revised(4) Scheme*).

‘(’, ‘%’’
: One scheme expression is `read` starting with this character. The `read` expression is evaluated, converted to a string (using `display`), and replaces the expression in the returned field.

‘;’
: Semicolon delimits a command. Using semicolons more than one command can appear on a line. Escaped semicolons and semicolons inside strings do not delimit commands.

The comment field differs from the previous fields in that it must be the first character of a command or appear after whitespace in order to be recognized. ⟨#⟩ can be part of fields if these conditions are not met. For instance, `ab#c` is just the field ab#c.

‘`#`’ Introduces a comment. The comment continues to the end of the line on which the semicolon appears. Comments are treated as whitespace by `read-dommand-line` and backslashes before ⟨newline⟩s in comments are also ignored.

read-options-file *filename* Function

`read-options-file` converts an *options file* into a list of strings suitable for parsing by `getopt`. The syntax of options files is the same as the syntax for command lines, except that ⟨newline⟩s do not terminate reading (only ⟨;⟩ or end of file).

If an end of file is encountered before any characters are found that can begin an object or comment, then an end of file object is returned.

3.4.4 Parameter lists

```
(require 'parameters)
```

Arguments to procedures in scheme are distinguished from each other by their position in the procedure call. This can be confusing when a procedure takes many arguments, many of which are not often used.

A *parameter-list* is a way of passing named information to a procedure. Procedures are also defined to set unused parameters to default values, check parameters, and combine parameter lists.

A *parameter* has the form `(`parameter-name value1 `...)`. This format allows for more than one value per parameter-name.

A *parameter-list* is a list of *parameter*s, each with a different *parameter-name*.

make-parameter-list *parameter-names* Function

Returns an empty parameter-list with slots for *parameter-names*.

parameter-list-ref *parameter-list parameter-name* Function

parameter-name must name a valid slot of *parameter-list*. `parameter-list-ref` returns the value of parameter *parameter-name* of *parameter-list*.

remove-parameter *parameter-name parameter-list* Function

Removes the parameter *parameter-name* from *parameter-list*. `remove-parameter` does not alter the argument *parameter-list*.

If there are more than one *parameter-name* parameters, an error is signaled.

adjoin-parameters! *parameter-list parameter1 . . .* Procedure

Returns *parameter-list* with *parameter1 . . .* merged in.

parameter-list-expand *expanders parameter-list* Procedure

expanders is a list of procedures whose order matches the order of the *parameter-names* in the call to `make-parameter-list` which created *parameter-list*. For each non-false element of *expanders* that procedure is mapped over the corresponding parameter value and the returned parameter lists are merged into *parameter-list*.

This process is repeated until *parameter-list* stops growing. The value returned from `parameter-list-expand` is unspecified.

fill-empty-parameters *defaulters parameter-list* Function

defaulters is a list of procedures whose order matches the order of the *parameter-names* in the call to `make-parameter-list` which created *parameter-list*. `fill-empty-parameters` returns a new parameter-list with each empty parameter replaced with the list returned by calling the corresponding *defaulter* with *parameter-list* as its argument.

check-parameters *checks parameter-list* Function

checks is a list of procedures whose order matches the order of the *parameter-names* in the call to `make-parameter-list` which created *parameter-list*.

`check-parameters` returns *parameter-list* if each *check* of the corresponding *parameter-list* returns non-false. If some *check* returns `#f` a warning is signaled.

In the following procedures *arities* is a list of symbols. The elements of `arities` can be:

`single`	Requires a single parameter.
`optional`	A single parameter or no parameter is acceptable.
`boolean`	A single boolean parameter or zero parameters is acceptable.
`nary`	Any number of parameters are acceptable.
`nary1`	One or more of parameters are acceptable.

parameter-list->arglist *positions arities parameter-list* Function

Returns *parameter-list* converted to an argument list. Parameters of *arity* type `single` and `boolean` are converted to the single value associated with them. The other *arity* types are converted to lists of the value(s).

positions is a list of positive integers whose order matches the order of the *parameter-names* in the call to `make-parameter-list` which created *parameter-list*. The integers specify in which argument position the corresponding parameter should appear.

3.4.5 Getopt Parameter lists

```
(require 'getopt-parameters)
```

getopt->parameter-list *argc argv optnames arities types aliases desc ...* Function

Returns *argv* converted to a parameter-list. *optnames* are the parameter-names. *arities* and *types* are lists of symbols corresponding to *optnames*.

aliases is a list of lists of strings or integers paired with elements of *optnames*. Each one-character string will be treated as a single '`-`' option by `getopt`. Longer strings will be treated as long-named options (see Section 3.4.1 [Getopt], page 51).

If the *aliases* association list has only strings as its `car`s, then all the option-arguments after an option (and before the next option) are adjoined to that option.

If the *aliases* association list has integers, then each (string) option will take at most one option-argument. Unoptioned arguments are collected in a list. A '`-1`' alias will take the last argument in this list; '`+1`' will take the first argument in the list. The aliases -2 then +2; -3 then +3; ... are tried so long as a positive or negative consecutive alias is found and arguments remain in the list. Finally a '`0`' alias, if found, absorbs any remaining arguments.

In all cases, if unclaimed arguments remain after processing, a warning is signaled and #f is returned.

getopt->arglist *argc argv optnames positions arities types defaulters checks aliases desc ...* Function

Like `getopt->parameter-list`, but converts *argv* to an argument-list as specified by *optnames*, *positions*, *arities*, *types*, *defaulters*, *checks*, and *aliases*. If the options supplied violate the *arities* or *checks* constraints, then a warning is signaled and #f is returned.

These `getopt` functions can be used with SLIB relational databases. For an example, See Section 5.2.2 [Using Databases], page 130.

If errors are encountered while processing options, directions for using the options (and argument strings *desc* ...) are printed to `current-error-port`.

```
(begin
  (set! *optind* 1)
  (getopt->parameter-list
   2
   '("cmd" "-?")
   '(flag number symbols symbols string flag2 flag3 num2 num3)
   '(boolean optional nary1 nary single boolean boolean nary nary)
   '(boolean integer symbol symbol string boolean boolean integer integer)
   '(("flag" flag)
     ("f" flag)
     ("Flag" flag2)
     ("B" flag3)
     ("optional" number)
     ("o" number)
     ("nary1" symbols)
     ("N" symbols)
     ("nary" symbols)
     ("n" symbols)
     ("single" string)
     ("s" string)
     ("a" num2)
     ("Abs" num3))))
 ⊣
Usage: cmd [OPTION ARGUMENT ...] ...

  -f, --flag
  -o, --optional=<number>
  -n, --nary=<symbols> ...
  -N, --nary1=<symbols> ...
  -s, --single=<string>
      --Flag
  -B
  -a        <num2> ...
      --Abs=<num3> ...

ERROR: getopt->parameter-list "unrecognized option" "-?"
```

3.4.6 Filenames

`(require 'filename)` or `(require 'glob)`

filename:match?? *pattern* Function
filename:match-ci?? *pattern* Function

Returns a predicate which returns a non-false value if its string argument matches (the string) *pattern*, false otherwise. Filename matching is like *glob* expansion described the bash manpage, except that names beginning with '`.`' are matched and '`/`' characters are not treated specially.

These functions interpret the following characters specially in *pattern* strings:

'`*`'	Matches any string, including the null string.
'`?`'	Matches any single character.
'`[...]`'	Matches any one of the enclosed characters. A pair of characters separated by a minus sign (-) denotes a range; any character lexically between those two characters, inclusive, is matched. If the first character following the '`[`' is a '`!`' or a '`^`' then any character not enclosed is matched. A '`-`' or '`]`' may be matched by including it as the first or last character in the set.

filename:substitute?? *pattern template* Function
filename:substitute-ci?? *pattern template* Function

Returns a function transforming a single string argument according to glob patterns *pattern* and *template*. *pattern* and *template* must have the same number of wildcard specifications, which need not be identical. *pattern* and *template* may have a different number of literal sections. If an argument to the function matches *pattern* in the sense of `filename:match??` then it returns a copy of *template* in which each wildcard specification is replaced by the part of the argument matched by the corresponding wildcard specification in *pattern*. A `*` wildcard matches the longest leftmost string possible. If the argument does not match *pattern* then false is returned.

template may be a function accepting the same number of string arguments as there are wildcard specifications in *pattern*. In the case of a match the result of applying *template* to a list of the substrings matched by wildcard specifications will be returned, otherwise *template* will not be called and `#f` will be returned.

```
((filename:substitute?? "scm_[0-9]*.html" "scm5c4_??.htm")
 "scm_10.html")
⇒ "scm5c4_10.htm"
((filename:substitute?? "??" "beg?mid?end") "AZ")
⇒ "begAmidZend"
((filename:substitute?? "*na*" "?NA?") "banana")
⇒ "banaNA"
((filename:substitute?? "?*?" (lambda (s1 s2 s3) (string-append s3 s1))) "ABZ"
⇒ "ZA"
```

replace-suffix *str old new* Function

str can be a string or a list of strings. Returns a new string (or strings) similar to `str` but with the suffix string *old* removed and the suffix string *new* appended. If the end of *str* does not match *old*, an error is signaled.

```
(replace-suffix "/usr/local/lib/slib/batch.scm" ".scm" ".c")
⇒ "/usr/local/lib/slib/batch.c"
```

3.4.7 Batch

```
(require 'batch)
```

The batch procedures provide a way to write and execute portable scripts for a variety of operating systems. Each `batch:` procedure takes as its first argument a parameter-list (see Section 3.4.4 [Parameter lists], page 54). This parameter-list argument *parms* contains named associations. Batch currently uses 2 of these:

`batch-port`
: The port on which to write lines of the batch file.

`batch-dialect`
: The syntax of batch file to generate. Currently supported are:

- unix
- dos
- vms
- amigaos
- system
- *unknown*

'`batch.scm`' uses 2 enhanced relational tables (see Section 5.2.2 [Using Databases], page 130) to store information linking the names of `operating-systems` to `batch-dialectes`.

batch:initialize! *database* Function
: Defines `operating-system` and `batch-dialect` tables and adds the domain `operating-system` to the enhanced relational database *database*.

batch:platform Variable
: Is batch's best guess as to which operating-system it is running under. `batch:platform` is set to `(software-type)` (see Section 1.5.3 [Configuration], page 6) unless `(software-type)` is `unix`, in which case finer distinctions are made.

batch:call-with-output-script *parms file proc* Function
: *proc* should be a procedure of one argument. If *file* is an output-port, `batch:call-with-output-script` writes an appropriate header to *file* and then calls *proc* with *file* as the only argument. If *file* is a string, `batch:call-with-output-script` opens a output-file of name *file*, writes an appropriate header to *file*, and then calls *proc* with the newly opened port as the only argument. Otherwise, `batch:call-with-output-script` acts as if it was called with the result of `(current-output-port)` as its third argument.

The rest of the `batch:` procedures write (or execute if `batch-dialect` is `system`) commands to the batch port which has been added to *parms* or `(copy-tree` *parms*`)` by the code:

`(adjoin-parameters!` *parms* `(list 'batch-port` *port*`))`

batch:command *parms string1 string2 . . .* Function
: Calls `batch:try-command` (below) with arguments, but signals an error if `batch:try-command` returns `#f`.

These functions return a non-false value if the command was successfully translated into the batch dialect and `#f` if not. In the case of the `system` dialect, the value is non-false if the operation suceeded.

batch:try-command *parms string1 string2 . . .* Function
: Writes a command to the `batch-port` in *parms* which executes the program named *string1* with arguments *string2 . . .*.

batch:try-chopped-command *parms arg1 arg2 . . . list* Function
: breaks the last argument *list* into chunks small enough so that the command:

arg1 arg2 . . . chunk

fits withing the platform's maximum command-line length.

`batch:try-chopped-command` calls `batch:try-command` with the command and returns non-false only if the commands all fit and `batch:try-command` of each command line returned non-false.

batch:run-script *parms string1 string2 . . .* Function

Writes a command to the `batch-port` in *parms* which executes the batch script named *string1* with arguments *string2 . . .*.

Note: `batch:run-script` and `batch:try-command` are not the same for some operating systems (VMS).

batch:comment *parms line1 . . .* Function

Writes comment lines *line1 . . .* to the `batch-port` in *parms*.

batch:lines->file *parms file line1 . . .* Function

Writes commands to the `batch-port` in *parms* which create a file named *file* with contents *line1 . . .*.

batch:delete-file *parms file* Function

Writes a command to the `batch-port` in *parms* which deletes the file named *file*.

batch:rename-file *parms old-name new-name* Function

Writes a command to the `batch-port` in *parms* which renames the file *old-name* to *new-name*.

In addition, batch provides some small utilities very useful for writing scripts:

truncate-up-to *path char* Function
truncate-up-to *path string* Function
truncate-up-to *path charlist* Function

path can be a string or a list of strings. Returns *path* sans any prefixes ending with a character of the second argument. This can be used to derive a filename moved locally from elsewhere.

```
(truncate-up-to "/usr/local/lib/slib/batch.scm" "/")
⇒ "batch.scm"
```

string-join *joiner string1 . . .* Function

Returns a new string consisting of all the strings *string1 . . .* in order appended together with the string *joiner* between each adjacent pair.

must-be-first *list1 list2* Function

Returns a new list consisting of the elements of *list2* ordered so that if some elements of *list1* are `equal?` to elements of *list2*, then those elements will appear first and in the order of *list1*.

must-be-last *list1 list2* Function

Returns a new list consisting of the elements of *list1* ordered so that if some elements of *list2* are `equal?` to elements of *list1*, then those elements will appear last and in the order of *list2*.

os->batch-dialect *osname* Function

Returns its best guess for the `batch-dialect` to be used for the operating-system named *osname*. `os->batch-dialect` uses the tables added to *database* by `batch:initialize!`.

Here is an example of the use of most of batch's procedures:

```
(require 'databases)
(require 'parameters)
(require 'batch)
(require 'glob)

(define batch (create-database #f 'alist-table))
(batch:initialize! batch)

(define my-parameters
  (list (list 'batch-dialect (os->batch-dialect batch:platform))
        (list 'platform batch:platform)
        (list 'batch-port (current-output-port)))) ;gets filled in later

(batch:call-with-output-script
 my-parameters
 "my-batch"
 (lambda (batch-port)
   (adjoin-parameters! my-parameters (list 'batch-port batch-port))
   (and
    (batch:comment my-parameters
                   "================ Write file with C program.")
    (batch:rename-file my-parameters "hello.c" "hello.c~")
    (batch:lines->file my-parameters "hello.c"
                       "#include <stdio.h>"
                       "int main(int argc, char **argv)"
                       "{"
                       "  printf(\"hello world\\n\");"
                       "  return 0;"
                       "}" )
    (batch:command my-parameters "cc" "-c" "hello.c")
    (batch:command my-parameters "cc" "-o" "hello"
                  (replace-suffix "hello.c" ".c" ".o"))
    (batch:command my-parameters "hello")
    (batch:delete-file my-parameters "hello")
    (batch:delete-file my-parameters "hello.c")
    (batch:delete-file my-parameters "hello.o")
    (batch:delete-file my-parameters "my-batch")
    )))
```

Produces the file 'my-batch':

```
#!/bin/sh
# "my-batch" script created by SLIB/batch Sun Oct 31 18:24:10 1999
# ================ Write file with C program.
mv -f hello.c hello.c~
```

```
rm -f hello.c
echo '#include <stdio.h>'>>hello.c
echo 'int main(int argc, char **argv)'>>hello.c
echo '{'>>hello.c
echo '  printf("hello world\n");'>>hello.c
echo '  return 0;'>>hello.c
echo '}'>>hello.c
cc -c hello.c
cc -o hello hello.o
hello
rm -f hello
rm -f hello.c
rm -f hello.o
rm -f my-batch
```

When run, '`my-batch`' prints:

```
bash$ my-batch
mv: hello.c: No such file or directory
hello world
```

3.5 HTML

```
(require 'html-form)
```

html:atval *txt* Function

Returns a string with character substitutions appropriate to send *txt* as an *attribute-value*.

html:plain *txt* Function

Returns a string with character substitutions appropriate to send *txt* as an *plain-text*.

html:meta *name content* Function

Returns a tag of meta-information suitable for passing as the third argument to `html:head`. The tag produced is '`<META NAME="`*name*`" CONTENT="`*content*`">`'. The string or symbol *name* can be '`author`', '`copyright`', '`keywords`', '`description`', '`date`', '`robots`', ...

html:http-equiv *name content* Function

Returns a tag of HTTP information suitable for passing as the third argument to `html:head`. The tag produced is '`<META HTTP-EQUIV="`*name*`" CONTENT="`*content*`">`'. The string or symbol *name* can be '`Expires`', '`PICS-Label`', '`Content-Type`', '`Refresh`', ...

html:meta-refresh *delay uri* Function
html:meta-refresh *delay* Function

Returns a tag suitable for passing as the third argument to `html:head`. If *uri* argument is supplied, then *delay* seconds after displaying the page with this tag, Netscape or IE browsers will fetch and display *uri*. Otherwise, *delay* seconds after displaying the page with this tag, Netscape or IE browsers will fetch and redisplay this page.

html:head *title backlink tags . . .* Function
html:head *title backlink* Function
html:head *title* Function

Returns header string for an HTML page named *title*. If *backlink* is a string, it is used verbatim between the '`H1`' tags; otherwise *title* is used. If string arguments *tags* ... are supplied, then they are included verbatim within the `<HEAD>` section.

html:body *body . . .* Function

Returns HTML string to end a page.

html:pre *line1 line . . .* Function

Returns the strings *line1*, *lines* as *PRE*formmated plain text (rendered in fixed-width font). Newlines are inserted between *line1*, *lines*. HTML tags ('`<tag>`') within *lines* will be visible verbatim.

html:comment *line1 line . . .* Function

Returns the strings *line1* as HTML comments.

3.6 HTML Forms

html:form *method action body . . .* Function

The symbol *method* is either `get`, `head`, `post`, `put`, or `delete`. The strings *body* form the body of the form. `html:form` returns the HTML *form*.

html:hidden *name value* Function

Returns HTML string which will cause *name*=*value* in form.

html:checkbox *pname default* Function

Returns HTML string for check box.

html:text *pname default size . . .* Function

Returns HTML string for one-line text box.

html:text-area *pname default-list* Function

Returns HTML string for multi-line text box.

html:select *pname arity default-list foreign-values* Function

Returns HTML string for pull-down menu selector.

html:buttons *pname arity default-list foreign-values* Function

Returns HTML string for any-of selector.

form:submit *submit-label command* Function
form:submit *submit-label* Function

The string or symbol *submit-label* appears on the button which submits the form. If the optional second argument *command* is given, then `*command*=`*command* and `*button*=`*submit-label* are set in the query. Otherwise, `*command*=`*submit-label* is set in the query.

form:image *submit-label image-src* Function

The *image-src* appears on the button which submits the form.

form:reset Function

Returns a string which generates a *reset* button.

form:element *pname arity default-list foreign-values* Function

Returns a string which generates an INPUT element for the field named *pname*. The element appears in the created form with its representation determined by its *arity* and domain. For domains which are foreign-keys:

`single`	select menu
`optional`	select menu
`nary`	check boxes
`nary1`	check boxes

If the foreign-key table has a field named '`visible-name`', then the contents of that field are the names visible to the user for those choices. Otherwise, the foreign-key itself is visible.

For other types of domains:

`single`	text area
`optional`	text area
`boolean`	check box
`nary`	text area
`nary1`	text area

form:delimited *pname doc aliat arity default-list foreign-values* Function

Returns a HTML string for a form element embedded in a line of a delimited list. Apply map `form:delimited` to the list returned by `command->p-specs`.

command->p-specs *rdb command-table command* Function

The symbol *command-table* names a command table in the *rdb* relational database. The symbol *command* names a key in *command-table*.

`command->p-specs` returns a list of lists of *pname*, *doc*, *aliat*, *arity*, *default-list*, and *foreign-values*. The returned list has one element for each parameter of command *command*.

This example demonstrates how to create a HTML-form for the '`build`' command.

```
(require (in-vicinity (implementation-vicinity) "build.scm"))
(call-with-output-file "buildscm.html"
  (lambda (port)
    (display
     (string-append
      (html:head 'commands)
      (html:body
       (sprintf #f "<H2>%s:</H2><BLOCKQUOTE>%s</BLOCKQUOTE>\\n"
(html:plain 'build)
(html:plain ((comtab 'get 'documentation) 'build)))
       (html:form
'post
(or "http://localhost:8081/buildscm" "/cgi-bin/build.cgi")
(apply html:delimited-list
       (apply map form:delimited
      (command->p-specs build '*commands* 'build)))
(form:submit 'build)
(form:reset))))
     port)))
```

3.7 HTML Tables

```
(require 'db->html)
```

html:table *options row . . .* Function

html:caption *caption align* Function
html:caption *caption* Function
align can be '`top`' or '`bottom`'.

html:heading *columns* Function
Outputs a heading row for the currently-started table.

html:href-heading *columns uris* Function
Outputs a heading row with column-names *columns* linked to URIs *uris*.

html:linked-row-converter *k foreigns* Function
The positive integer *k* is the primary-key-limit (number of primary-keys) of the table. *foreigns* is a list of the filenames of foreign-key field pages and #f for non foreign-key fields.

`html:linked-row-converter` returns a procedure taking a row for its single argument. This returned procedure returns the html string for that table row.

table-name->filename *table-name* Function

Returns the symbol *table-name* converted to a filename.

table->linked-html *caption db table-name match-key1 . . .* Function

Returns HTML string for *db* table *table-name*. Every foreign-key value is linked to the page (of the table) defining that key.

The optional *match-key1* . . . arguments restrict actions to a subset of the table. See Section 5.2.5 [Table Operations], page 135.

table->linked-page *db table-name index-filename arg . . .* Function

Returns a complete HTML page. The string *index-filename* names the page which refers to this one.

The optional *args* . . . arguments restrict actions to a subset of the table. See Section 5.2.5 [Table Operations], page 135.

catalog->html *db caption arg . . .* Function

Returns HTML string for the catalog table of *db*.

3.7.1 HTML editing tables

A client can modify one row of an editable table at a time. For any change submitted, these routines check if that row has been modified during the time the user has been editing the form. If so, an error page results.

The behavior of edited rows is:

- If no fields are changed, then no change is made to the table.
- If the primary keys equal null-keys (parameter defaults), and no other user has modified that row, then that row is deleted.
- If only primary keys are changed, there are non-key fields, and no row with the new keys is in the table, then the old row is deleted and one with the new keys is inserted.
- If only non-key fields are changed, and that row has not been modified by another user, then the row is changed to reflect the fields.
- If both keys and non-key fields are changed, and no row with the new keys is in the table, then a row is created with the new keys and fields.
- If fields are changed, all fields are primary keys, and no row with the new keys is in the table, then a row is created with the new keys.

After any change to the table, a `sync-database` of the database is performed.

command:modify-table *table-name null-keys update delete retrieve* Function
command:modify-table *table-name null-keys update delete* Function
command:modify-table *table-name null-keys update* Function
command:modify-table *table-name null-keys* Function

Returns procedure (of *db*) which returns procedure to modify row of *table-name*. *null-keys* is the list of *null* keys which indicate that the row is to be deleted. Optional arguments *update*, *delete*, and *retrieve* default to the `row:update`, `row:delete`, and `row:retrieve` of *table-name* in *db*.

command:make-editable-table *rdb table-name arg . . .* Function

Given *table-name* in *rdb*, creates parameter and `*command*` tables for editing one row of *table-name* at a time. `command:make-editable-table` returns a procedure taking a row argument which returns the HTML string for editing that row.

Optional *args* are expressions (lists) added to the call to `command:modify-table`.

The domain name of a column determines the expected arity of the data stored in that column. Domain names ending in:

‘`*`’	have arity ‘`nary`’;
‘`+`’	have arity ‘`nary1`’.

html:editable-row-converter *k names edit-point edit-converter* Function

The positive integer *k* is the primary-key-limit (number of primary-keys) of the table. *names* is a list of the field-names. *edit-point* is the list of primary-keys denoting the row to edit (or #f). *edit-converter* is the procedure called with *k*, *names*, and the row to edit.

`html:editable-row-converter` returns a procedure taking a row for its single argument. This returned procedure returns the html string for that table row.

Each HTML table constructed using `html:editable-row-converter` has first *k* fields (typically the primary key fields) of each row linked to a text encoding of these fields (the result of calling `row->anchor`). The page so referenced typically allows the user to edit fields of that row.

3.7.2 HTML databases

db->html-files *db dir index-filename caption* Function

db must be a relational database. *dir* must be #f or a non-empty string naming an existing sub-directory of the current directory.

`db->html-files` creates an html page for each table in the database *db* in the sub-directory named *dir*, or the current directory if *dir* is #f. The top level page with the catalog of tables (captioned *caption*) is written to a file named *index-filename*.

db->html-directory *db dir index-filename* Function
db->html-directory *db dir* Function

db must be a relational database. *dir* must be a non-empty string naming an existing sub-directory of the current directory or one to be created. The optional string *index-filename* names the filename of the top page, which defaults to ‘`index.html`’.

`db->html-directory` creates sub-directory *dir* if neccessary, and calls `(db->html-files` *db dir index-filename dir*`)`. The ‘`file:`’ URI of *index-filename* is returned.

db->netscape *db dir index-filename* Function
db->netscape *db dir* Function

`db->netscape` is just like `db->html-directory`, but calls `browse-url-netscape` with the uri for the top page after the pages are created.

3.8 HTTP and CGI

`(require 'http)` or `(require 'cgi)`

http:header *alist* Function

Returns a string containing lines for each element of *alist*; the `car` of which is followed by '`: `', then the `cdr`.

http:content *alist body . . .* Function

Returns the concatenation of strings *body* with the `(http:header` *alist*`)` and the '`Content-Length`' prepended.

http:byline Variable

String appearing at the bottom of error pages.

http:error-page *status-code reason-phrase html-string . . .* Function

status-code and *reason-phrase* should be an integer and string as specified in *RFC 2068*. The returned page (string) will show the *status-code* and *reason-phrase* and any additional *html-strings . . .*; with **http:byline** or SLIB's default at the bottom.

http:forwarding-page *title delay uri html-string . . .* Function

The string or symbol *title* is the page title. *delay* is a non-negative integer. The *html-strings . . .* are typically used to explain to the user why this page is being forwarded.

`http:forwarding-page` returns an HTML string for a page which automatically forwards to *uri* after *delay* seconds. The returned page (string) contains any *html-strings . . .* followed by a manual link to *uri*, in case the browser does not forward automatically.

http:serve-query *serve-proc input-port output-port* Function

reads the *URI* and *query-string* from *input-port*. If the query is a valid '`"POST"`' or '`"GET"`' query, then `http:serve-query` calls *serve-proc* with three arguments, the *request-line*, *query-string*, and *header-alist*. Otherwise, `http:serve-query` calls *serve-proc* with the *request-line*, #f, and *header-alist*.

If *serve-proc* returns a string, it is sent to *output-port*. If *serve-proc* returns a list, then an error page with number 525 and strings from the list. If *serve-proc* returns #f, then a '`Bad Request`' (400) page is sent to *output-port*.

Otherwise, `http:serve-query` replies (to *output-port*) with appropriate HTML describing the problem.

This example services HTTP queries from *port-number*:

```
(define socket (make-stream-socket AF_INET 0))
(and (socket:bind socket port-number) ; AF_INET INADDR_ANY
     (socket:listen socket 10)        ; Queue up to 10 requests.
     (dynamic-wind
         (lambda () #f)
         (lambda ()
           (do ((port (socket:accept socket) (socket:accept socket)))
               (#f)
             (let ((iport (duplicate-port port "r"))
                   (oport (duplicate-port port "w")))
               (http:serve-query build:serve iport oport)
               (close-port iport)
               (close-port oport))
             (close-port port)))
         (lambda () (close-port socket))))
```

cgi:serve-query *serve-proc* Function

reads the *URI* and *query-string* from `(current-input-port)`. If the query is a valid '`"POST"`' or '`"GET"`' query, then `cgi:serve-query` calls *serve-proc* with three arguments, the *request-line*, *query-string*, and *header-alist*. Otherwise, `cgi:serve-query` calls *serve-proc* with the *request-line*, #f, and *header-alist*.

If *serve-proc* returns a string, it is sent to `(current-input-port)`. If *serve-proc* returns a list, then an error page with number 525 and strings from the list. If *serve-proc* returns #f, then a '`Bad Request`' (400) page is sent to `(current-input-port)`.

Otherwise, `cgi:serve-query` replies (to `(current-input-port)`) with appropriate HTML describing the problem.

make-query-alist-command-server *rdb command-table* Function
make-query-alist-command-server *rdb command-table #t* Function

Returns a procedure of one argument. When that procedure is called with a *query-alist* (as returned by `uri:decode-query`, the value of the '`*command*`' association will be the command invoked in *command-table*. If '`*command*`' is not in the *query-alist* then the value of '`*suggest*`' is tried. If neither name is in the *query-alist*, then the literal value '`*default*`' is tried in *command-table*.

If optional third argument is non-false, then the command is called with just the parameter-list; otherwise, command is called with the arguments described in its table.

3.9 URI

```
(require 'uri)
```

Implements *Uniform Resource Identifiers* (URI) as described in RFC 2396.

make-uri Function
make-uri *fragment* Function
make-uri *query fragment* Function
make-uri *path query fragment* Function
make-uri *authority path query fragment* Function
make-uri *scheme authority path query fragment* Function

Returns a Uniform Resource Identifier string from component arguments.

html:anchor *name* Function

Returns a string which defines this location in the (HTML) file as *name*. The hypertext '`<A HREF="#name">`' will link to this point.

```
(html:anchor "(section 7)")
⇒
"<A NAME=\"(section%207)\"></A>"
```

html:link *uri highlighted* Function

Returns a string which links the *highlighted* text to *uri*.

```
(html:link (make-uri "(section 7)") "section 7")
⇒
"<A HREF=\"#(section%207)\">section 7</A>"
```

html:base *uri* Function

Returns a string specifying the *base uri* of a document, for inclusion in the HEAD of the document (see Section 3.5 [HTML], page 62).

html:isindex *prompt* Function

Returns a string specifying the search *prompt* of a document, for inclusion in the HEAD of the document (see Section 3.5 [HTML], page 62).

uri->tree *uri-reference base-tree . . .* Function

Returns a list of 5 elements corresponding to the parts (*scheme authority path query fragment*) of string *uri-reference*. Elements corresponding to absent parts are #f.

The *path* is a list of strings. If the first string is empty, then the path is absolute; otherwise relative.

If the *authority* component is a *Server-based Naming Authority*, then it is a list of the *userinfo*, *host*, and *port* strings (or #f). For other types of *authority* components the *authority* will be a string.

```
(uri->tree "http://www.ics.uci.edu/pub/ietf/uri/#Related")
⇒
(http "www.ics.uci.edu" ("" "pub" "ietf" "uri" "") #f "Related")
```

`uric:` prefixes indicate procedures dealing with URI-components.

uric:encode *uri-component allows* Function

Returns a copy of the string *uri-component* in which all *unsafe* octets (as defined in RFC 2396) have been '%' *escaped*. `uric:decode` decodes strings encoded by `uric:encode`.

uric:decode *uri-component* Function

Returns a copy of the string *uri-component* in which each '%' escaped characters in *uri-component* is replaced with the character it encodes. This routine is useful for showing URI contents on error pages.

3.10 Printing Scheme

3.10.1 Generic-Write

```
(require 'generic-write)
```

`generic-write` is a procedure that transforms a Scheme data value (or Scheme program expression) into its textual representation and prints it. The interface to the procedure is sufficiently general to easily implement other useful formatting procedures such as pretty printing, output to a string and truncated output.

generic-write *obj display? width output* Procedure

obj	Scheme data value to transform.
display?	Boolean, controls whether characters and strings are quoted.
width	Extended boolean, selects format:
	#f — single line format
	integer > 0 — pretty-print (value = max nb of chars per line)
output	Procedure of 1 argument of string type, called repeatedly with successive substrings of the textual representation. This procedure can return `#f` to stop the transformation.

The value returned by `generic-write` is undefined.

Examples:

```
(write obj) ≡ (generic-write obj #f #f display-string)
(display obj) ≡ (generic-write obj #t #f display-string)
```

where

```
display-string ≡
(lambda (s) (for-each write-char (string->list s)) #t)
```

3.10.2 Object-To-String

```
(require 'object->string)
```

object->string *obj* Function

Returns the textual representation of *obj* as a string.

object->limited-string *obj limit* Function

Returns the textual representation of *obj* as a string of length at most *limit*.

3.10.3 Pretty-Print

```
(require 'pretty-print)
```

pretty-print *obj* Procedure
pretty-print *obj port* Procedure

`pretty-print`s *obj* on *port*. If *port* is not specified, `current-output-port` is used.

Example:

```
(pretty-print '((1 2 3 4 5) (6 7 8 9 10) (11 12 13 14 15)
                (16 17 18 19 20) (21 22 23 24 25)))
   ⊣ ((1 2 3 4 5)
   ⊣  (6 7 8 9 10)
   ⊣  (11 12 13 14 15)
   ⊣  (16 17 18 19 20)
   ⊣  (21 22 23 24 25))
```

pretty-print->string *obj* Procedure
pretty-print->string *obj width* Procedure

Returns the string of *obj* `pretty-print`ed in *width* columns. If *width* is not specified, `(output-port-width)` is used.

Example:

```
(pretty-print->string '((1 2 3 4 5) (6 7 8 9 10) (11 12 13 14 15)
                        (16 17 18 19 20) (21 22 23 24 25)))
⇒
"((1 2 3 4 5)
 (6 7 8 9 10)
 (11 12 13 14 15)
 (16 17 18 19 20)
 (21 22 23 24 25))
"
```

```
(pretty-print->string '((1 2 3 4 5) (6 7 8 9 10) (11 12 13 14 15)
                        (16 17 18 19 20) (21 22 23 24 25))
                      16)
⇒
"((1 2 3 4 5)
 (6 7 8 9 10)
 (11
  12
  13
  14
  15)
 (16
  17
  18
  19
  20)
 (21
  22
  23
  24
  25))
"
```

```
(require 'pprint-file)
```

pprint-file *infile* Procedure
pprint-file *infile outfile* Procedure

Pretty-prints all the code in *infile*. If *outfile* is specified, the output goes to *outfile*, otherwise it goes to `(current-output-port)`.

pprint-filter-file *infile proc outfile* Function
pprint-filter-file *infile proc* Function

infile is a port or a string naming an existing file. Scheme source code expressions and definitions are read from the port (or file) and *proc* is applied to them sequentially.

outfile is a port or a string. If no *outfile* is specified then `current-output-port` is assumed. These expanded expressions are then `pretty-print`ed to this port.

Whitepsace and comments (introduced by `;`) which are not part of scheme expressions are reproduced in the output. This procedure does not affect the values returned by `current-input-port` and `current-output-port`.

`pprint-filter-file` can be used to pre-compile macro-expansion and thus can reduce loading time. The following will write into '`exp-code.scm`' the result of expanding all defmacros in '`code.scm`'.

```
(require 'pprint-file)
(require 'defmacroexpand)
(defmacro:load "my-macros.scm")
(pprint-filter-file "code.scm" defmacro:expand* "exp-code.scm")
```

3.11 Time and Date

If `(provided? 'current-time)`:

The procedures `current-time`, `difftime`, and `offset-time` deal with a *calendar time* datatype which may or may not be disjoint from other Scheme datatypes.

current-time Function

Returns the time since 00:00:00 GMT, January 1, 1970, measured in seconds. Note that the reference time is different from the reference time for `get-universal-time` in Section 3.11.3 [Common-Lisp Time], page 77.

difftime *caltime1 caltime0* Function

Returns the difference (number of seconds) between twe calendar times: *caltime1* - *caltime0*. *caltime0* may also be a number.

offset-time *caltime offset* Function

Returns the calendar time of *caltime* offset by *offset* number of seconds `(+ caltime offset)`.

3.11.1 Time Zone

(require 'time-zone)

TZ-string Data Format

POSIX standards specify several formats for encoding time-zone rules.

`:`<*pathname*>
: If the first character of <*pathname*> is '`/`', then <*pathname*> specifies the absolute pathname of a tzfile(5) format time-zone file. Otherwise, <*pathname*> is interpreted as a pathname within *tzfile:vicinity* (/usr/lib/zoneinfo/) naming a tzfile(5) format time-zone file.

<*std*><*offset*>
: The string <*std*> consists of 3 or more alphabetic characters. <*offset*> specifies the time difference from GMT. The <*offset*> is positive if the local time zone is west of the Prime Meridian and negative if it is east. <*offset*> can be the number of hours or hours and minutes (and optionally seconds) separated by '`:`'. For example, `-4:30`.

<*std*><*offset*><*dst*>
: <*dst*> is the at least 3 alphabetic characters naming the local daylight-savings-time.

<*std*><*offset*><*dst*><*doffset*>
: <*doffset*> specifies the offset from the Prime Meridian when daylight-savings-time is in effect.

The non-tzfile formats can optionally be followed by transition times specifying the day and time when a zone changes from standard to daylight-savings and back again.

`,<`*date*`>/<`*time*`>,<`*date*`>/<`*time*`>`
: The <*time*>s are specified like the <*offset*>s above, except that leading '`+`' and '`-`' are not allowed.

  Each <*date*> has one of the formats:

  `J<`*day*`>`
  : specifies the Julian day with <*day*> between 1 and 365. February 29 is never counted and cannot be referenced.

  `<`*day*`>`
  : This specifies the Julian day with n between 0 and 365. February 29 is counted in leap years and can be specified.

  `M<`*month*`>.<`*week*`>.<`*day*`>`
  : This specifies day <*day*> (0 <= <*day*> <= 6) of week <*week*> (1 <= <*week*> <= 5) of month <*month*> (1 <= <*month*> <= 12). Week 1 is the first week in which day d occurs and week 5 is the last week in which day <*day*> occurs. Day 0 is a Sunday.

time-zone Data Type

is a datatype encoding how many hours from Greenwich Mean Time the local time is, and the *Daylight Savings Time* rules for changing it.

time-zone *TZ-string* Function

Creates and returns a time-zone object specified by the string *TZ-string*. If `time-zone` cannot interpret *TZ-string*, `#f` is returned.

tz:params *caltime tz* Function

tz is a time-zone object. `tz:params` returns a list of three items:

0. An integer. 0 if standard time is in effect for timezone *tz* at *caltime*; 1 if daylight savings time is in effect for timezone *tz* at *caltime*.
1. The number of seconds west of the Prime Meridian timezone *tz* is at *caltime*.
2. The name for timezone *tz* at *caltime*.

`tz:params` is unaffected by the default timezone; inquiries can be made of any timezone at any calendar time.

The rest of these procedures and variables are provided for POSIX compatability. Because of shared state they are not thread-safe.

tzset Function

Returns the default time-zone.

tzset *tz* Function

Sets (and returns) the default time-zone to *tz*.

tzset *TZ-string* Function

Sets (and returns) the default time-zone to that specified by *TZ-string*.

`tzset` also sets the variables **timezone**, *daylight?*, and *tzname*. This function is automatically called by the time conversion procedures which depend on the time zone (see Section 3.11 [Time and Date], page 74).

timezone Variable

Contains the difference, in seconds, between Greenwich Mean Time and local standard time (for example, in the U.S. Eastern time zone (EST), timezone is 5*60*60). `*timezone*` is initialized by `tzset`.

daylight? Variable

is `#t` if the default timezone has rules for *Daylight Savings Time*. *Note: daylight?* does not tell you when Daylight Savings Time is in effect, just that the default zone sometimes has Daylight Savings Time.

tzname Variable

is a vector of strings. Index 0 has the abbreviation for the standard timezone; If *daylight?*, then index 1 has the abbreviation for the Daylight Savings timezone.

3.11.2 Posix Time

```
(require 'posix-time)
```

Calendar-Time Data Type

is a datatype encapsulating time.

Coordinated Universal Time Data Type

(abbreviated *UTC*) is a vector of integers representing time:

0. seconds (0 - 61)
1. minutes (0 - 59)
2. hours since midnight (0 - 23)
3. day of month (1 - 31)
4. month (0 - 11). Note difference from `decode-universal-time`.
5. the number of years since 1900. Note difference from `decode-universal-time`.
6. day of week (0 - 6)
7. day of year (0 - 365)
8. 1 for daylight savings, 0 for regular time

gmtime *caltime* Function

Converts the calendar time *caltime* to UTC and returns it.

localtime *caltime tz* Function

Returns *caltime* converted to UTC relative to timezone *tz*.

localtime *caltime* Function

converts the calendar time *caltime* to a vector of integers expressed relative to the user's time zone. `localtime` sets the variable **timezone** with the difference between Coordinated Universal Time (UTC) and local standard time in seconds (see Section 3.11.1 [Time Zone], page 74).

gmktime *univtime* Function

Converts a vector of integers in GMT Coordinated Universal Time (UTC) format to a calendar time.

mktime *univtime* Function

Converts a vector of integers in local Coordinated Universal Time (UTC) format to a calendar time.

mktime *univtime tz* Function

Converts a vector of integers in Coordinated Universal Time (UTC) format (relative to time-zone *tz*) to calendar time.

asctime *univtime* Function

Converts the vector of integers *caltime* in Coordinated Universal Time (UTC) format into a string of the form `"Wed Jun 30 21:49:08 1993"`.

gtime *caltime* Function
ctime *caltime* Function
ctime *caltime tz* Function

Equivalent to `(asctime (gmtime` *caltime*`))`, `(asctime (localtime` *caltime*`))`, and `(asctime (localtime` *caltime tz*`))`, respectively.

3.11.3 Common-Lisp Time

get-decoded-time Function

Equivalent to `(decode-universal-time (get-universal-time))`.

get-universal-time Function

Returns the current time as *Universal Time*, number of seconds since 00:00:00 Jan 1, 1900 GMT. Note that the reference time is different from `current-time`.

decode-universal-time *univtime* Function

Converts *univtime* to *Decoded Time* format. Nine values are returned:

0. seconds (0 - 61)
1. minutes (0 - 59)
2. hours since midnight
3. day of month
4. month (1 - 12). Note difference from `gmtime` and `localtime`.

5. year (A.D.). Note difference from `gmtime` and `localtime`.
6. day of week (0 - 6)
7. #t for daylight savings, #f otherwise
8. hours west of GMT (-24 - +24)

Notice that the values returned by `decode-universal-time` do not match the arguments to `encode-universal-time`.

encode-universal-time *second minute hour date month year* Function
encode-universal-time *second minute hour date month year time-zone* Function

Converts the arguments in Decoded Time format to Universal Time format. If *time-zone* is not specified, the returned time is adjusted for daylight saving time. Otherwise, no adjustment is performed.

Notice that the values returned by `decode-universal-time` do not match the arguments to `encode-universal-time`.

3.12 Schmooz

Schmooz is a simple, lightweight markup language for interspersing Texinfo documentation with Scheme source code. Schmooz does not create the top level Texinfo file; it creates '`txi`' files which can be imported into the documentation using the Texinfo command '`@include`'.

`(require 'schmooz)` defines the function `schmooz`, which is used to process files. Files containing schmooz documentation should not contain `(require 'schmooz)`.

schmooz *filename*.scm . . . Procedure

Filename.scm should be a string ending with '`.scm`' naming an existing file containing Scheme source code. `schmooz` extracts top-level comments containing schmooz commands from *filename*.scm and writes the converted Texinfo source to a file named *filename*.txi.

schmooz *filename*.texi . . . Procedure
schmooz *filename*.tex . . . Procedure
schmooz *filename*.txi . . . Procedure

Filename should be a string naming an existing file containing Texinfo source code. For every occurrence of the string '`@include` *filename*`.txi`' within that file, `schmooz` calls itself with the argument '*filename*`.scm`'.

Schmooz comments are distinguished (from non-schmooz comments) by their first line, which must start with an at-sign (`@`) preceded by one or more semicolons (`;`). A schmooz comment ends at the first subsequent line which does *not* start with a semicolon. Currently schmooz comments are recognized only at top level.

Schmooz comments are copied to the Texinfo output file with the leading contiguous semicolons removed. Certain character sequences starting with at-sign are treated specially. Others are copied unchanged.

A schmooz comment starting with '@body' must be followed by a Scheme definition. All comments between the '@body' line and the definition will be included in a Texinfo definition, either a '@defun' or a '@defvar', depending on whether a procedure or a variable is being defined.

Within the text of that schmooz comment, at-sign followed by '0' will be replaced by @code{procedure-name} if the following definition is of a procedure; or @var{variable} if defining a variable.

An at-sign followed by a non-zero digit will expand to the variable citation of that numbered argument: '@var{argument-name}'.

If more than one definition follows a '@body' comment line without an intervening blank or comment line, then those definitions will be included in the same Texinfo definition using '@defvarx' or '@defunx', depending on whether the first definition is of a variable or of a procedure.

Schmooz can figure out whether a definition is of a procedure if it is of the form:

'(define (<identifier> <arg> ...) <expression>)'

or if the left hand side of the definition is some form ending in a lambda expression. Obviously, it can be fooled. In order to force recognition of a procedure definition, start the documentation with '@args' instead of '@body'. '@args' should be followed by the argument list of the function being defined, which may be enclosed in parentheses and delimited by whitespace, (as in Scheme), enclosed in braces and separated by commas, (as in Texinfo), or consist of the remainder of the line, separated by whitespace.

For example:

```
;;@args arg1 args ...
;;@0 takes argument @1 and any number of @2
(define myfun (some-function-returning-magic))
```

Will result in:

```
@defun myfun arg1 args @dots{}

@code{myfun} takes argument @var{arg1} and any number of @var{args}
@end defun
```

'@args' may also be useful for indicating optional arguments by name. If '@args' occurs inside a schmooz comment section, rather than at the beginning, then it will generate a '@defunx' line with the arguments supplied.

If the first at-sign in a schmooz comment is immediately followed by whitespace, then the comment will be expanded to whatever follows that whitespace. If the at-sign is followed by a non-whitespace character then the at-sign will be included as the first character of the expansion. This feature is intended to make it easy to include Texinfo directives in schmooz comments.

4 Mathematical Packages

4.1 Bit-Twiddling

```
(require 'logical)
```

The bit-twiddling functions are made available through the use of the `logical` package. `logical` is loaded by inserting `(require 'logical)` before the code that uses these functions. These functions behave as though operating on integers in two's-complement representation.

4.1.1 Bitwise Operations

logand *n1 n1* Function

Returns the integer which is the bit-wise AND of the two integer arguments.

Example:

```
(number->string (logand #b1100 #b1010) 2)
   ⇒ "1000"
```

logior *n1 n2* Function

Returns the integer which is the bit-wise OR of the two integer arguments.

Example:

```
(number->string (logior #b1100 #b1010) 2)
   ⇒ "1110"
```

logxor *n1 n2* Function

Returns the integer which is the bit-wise XOR of the two integer arguments.

Example:

```
(number->string (logxor #b1100 #b1010) 2)
   ⇒ "110"
```

lognot *n* Function

Returns the integer which is the 2s-complement of the integer argument.

Example:

```
(number->string (lognot #b10000000) 2)
   ⇒ "-10000001"
(number->string (lognot #b0) 2)
   ⇒ "-1"
```

bitwise-if *mask n0 n1* Function

Returns an integer composed of some bits from integer *n0* and some from integer *n1*. A bit of the result is taken from *n0* if the corresponding bit of integer *mask* is 1 and from *n1* if that bit of *mask* is 0.

logtest *j k* Function

```
(logtest j k) ≡ (not (zero? (logand j k)))

(logtest #b0100 #b1011) ⇒ #f
(logtest #b0100 #b0111) ⇒ #t
```

logcount *n* Function

Returns the number of bits in integer *n*. If integer is positive, the 1-bits in its binary representation are counted. If negative, the 0-bits in its two's-complement binary representation are counted. If 0, 0 is returned.

Example:

```
(logcount #b10101010)
   ⇒ 4
(logcount 0)
   ⇒ 0
(logcount -2)
   ⇒ 1
```

4.1.2 Bit Within Word

logbit? *index j* Function

```
(logbit? index j) ≡ (logtest (integer-expt 2 index) j)

(logbit? 0 #b1101) ⇒ #t
(logbit? 1 #b1101) ⇒ #f
(logbit? 2 #b1101) ⇒ #t
(logbit? 3 #b1101) ⇒ #t
(logbit? 4 #b1101) ⇒ #f
```

copy-bit *index from bit* Function

Returns an integer the same as *from* except in the *index*th bit, which is 1 if *bit* is `#t` and 0 if *bit* is `#f`.

Example:

```
(number->string (copy-bit 0 0 #t) 2)       ⇒ "1"
(number->string (copy-bit 2 0 #t) 2)       ⇒ "100"
(number->string (copy-bit 2 #b1111 #f) 2)  ⇒ "1011"
```

4.1.3 Fields of Bits

bit-field *n start end* Function

Returns the integer composed of the *start* (inclusive) through *end* (exclusive) bits of *n*. The *start*th bit becomes the 0-th bit in the result.

This function was called `bit-extract` in previous versions of SLIB.

Example:

```
(number->string (bit-field #b1101101010 0 4) 2)
   ⇒ "1010"
(number->string (bit-field #b1101101010 4 9) 2)
   ⇒ "10110"
```

copy-bit-field *to start end from* Function

Returns an integer the same as *to* except possibly in the *start* (inclusive) through *end* (exclusive) bits, which are the same as those of *from*. The 0-th bit of *from* becomes the *start*th bit of the result.

Example:

```
(number->string (copy-bit-field #b1101101010 0 4 0) 2)
        ⇒ "1101100000"
(number->string (copy-bit-field #b1101101010 0 4 -1) 2)
        ⇒ "1101101111"
```

ash *int count* Function

Returns an integer equivalent to `(inexact->exact (floor (*` *int* `(expt 2` *count*`))))`.

Example:

```
(number->string (ash #b1 3) 2)
   ⇒ "1000"
(number->string (ash #b1010 -1) 2)
   ⇒ "101"
```

integer-length *n* Function

Returns the number of bits neccessary to represent *n*.

Example:

```
(integer-length #b10101010)
   ⇒ 8
(integer-length 0)
   ⇒ 0
(integer-length #b1111)
   ⇒ 4
```

integer-expt *n k* Function

Returns *n* raised to the non-negative integer exponent *k*.

Example:

```
(integer-expt 2 5)
   ⇒ 32
(integer-expt -3 3)
   ⇒ -27
```

4.1.4 Bit order and Lamination

bit-reverse *k n* Function

Returns the low-order *k* bits of *n* with the bit order reversed. The low-order bit of *n* is the high order bit of the returned value.

```
(number->string (bit-reverse 8 #xa7) 16)
  ⇒ "e5"
```

integer->list *k len* Function
integer->list *k* Function

`integer->list` returns a list of *len* booleans corresponding to each bit of the given integer. #t is coded for each 1; #f for 0. The *len* argument defaults to `(integer-length` *k*`)`.

list->integer *list* Function

`list->integer` returns an integer formed from the booleans in the list *list*, which must be a list of booleans. A 1 bit is coded for each #t; a 0 bit for #f.

`integer->list` and `list->integer` are inverses so far as `equal?` is concerned.

booleans->integer *bool1 . . .* Function

Returns the integer coded by the *bool1* . . . arguments.

bitwise:laminate *k1 . . .* Function

Returns an integer composed of the bits of *k1* . . . interlaced in argument order. Given *k1*, . . . *kn*, the n low-order bits of the returned value will be the lowest-order bit of each argument.

bitwise:delaminate *count k* Function

Returns a list of *count* integers comprised of every *count*h bit of the integer *k*.

For any non-negative integers *k* and *count*:

```
(eqv? k (bitwise:laminate (bitwise:delaminate count k)))
```

4.1.5 Gray code

A *Gray code* is an ordering of non-negative integers in which exactly one bit differs between each pair of successive elements. There are multiple Gray codings. An n-bit Gray code corresponds to a Hamiltonian cycle on an n-dimensional hypercube.

Gray codes find use communicating incrementally changing values between asynchronous agents. De-laminated Gray codes comprise the coordinates of Hilbert's space-filling curves.

integer->gray-code *k* Function

Converts *k* to a Gray code of the same `integer-length` as *k*.

gray-code->integer *k* Function

Converts the Gray code *k* to an integer of the same `integer-length` as *k*.

For any non-negative integer *k*,

```
(eqv? k (gray-code->integer (integer->gray-code k)))
```

`=` *k1 k2* Function
gray-code<? *k1 k2* Function
gray-code>? *k1 k2* Function
gray-code<=? *k1 k2* Function
gray-code>=? *k1 k2* Function

These procedures return #t if their Gray code arguments are (respectively): equal, monotonically increasing, monotonically decreasing, monotonically nondecreasing, or monotonically nonincreasing.

For any non-negative integers *k1* and *k2*, the Gray code predicate of `(integer->gray-code k1)` and `(integer->gray-code k2)` will return the same value as the corresponding predicate of *k1* and *k2*.

4.2 Modular Arithmetic

```
(require 'modular)
```

mod *x1 x2* Function
rem *x1 x2* Function

These procedures implement the Common-Lisp functions of the same names. The real number *x2* must be non-zero. `mod` returns `(- x1 (* x2 (floor (/ x1 x2))))`. `rem` returns `(- x1 (* x2 (truncate (/ x1 x2))))`.

If *x1* and *x2* are integers, then `mod` behaves like `modulo` and `rem` behaves like `remainder`.

```
(mod -90 360)                    ⇒ 270
(rem -90 180)                    ⇒ -90

(mod 540 360)                    ⇒ 180
(rem 540 360)                    ⇒ 180

(mod (* 5/2 pi) (* 2 pi))        ⇒ 1.5707963267948965
(rem (* -5/2 pi) (* 2 pi))       ⇒ -1.5707963267948965
```

extended-euclid *n1 n2* Function

Returns a list of 3 integers `(d x y)` such that d = gcd(*n1*, *n2*) = *n1* * x + *n2* * y.

symmetric:modulus *n* Function

Returns `(quotient (+ -1 n) -2)` for positive odd integer *n*.

modulus->integer *modulus* Function

Returns the non-negative integer characteristic of the ring formed when *modulus* is used with `modular:` procedures.

modular:normalize *modulus n* Function

Returns the integer `(modulo` *n* `(modulus->integer` *modulus*`))` in the representation specified by *modulus*.

The rest of these functions assume normalized arguments; That is, the arguments are constrained by the following table:

For all of these functions, if the first argument (*modulus*) is:

`positive?`
: Work as before. The result is between 0 and *modulus*.

`zero?`
: The arguments are treated as integers. An integer is returned.

`negative?`
: The arguments and result are treated as members of the integers modulo `(+ 1 (* -2` *modulus*`))`, but with *symmetric* representation; i.e. `(<= (-` *modulus*`)` *n* *modulus*`)`.

If all the arguments are fixnums the computation will use only fixnums.

modular:invertable? *modulus k* Function

Returns `#t` if there exists an integer n such that $k * n \equiv 1 \bmod modulus$, and `#f` otherwise.

modular:invert *modulus n2* Function

Returns an integer n such that 1 = (n * *n2*) mod *modulus*. If *n2* has no inverse mod *modulus* an error is signaled.

modular:negate *modulus n2* Function

Returns (−*n2*) mod *modulus*.

modular:+ *modulus n2 n3* Function

Returns (*n2* + *n3*) mod *modulus*.

modular:− *modulus n2 n3* Function

Returns (*n2* − *n3*) mod *modulus*.

modular:* *modulus n2 n3* Function

Returns (*n2* * *n3*) mod *modulus*.

The Scheme code for `modular:*` with negative *modulus* is not completed for fixnum-only implementations.

modular:expt *modulus n2 n3* Function

Returns (*n2* ^ *n3*) mod *modulus*.

4.3 Prime Numbers

```
(require 'factor)
```

prime:prngs Variable

prime:prngs is the random-state (see Section 4.4 [Random Numbers], page 86) used by these procedures. If you call these procedures from more than one thread (or from interrupt), `random` may complain about reentrant calls.

Note: The prime test and generation procedures implement (or use) the Solovay-Strassen primality test. See

- Robert Solovay and Volker Strassen, *A Fast Monte-Carlo Test for Primality*, SIAM Journal on Computing, 1977, pp 84-85.

jacobi-symbol *p q* Function

Returns the value (+1, −1, or 0) of the Jacobi-Symbol of exact non-negative integer *p* and exact positive odd integer *q*.

prime:trials Variable

prime:trials the maxinum number of iterations of Solovay-Strassen that will be done to test a number for primality.

prime? *n* Function

Returns `#f` if *n* is composite; `#t` if *n* is prime. There is a slight chance `(expt 2 (- prime:trials))` that a composite will return `#t`.

primes< *start count* Function

Returns a list of the first *count* prime numbers less than *start*. If there are fewer than *count* prime numbers less than *start*, then the returned list will have fewer than *start* elements.

primes> *start count* Function

Returns a list of the first *count* prime numbers greater than *start*.

factor *k* Function

Returns a list of the prime factors of *k*. The order of the factors is unspecified. In order to obtain a sorted list do `(sort! (factor k) <)`.

4.4 Random Numbers

```
(require 'random)
```

A pseudo-random number generator is only as good as the tests it passes. George Marsaglia of Florida State University developed a battery of tests named *DIEHARD* (

`http://stat.fsu.edu/~geo/diehard.html`). '`diehard.c`' has a bug which the patch `http://swissnet.ai.mit.edu/ftpdir/users/jaffer/diehard.c.pat` corrects.

SLIB's new PRNG generates 8 bits at a time. With the degenerate seed '`0`', the numbers generated pass DIEHARD; but when bits are combined from sequential bytes, tests fail. With the seed '`http://swissnet.ai.mit.edu/~jaffer/SLIB.html`', all of those tests pass.

random *n* Function
random *n state* Function

Accepts a positive integer or real *n* and returns a number of the same type between zero (inclusive) and *n* (exclusive). The values returned by `random` are uniformly distributed from 0 to *n*.

The optional argument *state* must be of the type returned by `(seed->random-state)` or `(make-random-state)`. It defaults to the value of the variable `*random-state*`. This object is used to maintain the state of the pseudo-random-number generator and is altered as a side effect of calls to `random`.

random-state Variable

Holds a data structure that encodes the internal state of the random-number generator that `random` uses by default. The nature of this data structure is implementation-dependent. It may be printed out and successfully read back in, but may or may not function correctly as a random-number state object in another implementation.

copy-random-state *state* Function

Returns a new copy of argument *state*.

copy-random-state Function

Returns a new copy of `*random-state*`.

seed->random-state *seed* Function

Returns a new object of type suitable for use as the value of the variable `*random-state*` or as a second argument to `random`. The number or string *seed* is used to initialize the state. If `seed->random-state` is called twice with arguments which are `equal?`, then the returned data structures will be `equal?`. Calling `seed->random-state` with unequal arguments will nearly always return unequal states.

make-random-state Function
make-random-state *obj* Function

Returns a new object of type suitable for use as the value of the variable `*random-state*` or as a second argument to `random`. If the optional argument *obj* is given, it should be a printable Scheme object; the first 50 characters of its printed representation will be used as the seed. Otherwise the value of `*random-state*` is used as the seed.

If inexact numbers are supported by the Scheme implementation, '`randinex.scm`' will be loaded as well. '`randinex.scm`' contains procedures for generating inexact distributions.

random:uniform Function
random:uniform *state* Function
Returns an uniformly distributed inexact real random number in the range between 0 and 1.

random:exp Function
random:exp *state* Function
Returns an inexact real in an exponential distribution with mean 1. For an exponential distribution with mean *u* use `(*` *u* `(random:exp))`.

random:normal Function
random:normal *state* Function
Returns an inexact real in a normal distribution with mean 0 and standard deviation 1. For a normal distribution with mean *m* and standard deviation *d* use `(+` *m* `(*` *d* `(random:normal)))`.

random:normal-vector! *vect* Function
random:normal-vector! *vect state* Function
Fills *vect* with inexact real random numbers which are independent and standard normally distributed (i.e., with mean 0 and variance 1).

random:hollow-sphere! *vect* Function
random:hollow-sphere! *vect state* Function
Fills *vect* with inexact real random numbers the sum of whose squares is equal to 1.0. Thinking of *vect* as coordinates in space of dimension n = `(vector-length` *vect*`)`, the coordinates are uniformly distributed over the surface of the unit n-shere.

random:solid-sphere! *vect* Function
random:solid-sphere! *vect state* Function
Fills *vect* with inexact real random numbers the sum of whose squares is less than 1.0. Thinking of *vect* as coordinates in space of dimension n = `(vector-length` *vect*`)`, the coordinates are uniformly distributed within the unit n-shere. The sum of the squares of the numbers is returned.

4.5 Fast Fourier Transform

```
(require 'fft)
```

fft *array* Function
array is an array of `(expt 2 n)` numbers. `fft` returns an array of complex numbers comprising the *Discrete Fourier Transform* of *array*.

fft-1 *array* Function
`fft-1` returns an array of complex numbers comprising the inverse Discrete Fourier Transform of *array*.

`(fft-1 (fft` *array*`))` will return an array of values close to *array*.

```
(fft '#(1 0+i -1 0-i 1 0+i -1 0-i)) ⇒

#(0.0 0.0 0.0+628.0783185208527e-18i 0.0
  0.0 0.0 8.0-628.0783185208527e-18i 0.0)

(fft-1 '#(0 0 0 0 0 0 8 0)) ⇒

#(1.0 -61.23031769111886e-18+1.0i -1.0 61.23031769111886e-18-1.0i
  1.0 -61.23031769111886e-18+1.0i -1.0 61.23031769111886e-18-1.0i)
```

4.6 Cyclic Checksum

`(require 'make-crc)`

make-port-crc — Function

make-port-crc *degree* — Function

Returns an expression for a procedure of one argument, a port. This procedure reads characters from the port until the end of file and returns the integer checksum of the bytes read.

The integer *degree*, if given, specifies the degree of the polynomial being computed – which is also the number of bits computed in the checksums. The default value is 32.

make-port-crc *generator* — Function

The integer *generator* specifies the polynomial being computed. The power of 2 generating each 1 bit is the exponent of a term of the polynomial. The value of *generator* must be larger than 127.

make-port-crc *degree generator* — Function

The integer *generator* specifies the polynomial being computed. The power of 2 generating each 1 bit is the exponent of a term of the polynomial. The bit at position *degree* is implicit and should not be part of *generator*. This allows systems with numbers limited to 32 bits to calculate 32 bit checksums. The default value of *generator* when *degree* is 32 (its default) is:

```
(make-port-crc 32 #b00000100110000010001110110110111)
```

Creates a procedure to calculate the P1003.2/D11.2 (POSIX.2) 32-bit checksum from the polynomial:

```
     32    26    23    22    16    12    11
  ( x   + x   + x   + x   + x   + x   + x   +

      10    8    7    5    4    2    1
     x   + x  + x  + x  + x  + x  + x  + 1 )  mod 2
```

```
(require 'make-crc)
(define crc32 (slib:eval (make-port-crc)))
(define (file-check-sum file) (call-with-input-file file crc32))
(file-check-sum (in-vicinity (library-vicinity) "ratize.scm"))
```

```
⇒ 157103930
```

4.7 Plotting

```
(require 'charplot)
```

charplot:dimensions Variable

A list of the maximum height (number of lines) and maximum width (number of columns) for the graph, its scales, and labels.

The default value for *charplot:dimensions* is the `output-port-height` and `output-port-width` of `current-output-port`.

plot *coords x-label y-label* Procedure

coords is a list or vector of coordinates, lists of x and y coordinates. *x-label* and *y-label* are strings with which to label the x and y axes.

Example:

```
(require 'charplot)
(set! charplot:dimensions '(19 45))

(define (make-points n)
  (if (zero? n)
      '()
      (cons (cons (/ n 6) (sin (/ n 6))) (make-points (1- n)))))

(plot (make-points 37) "x" "Sin(x)")
⊣
```

```
Sin(x)   ____________________________________________
    1.25|-                                           |
        |                                            |
       1|-       ****                                |
        |      **    **                              |
    0.75|-    *        *                             |
        |    *          *                            |
     0.5|-  *            *                           |
        |  *                                         |
    0.25|-                *                          |
        | *                *                         |
       0|-------------------*------------------------|
        |                                     *      |
   -0.25|-                   *               *       |
        |                     *             *        |
    -0.5|-                     *                     |
        |                       *          *         |
   -0.75|-                       *        *          |
        |                         **    **           |
      -1|-                          ****             |
        |:_____._____:_____._____:_____._____:_____._|
   x                 2           4           6
```

plot *func x1 x2* Procedure
plot *func x1 x2 npts* Procedure

Plots the function of one argument *func* over the range *x1* to *x2*. If the optional integer argument *npts* is supplied, it specifies the number of points to evaluate *func* at.

```
(plot sin 0 (* 2 pi))
⊣
```

```
      --------------------------------------------
 1.25|-:                                          |
     | :                                          |
    1|-:       ****                               |
     | :     **    **                             |
 0.75|-:    *        *                            |
     | :   *          *                           |
  0.5|-:  **          **                          |
     | : *             *                          |
 0.25|-:**              **                        |
     | :*                *                        |
    0|-*------------------*-----------------------|
     | :                  *                 *     |
-0.25|-:                   **              **     |
     | :                    *             *       |
 -0.5|-:                     *           **       |
     | :                      *          *        |
-0.75|-:                       *       **         |
     | :                        **    **          |
   -1|-:                          ****            |
     |_:_____._____:_____._____:_____._____:_____.|
       0           2           4           6
```

histograph *data label* Procedure

Creates and displays a histogram of the numerical values contained in vector or list *data*

```
(require 'random)
(histograph (do ((idx 99 (+ -1 idx))
                 (lst '() (cons (* .02 (random:normal)) lst)))
                ((negative? idx) lst))
            "normal")
⊣
```

```
         --------------------------------------------
       9|-                  :                        |
        |                   :                        |
       8|-               I  :                        |
        |                I  :                        |
       7|-             III II I                      |
        |              III II I                      |
       6|-             III II I   I                  |
        |              III II I   I                  |
       5|-             III II I  III                 |
        |              III II I  III                 |
       4|-            IIII IIIII III                 |
        |             IIII IIIII III                 |
       3|-            IIII IIIIIIIII                 |
        |             IIII IIIIIIIII                 |
       2|-         IIIIIII IIIIIIIII     I I I       |
        |          IIIIIII IIIIIIIII     I I I       |
       1|-II I III IIIIIII IIIIIIIIIII IIIIIII       |
        | II I III IIIIIII IIIIIIIIIII IIIIIII       |
       0|-IIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIII-------|
        |____.____:____.____:____.____:____.____:____|
normal          -0.025      0       0.025      0.05
```

4.8 Solid Modeling

    (require 'solid)

`http://swissnet.ai.mit.edu/~jaffer/Solid/#Example` gives an example use of this package.

vrml *node . . .* Function

Returns the VRML97 string (including header) of the concatenation of strings *nodes*, ...

vrml-append *node1 node2 . . .* Function

Returns the concatenation with interdigitated newlines of strings *node1*, *node2*, ...

vrml-to-file *file node . . .* Function

Writes to file named *file* the VRML97 string (including header) of the concatenation of strings *nodes*, ...

world:info *title info . . .* Function

Returns a VRML97 string setting the title of the file in which it appears to *title*. Additional strings *info*, . . . are comments.

VRML97 strings passed to `vrml` and `vrml-to-file` as arguments will appear in the resulting VRML code. This string turns off the headlight at the viewpoint:

```
" NavigationInfo {headlight FALSE}"
```

scene:panorama *front right back left top bottom* — Function

Specifies the distant images on the inside faces of the cube enclosing the virtual world.

scene:sphere *colors angles* — Function

colors is a list of color objects. Each may be of type Section 4.9.1 [Color Data-Type], page 99, a 24-bit sRGB integer, or a list of 3 numbers between 0.0 and 1.0.

angles is a list of non-increasing angles the same length as *colors*. Each angle is between 90 and -90 degrees. If 90 or -90 are not elements of *angles*, then the color at the zenith and nadir are taken from the colors paired with the angles nearest them.

`scene:sphere` fills horizontal bands with interpolated colors on the backgroud sphere encasing the world.

scene:sky-and-dirt — Function

Returns a blue and brown backgroud sphere encasing the world.

scene:sky-and-grass — Function

Returns a blue and green backgroud sphere encasing the world.

scene:sun *latitude julian-day hour turbidity strength* — Function
scene:sun *latitude julian-day hour turbidity* — Function

latitude is the virtual place's latitude in degrees. *julian-day* is an integer from 0 to 366, the day of the year. *hour* is a real number from 0 to 24 for the time of day; 12 is noon. *turbidity* is the degree of fogginess described in See Section 4.9.7 [Daylight], page 113.

`scene:sun` returns a bright yellow, distant sphere where the sun would be at *hour* on *julian-day* at *latitude*. If *strength* is positive, included is a light source of *strength* (default 1).

scene:overcast *latitude julian-day hour turbidity strength* — Function
scene:overcast *latitude julian-day hour turbidity* — Function

latitude is the virtual place's latitude in degrees. *julian-day* is an integer from 0 to 366, the day of the year. *hour* is a real number from 0 to 24 for the time of day; 12 is noon. *turbidity* is the degree of cloudiness described in See Section 4.9.7 [Daylight], page 113.

`scene:overcast` returns an overcast sky as it might look at *hour* on *julian-day* at *latitude*. If *strength* is positive, included is an ambient light source of *strength* (default 1).

Viewpoints are objects in the virtual world, and can be transformed individually or with solid objects.

scene:viewpoint *name distance compass pitch* Function
scene:viewpoint *name distance compass* Function

Returns a viewpoint named *name* facing the origin and placed *distance* from it. *compass* is a number from 0 to 360 giving the compass heading. *pitch* is a number from -90 to 90, defaulting to 0, specifying the angle from the horizontal.

scene:viewpoints *proximity* Function

Returns 6 viewpoints, one at the center of each face of a cube with sides 2 * *proximity*, centered on the origin.

Light Sources

In VRML97, lights shine only on objects within the same children node and descendants of that node. Although it would have been convenient to let light direction be rotated by `solid:rotation`, this restricts a rotated light's visibility to objects rotated with it.

To workaround this limitation, these directional light source procedures accept either Cartesian or spherical coordinates for direction. A spherical coordinate is a list `(`*theta azimuth*`)`; where *theta* is the angle in degrees from the zenith, and *azimuth* is the angle in degrees due west of south.

It is sometimes useful for light sources to be brighter than '`1`'. When *intensity* arguments are greater than 1, these functions gang multiple sources to reach the desired strength.

light:ambient *color intensity* Function
light:ambient *color* Function

Ambient light shines on all surfaces with which it is grouped.

color is a an object of type Section 4.9.1 [Color Data-Type], page 99, a 24-bit sRGB integer, or a list of 3 numbers between 0.0 and 1.0. If *color* is #f, then the default color will be used. *intensity* is a real non-negative number defaulting to '`1`'.

`light:ambient` returns a light source or sources of *color* with total strength of *intensity* (or 1 if omitted).

light:directional *color direction intensity* Function
light:directional *color direction* Function
light:directional *color* Function

Directional light shines parallel rays with uniform intensity on all objects with which it is grouped.

color is a an object of type Section 4.9.1 [Color Data-Type], page 99, a 24-bit sRGB integer, or a list of 3 numbers between 0.0 and 1.0. If *color* is #f, then the default color will be used.

direction must be a list or vector of 2 or 3 numbers specifying the direction to this light. If *direction* has 2 numbers, then these numbers are the angle from zenith and the azimuth in degrees; if *direction* has 3 numbers, then these are taken as a Cartesian vector specifying the direction to the light source. The default direction is upwards; thus its light will shine down.

intensity is a real non-negative number defaulting to '`1`'.

`light:directional` returns a light source or sources of *color* with total strength of *intensity*, shining from *direction*.

light:beam *attenuation radius aperture peak* Function
light:beam *attenuation radius aperture* Function
light:beam *attenuation radius* Function
light:beam *attenuation* Function

attenuation is a list or vector of three nonnegative real numbers specifying the reduction of intensity, the reduction of intensity with distance, and the reduction of intensity as the square of distance. *radius* is the distance beyond which the light does not shine. *radius* defaults to '`100`'.

aperture is a real number between 0 and 180, the angle centered on the light's axis through which it sheds some light. *peak* is a real number between 0 and 90, the angle of greatest illumination.

light:point *location color intensity beam* Function
light:point *location color intensity* Function
light:point *location color* Function
light:point *location* Function

Point light radiates from *location*, intensity decreasing with distance, towards all objects with which it is grouped.

color is a an object of type Section 4.9.1 [Color Data-Type], page 99, a 24-bit sRGB integer, or a list of 3 numbers between 0.0 and 1.0. If *color* is #f, then the default color will be used. *intensity* is a real non-negative number defaulting to '`1`'. *beam* is a structure returned by `light:beam` or #f.

`light:point` returns a light source or sources at *location* of *color* with total strength *intensity* and *beam* properties. Note that the pointlight itself is not visible. To make it so, place an object with emissive appearance at *location*.

light:spot *location direction color intensity beam* Function
light:spot *location direction color intensity* Function
light:spot *location direction color* Function
light:spot *location direction* Function
light:spot *location* Function

Spot light radiates from *location* towards *direction*, intensity decreasing with distance, illuminating objects with which it is grouped.

direction must be a list or vector of 2 or 3 numbers specifying the direction to this light. If *direction* has 2 numbers, then these numbers are the angle from zenith and the azimuth in degrees; if *direction* has 3 numbers, then these are taken as a Cartesian vector specifying the direction to the light source. The default direction is upwards; thus its light will shine down.

color is a an object of type Section 4.9.1 [Color Data-Type], page 99, a 24-bit sRGB integer, or a list of 3 numbers between 0.0 and 1.0. If *color* is #f, then the default color will be used.

intensity is a real non-negative number defaulting to '`1`'.

`light:spot` returns a light source or sources at *location* of *direction* with total strength *color*. Note that the spotlight itself is not visible. To make it so, place an object with emissive appearance at *location*.

Object Primitives

solid:box *geometry appearance* Function
solid:box *geometry* Function

geometry must be a number or a list or vector of three numbers. If *geometry* is a number, the `solid:box` returns a cube with sides of length *geometry* centered on the origin. Otherwise, `solid:box` returns a rectangular box with dimensions *geometry* centered on the origin. *appearance* determines the surface properties of the returned object.

solid:cylinder *radius height appearance* Function
solid:cylinder *radius height* Function

Returns a right cylinder with dimensions *radius* and `(abs` *height*`)` centered on the origin. If *height* is positive, then the cylinder ends will be capped. *appearance* determines the surface properties of the returned object.

solid:disk *radius thickness appearance* Function
solid:disk *radius thickness* Function

thickness must be a positive real number. `solid:disk` returns a circular disk with dimensions *radius* and *thickness* centered on the origin. *appearance* determines the surface properties of the returned object.

solid:cone *radius height appearance* Function
solid:cone *radius height* Function

Returns an isosceles cone with dimensions *radius* and *height* centered on the origin. *appearance* determines the surface properties of the returned object.

solid:pyramid *side height appearance* Function
solid:pyramid *side height* Function

Returns an isosceles pyramid with dimensions *side* and *height* centered on the origin. *appearance* determines the surface properties of the returned object.

solid:sphere *radius appearance* Function
solid:sphere *radius* Function

Returns a sphere of radius *radius* centered on the origin. *appearance* determines the surface properties of the returned object.

solid:ellipsoid *geometry appearance* Function
solid:ellipsoid *geometry* Function

geometry must be a number or a list or vector of three numbers. If *geometry* is a number, the `solid:ellipsoid` returns a sphere of diameter *geometry* centered on the origin. Otherwise, `solid:ellipsoid` returns an ellipsoid with diameters *geometry* centered on the origin. *appearance* determines the surface properties of the returned object.

Surface Attributes

solid:color *diffuseColor ambientIntensity specularColor shininess emissiveColor transparency* — Function
solid:color *diffuseColor ambientIntensity specularColor shininess emissiveColor* — Function
solid:color *diffuseColor ambientIntensity specularColor shininess* — Function
solid:color *diffuseColor ambientIntensity specularColor* — Function
solid:color *diffuseColor ambientIntensity* — Function
solid:color *diffuseColor* — Function

Returns an *appearance*, the optical properties of the objects with which it is associated. *ambientIntensity*, *shininess*, and *transparency* must be numbers between 0 and 1. *diffuseColor*, *specularColor*, and *emissiveColor* are objects of type Section 4.9.1 [Color Data-Type], page 99, 24-bit sRGB integers or lists of 3 numbers between 0.0 and 1.0. If a color argument is omitted or #f, then the default color will be used.

solid:texture *image color scale rotation center translation* — Function
solid:texture *image color scale rotation center* — Function
solid:texture *image color scale rotation* — Function
solid:texture *image color scale* — Function
solid:texture *image color* — Function
solid:texture *image* — Function

Returns an *appearance*, the optical properties of the objects with which it is associated. *image* is a string naming a JPEG or PNG image resource. *color* is #f, a color, or the string returned by `solid:color`. The rest of the optional arguments specify 2-dimensional transforms applying to the *image*.

scale must be #f, a number, or list or vector of 2 numbers specifying the scale to apply to *image*. *rotation* must be #f or the number of degrees to rotate *image*. *center* must be #f or a list or vector of 2 numbers specifying the center of *image* relative to the *image* dimensions. *translation* must be #f or a list or vector of 2 numbers specifying the translation to apply to *image*.

Aggregating Objects

solid:center-row-of *number solid spacing* — Function

Returns a row of *number solid* objects spaced evenly *spacing* apart.

solid:center-array-of *number-a number-b solid spacing-a spacing-b* — Function

Returns *number-b* rows, *spacing-b* apart, of *number-a solid* objects *spacing-a* apart.

solid:center-pile-of *number-a number-b number-c solid spacing-a spacing-b spacing-c* — Function

Returns *number-c* planes, *spacing-c* apart, of *number-b* rows, *spacing-b* apart, of *number-a solid* objects *spacing-a* apart.

solid:arrow *center* — Function

center must be a list or vector of three numbers. Returns an upward pointing metallic arrow centered at *center*.

solid:arrow — Function

Returns an upward pointing metallic arrow centered at the origin.

Spatial Transformations

solid:translation *center solid . . .* — Function

center must be a list or vector of three numbers. `solid:translation` Returns an aggregate of *solids*, . . . with their origin moved to *center*.

solid:scale *scale solid . . .* — Function

scale must be a number or a list or vector of three numbers. `solid:scale` Returns an aggregate of *solids*, . . . scaled per *scale*.

solid:rotation *axis angle solid . . .* — Function

axis must be a list or vector of three numbers. `solid:rotation` Returns an aggregate of *solids*, . . . rotated *angle* degrees around the axis *axis*.

4.9 Color

```
http://swissnet.ai.mit.edu/~jaffer/Color
```

The goals of this package are to provide methods to specify, compute, and transform colors in a core set of additive color spaces. The color spaces supported should be sufficient for working with the color data encountered in practice and the literature.

4.9.1 Color Data-Type

```
(require 'color)
```

color? *obj* — Function

Returns #t if *obj* is a color.

color? *obj typ* — Function

Returns #t if *obj* is a color of color-space *typ*. The symbol *typ* must be one of:

- CIEXYZ
- RGB709

- L*a*b*
- L*u*v*
- sRGB
- e-sRGB
- L*C*h

make-color *space arg . . .* Function

Returns a color of type *space*.

color-space *color* Function

Returns the symbol for the color-space in which *color* is embedded.

color-precision *color* Function

For colors in digital color-spaces, `color-precision` returns the number of bits used for each of the R, G, and B channels of the encoding. Otherwise, `color-precision` returns #f

color-white-point *color* Function

Returns the white-point of *color* in all color-spaces except CIEXYZ.

convert-color *color space white-point* Function
convert-color *color space* Function
convert-color *color e-sRGB precision* Function

Converts *color* into *space* at optional *white-point*.

4.9.1.1 External Representation

Each color encoding has an external, case-insensitive representation. To ensure portability, the white-point for all color strings is D65.[1]

Color Space	External Representation
CIEXYZ	CIEXYZ:<*X*>/<*Y*>/<*Z*>
RGB709	RGBi:<*R*>/<*G*>/<*B*>
L*a*b*	CIELAB:<*L*>/<*a*>/<*b*>
L*u*v*	CIELuv:<*L*>/<*u*>/<*v*>
L*C*h	CIELCh:<*L*>/<*C*>/<*h*>

The X, Y, Z, L, a, b, u, v, C, h, R, G, and B fields are (Scheme) real numbers within the appropriate ranges.

[1] Readers may recognize these color string formats from Xlib. X11's color management system was doomed by its fiction that CRT monitors' (and X11 default) color-spaces were linear RGBi. Unable to shed this legacy, the only practical way to view pictures on X is to ignore its color management system and use an sRGB monitor. In this implementation the device-independent RGB709 and sRGB spaces replace the device-dependent RGBi and RGB spaces of Xlib.

Color Space	External Representation
sRGB	sRGB:<*R*>/<*G*>/<*B*>
e-sRGB10	e-sRGB10:<*R*>/<*G*>/<*B*>
e-sRGB12	e-sRGB12:<*R*>/<*G*>/<*B*>
e-sRGB16	e-sRGB16:<*R*>/<*G*>/<*B*>

The *R*, *G*, and *B*, fields are non-negative exact decimal integers within the appropriate ranges.

Several additional syntaxes are supported by `string->color`:

Color Space	External Representation
sRGB	sRGB:<*RRGGBB*>
sRGB	#<*RRGGBB*>
sRGB	0x<*RRGGBB*>
sRGB	#x<*RRGGBB*>

Where *RRGGBB* is a non-negative six-digit hexadecimal number.

color->string *color* Function

Returns a string representation of *color*.

string->color *string* Function

Returns the color represented by *string*. If *string* is not a syntactically valid notation for a color, then `string->color` returns #f.

4.9.1.2 White

We experience color relative to the illumination around us. CIEXYZ coordinates, although subject to uniform scaling, are objective. Thus other color spaces are specified relative to a *white point* in CIEXYZ coordinates.

The white point for digital color spaces is set to D65. For the other spaces a *white-point* argument can be specified. The default if none is specified is the white-point with which the color was created or last converted; and D65 if none has been specified.

D65 Constant

Is the color of 6500.K (blackbody) illumination. D65 is close to the average color of daylight.

D50 Constant

Is the color of 5000.K (blackbody) illumination. D50 is the color of indoor lighting by incandescent bulbs, whose filaments have temperatures around 5000.K.

4.9.2 Color Spaces

Measurement-based Color Spaces

The *tristimulus* color spaces are those whose component values are proportional measurements of light intensity. The CIEXYZ(1931) system provides 3 sets of spectra to convolve with a spectrum of interest. The result of those convolutions is coordinates in CIEXYZ space. All tristimuls color spaces are related to CIEXYZ by linear transforms, namely matrix multiplication. Of the color spaces listed here, CIEXYZ and RGB709 are tristimulus spaces.

CIEXYZ Color Space

The CIEXYZ color space covers the full *gamut*. It is the basis for color-space conversions.

CIEXYZ is a list of three inexact numbers between 0 and 1.1. '(0. 0. 0.) is black; '(1. 1. 1.) is white.

ciexyz->color *xyz* Function

xyz must be a list of 3 numbers. If *xyz* is valid CIEXYZ coordinates, then `ciexyz->color` returns the color specified by *xyz*; otherwise returns #f.

color:ciexyz *x y z* Function

Returns the CIEXYZ color composed of *x*, *y*, *z*. If the coordinates do not encode a valid CIEXYZ color, then an error is signaled.

color->ciexyz *color* Function

Returns the list of 3 numbers encoding *color* in CIEXYZ.

RGB709 Color Space

BT.709-4 (03/00) *Parameter values for the HDTV standards for production and international programme exchange* specifies parameter values for chromaticity, sampling, signal format, frame rates, etc., of high definition television signals.

An RGB709 color is represented by a list of three inexact numbers between 0 and 1. '(0. 0. 0.) is black '(1. 1. 1.) is white.

rgb709->color *rgb* Function

rgb must be a list of 3 numbers. If *rgb* is valid RGB709 coordinates, then `rgb709->color` returns the color specified by *rgb*; otherwise returns #f.

color:rgb709 *r g b* Function

Returns the RGB709 color composed of *r*, *g*, *b*. If the coordinates do not encode a valid RGB709 color, then an error is signaled.

color->rgb709 *color* Function

Returns the list of 3 numbers encoding *color* in RGB709.

Perceptual Uniformity

Although properly encoding the chromaticity, tristimulus spaces do not match the logarithmic response of human visual systems to intensity. Minimum detectable differences between colors correspond to a smaller range of distances (6:1) in the L*a*b* and L*u*v* spaces than in tristimulus spaces (80:1). For this reason, color distances are computed in L*a*b* (or L*C*h).

L*a*b* Color Space

Is a CIE color space which better matches the human visual system's perception of color. It is a list of three numbers:

- 0 <= L* <= 100 (CIE *Lightness*)
- -500 <= a* <= 500
- -200 <= b* <= 200

l*a*b*->color *L*a*b* white-point* Function

*L*a*b** must be a list of 3 numbers. If *L*a*b** is valid L*a*b* coordinates, then `l*a*b*->color` returns the color specified by *L*a*b**; otherwise returns #f.

color:l*a*b* *L* a* b* white-point* Function

Returns the L*a*b* color composed of *L**, *a**, *b** with *white-point*.

color:l*a*b* *L* a* b** Function

Returns the L*a*b* color composed of *L**, *a**, *b**. If the coordinates do not encode a valid L*a*b* color, then an error is signaled.

color->l*a*b* *color white-point* Function

Returns the list of 3 numbers encoding *color* in L*a*b* with *white-point*.

color->l*a*b* *color* Function

Returns the list of 3 numbers encoding *color* in L*a*b*.

L*u*v* Color Space

Is another CIE encoding designed to better match the human visual system's perception of color.

l*u*v*->color *L*u*v* white-point* Function

*L*u*v** must be a list of 3 numbers. If *L*u*v** is valid L*u*v* coordinates, then `l*u*v*->color` returns the color specified by *L*u*v**; otherwise returns #f.

color:l*u*v* *L* u* v* white-point* Function

Returns the L*u*v* color composed of *L**, *u**, *v** with *white-point*.

color:l*u*v* *L* u* v** Function

Returns the L*u*v* color composed of *L**, *u**, *v**. If the coordinates do not encode a valid L*u*v* color, then an error is signaled.

color->l*u*v* *color white-point* Function

Returns the list of 3 numbers encoding *color* in L*u*v* with *white-point*.

color->l*u*v* *color* Function

Returns the list of 3 numbers encoding *color* in L*u*v*.

Cylindrical Coordinates

HSL (Hue Saturation Lightness), HSV (Hue Saturation Value), HSI (Hue Saturation Intensity) and HCI (Hue Chroma Intensity) are cylindrical color spaces (with angle hue). But these spaces are all defined in terms device-dependent RGB spaces.

One might wonder if there is some fundamental reason why intuitive specification of color must be device-dependent. But take heart! A cylindrical system can be based on L*a*b* and is used for predicting how close colors seem to observers.

L*C*h Color Space

Expresses the *a and b* of L*a*b* in polar coordinates. It is a list of three numbers:

- 0 <= L* <= 100 (CIE *Lightness*)
- C* (CIE *Chroma*) is the distance from the neutral (gray) axis.
- 0 <= h <= 360 (CIE *Hue*) is the angle.

The colors by quadrant of h are:

0	red, orange, yellow	90
90	yellow, yellow-green, green	180
180	green, cyan (blue-green), blue	270
270	blue, purple, magenta	360

l*c*h->color *L*C*h white-point* Function

*L*C*h* must be a list of 3 numbers. If *L*C*h* is valid L*C*h coordinates, then `l*c*h->color` returns the color specified by *L*C*h*; otherwise returns #f.

color:l*c*h *L* C* h white-point* Function

Returns the L*C*h color composed of *L**, *C**, *h* with *white-point*.

color:l*c*h *L* C* h* Function

Returns the L*C*h color composed of *L**, *C**, *h*. If the coordinates do not encode a valid L*C*h color, then an error is signaled.

color->l*c*h *color white-point* Function

Returns the list of 3 numbers encoding *color* in L*C*h with *white-point*.

color->l*c*h *color* Function

Returns the list of 3 numbers encoding *color* in L*C*h.

Digital Color Spaces

The color spaces discussed so far are impractical for image data because of numerical precision and computational requirements. In 1998 the IEC adopted *A Standard Default Color Space for the Internet - sRGB* (`http://www.w3.org/Graphics/Color/sRGB`). sRGB was cleverly designed to employ the 24-bit (256x256x256) color encoding already in widespread use; and the 2.2 gamma intrinsic to CRT monitors.

Conversion from CIEXYZ to digital (sRGB) color spaces is accomplished by conversion first to a RGB709 tristimulus space with D65 white-point; then each coordinate is individually subjected to the same non-linear mapping. Inverse operations in the reverse order create the inverse transform.

sRGB Color Space

Is "A Standard Default Color Space for the Internet". Most display monitors will work fairly well with sRGB directly. Systems using ICC profiles [2] should work very well with sRGB.

srgb->color *rgb* Function

rgb must be a list of 3 numbers. If *rgb* is valid sRGB coordinates, then `srgb->color` returns the color specified by *rgb*; otherwise returns #f.

color:srgb *r g b* Function

Returns the sRGB color composed of *r*, *g*, *b*. If the coordinates do not encode a valid sRGB color, then an error is signaled.

xRGB Color Space

Represents the equivalent sRGB color with a single 24-bit integer. The most significant 8 bits encode red, the middle 8 bits blue, and the least significant 8 bits green.

color->srgb *color* Function

Returns the list of 3 integers encoding *color* in sRGB.

color->xrgb *color* Function

Returns the 24-bit integer encoding *color* in sRGB.

xrgb->color *k* Function

Returns the sRGB color composed of the 24-bit integer *k*.

---

[2] A comprehensive encoding of transforms between CIEXYZ and device color spaces is the International Color Consortium profile format, ICC.1:1998-09:

> The intent of this format is to provide a cross-platform device profile format. Such device profiles can be used to translate color data created on one device into another device's native color space.

e-sRGB Color Space

Is "Photography - Electronic still picture imaging - Extended sRGB color encoding" (PIMA 7667:2001). It extends the gamut of sRGB; and its higher precision numbers provide a larger dynamic range.

A triplet of integers represent e-sRGB colors. Three precisions are supported:

e-sRGB10	0 to 1023
e-sRGB12	0 to 4095
e-sRGB16	0 to 65535

e-srgb->color *precision rgb* Function

precision must be the integer 10, 12, or 16. *rgb* must be a list of 3 numbers. If *rgb* is valid e-sRGB coordinates, then `e-srgb->color` returns the color specified by *rgb*; otherwise returns #f.

color:e-srgb *10 r g b* Function

Returns the e-sRGB10 color composed of integers *r*, *g*, *b*.

color:e-srgb *12 r g b* Function

Returns the e-sRGB12 color composed of integers *r*, *g*, *b*.

color:e-srgb *16 r g b* Function

Returns the e-sRGB16 color composed of integers *r*, *g*, *b*. If the coordinates do not encode a valid e-sRGB color, then an error is signaled.

color->e-srgb *precision color* Function

precision must be the integer 10, 12, or 16. `color->e-srgb` returns the list of 3 integers encoding *color* in sRGB10, sRGB12, or sRGB16.

4.9.3 Spectra

The following functions compute colors from spectra, scale color luminance, and extract chromaticity. XYZ is used in the names of procedures for unnormalized colors; the coordinates of CIEXYZ colors are constrained as described in Section 4.9.2 [Color Spaces], page 101.

```
(require 'color-space)
```

A spectrum may be represented as:

- A procedure of one argument accepting real numbers from 380e-9 to 780e-9, the wavelength in meters; or
- A vector of real numbers representing intensity samples evenly spaced over some range of wavelengths overlapping the range 380e-9 to 780e-9.

CIEXYZ values are calculated as dot-product with the X, Y (Luminance), and Z *Spectral Tristimulus Values*. The files '`cie1931.xyz`' and '`cie1964.xyz`' in the distribution contain these CIE-defined values.

cie1964 Feature

Loads the Spectral Tristimulus Values defining *CIE 1964 Supplementary Standard Colorimetric Observer.* **cie1931**

Loads the Spectral Tristimulus Values defining *CIE 1931 Supplementary Standard Colorimetric Observer.* **ciexyz**

Requires Spectral Tristimulus Values, defaulting to cie1931.

`(require 'cie1964)` or `(require 'cie1931)` will load specific values used by the following spectrum conversion procedures. The spectrum conversion procedures `(require 'ciexyz)` to assure that a set is loaded.

spectrum->XYZ *proc* Function

proc must be a function of one argument. `spectrum->XYZ` computes the CIEXYZ(1931) values for the spectrum returned by *proc* when called with arguments from 380e-9 to 780e-9, the wavelength in meters.

spectrum->XYZ *spectrum x1 x2* Function

x1 and *x2* must be positive real numbers specifying the wavelengths (in meters) corresponding to the zeroth and last elements of vector or list *spectrum*. `spectrum->XYZ` returns the CIEXYZ(1931) values for a light source with spectral values proportional to the elements of *spectrum* at evenly spaced wavelengths between *x1* and *x2*.

Compute the colors of 6500.K and 5000.K blackbody radiation:

```
(require 'color-space)
(define xyz (spectrum->XYZ (blackbody-spectrum 6500)))
(define y_n (cadr xyz))
(map (lambda (x) (/ x y_n)) xyz)
    ⇒ (0.9687111145512467 1.0 1.1210875945303613)

(define xyz (spectrum->XYZ (blackbody-spectrum 5000)))
(map (lambda (x) (/ x y_n)) xyz)
    ⇒ (0.2933441826889158 0.2988931825387761 0.25783646831201573)
```

spectrum->CIEXYZ *spectrum x1 x2* Function

spectrum->CIEXYZ *proc* Function

`spectrum->CIEXYZ` computes the CIEXYZ(1931) values for the spectrum, scaled to be just inside the RGB709 gamut.

wavelength->XYZ *w* Function

w must be a number between 380e-9 to 780e-9. `wavelength->XYZ` returns (unnormalized) XYZ values for a monochromatic light source with wavelength *w*.

blackbody-spectrum *temp* Function

blackbody-spectrum *temp span* Function

Returns a procedure of one argument (wavelength in meters), which returns the radiance of a black body at *temp*.

The optional argument *span* is the wavelength analog of bandwidth. With the default *span* of 1.nm (1e-9.m), the values returned by the procedure correspond to the power of the photons with wavelengths *w* to *w*+1e-9.

temperature->XYZ *x* Function

The positive number *x* is a temperature in degrees kelvin. `temperature->XYZ` computes the CIEXYZ(1931) values for the spectrum of a black body at temperature *x*.

Compute the chromaticities of 6500.K and 5000.K blackbody radiation:

```
(require 'color-space)
(XYZ->chromaticity (temperature->XYZ 6500))
    ⇒ (0.3135191660557008 0.3236456786200268)

(XYZ->chromaticity (temperature->XYZ 5000))
    ⇒ (0.34508082841161052 0.3516084965163377)
```

temperature->CIEXYZ *x* Function

The positive number *x* is a temperature in degrees kelvin. `temperature->CIEXYZ` computes the CIEXYZ(1931) values for the spectrum of a black body at temperature *x*, scaled to be just inside the RGB709 gamut.

XYZ:normalize *xyz* Function

xyz is a list of three non-negative real numbers. `XYZ:normalize` returns a list of numbers proportional to *xyz*; scaled so their sum is 1.

XYZ:normalize-colors *colors . . .* Function

colors is a list of XYZ triples. `XYZ:normalize-colors` scales the triples in the list so the maximum sum of numbers in a triple is 1.

XYZ->chromaticity *xyz* Function

Returns a two element list: the x and y components of *xyz* normalized to 1 (= *x* + *y* + *z*).

chromaticity->CIEXYZ *x y* Function

Returns the largest CIEXYZ(1931) values having chromaticity *x* and *y* which are within the RGB709 gamut.

Many color datasets are expressed in *xyY* format; chromaticity with CIE luminance (Y). But xyY is not a CIE standard like CIEXYZ, CIELAB, and CIELUV. Although chrominance is well defined, the luminance component is sometimes scaled to 1, sometimes to 100, but usually has no obvious range. With no given whitepoint, the only reasonable course is to ascertain the luminance range of a dataset and normalize the values to lie from 0 to 1.

XYZ->xyY *xyz* Function

Returns a three element list: the *x* and *y* components of *XYZ* normalized to 1, and CIE luminance *Y*.

xyY->XYZ *xyY* Function

xyY:normalize-colors *colors* Function

colors is a list of xyY triples. `xyY:normalize-colors` scales each chromaticity so it sums to 1 or less; and divides the Y values by the maximum Y in the dataset, so all lie between 0 and 1.

xyY:normalize-colors *colors n* Function

If *n* is positive real, then `xyY:normalize-colors` divides the Y values by *n* times the maximum Y in the dataset.

If *n* is an exact non-positive integer, then `xyY:normalize-colors` divides the Y values by the maximum of the Ys in the dataset excepting the `-`*n* largest Y values.

In all cases, returned Y values are limited to lie from 0 to 1.

Why would one want to normalize to other than 1? If the sun or its reflection is the brightest object in a scene, then normalizing to its luminance will tend to make the rest of the scene very dark. As with photographs, limiting the specular highlights looks better than darkening everything else.

The results of measurements being what they are, `xyY:normalize-colors` is extremely tolerant. Negative numbers are replaced with zero, and chromaticities with sums greater than one are scaled to sum to one.

4.9.4 Color Difference Metrics

CIE:DE* *color1 color2 white-point* Function
CIE:DE* *color1 color2* Function

Returns the Euclidean distance in L*a*b* space between *color1* and *color2*.

CIE:DE*94 *color1 color2 parametric-factors* Function
CIE:DE*94 *color1 color2* Function

`CIE:DE*94` measures distance in the L*C*h cylindrical color-space. The three axes are individually scaled (depending on C*) in their contributions to the total distance.

The CIE has defined reference conditions under which the metric with default parameters can be expected to perform well. These are:

- The specimens are homogeneous in colour.
- The colour difference (CIELAB) is `<=` 5 units.
- They are placed in direct edge contact.
- Each specimen subtends an angle of `>`4 degrees to the assessor, whose colour vision is normal.
- They are illuminated at 1000 lux, and viewed against a background of uniform grey, with L* of 50, under illumination simulating D65.

The *parametric-factors* argument is a list of 3 quantities kL, kC and kH. *parametric-factors* independently adjust each colour-difference term to account for any deviations from the reference viewing conditions. Under the reference conditions explained above, the default is kL = kC = kH = 1.

The Color Measurement Committee of The Society of Dyers and Colorists in Great Britain created a more sophisticated color-distance function for use in judging the consistency of dye lots. With CMC:DE* it is possible to use a single value pass/fail tolerance for all shades.

CMC:DE* *color1 color2 l c* Function
CMC:DE* *color1 color2* Function

CMC:DE* is also a L*C*h metric. The *parametric-factors* argument is a list of 2 numbers *l* and *c*. *l* and *c* parameterize this metric. 1 and 1 are recommended for perceptibility; the default, 2 and 1, for acceptability.

4.9.5 Color Conversions

This package contains the low-level color conversion and color metric routines operating on lists of 3 numbers. There is no type or range checking.

```
(require 'color-space)
```

CIEXYZ:D65 Constant

Is the color of 6500.K (blackbody) illumination. D65 is close to the average color of daylight.

CIEXYZ:D50 Constant

Is the color of 5000.K (blackbody) illumination. D50 is the color of indoor lighting by incandescent bulbs.

CIEXYZ->RGB709 *xyz* Function
RGB709->CIEXYZ *srgb* Function

CIEXYZ->L*u*v* *xyz white-point* Function
CIEXYZ->L*u*v* *xyz* Function
L*u*v*->CIEXYZ *L*u*v* white-point* Function
L*u*v*->CIEXYZ *L*u*v** Function

The *white-point* defaults to CIEXYZ:D65.

CIEXYZ->L*a*b* *xyz white-point* Function
CIEXYZ->L*a*b* *xyz* Function
L*a*b*->CIEXYZ *L*a*b* white-point* Function
L*a*b*->CIEXYZ *L*a*b** Function

The XYZ *white-point* defaults to CIEXYZ:D65.

L*a*b*->L*C*h *L*a*b** Function
L*C*h->L*a*b* *L*C*h* Function

CIEXYZ->sRGB *xyz* Function
sRGB->CIEXYZ *srgb* Function

CIEXYZ->e-sRGB *n xyz* Function
e-sRGB->CIEXYZ *n srgb* Function

sRGB->e-sRGB *n srgb* Function
e-sRGB->sRGB *n srgb* Function

The integer *n* must be 10, 12, or 16. Because sRGB and e-sRGB use the same RGB709 chromaticities, conversion between them is simpler than conversion through CIEXYZ.

Do not convert e-sRGB precision through `e-sRGB->sRGB` then `sRGB->e-sRGB` – values would be truncated to 8-bits!

e-sRGB->e-sRGB *n1 srgb n2* Function

The integers *n1* and *n2* must be 10, 12, or 16. `e-sRGB->e-sRGB` converts *srgb* to e-sRGB of precision *n2*.

L*a*b*:DE *lab1 lab2* Function

Returns the Euclidean distance between *lab1* and *lab2*.

L*C*h:DE*94 *lch1 lch2 parametric-factors* Function
L*C*h:DE*94 *lch1 lch2* Function

`L*C*h:DE*94` measures distance in the L*C*h cylindrical color-space between *lch1* and *lch2*. The three axes are individually scaled (depending on C*) in their contributions to the total distance.

CMC-DE *lch1 lch2 parametric-factors* Function
CMC-DE *lch1 lch2 l c* Function
CMC-DE *lch1 lch2 l* Function
CMC-DE *lch1 lch2* Function

`CMC:DE` is a L*C*h metric. The *parametric-factors* argument is a list of 2 numbers *l* and *c*. *l* and *c* parameterize this metric. 1 and 1 are recommended for perceptibility; the default, 2 and 1, for acceptability.

4.9.6 Color Names

```
(require 'color-names)
```

Rather than ballast the color dictionaries with numbered grays, `file->color-dictionary` discards them. They are provided through the `grey` procedure:

grey *k* Function

Returns `(inexact->exact (round (* k 2.55)))`, the X11 color grey<*k*>.

A color dictionary is a database table relating *canonical* color-names to color-strings (see Section 4.9.1 [Color Data-Type], page 99).

The column names in a color dictionary are unimportant; the first field is the key, and the second is the color-string.

color-name:canonicalize *name* Function

Returns a downcased copy of the string or symbol *name* with '`_`', '`-`', and whitespace removed.

color-name->color *name table1 table2 . . .* Function

table1, *table2*, . . . must be color-dictionary tables. `color-name->color` searches for the canonical form of *name* in *table1*, *table2*, . . . in order; returning the color-string of the first matching record; #f otherwise.

color-dictionaries->lookup *table1 table2 . . .* Function

table1, *table2*, . . . must be color-dictionary tables. `color-dictionaries->lookup` returns a procedure which searches for the canonical form of its string argument in *table1*, *table2*, . . .; returning the color-string of the first matching record; and #f otherwise.

color-dictionary *name rdb base-table-type* Function

rdb must be a string naming a relational database file; and the symbol *name* a table therein. The database will be opened as *base-table-type*. `color-dictionary` returns the read-only table *name* in database *name* if it exists; #f otherwise.

color-dictionary *name rdb* Function

rdb must be an open relational database or a string naming a relational database file; and the symbol *name* a table therein. `color-dictionary` returns the read-only table *name* in database *name* if it exists; #f otherwise.

load-color-dictionary *name rdb base-table-type* Function
load-color-dictionary *name rdb* Function

rdb must be a string naming a relational database file; and the symbol *name* a table therein. If the symbol *base-table-type* is provided, the database will be opened as *base-table-type*. `load-color-dictionary` creates a top-level definition of the symbol *name* to a lookup procedure for the color dictionary *name* in *rdb*.

The value returned by `load-color-dictionary` is unspecified.

Dictionary Creation

file->color-dictionary *file table-name rdb base-table-type* Function
file->color-dictionary *file table-name rdb* Function

rdb must be an open relational database or a string naming a relational database file, *table-name* a symbol, and the string *file* must name an existing file with color-names and their corresponding xRGB (6-digit hex) values. `file->color-dictionary` creates a table *table-name* in *rdb* and enters the associations found in *file* into it.

url->color-dictionary *url table-name rdb base-table-type* Function
url->color-dictionary *url table-name rdb* Function

rdb must be an open relational database or a string naming a relational database file and *table-name* a symbol. `url->color-dictionary` retrieves the resource named by the string *url* using the *wget* program; then calls `file->color-dictionary` to enter its associations in *table-name* in *url*.

This section has detailed the procedures for creating and loading color dictionaries. So where are the dictionaries to load?

`http://swissnet.ai.mit.edu/~jaffer/Color/Dictionaries.html`

Describes and evaluates several color-name dictionaries on the web. The following procedure creates a database containing two of these dictionaries.

make-slib-color-name-db Function

Creates an alist-table relational database in library-vicinity containing the Resene and Hollasch color-name dictionaries.

If the files '`resenecolours.txt`' and '`hollasch.txt`' exist in the library-vicinity, then they used as the source of color-name data. Otherwise, `make-slib-color-name-db` calls url->color-dictionary with the URLs of appropriate source files.

The Short List

```
(require 'hollasch)
```

hollasch *name* Function

Looks for *name* among the 190 entries in the Hollasch color-name dictionary (`http://swissnet.ai.mit.edu/~jaffer/Color/hollasch.pdf`). If *name* is found, the corresponding color is returned. Otherwise #f is returned. Hollasch is well suited for light source colors.

Resene Paints Limited, New Zealand's largest privately-owned and operated paint manufacturing company, has generously made their *Resene RGB Values List* available.

```
(require 'resene)
```

resene *name* Function

Looks for *name* among the 1300 entries in the Resene color-name dictionary (`http://swissnet.ai.mit.edu/~jaffer/Color/resene.pdf`). If *name* is found, the corresponding color is returned. Otherwise #f is returned. The *Resene RGB Values List* is an excellent source for surface colors.

4.9.7 Daylight

```
(require 'daylight)
```

This package calculates the colors of sky as detailed in: `http://www.cs.utah.edu/vissim/papers/sunsky/`
A Practical Analytic Model for Daylight A. J. Preetham, Peter Shirley, Brian Smits

solar-hour *julian-day hour* Function

Returns the solar-time in hours given the integer *julian-day* in the range 1 to 366, and the local time in hours.

To be meticulous, subtract 4 minutes for each degree of longitude west of the standard meridian of your time zone.

solar-declination *julian-day* Function

solar-polar *declination latitude solar-hour* Function

Returns a list of *theta_s*, the solar angle from the zenith, and *phi_s*, the solar azimuth. 0 <= *theta_s* measured in degrees. *phi_s* is measured in degrees from due south; west of south being positive.

In the following procedures, the number 0 <= *theta_s* <= 90 is the solar angle from the zenith in degrees.

Turbidity is a measure of the fraction of scattering due to haze as opposed to molecules. This is a convenient quantity because it can be estimated based on visibility of distant objects. This model fails for turbidity values less than 1.3.

```
    ---------------------------------------------------------------
512|-:                                                             |
   | * pure-air                                                    |
256|-:**                                                           |
   | : ** exceptionally-clear                                      |
128|-:   *                                                         |
   | :    **                                                       |
 64|-:      *                                                      |
   | :       ** very-clear                                         |
 32|-:         **                                                  |
   | :           **                                                |
 16|-:             *** clear                                       |
   | :               ****                                          |
  8|-:                  ****                                       |
   | :                     **** light-haze                         |
  4|-:                         ****                                |
   | :                             ******                          |
  2|-:                                  ******** haze         thin-|
   | :                                        ***********    fog |
  1|-:----------------------------------------------------*******--|
   |_:____.____:____.____:____.____:____.____:____.____:____.____:_|
     1         2         4         8        16        32        64
              Meterorological range (km) versus Turbidity
```

sunlight-spectrum *turbidity theta_s* Function

Returns a vector of 41 values, the spectrum of sunlight from 380.nm to 790.nm for a given *turbidity* and *theta_s*.

sunlight-xyz *turbidity theta_s* Function

Returns (unnormalized) XYZ values for color of sunlight for a given *turbidity* and *theta_s*.

sunlight-ciexyz *turbidity theta_s* Function

Given *turbidity* and *theta_s*, `sunlight-ciexyz` returns the CIEXYZ triple for color of sunlight scaled to be just inside the RGB709 gamut.

zenith-xyy *turbidity theta_s* Function

Returns the xyY (chromaticity and luminance) at the zenith. The Luminance has units kcd/m^2.

overcast-sky-color-xyy *turbidity theta_s* Function

turbidity is a positive real number expressing the amount of light scattering. The real number *theta_s* is the solar angle from the zenith in degrees.

`overcast-sky-color-xyy` returns a function of one angle *theta*, the angle from the zenith of the viewing direction (in degrees); and returning the xyY value for light coming from that elevation of the sky.

clear-sky-color-xyy *turbidity theta_s phi_s* Function
sky-color-xyy *turbidity theta_s phi_s* Function

turbidity is a positive real number expressing the amount of light scattering. The real number *theta_s* is the solar angle from the zenith in degrees. The real number *phi_s* is the solar angle from south.

`clear-sky-color-xyy` returns a function of two angles, *theta* and *phi* which specify the angles from the zenith and south meridian of the viewing direction (in degrees); returning the xyY value for light coming from that direction of the sky.

`sky-color-xyY` calls `overcast-sky-color-xyY` for *turbidity* <= 20; otherwise the `clear-sky-color-xyy` function.

4.10 Root Finding

```
(require 'root)
```

newtown:find-integer-root *f df/dx x0* Function

Given integer valued procedure *f*, its derivative (with respect to its argument) *df/dx*, and initial integer value *x0* for which *df/dx*(*x0*) is non-zero, returns an integer *x* for which *f*(*x*) is closer to zero than either of the integers adjacent to *x*; or returns `#f` if such an integer can't be found.

To find the closest integer to a given integers square root:

```
(define (integer-sqrt y)
  (newton:find-integer-root
   (lambda (x) (- (* x x) y))
   (lambda (x) (* 2 x))
   (ash 1 (quotient (integer-length y) 2))))

(integer-sqrt 15) ⇒ 4
```

integer-sqrt *y* Function

Given a non-negative integer *y*, returns the rounded square-root of *y*.

newton:find-root *f df/dx x0 prec* Function

Given real valued procedures *f*, *df/dx* of one (real) argument, initial real value *x0* for which *df/dx*(*x0*) is non-zero, and positive real number *prec*, returns a real *x* for which `abs`(*f*(*x*)) is less than *prec*; or returns `#f` if such a real can't be found.

If *prec* is instead a negative integer, `newton:find-root` returns the result of -*prec* iterations.

H. J. Orchard, *The Laguerre Method for Finding the Zeros of Polynomials*, IEEE Transactions on Circuits and Systems, Vol. 36, No. 11, November 1989, pp 1377-1381.

There are 2 errors in Orchard's Table II. Line k=2 for starting value of 1000+j0 should have Z_k of 1.0475 + j4.1036 and line k=2 for starting value of 0+j1000 should have Z_k of 1.0988 + j4.0833.

laguerre:find-root *f df/dz ddf/dz^2 z0 prec* Function

Given complex valued procedure *f* of one (complex) argument, its derivative (with respect to its argument) *df/dx*, its second derivative *ddf/dz^2*, initial complex value *z0*, and positive real number *prec*, returns a complex number *z* for which `magnitude`(*f*(*z*)) is less than *prec*; or returns `#f` if such a number can't be found.

If *prec* is instead a negative integer, `laguerre:find-root` returns the result of -*prec* iterations.

laguerre:find-polynomial-root *deg f df/dz ddf/dz^2 z0 prec* Function

Given polynomial procedure *f* of integer degree *deg* of one argument, its derivative (with respect to its argument) *df/dx*, its second derivative *ddf/dz^2*, initial complex value *z0*, and positive real number *prec*, returns a complex number *z* for which `magnitude`(*f*(*z*)) is less than *prec*; or returns `#f` if such a number can't be found.

If *prec* is instead a negative integer, `laguerre:find-polynomial-root` returns the result of -*prec* iterations.

secant:find-root *f x0 x1 prec* Function
secant:find-bracketed-root *f x0 x1 prec* Function

Given a real valued procedure *f* and two real valued starting points *x0* and *x1*, returns a real *x* for which `(abs (f x))` is less than *prec*; or returns `#f` if such a real can't be found.

If *x0* and *x1* are chosen such that they bracket a root, that is

```
(or (< (f x0) 0 (f x1))
    (< (f x1) 0 (f x0)))
```

then the root returned will be between *x0* and *x1*, and *f* will not be passed an argument outside of that interval.

`secant:find-bracketed-root` will return `#f` unless *x0* and *x1* bracket a root.

The secant method is used until a bracketing interval is found, at which point a modified *regula falsi* method is used.

If *prec* is instead a negative integer, `secant:find-root` returns the result of -*prec* iterations.

If *prec* is a procedure it should accept 5 arguments: *x0 f0 x1 f1* and *count*, where *f0* will be `(f x0)`, *f1* `(f x1)`, and *count* the number of iterations performed so far. *prec* should return non-false if the iteration should be stopped.

4.11 Minimizing

```
(require 'minimize)
```

The Golden Section Search[3] algorithm finds minima of functions which are expensive to compute or for which derivatives are not available. Although optimum for the general case, convergence is slow, requiring nearly 100 iterations for the example (x^3-2x-5).

If the derivative is available, Newton-Raphson is probably a better choice. If the function is inexpensive to compute, consider approximating the derivative.

golden-section-search *f x0 x1 prec* Function

x_0 are *x_1* real numbers. The (single argument) procedure *f* is unimodal over the open interval (*x_0*, *x_1*). That is, there is exactly one point in the interval for which the derivative of *f* is zero.

`golden-section-search` returns a pair $(x\ .\ f(x))$ where $f(x)$ is the minimum. The *prec* parameter is the stop criterion. If *prec* is a positive number, then the iteration continues until x is within *prec* from the true value. If *prec* is a negative integer, then the procedure will iterate -*prec* times or until convergence. If *prec* is a procedure of seven arguments, *x0*, *x1*, *a*, *b*, *fa*, *fb*, and *count*, then the iterations will stop when the procedure returns `#t`.

Analytically, the minimum of x^3-2x-5 is 0.816497.

```
(define func (lambda (x) (+ (* x (+ (* x x) -2)) -5)))
(golden-section-search func 0 1 (/ 10000))
      ==> (816.4883855245578e-3 . -6.0886621077391165)
(golden-section-search func 0 1 -5)
      ==> (819.6601125010515e-3 . -6.088637561916407)
(golden-section-search func 0 1
                       (lambda (a b c d e f g ) (= g 500)))
      ==> (816.4965933140557e-3 . -6.088662107903635)
```

4.12 Commutative Rings

[3] David Kahaner, Cleve Moler, and Stephen Nash *Numerical Methods and Software* Prentice-Hall, 1989, ISBN 0-13-627258-4

Scheme provides a consistent and capable set of numeric functions. Inexacts implement a field; integers a commutative ring (and Euclidean domain). This package allows one to use basic Scheme numeric functions with symbols and non-numeric elements of commutative rings.

```
(require 'commutative-ring)
```

The *commutative-ring* package makes the procedures `+`, `-`, `*`, `/`, and `^` *careful* in the sense that any non-numeric arguments they do not reduce appear in the expression output. In order to see what working with this package is like, self-set all the single letter identifiers (to their corresponding symbols).

```
(define a 'a)
...
(define z 'z)
```

Or just `(require 'self-set)`. Now try some sample expressions:

```
(+ (+ a b) (- a b)) ⇒ (* a 2)
(* (+ a b) (+ a b)) ⇒ (^ (+ a b) 2)
(* (+ a b) (- a b)) ⇒ (* (+ a b) (- a b))
(* (- a b) (- a b)) ⇒ (^ (- a b) 2)
(* (- a b) (+ a b)) ⇒ (* (+ a b) (- a b))
(/ (+ a b) (+ c d)) ⇒ (/ (+ a b) (+ c d))
(^ (+ a b) 3) ⇒ (^ (+ a b) 3)
(^ (+ a 2) 3) ⇒ (^ (+ 2 a) 3)
```

Associative rules have been applied and repeated addition and multiplication converted to multiplication and exponentiation.

We can enable distributive rules, thus expanding to sum of products form:

```
(set! *ruleset* (combined-rulesets distribute* distribute/))

(* (+ a b) (+ a b)) ⇒ (+ (* 2 a b) (^ a 2) (^ b 2))
(* (+ a b) (- a b)) ⇒ (- (^ a 2) (^ b 2))
(* (- a b) (- a b)) ⇒ (- (+ (^ a 2) (^ b 2)) (* 2 a b))
(* (- a b) (+ a b)) ⇒ (- (^ a 2) (^ b 2))
(/ (+ a b) (+ c d)) ⇒ (+ (/ a (+ c d)) (/ b (+ c d)))
(/ (+ a b) (- c d)) ⇒ (+ (/ a (- c d)) (/ b (- c d)))
(/ (- a b) (- c d)) ⇒ (- (/ a (- c d)) (/ b (- c d)))
(/ (- a b) (+ c d)) ⇒ (- (/ a (+ c d)) (/ b (+ c d)))
(^ (+ a b) 3) ⇒ (+ (* 3 a (^ b 2)) (* 3 b (^ a 2)) (^ a 3) (^ b 3))
(^ (+ a 2) 3) ⇒ (+ 8 (* a 12) (* (^ a 2) 6) (^ a 3))
```

Use of this package is not restricted to simple arithmetic expressions:

```
(require 'determinant)

(determinant '((a b c) (d e f) (g h i))) ⇒
(- (+ (* a e i) (* b f g) (* c d h)) (* a f h) (* b d i) (* c e g))
```

Currently, only `+`, `-`, `*`, `/`, and `^` support non-numeric elements. Expressions with `-` are converted to equivalent expressions without `-`, so behavior for `-` is not defined separately. `/` expressions are handled similarly.

This list might be extended to include `quotient`, `modulo`, `remainder`, `lcm`, and `gcd`; but these work only for the more restrictive Euclidean (Unique Factorization) Domain.

4.13 Rules and Rulesets

The *commutative-ring* package allows control of ring properties through the use of *rulesets*.

ruleset Variable

Contains the set of rules currently in effect. Rules defined by `cring:define-rule` are stored within the value of *ruleset* at the time `cring:define-rule` is called. If **ruleset** is `#f`, then no rules apply.

make-ruleset *rule1 ...* Function
make-ruleset *name rule1 ...* Function

Returns a new ruleset containing the rules formed by applying `cring:define-rule` to each 4-element list argument *rule*. If the first argument to `make-ruleset` is a symbol, then the database table created for the new ruleset will be named *name*. Calling `make-ruleset` with no rule arguments creates an empty ruleset.

combined-rulesets *ruleset1 ...* Function
combined-rulesets *name ruleset1 ...* Function

Returns a new ruleset containing the rules contained in each ruleset argument *ruleset*. If the first argument to `combined-ruleset` is a symbol, then the database table created for the new ruleset will be named *name*. Calling `combined-ruleset` with no ruleset arguments creates an empty ruleset.

Two rulesets are defined by this package.

distribute* Constant

Contain the ruleset to distribute multiplication over addition and subtraction. **distribute/** Const

Contain the ruleset to distribute division over addition and subtraction.

Take care when using both *distribute** and *distribute/* simultaneously. It is possible to put `/` into an infinite loop.

You can specify how sum and product expressions containing non-numeric elements simplify by specifying the rules for `+` or `*` for cases where expressions involving objects reduce to numbers or to expressions involving different non-numeric elements.

cring:define-rule *op sub-op1 sub-op2 reduction* Function

Defines a rule for the case when the operation represented by symbol *op* is applied to lists whose `cars` are *sub-op1* and *sub-op2*, respectively. The argument *reduction* is a procedure accepting 2 arguments which will be lists whose `car`s are *sub-op1* and *sub-op2*.

cring:define-rule *op sub-op1 'identity reduction* Function

Defines a rule for the case when the operation represented by symbol *op* is applied to a list whose `car` is *sub-op1*, and some other argument. *Reduction* will be called with the list whose `car` is *sub-op1* and some other argument.

If *reduction* returns `#f`, the reduction has failed and other reductions will be tried. If *reduction* returns a non-false value, that value will replace the two arguments in arithmetic (`+`, `-`, and `*`) calculations involving non-numeric elements.

The operations `+` and `*` are assumed commutative; hence both orders of arguments to *reduction* will be tried if necessary.

The following rule is the definition for distributing `*` over `+`.

```
(cring:define-rule
 '* '+ 'identity
 (lambda (exp1 exp2)
   (apply + (map (lambda (trm) (* trm exp2)) (cdr exp1))))))
```

4.14 How to Create a Commutative Ring

The first step in creating your commutative ring is to write procedures to create elements of the ring. A non-numeric element of the ring must be represented as a list whose first element is a symbol or string. This first element identifies the type of the object. A convenient and clear convention is to make the type-identifying element be the same symbol whose top-level value is the procedure to create it.

```
(define (n . list1)
  (cond ((and (= 2 (length list1))
              (eq? (car list1) (cadr list1)))
         0)
        ((not (term< (first list1) (last1 list1)))
         (apply n (reverse list1)))
        (else (cons 'n list1))))

(define (s x y) (n x y))

(define (m . list1)
  (cond ((neq? (first list1) (term_min list1))
         (apply m (cyclicrotate list1)))
        ((term< (last1 list1) (cadr list1))
         (apply m (reverse (cyclicrotate list1))))
        (else (cons 'm list1))))
```

Define a procedure to multiply 2 non-numeric elements of the ring. Other multiplicatons are handled automatically. Objects for which rules have *not* been defined are not changed.

```
(define (n*n ni nj)
  (let ((list1 (cdr ni)) (list2 (cdr nj)))
    (cond ((null? (intersection list1 list2)) #f)
          ((and (eq? (last1 list1) (first list2))
                (neq? (first list1) (last1 list2)))
           (apply n (splice list1 list2)))
          ((and (eq? (first list1) (first list2))
                (neq? (last1 list1) (last1 list2)))
           (apply n (splice (reverse list1) list2)))
          ((and (eq? (last1 list1) (last1 list2))
                (neq? (first list1) (first list2)))
           (apply n (splice list1 (reverse list2))))
          ((and (eq? (last1 list1) (first list2))
                (eq? (first list1) (last1 list2)))
           (apply m (cyclicsplice list1 list2)))
          ((and (eq? (first list1) (first list2))
                (eq? (last1 list1) (last1 list2)))
           (apply m (cyclicsplice (reverse list1) list2)))
          (else #f))))
```

Test the procedures to see if they work.

```
;;; where cyclicrotate(list) is cyclic rotation of the list one step
;;; by putting the first element at the end
(define (cyclicrotate list1)
  (append (rest list1) (list (first list1))))
;;; and where term_min(list) is the element of the list which is
;;; first in the term ordering.
(define (term_min list1)
  (car (sort list1 term<)))
(define (term< sym1 sym2)
  (string<? (symbol->string sym1) (symbol->string sym2)))
(define first car)
(define rest cdr)
(define (last1 list1) (car (last-pair list1)))
(define (neq? obj1 obj2) (not (eq? obj1 obj2)))
;;; where splice is the concatenation of list1 and list2 except that their
;;; common element is not repeated.
(define (splice list1 list2)
  (cond ((eq? (last1 list1) (first list2))
         (append list1 (cdr list2)))
        (else (error 'splice list1 list2))))
;;; where cyclicsplice is the result of leaving off the last element of
;;; splice(list1,list2).
(define (cyclicsplice list1 list2)
  (cond ((and (eq? (last1 list1) (first list2))
              (eq? (first list1) (last1 list2)))
         (butlast (splice list1 list2) 1))
        (else (error 'cyclicsplice list1 list2))))

(N*N (S a b) (S a b)) ⇒ (m a b)
```

Then register the rule for multiplying type N objects by type N objects.

```
(cring:define-rule '* 'N 'N N*N))
```

Now we are ready to compute!

```
(define (t)
  (define detM
    (+ (* (S g b)
          (+ (* (S f d)
                (- (* (S a f) (S d g)) (* (S a g) (S d f))))
             (* (S f f)
                (- (* (S a g) (S d d)) (* (S a d) (S d g))))
             (* (S f g)
                (- (* (S a d) (S d f)) (* (S a f) (S d d))))))
       (* (S g d)
          (+ (* (S f b)
                (- (* (S a g) (S d f)) (* (S a f) (S d g))))
             (* (S f f)
                (- (* (S a b) (S d g)) (* (S a g) (S d b))))
             (* (S f g)
                (- (* (S a f) (S d b)) (* (S a b) (S d f))))))
       (* (S g f)
          (+ (* (S f b)
                (- (* (S a d) (S d g)) (* (S a g) (S d d))))
             (* (S f d)
                (- (* (S a g) (S d b)) (* (S a b) (S d g))))
             (* (S f g)
                (- (* (S a b) (S d d)) (* (S a d) (S d b))))))
       (* (S g g)
          (+ (* (S f b)
                (- (* (S a f) (S d d)) (* (S a d) (S d f))))
             (* (S f d)
                (- (* (S a b) (S d f)) (* (S a f) (S d b))))
             (* (S f f)
                (- (* (S a d) (S d b)) (* (S a b) (S d d))))))))
  (* (S b e) (S c a) (S e c)
     detM
     ))
(pretty-print (t))
⊣
(- (+ (m a c e b d f g)
      (m a c e b d g f)
      (m a c e b f d g)
      (m a c e b f g d)
      (m a c e b g d f)
      (m a c e b g f d))
   (* 2 (m a b e c) (m d f g))
   (* (m a c e b d) (m f g))
   (* (m a c e b f) (m d g))
   (* (m a c e b g) (m d f)))
```

4.15 Matrix Algebra

```
(require 'determinant)
```

A Matrix can be either a list of lists (rows) or an array. As with linear-algebra texts, this package uses 1-based coordinates.

matrix->lists *matrix* Function

Returns the list-of-lists form of *matrix*.

matrix->array *matrix* Function

Returns the (ones-based) array form of *matrix*.

determinant *matrix* Function

matrix must be a square matrix. `determinant` returns the determinant of *matrix*.

```
(require 'determinant)
(determinant '((1 2) (3 4))) ⇒ -2
(determinant '((1 2 3) (4 5 6) (7 8 9))) ⇒ 0
```

transpose *matrix* Function

Returns a copy of *matrix* flipped over the diagonal containing the 1,1 element.

matrix:product *m1 m2* Function

Returns the product of matrices *m1* and *m2*.

matrix:inverse *matrix* Function

matrix must be a square matrix. If *matrix* is singlar, then `matrix:inverse` returns #f; otherwise `matrix:inverse` returns the `matrix:product` inverse of *matrix*.

5 Database Packages

5.1 Base Table

A *base-table* is the primitive database layer upon which SLIB relational databases are built. At the minimum, it must support the types integer, symbol, string, and boolean. The base-table may restrict the size of integers, symbols, and strings it supports.

A base table implementation is available as the value of the identifier naming it (eg. *alist-table*) after requiring the symbol of that name.

alist-table Feature

`(require 'alist-table)`

Association-list base tables support all Scheme types and are suitable for small databases. In order to be retrieved after being written to a file, the data stored should include only objects which are readable and writeable in the Scheme implementation.

The *alist-table* base-table-type is included in the SLIB distribution.

wb-table Feature

`(require 'wb-table)`

WB is a B-tree database package supported by the SCM Scheme implementation. It supports scheme expressions for keys and values whose text representations are less than 255 characters in length. Being disk-based, wb-table readily stores hundreds of megabytes of data.

This rest of this section documents the interface for a base table implementation from which the Section 5.2 [Relational Database], page 128 package constructs a Relational system. It will be of interest primarily to those wishing to port or write new base-table implementations.

base-table-implementations Variable

To support automatic dispatch for `open-database`, each base-table module adds an association to **base-table-implementations** when loaded. This association is the list of the base-table symbol and the value returned by `(make-relational-system` *base-table*`)`.

All of these functions are accessed through a single procedure by calling that procedure with the symbol name of the operation. A procedure will be returned if that operation is supported and `#f` otherwise. For example:

```
(require 'alist-table)
(define open-base (alist-table 'make-base))
make-base       ⇒ *a procedure*
(define foo (alist-table 'foo))
foo             ⇒ #f
```

make-base *filename key-dimension column-types* Function

Returns a new, open, low-level database (collection of tables) associated with *filename*. This returned database has an empty table associated with *catalog-id*. The positive integer *key-dimension* is the number of keys composed to make a *primary-key* for the catalog table. The list of symbols *column-types* describes the types of each column for that table. If the database cannot be created as specified, `#f` is returned.

Calling the `close-base` method on this database and possibly other operations will cause *filename* to be written to. If *filename* is `#f` a temporary, non-disk based database will be created if such can be supported by the base table implelentation.

open-base *filename mutable* Function

Returns an open low-level database associated with *filename*. If *mutable* is `#t`, this database will have methods capable of effecting change to the database. If *mutable* is `#f`, only methods for inquiring the database will be available. If the database cannot be opened as specified `#f` is returned.

Calling the `close-base` (and possibly other) method on a *mutable* database will cause *filename* to be written to.

write-base *lldb filename* Function

Causes the low-level database *lldb* to be written to *filename*. If the write is successful, also causes *lldb* to henceforth be associated with *filename*. Calling the `close-database` (and possibly other) method on *lldb* may cause *filename* to be written to. If *filename* is `#f` this database will be changed to a temporary, non-disk based database if such can be supported by the underlying base table implelentation. If the operations completed successfully, `#t` is returned. Otherwise, `#f` is returned.

sync-base *lldb* Function

Causes the file associated with the low-level database *lldb* to be updated to reflect its current state. If the associated filename is `#f`, no action is taken and `#f` is returned. If this operation completes successfully, `#t` is returned. Otherwise, `#f` is returned.

close-base *lldb* Function

Causes the low-level database *lldb* to be written to its associated file (if any). If the write is successful, subsequent operations to *lldb* will signal an error. If the operations complete successfully, `#t` is returned. Otherwise, `#f` is returned.

make-table *lldb key-dimension column-types* Function

Returns the *base-id* for a new base table, otherwise returns `#f`. The base table can then be opened using `(open-table` *lldb base-id*`)`. The positive integer *key-dimension* is the number of keys composed to make a *primary-key* for this table. The list of symbols *column-types* describes the types of each column.

catalog-id Constant

A constant *base-id* suitable for passing as a parameter to `open-table`. *catalog-id* will be used as the base table for the system catalog.

open-table *lldb base-id key-dimension column-types* Function

Returns a *handle* for an existing base table in the low-level database *lldb* if that table exists and can be opened in the mode indicated by *mutable*, otherwise returns `#f`.

As with `make-table`, the positive integer *key-dimension* is the number of keys composed to make a *primary-key* for this table. The list of symbols *column-types* describes the types of each column.

kill-table *lldb base-id key-dimension column-types* Function

Returns `#t` if the base table associated with *base-id* was removed from the low level database *lldb*, and `#f` otherwise.

make-keyifier-1 *type* Function

Returns a procedure which accepts a single argument which must be of type *type*. This returned procedure returns an object suitable for being a *key* argument in the functions whose descriptions follow.

Any 2 arguments of the supported type passed to the returned function which are not `equal?` must result in returned values which are not `equal?`.

make-list-keyifier *key-dimension types* Function

The list of symbols *types* must have at least *key-dimension* elements. Returns a procedure which accepts a list of length *key-dimension* and whose types must corresopond to the types named by *types*. This returned procedure combines the elements of its list argument into an object suitable for being a *key* argument in the functions whose descriptions follow.

Any 2 lists of supported types (which must at least include symbols and non-negative integers) passed to the returned function which are not `equal?` must result in returned values which are not `equal?`.

make-key-extractor *key-dimension types column-number* Function

Returns a procedure which accepts objects produced by application of the result of `(make-list-keyifier` *key-dimension types*`)`. This procedure returns a *key* which is `equal?` to the *column-number*th element of the list which was passed to create *combined-key*. The list *types* must have at least *key-dimension* elements.

make-key->list *key-dimension types* Function

Returns a procedure which accepts objects produced by application of the result of `(make-list-keyifier` *key-dimension types*`)`. This procedure returns a list of *key*s which are elementwise `equal?` to the list which was passed to create *combined-key*.

In the following functions, the *key* argument can always be assumed to be the value returned by a call to a *keyify* routine.

In contrast, a *match-keys* argument is a list of length equal to the number of primary keys. The *match-keys* restrict the actions of the table command to those records whose primary keys all satisfy the corresponding element of the *match-keys* list. The elements and their actions are:

`#f`	The false value matches any key in the corresponding position.
an object of type procedure	This procedure must take a single argument, the key in the corresponding position. Any key for which the procedure returns a non-false value is a match; Any key for which the procedure returns a `#f` is not.
other values	Any other value matches only those keys `equal?` to it.

The *key-dimension* and *column-types* arguments are needed to decode the combined-keys for matching with *match-keys*.

for-each-key *handle procedure key-dimension column-types match-keys* Function

Calls *procedure* once with each *key* in the table opened in *handle* which satisfy *match-keys* in an unspecified order. An unspecified value is returned.

map-key *handle procedure key-dimension column-types match-keys* Function

Returns a list of the values returned by calling *procedure* once with each *key* in the table opened in *handle* which satisfy *match-keys* in an unspecified order.

ordered-for-each-key *handle procedure key-dimension column-types match-keys* Function

Calls *procedure* once with each *key* in the table opened in *handle* which satisfy *match-keys* in the natural order for the types of the primary key fields of that table. An unspecified value is returned.

delete* *handle key-dimension column-types match-keys* Function

Removes all rows which satisfy *match-keys* from the table opened in *handle*. An unspecified value is returned.

present? *handle key* Function

Returns a non-`#f` value if there is a row associated with *key* in the table opened in *handle* and `#f` otherwise.

delete *handle key* Function

Removes the row associated with *key* from the table opened in *handle*. An unspecified value is returned.

make-getter *key-dimension types* Function

Returns a procedure which takes arguments *handle* and *key*. This procedure returns a list of the non-primary values of the relation (in the base table opened in *handle*) whose primary key is *key* if it exists, and `#f` otherwise.

make-putter *key-dimension types* Function

Returns a procedure which takes arguments *handle* and *key* and *value-list*. This procedure associates the primary key *key* with the values in *value-list* (in the base table opened in *handle*) and returns an unspecified value.

supported-type? *symbol* Function

Returns `#t` if *symbol* names a type allowed as a column value by the implementation, and `#f` otherwise. At a minimum, an implementation must support the types `integer`, `symbol`, `string`, `boolean`, and `base-id`.

supported-key-type? *symbol* Function

Returns `#t` if *symbol* names a type allowed as a key value by the implementation, and `#f` otherwise. At a minimum, an implementation must support the types `integer`, and `symbol`.

`integer`	Scheme exact integer.
`symbol`	Scheme symbol.
`boolean`	`#t` or `#f`.
`base-id`	Objects suitable for passing as the *base-id* parameter to `open-table`. The value of *catalog-id* must be an acceptable `base-id`.

5.2 Relational Database

```
(require 'relational-database)
```

This package implements a database system inspired by the Relational Model (*E. F. Codd, A Relational Model of Data for Large Shared Data Banks*). An SLIB relational database implementation can be created from any Section 5.1 [Base Table], page 124 implementation.

Why relational database? For motivations and design issues see `http://swissnet.ai.mit.edu/~jaffe`

5.2.1 Using Databases

```
(require 'databases)
```

This enhancement wraps a utility layer on `relational-database` which provides:

- Identification of open databases by filename.
- Automatic sharing of open (immutable) databases.
- Automatic loading of base-table package when creating a database.
- Detection and automatic loading of the appropriate base-table package when opening a database.
- Table and data definition from Scheme lists.

Database Sharing

Auto-sharing refers to a call to the procedure `open-database` returning an already open database (procedure), rather than opening the database file a second time.

> *Note:* Databases returned by `open-database` do not include wrappers applied by packages like Section 5.2.7 [Embedded Commands], page 139. But wrapped databases do work as arguments to these functions.

When a database is created, it is mutable by the creator and not auto-sharable. A database opened mutably is also not auto-sharable. But any number of readers can (open) share a non-mutable database file.

This next set of procedures mirror the whole-database methods in Section 5.2.4 [Database Operations], page 133. Except for `create-database`, each procedure will accept either a filename or database procedure for its first argument.

create-database *filename base-table-type* Function

> Returns an open relational database (with base-table type *base-table-type*) associated with *filename*.

Only `alist-table` and base-table modules which have been loaded will dispatch correctly from the `open-database` procedures. Therefore, either pass two arguments to `open-database`, or require the base-table your database file uses before calling `open-database` with one argument.

open-database! *rdb base-table-type* Function

> Returns *mutable* open relational database or #f.

open-database *rdb base-table-type* Function

> Returns an open relational database associated with *rdb*. The database will be opened with base-table type *base-table-type*).

open-database *rdb* Function

> Returns an open relational database associated with *rdb*. `open-database` will attempt to deduce the correct base-table-type.

write-database *rdb filename* Function

> Writes the mutable relational-database *rdb* to *filename*.

sync-database *rdb* Function

> Writes the mutable relational-database *rdb* to the filename it was opened with.

solidify-database *rdb* Function

> Syncs *rdb* and makes it immutable.

close-database *rdb* Function

> *rdb* will only be closed when the count of `open-database` - `close-database` calls for *rdb* (and its filename) is 0.

mdbm:report Function

Prints a table of open database files. The columns are the base-table type, number of opens, '`!`' for mutable, and the filename.

```
(mdbm:report)
⊣
   alist-table 003   /usr/local/lib/slib/clrnamdb.scm
   alist-table 001 ! sdram.db
```

Defining Tables

define-tables *rdb spec-0 . . .* Function

Adds tables as specified in *spec-0 . . .* to the open relational-database *rdb*. Each *spec* has the form:

(<name> <descriptor-name> <descriptor-name> <rows>)

or

(<name> <primary-key-fields> <other-fields> <rows>)

where <name> is the table name, <descriptor-name> is the symbol name of a descriptor table, <primary-key-fields> and <other-fields> describe the primary keys and other fields respectively, and <rows> is a list of data rows to be added to the table.

<primary-key-fields> and <other-fields> are lists of field descriptors of the form:

(<column-name> <domain>)

or

(<column-name> <domain> <column-integrity-rule>)

where <column-name> is the column name, <domain> is the domain of the column, and <column-integrity-rule> is an expression whose value is a procedure of one argument (which returns `#f` to signal an error).

If <domain> is not a defined domain name and it matches the name of this table or an already defined (in one of *spec-0 . . .*) single key field table, a foriegn-key domain will be created for it.

Listing Tables

list-table-definition *rdb table-name* Function

If symbol *table-name* exists in the open relational-database *rdb*, then returns a list of the table-name, its primary key names and domains, its other key names and domains, and the table's records (as lists). Otherwise, returns #f.

The list returned by `list-table-definition`, when passed as an argument to `define-tables`, will recreate the table.

5.2.2 Relational Database Objects

make-relational-system *base-table-implementation* Function

Returns a procedure implementing a relational database using the *base-table-implementation*.

All of the operations of a base table implementation are accessed through a procedure defined by `require`ing that implementation. Similarly, all of the operations of the relational database implementation are accessed through the procedure returned by `make-relational-system`. For instance, a new relational database could be created from the procedure returned by `make-relational-system` by:

```
(require 'alist-table)
(define relational-alist-system
        (make-relational-system alist-table))
(define create-alist-database
        (relational-alist-system 'create-database))
(define my-database
        (create-alist-database "mydata.db"))
```

What follows are the descriptions of the methods available from relational system returned by a call to `make-relational-system`.

create-database *filename* Function

Returns an open, nearly empty relational database associated with *filename*. The only tables defined are the system catalog and domain table. Calling the `close-database` method on this database and possibly other operations will cause *filename* to be written to. If *filename* is `#f` a temporary, non-disk based database will be created if such can be supported by the underlying base table implelentation. If the database cannot be created as specified `#f` is returned. For the fields and layout of descriptor tables, Section 5.2.6 [Catalog Representation], page 138

open-database *filename mutable?* Function

Returns an open relational database associated with *filename*. If *mutable?* is `#t`, this database will have methods capable of effecting change to the database. If *mutable?* is `#f`, only methods for inquiring the database will be available. Calling the `close-database` (and possibly other) method on a *mutable?* database will cause *filename* to be written to. If the database cannot be opened as specified `#f` is returned.

5.2.3 Database Operations

These are the descriptions of the methods available from an open relational database. A method is retrieved from a database by calling the database with the symbol name of the operation. For example:

```
(define my-database
        (create-alist-database "mydata.db"))
(define telephone-table-desc
        ((my-database 'create-table) 'telephone-table-desc))
```

close-database Function

Causes the relational database to be written to its associated file (if any). If the write is successful, subsequent operations to this database will signal an error. If the operations completed successfully, `#t` is returned. Otherwise, `#f` is returned.

write-database *filename* Function

Causes the relational database to be written to *filename*. If the write is successful, also causes the database to henceforth be associated with *filename*. Calling the `close-database` (and possibly other) method on this database will cause *filename* to be written to. If *filename* is `#f` this database will be changed to a temporary, non-disk based database if such can be supported by the underlying base table implelentation. If the operations completed successfully, `#t` is returned. Otherwise, `#f` is returned.

sync-database Function

Causes any pending updates to the database file to be written out. If the operations completed successfully, `#t` is returned. Otherwise, `#f` is returned.

solidify-database Function

Causes any pending updates to the database file to be written out. If the writes completed successfully, then the database is changed to be immutable and `#t` is returned. Otherwise, `#f` is returned.

table-exists? *table-name* Function

Returns `#t` if *table-name* exists in the system catalog, otherwise returns `#f`.

open-table *table-name mutable?* Function

Returns a *methods* procedure for an existing relational table in this database if it exists and can be opened in the mode indicated by *mutable?*, otherwise returns `#f`.

These methods will be present only in mutable databases.

delete-table *table-name* Function

Removes and returns the *table-name* row from the system catalog if the table or view associated with *table-name* gets removed from the database, and `#f` otherwise.

create-table *table-desc-name* Function

Returns a methods procedure for a new (open) relational table for describing the columns of a new base table in this database, otherwise returns `#f`. For the fields and layout of descriptor tables, See Section 5.2.6 [Catalog Representation], page 138.

create-table *table-name table-desc-name* Function

Returns a methods procedure for a new (open) relational table with columns as described by *table-desc-name*, otherwise returns `#f`.

create-view *??* Function
project-table *??* Function
restrict-table *??* Function
cart-prod-tables *??* Function

Not yet implemented.

5.2.4 Table Operations

These are the descriptions of the methods available from an open relational table. A method is retrieved from a table by calling the table with the symbol name of the operation. For example:

```
(define telephone-table-desc
        ((my-database 'create-table) 'telephone-table-desc))
(require 'common-list-functions)
(define ndrp (telephone-table-desc 'row:insert))
(ndrp '(1 #t name #f string))
(ndrp '(2 #f telephone
          (lambda (d)
            (and (string? d) (> (string-length d) 2)
                 (every
                  (lambda (c)
                    (memv c '(#\0 #\1 #\2 #\3 #\4 #\5 #\6 #\7 #\8 #\9
                                  #\+ #\( #\  #\) #\-)))
                  (string->list d))))
          string))
```

Some operations described below require primary key arguments. Primary keys arguments are denoted *key1 key2* .... It is an error to call an operation for a table which takes primary key arguments with the wrong number of primary keys for that table.

The term *row* used below refers to a Scheme list of values (one for each column) in the order specified in the descriptor (table) for this table. Missing values appear as `#f`. Primary keys must not be missing.

get *column-name* Function

Returns a procedure of arguments *key1 key2* ... which returns the value for the *column-name* column of the row associated with primary keys *key1*, *key2* ... if that row exists in the table, or `#f` otherwise.

```
((plat 'get 'processor) 'djgpp) ⇒ i386
((plat 'get 'processor) 'be-os) ⇒ #f
```

get* *column-name* Function

Returns a procedure of optional arguments *match-key1* ... which returns a list of the values for the specified column for all rows in this table. The optional *match-key1* ... arguments restrict actions to a subset of the table. See the match-key description below for details.

```
((plat 'get* 'processor)) ⇒
(i386 8086 i386 8086 i386 i386 8086 m68000
 m68000 m68000 m68000 m68000 powerpc)

((plat 'get* 'processor) #f) ⇒
(i386 8086 i386 8086 i386 i386 8086 m68000
 m68000 m68000 m68000 m68000 powerpc)

(define (a-key? key)
   (char=? #\a (string-ref (symbol->string key) 0)))

((plat 'get* 'processor) a-key?) ⇒
(m68000 m68000 m68000 m68000 m68000 powerpc)

((plat 'get* 'name) a-key?) ⇒
(atari-st-turbo-c atari-st-gcc amiga-sas/c-5.10
 amiga-aztec amiga-dice-c aix)
```

row:retrieve Function

Returns a procedure of arguments *key1 key2* ... which returns the row associated with primary keys *key1*, *key2* ... if it exists, or `#f` otherwise.

```
((plat 'row:retrieve) 'linux) ⇒ (linux i386 linux gcc)
((plat 'row:retrieve) 'multics) ⇒ #f
```

row:retrieve* Function

Returns a procedure of optional arguments *match-key1* ... which returns a list of all rows in this table. The optional *match-key1* ... arguments restrict actions to a subset of the table. See the match-key description below for details.

```
((plat 'row:retrieve*) a-key?) ⇒
((atari-st-turbo-c m68000 atari turbo-c)
 (atari-st-gcc m68000 atari gcc)
 (amiga-sas/c-5.10 m68000 amiga sas/c)
 (amiga-aztec m68000 amiga aztec)
 (amiga-dice-c m68000 amiga dice-c)
 (aix powerpc aix -))
```

row:remove Function

Returns a procedure of arguments *key1 key2* ... which removes and returns the row associated with primary keys *key1*, *key2* ... if it exists, or `#f` otherwise.

row:remove* Function

Returns a procedure of optional arguments *match-key1* ... which removes and returns a list of all rows in this table. The optional *match-key1* ... arguments restrict actions to a subset of the table. See the match-key description below for details.

row:delete Function

Returns a procedure of arguments *key1 key2* ... which deletes the row associated with primary keys *key1*, *key2* ... if it exists. The value returned is unspecified.

row:delete* Function

Returns a procedure of optional arguments *match-key1* . . . which Deletes all rows from this table. The optional *match-key1* . . . arguments restrict deletions to a subset of the table. See the match-key description below for details. The value returned is unspecified. The descriptor table and catalog entry for this table are not affected.

row:update Function

Returns a procedure of one argument, *row*, which adds the row, *row*, to this table. If a row for the primary key(s) specified by *row* already exists in this table, it will be overwritten. The value returned is unspecified.

row:update* Function

Returns a procedure of one argument, *rows*, which adds each row in the list of rows, *rows*, to this table. If a row for the primary key specified by an element of *rows* already exists in this table, it will be overwritten. The value returned is unspecified.

row:insert Function

Adds the row *row* to this table. If a row for the primary key(s) specified by *row* already exists in this table an error is signaled. The value returned is unspecified.

row:insert* Function

Returns a procedure of one argument, *rows*, which adds each row in the list of rows, *rows*, to this table. If a row for the primary key specified by an element of *rows* already exists in this table, an error is signaled. The value returned is unspecified.

for-each-row Function

Returns a procedure of arguments *proc match-key1* . . . which calls *proc* with each *row* in this table in the (implementation-dependent) natural ordering for rows. The optional *match-key1* . . . arguments restrict actions to a subset of the table. See the match-key description below for details.

Real relational programmers would use some least-upper-bound join for every row to get them in order; But we don't have joins yet.

The (optional) *match-key1* . . . arguments are used to restrict actions of a whole-table operation to a subset of that table. Those procedures (returned by methods) which accept match-key arguments will accept any number of match-key arguments between zero and the number of primary keys in the table. Any unspecified *match-key* arguments default to `#f`.

The *match-key1* . . . restrict the actions of the table command to those records whose primary keys each satisfy the corresponding *match-key* argument. The arguments and their actions are:

`#f` The false value matches any key in the corresponding position.

an object of type procedure

This procedure must take a single argument, the key in the corresponding position. Any key for which the procedure returns a non-false value is a match; Any key for which the procedure returns a `#f` is not.

other values
: Any other value matches only those keys `equal?` to it.

close-table Function
: Subsequent operations to this table will signal an error.

column-names Constant
column-foreigns Constant
column-domains Constant
column-types Constant
: Return a list of the column names, foreign-key table names, domain names, or type names respectively for this table. These 4 methods are different from the others in that the list is returned, rather than a procedure to obtain the list.

primary-limit Constant
: Returns the number of primary keys fields in the relations in this table.

5.2.5 Catalog Representation

Each database (in an implementation) has a *system catalog* which describes all the user accessible tables in that database (including itself).

The system catalog base table has the following fields. `PRI` indicates a primary key for that table.

```
PRI table-name
    column-limit            the highest column number
    coltab-name             descriptor table name
    bastab-id               data base table identifier
    user-integrity-rule
    view-procedure          A scheme thunk which, when called,
                            produces a handle for the view.  coltab
                            and bastab are specified if and only if
                            view-procedure is not.
```

Descriptors for base tables (not views) are tables (pointed to by system catalog). Descriptor (base) tables have the fields:

```
PRI column-number           sequential integers from 1
    primary-key?            boolean TRUE for primary key components
    column-name
    column-integrity-rule
    domain-name
```

A *primary key* is any column marked as `primary-key?` in the corresponding descriptor table. All the `primary-key?` columns must have lower column numbers than any non-`primary-key?` columns. Every table must have at least one primary key. Primary keys must be sufficient to distinguish all rows from each other in the table. All of the system defined tables have a single primary key.

This package currently supports tables having from 1 to 4 primary keys if there are non-primary columns, and any (natural) number if *all* columns are primary keys. If you need more than 4 primary keys, I would like to hear what you are doing!

A *domain* is a category describing the allowable values to occur in a column. It is described by a (base) table with the fields:

```
PRI domain-name
    foreign-table
    domain-integrity-rule
    type-id
    type-param
```

The *type-id* field value is a symbol. This symbol may be used by the underlying base table implementation in storing that field.

If the `foreign-table` field is non-`#f` then that field names a table from the catalog. The values for that domain must match a primary key of the table referenced by the *type-param* (or `#f`, if allowed). This package currently does not support composite foreign-keys.

The types for which support is planned are:

```
atom
symbol
string                  [<length>]
number                  [<base>]
money                   <currency>
date-time
boolean

foreign-key             <table-name>
expression
virtual                 <expression>
```

5.2.6 Embedded Commands

```
(require 'database-commands)
```

This enhancement wraps a utility layer on `relational-database` which provides:

- Automatic execution of initialization commands stored in database.
- Transparent execution of database commands stored in `*commands*` table in database.

When an enhanced relational-database is called with a symbol which matches a *name* in the `*commands*` table, the associated procedure expression is evaluated and applied to the enhanced relational-database. A procedure should then be returned which the user can invoke on (optional) arguments.

The command `*initialize*` is special. If present in the `*commands*` table, `open-database` or `open-database!` will return the value of the `*initialize*` command. Notice that arbitrary code can be run when the `*initialize*` procedure is automatically applied to the enhanced relational-database.

Note also that if you wish to shadow or hide from the user relational-database methods described in Section 5.2.4 [Database Operations], page 133, this can be done by a dispatch in the closure returned by the `*initialize*` expression rather than by entries in the `*commands*` table if it is desired that the underlying methods remain accessible to code in the `*commands*` table.

5.2.6.1 Database Extension

wrap-command-interface *rdb* Function

Returns relational database *rdb* wrapped with additional commands defined in its *commands* table.

add-command-tables *rdb* Function

The relational database *rdb* must be mutable. *add-command-tables* adds a *command* table to *rdb*; then returns `(wrap-command-interface` *rdb*`)`.

open-command-database *filename* Function
open-command-database *filename base-table-type* Function

Returns an open enhanced relational database associated with *filename*. The database will be opened with base-table type *base-table-type*) if supplied. If *base-table-type* is not supplied, `open-command-database` will attempt to deduce the correct base-table-type. If the database can not be opened or if it lacks the `*commands*` table, `#f` is returned.

open-command-database! *filename* Function
open-command-database! *filename base-table-type* Function

Returns *mutable* open enhanced relational database . . .

open-command-database *database* Function

Returns *database* if it is an immutable relational database; #f otherwise.

open-command-database! *database* Function

Returns *database* if it is a mutable relational database; #f otherwise.

5.2.6.2 Command Intrinsics

Some commands are defined in all extended relational-databases. The are called just like Section 5.2.4 [Database Operations], page 133.

add-domain *domain-row* Function

Adds *domain-row* to the *domains* table if there is no row in the domains table associated with key `(car` *domain-row*`)` and returns `#t`. Otherwise returns `#f`.

For the fields and layout of the domain table, See Section 5.2.6 [Catalog Representation], page 138. Currently, these fields are

- domain-name

- foreign-table
- domain-integrity-rule
- type-id
- type-param

The following example adds 3 domains to the 'build' database. 'Optstring' is either a string or #f. filename is a string and build-whats is a symbol.

```
(for-each (build 'add-domain)
          '((optstring #f
                       (lambda (x) (or (not x) (string? x)))
                       string
                       #f)
            (filename #f #f string #f)
            (build-whats #f #f symbol #f)))
```

delete-domain *domain-name* Function
Removes and returns the *domain-name* row from the *domains* table.

domain-checker *domain* Function
Returns a procedure to check an argument for conformance to domain *domain*.

5.2.6.3 Define-tables Example

The following example shows a new database with the name of 'foo.db' being created with tables describing processor families and processor/os/compiler combinations.

The database command define-tables is defined to call define-tables with its arguments. The database is also configured to print 'Welcome' when the database is opened. The database is then closed and reopened.

```
(require 'databases)
(define my-rdb (create-database "foo.db" 'alist-table))

(define-tables my-rdb
  '(*commands*
    ((name symbol))
    ((parameters parameter-list)
     (procedure expression)
     (documentation string))
    ((define-tables
      no-parameters
      no-parameter-names
      (lambda (rdb) (lambda specs (apply define-tables rdb specs)))
      "Create or Augment tables from list of specs")
     (*initialize*
      no-parameters
      no-parameter-names
      (lambda (rdb) (display "Welcome") (newline) rdb)
```

```
          "Print Welcome"))))

((my-rdb 'define-tables)
 '(processor-family
   ((family    atom))
   ((also-ran  processor-family))
   ((m68000           #f)
    (m68030           m68000)
    (i386             8086)
    (8086             #f)
    (powerpc          #f)))

 '(platform
   ((name      symbol))
   ((processor processor-family)
    (os        symbol)
    (compiler  symbol))
   ((aix              powerpc aix     -)
    (amiga-dice-c     m68000  amiga   dice-c)
    (amiga-aztec      m68000  amiga   aztec)
    (amiga-sas/c-5.10 m68000  amiga   sas/c)
    (atari-st-gcc     m68000  atari   gcc)
    (atari-st-turbo-c m68000  atari   turbo-c)
    (borland-c-3.1    8086    ms-dos  borland-c)
    (djgpp            i386    ms-dos  gcc)
    (linux            i386    linux   gcc)
    (microsoft-c      8086    ms-dos  microsoft-c)
    (os/2-emx         i386    os/2    gcc)
    (turbo-c-2        8086    ms-dos  turbo-c)
    (watcom-9.0       i386    ms-dos  watcom))))

((my-rdb 'close-database))

(set! my-rdb (open-database "foo.db" 'alist-table))
⊣
Welcome
```

5.2.6.4 The *commands* Table

The table `*commands*` in an *enhanced* relational-database has the fields (with domains):

```
PRI name          symbol
    parameters    parameter-list
    procedure     expression
    documentation string
```

The `parameters` field is a foreign key (domain `parameter-list`) of the `*catalog-data*` table and should have the value of a table described by `*parameter-columns*`. This `parameter-list` table describes the arguments suitable for passing to the associated command. The intent of this table is to be of a form such that different user-interfaces (for instance, pull-down menus or plain-text queries) can operate from the same table. A `parameter-list` table has the following fields:

```
PRI index       uint
    name        symbol
    arity       parameter-arity
    domain      domain
    defaulter   expression
    expander    expression
    documentation string
```

The `arity` field can take the values:

`single` Requires a single parameter of the specified domain.

`optional` A single parameter of the specified domain or zero parameters is acceptable.

`boolean` A single boolean parameter or zero parameters (in which case `#f` is substituted) is acceptable.

`nary` Any number of parameters of the specified domain are acceptable. The argument passed to the command function is always a list of the parameters.

`nary1` One or more of parameters of the specified domain are acceptable. The argument passed to the command function is always a list of the parameters.

The `domain` field specifies the domain which a parameter or parameters in the `index`th field must satisfy.

The `defaulter` field is an expression whose value is either `#f` or a procedure of one argument (the parameter-list) which returns a *list* of the default value or values as appropriate. Note that since the `defaulter` procedure is called every time a default parameter is needed for this column, *sticky* defaults can be implemented using shared state with the domain-integrity-rule.

5.2.6.5 Command Service

make-command-server *rdb table-name* Function

Returns a procedure of 2 arguments, a (symbol) command and a call-back procedure. When this returned procedure is called, it looks up *command* in table *table-name* and calls the call-back procedure with arguments:

command The *command*

command-value
The result of evaluating the expression in the *procedure* field of *table-name* and calling it with *rdb*.

parameter-name
A list of the *official* name of each parameter. Corresponds to the `name` field of the *command*'s parameter-table.

positions A list of the positive integer index of each parameter. Corresponds to the `index` field of the *command*'s parameter-table.

arities A list of the arities of each parameter. Corresponds to the `arity` field of the *command*'s parameter-table. For a description of `arity` see table above.

types A list of the type name of each parameter. Correspnds to the `type-id` field of the contents of the `domain` of the *command*'s parameter-table.

defaulters A list of the defaulters for each parameter. Corresponds to the `defaulters` field of the *command*'s parameter-table.

domain-integrity-rules
A list of procedures (one for each parameter) which tests whether a value for a parameter is acceptable for that parameter. The procedure should be called with each datum in the list for `nary` arity parameters.

aliases A list of lists of `(alias parameter-name)`. There can be more than one alias per *parameter-name*.

For information about parameters, See Section 3.4.4 [Parameter lists], page 54.

5.2.6.6 Command Example

Here is an example of setting up a command with arguments and parsing those arguments from a `getopt` style argument list (see Section 3.4.1 [Getopt], page 51).

```
(require 'databases)
(require 'fluid-let)
(require 'parameters)
(require 'getopt)

(define my-rdb (create-command-database #f 'alist-table))

(define-tables my-rdb
  '(foo-params
    *parameter-columns*
    *parameter-columns*
    ((1 single-string single string
        (lambda (pl) '("str")) #f "single string")
     (2 nary-symbols nary symbol
        (lambda (pl) '()) #f "zero or more symbols")
     (3 nary1-symbols nary1 symbol
        (lambda (pl) '(symb)) #f "one or more symbols")
     (4 optional-number optional uint
        (lambda (pl) '()) #f "zero or one number")
     (5 flag boolean boolean
        (lambda (pl) '(#f)) #f "a boolean flag")))
  '(foo-pnames
    ((name string))
    ((parameter-index uint))
    (("s" 1)
     ("single-string" 1)
     ("n" 2)
     ("nary-symbols" 2)
     ("N" 3)
     ("nary1-symbols" 3)
     ("o" 4)
     ("optional-number" 4)
     ("f" 5)
     ("flag" 5)))
  '(my-commands
    ((name symbol))
    ((parameters parameter-list)
     (parameter-names parameter-name-translation)
     (procedure expression)
     (documentation string))
    ((foo
      foo-params
      foo-pnames
      (lambda (rdb) (lambda args (print args)))
      "test command arguments"))))

(define (dbutil:serve-command-line rdb command-table
                                   command argc argv)
  (set! argv (if (vector? argv) (vector->list argv) argv))
  ((make-command-server rdb command-table)
   command
   (lambda (comname comval options positions
                    arities types defaulters dirs aliases)
     (apply comval (getopt->arglist
                    argc argv options positions
                    arities types defaulters dirs aliases)))))

(define (cmd . opts)
  (fluid-let ((*optind* 1))
    (printf "%-34s ⇒ "
            (call-with-output-string
             (lambda (pt) (write (cons 'cmd opts) pt))))
    (set! opts (cons "cmd" opts))
    (force-output)
    (dbutil:serve-command-line
     my-rdb 'my-commands 'foo (length opts) opts)))

(cmd)                              ⇒ ("str" () (symb) () #f)
(cmd "-f")                         ⇒ ("str" () (symb) () #t)
(cmd "--flag")                     ⇒ ("str" () (symb) () #t)
(cmd "-o177")                      ⇒ ("str" () (symb) (177) #f)
(cmd "-o" "177")                   ⇒ ("str" () (symb) (177) #f)
```

```
(cmd "--optional" "621")          ⇒ ("str" () (symb) (621) #f)
(cmd "--optional=621")            ⇒ ("str" () (symb) (621) #f)
(cmd "-s" "speciality")           ⇒ ("speciality" () (symb) () #f)
(cmd "-sspeciality")              ⇒ ("speciality" () (symb) () #f)
(cmd "--single" "serendipity")    ⇒ ("serendipity" () (symb) () #f)
(cmd "--single=serendipity")      ⇒ ("serendipity" () (symb) () #f)
(cmd "-n" "gravity" "piety")      ⇒ ("str" () (piety gravity) () #f)
(cmd "-ngravity" "piety")         ⇒ ("str" () (piety gravity) () #f)
(cmd "--nary" "chastity")         ⇒ ("str" () (chastity) () #f)
(cmd "--nary=chastity" "")        ⇒ ("str" () ( chastity) () #f)
(cmd "-N" "calamity")             ⇒ ("str" () (calamity) () #f)
(cmd "-Ncalamity")                ⇒ ("str" () (calamity) () #f)
(cmd "--nary1" "surety")          ⇒ ("str" () (surety) () #f)
(cmd "--nary1=surety")            ⇒ ("str" () (surety) () #f)
(cmd "-N" "levity" "fealty")      ⇒ ("str" () (fealty levity) () #f)
(cmd "-Nlevity" "fealty")         ⇒ ("str" () (fealty levity) () #f)
(cmd "--nary1" "surety" "brevity") ⇒ ("str" () (brevity surety) () #f)
(cmd "--nary1=surety" "brevity")  ⇒ ("str" () (brevity surety) () #f)
(cmd "-?")
⊣
Usage: cmd [OPTION ARGUMENT ...] ...

  -f, --flag
  -o, --optional[=]<number>
  -n, --nary[=]<symbols> ...
  -N, --nary1[=]<symbols> ...
  -s, --single[=]<string>

ERROR: getopt->parameter-list "unrecognized option" "-?"
```

5.2.7 Database Reports

Code for generating database reports is in '`report.scm`'. After writing it using `format`, I discovered that Common-Lisp `format` is not useable for this application because there is no mechanismm for truncating fields. '`report.scm`' needs to be rewritten using `printf`.

create-report *rdb destination report-name table* Procedure
create-report *rdb destination report-name* Procedure

The symbol *report-name* must be primary key in the table named `*reports*` in the relational database *rdb*. *destination* is a port, string, or symbol. If *destination* is a:

port	The table is created as ascii text and written to that port.
string	The table is created as ascii text and written to the file named by *destination*.
symbol	*destination* is the primary key for a row in the table named *printers*.

The report is prepared as follows:

- `Format` (see Section 3.2 [Format], page 39) is called with the `header` field and the (list of) `column-names` of the table.
- `Format` is called with the `reporter` field and (on successive calls) each record in the natural order for the table. A count is kept of the number of newlines output by format. When the number of newlines to be output exceeds the number of lines per page, the set of lines will be broken if there are more than `minimum-break` left on this page and the number of lines for this row is larger or equal to twice `minimum-break`.
- `Format` is called with the `footer` field and the (list of) `column-names` of the table. The footer field should not output a newline.
- A new page is output.
- This entire process repeats until all the rows are output.

Each row in the table *reports* has the fields:

name
: The report name.

default-table
: The table to report on if none is specified.

header, footer
: A `format` string. At the beginning and end of each page respectively, `format` is called with this string and the (list of) column-names of this table.

reporter
: A `format` string. For each row in the table, `format` is called with this string and the row.

minimum-break
: The minimum number of lines into which the report lines for a row can be broken. Use `0` if a row's lines should not be broken over page boundaries.

Each row in the table *printers* has the fields:

name
: The printer name.

print-procedure
: The procedure to call to actually print.

5.2.8 Database Browser

(require 'database-browse)

browse *database* Procedure
: Prints the names of all the tables in *database* and sets browse's default to *database*.

browse Procedure
: Prints the names of all the tables in the default database.

browse *table-name* Procedure
: For each record of the table named by the symbol *table-name*, prints a line composed of all the field values.

browse *pathname* Procedure

Opens the database named by the string *pathname*, prints the names of all its tables, and sets browse's default to the database.

browse *database table-name* Procedure

Sets browse's default to *database* and prints the records of the table named by the symbol *table-name*.

browse *pathname table-name* Procedure

Opens the database named by the string *pathname* and sets browse's default to it; `browse` prints the records of the table named by the symbol *table-name*.

5.3 Weight-Balanced Trees

```
(require 'wt-tree)
```

Balanced binary trees are a useful data structure for maintaining large sets of ordered objects or sets of associations whose keys are ordered. MIT Scheme has an comprehensive implementation of weight-balanced binary trees which has several advantages over the other data structures for large aggregates:

- In addition to the usual element-level operations like insertion, deletion and lookup, there is a full complement of collection-level operations, like set intersection, set union and subset test, all of which are implemented with good orders of growth in time and space. This makes weight balanced trees ideal for rapid prototyping of functionally derived specifications.
- An element in a tree may be indexed by its position under the ordering of the keys, and the ordinal position of an element may be determined, both with reasonable efficiency.
- Operations to find and remove minimum element make weight balanced trees simple to use for priority queues.
- The implementation is *functional* rather than *imperative*. This means that operations like 'inserting' an association in a tree do not destroy the old tree, in much the same way that `(+ 1 x)` modifies neither the constant 1 nor the value bound to `x`. The trees are referentially transparent thus the programmer need not worry about copying the trees. Referential transparency allows space efficiency to be achieved by sharing subtrees.

These features make weight-balanced trees suitable for a wide range of applications, especially those that require large numbers of sets or discrete maps. Applications that have a few global databases and/or concentrate on element-level operations like insertion and lookup are probably better off using hash-tables or red-black trees.

The *size* of a tree is the number of associations that it contains. Weight balanced binary trees are balanced to keep the sizes of the subtrees of each node within a constant factor of each other. This ensures logarithmic times for single-path operations (like lookup and insertion). A weight balanced tree takes space that is proportional to the number of associations in the tree. For the current implementation, the constant of proportionality is six words per association.

Weight balanced trees can be used as an implementation for either discrete sets or discrete maps (associations). Sets are implemented by ignoring the datum that is associated with the key. Under this scheme if an associations exists in the tree this indicates that the key of the association is a member of the set. Typically a value such as `()`, `#t` or `#f` is associated with the key.

Many operations can be viewed as computing a result that, depending on whether the tree arguments are thought of as sets or maps, is known by two different names. An example is `wt-tree/member?`, which, when regarding the tree argument as a set, computes the set membership operation, but, when regarding the tree as a discrete map, `wt-tree/member?` is the predicate testing if the map is defined at an element in its domain. Most names in this package have been chosen based on interpreting the trees as sets, hence the name `wt-tree/member?` rather than `wt-tree/defined-at?`.

The weight balanced tree implementation is a run-time-loadable option. To use weight balanced trees, execute

```
(load-option 'wt-tree)
```

once before calling any of the procedures defined here.

5.3.1 Construction of Weight-Balanced Trees

Binary trees require there to be a total order on the keys used to arrange the elements in the tree. Weight balanced trees are organized by *types*, where the type is an object encapsulating the ordering relation. Creating a tree is a two-stage process. First a tree type must be created from the predicate which gives the ordering. The tree type is then used for making trees, either empty or singleton trees or trees from other aggregate structures like association lists. Once created, a tree 'knows' its type and the type is used to test compatibility between trees in operations taking two trees. Usually a small number of tree types are created at the beginning of a program and used many times throughout the program's execution.

make-wt-tree-type *key<?* procedure+

This procedure creates and returns a new tree type based on the ordering predicate *key<?*. *Key<?* must be a total ordering, having the property that for all key values `a`, `b` and `c`:

```
(key<? a a)                         ⇒ #f
(and (key<? a b) (key<? b a))       ⇒ #f
(if (and (key<? a b) (key<? b c))
    (key<? a c)
    #t)                             ⇒ #t
```

Two key values are assumed to be equal if neither is less than the other by *key<?*.

Each call to `make-wt-tree-type` returns a distinct value, and trees are only compatible if their tree types are `eq?`. A consequence is that trees that are intended to be used in binary tree operations must all be created with a tree type originating from the same call to `make-wt-tree-type`.

number-wt-type variable+

A standard tree type for trees with numeric keys. `Number-wt-type` could have been defined by

```
(define number-wt-type (make-wt-tree-type  <))
```

string-wt-type variable+

A standard tree type for trees with string keys. `String-wt-type` could have been defined by

```
(define string-wt-type (make-wt-tree-type  string<?))
```

make-wt-tree *wt-tree-type* procedure+

This procedure creates and returns a newly allocated weight balanced tree. The tree is empty, i.e. it contains no associations. *Wt-tree-type* is a weight balanced tree type obtained by calling `make-wt-tree-type`; the returned tree has this type.

singleton-wt-tree *wt-tree-type key datum* procedure+

This procedure creates and returns a newly allocated weight balanced tree. The tree contains a single association, that of *datum* with *key*. *Wt-tree-type* is a weight balanced tree type obtained by calling `make-wt-tree-type`; the returned tree has this type.

alist->wt-tree *tree-type alist* procedure+

Returns a newly allocated weight-balanced tree that contains the same associations as *alist*. This procedure is equivalent to:

```
(lambda (type alist)
  (let ((tree (make-wt-tree type)))
    (for-each (lambda (association)
                (wt-tree/add! tree
                              (car association)
                              (cdr association)))
              alist)
    tree))
```

5.3.2 Basic Operations on Weight-Balanced Trees

This section describes the basic tree operations on weight balanced trees. These operations are the usual tree operations for insertion, deletion and lookup, some predicates and a procedure for determining the number of associations in a tree.

wt-tree? *object* procedure+

Returns `#t` if *object* is a weight-balanced tree, otherwise returns `#f`.

wt-tree/empty? *wt-tree* procedure+

Returns `#t` if *wt-tree* contains no associations, otherwise returns `#f`.

wt-tree/size *wt-tree* procedure+

Returns the number of associations in *wt-tree*, an exact non-negative integer. This operation takes constant time.

wt-tree/add *wt-tree key datum* procedure+

Returns a new tree containing all the associations in *wt-tree* and the association of *datum* with *key*. If *wt-tree* already had an association for *key*, the new association overrides the old. The average and worst-case times required by this operation are proportional to the logarithm of the number of associations in *wt-tree*.

wt-tree/add! *wt-tree key datum* procedure+

Associates *datum* with *key* in *wt-tree* and returns an unspecified value. If *wt-tree* already has an association for *key*, that association is replaced. The average and worst-case times required by this operation are proportional to the logarithm of the number of associations in *wt-tree*.

wt-tree/member? *key wt-tree* procedure+

Returns `#t` if *wt-tree* contains an association for *key*, otherwise returns `#f`. The average and worst-case times required by this operation are proportional to the logarithm of the number of associations in *wt-tree*.

wt-tree/lookup *wt-tree key default* procedure+

Returns the datum associated with *key* in *wt-tree*. If *wt-tree* doesn't contain an association for *key*, *default* is returned. The average and worst-case times required by this operation are proportional to the logarithm of the number of associations in *wt-tree*.

wt-tree/delete *wt-tree key* procedure+

Returns a new tree containing all the associations in *wt-tree*, except that if *wt-tree* contains an association for *key*, it is removed from the result. The average and worst-case times required by this operation are proportional to the logarithm of the number of associations in *wt-tree*.

wt-tree/delete! *wt-tree key* procedure+

If *wt-tree* contains an association for *key* the association is removed. Returns an unspecified value. The average and worst-case times required by this operation are proportional to the logarithm of the number of associations in *wt-tree*.

5.3.3 Advanced Operations on Weight-Balanced Trees

In the following the *size* of a tree is the number of associations that the tree contains, and a *smaller* tree contains fewer associations.

wt-tree/split< *wt-tree bound* procedure+

Returns a new tree containing all and only the associations in *wt-tree* which have a key that is less than *bound* in the ordering relation of the tree type of *wt-tree*. The average and worst-case times required by this operation are proportional to the logarithm of the size of *wt-tree*.

wt-tree/split> *wt-tree bound* procedure+

Returns a new tree containing all and only the associations in *wt-tree* which have a key that is greater than *bound* in the ordering relation of the tree type of *wt-tree*. The average and worst-case times required by this operation are proportional to the logarithm of size of *wt-tree*.

wt-tree/union *wt-tree-1 wt-tree-2* procedure+

Returns a new tree containing all the associations from both trees. This operation is asymmetric: when both trees have an association for the same key, the returned tree associates the datum from *wt-tree-2* with the key. Thus if the trees are viewed as discrete maps then `wt-tree/union` computes the map override of *wt-tree-1* by *wt-tree-2*. If the trees are viewed as sets the result is the set union of the arguments. The worst-case time required by this operation is proportional to the sum of the sizes of both trees. If the minimum key of one tree is greater than the maximum key of the other tree then the time required is at worst proportional to the logarithm of the size of the larger tree.

wt-tree/intersection *wt-tree-1 wt-tree-2* procedure+

Returns a new tree containing all and only those associations from *wt-tree-1* which have keys appearing as the key of an association in *wt-tree-2*. Thus the associated data in the result are those from *wt-tree-1*. If the trees are being used as sets the result is the set intersection of the arguments. As a discrete map operation, `wt-tree/intersection` computes the domain restriction of *wt-tree-1* to (the domain of) *wt-tree-2*. The time required by this operation is never worse that proportional to the sum of the sizes of the trees.

wt-tree/difference *wt-tree-1 wt-tree-2* procedure+

Returns a new tree containing all and only those associations from *wt-tree-1* which have keys that *do not* appear as the key of an association in *wt-tree-2*. If the trees are viewed as sets the result is the asymmetric set difference of the arguments. As a discrete map operation, it computes the domain restriction of *wt-tree-1* to the complement of (the domain of) *wt-tree-2*. The time required by this operation is never worse that proportional to the sum of the sizes of the trees.

wt-tree/subset? *wt-tree-1 wt-tree-2* procedure+

Returns `#t` iff the key of each association in *wt-tree-1* is the key of some association in *wt-tree-2*, otherwise returns `#f`. Viewed as a set operation, `wt-tree/subset?` is the improper subset predicate. A proper subset predicate can be constructed:

```
(define (proper-subset? s1 s2)
  (and (wt-tree/subset? s1 s2)
              (< (wt-tree/size s1) (wt-tree/size s2))))
```

As a discrete map operation, `wt-tree/subset?` is the subset test on the domain(s) of the map(s). In the worst-case the time required by this operation is proportional to the size of *wt-tree-1*.

wt-tree/set-equal? *wt-tree-1 wt-tree-2* procedure+

Returns `#t` iff for every association in *wt-tree-1* there is an association in *wt-tree-2* that has the same key, and *vice versa*.

Viewing the arguments as sets `wt-tree/set-equal?` is the set equality predicate. As a map operation it determines if two maps are defined on the same domain.

This procedure is equivalent to

```
(lambda (wt-tree-1 wt-tree-2)
  (and (wt-tree/subset? wt-tree-1 wt-tree-2
       (wt-tree/subset? wt-tree-2 wt-tree-1)))
```

In the worst-case the time required by this operation is proportional to the size of the smaller tree.

wt-tree/fold *combiner initial wt-tree* procedure+

This procedure reduces *wt-tree* by combining all the associations, using an reverse in-order traversal, so the associations are visited in reverse order. *Combiner* is a procedure of three arguments: a key, a datum and the accumulated result so far. Provided *combiner* takes time bounded by a constant, `wt-tree/fold` takes time proportional to the size of *wt-tree*.

A sorted association list can be derived simply:

```
(wt-tree/fold  (lambda (key datum list)
                 (cons (cons key datum) list))
               '()
               wt-tree))
```

The data in the associations can be summed like this:

```
(wt-tree/fold  (lambda (key datum sum) (+ sum datum))
               0
               wt-tree)
```

wt-tree/for-each *action wt-tree* procedure+

This procedure traverses the tree in-order, applying *action* to each association. The associations are processed in increasing order of their keys. *Action* is a procedure of two arguments which take the key and datum respectively of the association. Provided *action* takes time bounded by a constant, `wt-tree/for-each` takes time proportional to in the size of *wt-tree*. The example prints the tree:

```
(wt-tree/for-each (lambda (key value)
                    (display (list key value)))
                  wt-tree))
```

5.3.4 Indexing Operations on Weight-Balanced Trees

Weight balanced trees support operations that view the tree as sorted sequence of associations. Elements of the sequence can be accessed by position, and the position of an element in the sequence can be determined, both in logarthmic time.

wt-tree/index *wt-tree index* procedure+
wt-tree/index-datum *wt-tree index* procedure+
wt-tree/index-pair *wt-tree index* procedure+

Returns the 0-based *index*th association of *wt-tree* in the sorted sequence under the tree's ordering relation on the keys. `wt-tree/index` returns the *index*th key, `wt-tree/index-datum` returns the datum associated with the *index*th key and `wt-tree/index-pair` returns a new pair (*key* . *datum*) which is the `cons` of the *index*th key and its datum. The average and worst-case times required by this operation are proportional to the logarithm of the number of associations in the tree.

These operations signal an error if the tree is empty, if *index*`<0`, or if *index* is greater than or equal to the number of associations in the tree.

Indexing can be used to find the median and maximum keys in the tree as follows:

```
median:   (wt-tree/index wt-tree
                         (quotient (wt-tree/size wt-tree) 2))

maximum:  (wt-tree/index wt-tree
                         (-1+ (wt-tree/size wt-tree)))
```

wt-tree/rank *wt-tree key* procedure+

Determines the 0-based position of *key* in the sorted sequence of the keys under the tree's ordering relation, or `#f` if the tree has no association with for *key*. This procedure returns either an exact non-negative integer or `#f`. The average and worst-case times required by this operation are proportional to the logarithm of the number of associations in the tree.

wt-tree/min *wt-tree* procedure+
wt-tree/min-datum *wt-tree* procedure+
wt-tree/min-pair *wt-tree* procedure+

Returns the association of *wt-tree* that has the least key under the tree's ordering relation. `wt-tree/min` returns the least key, `wt-tree/min-datum` returns the datum associated with the least key and `wt-tree/min-pair` returns a new pair `(key . datum)` which is the `cons` of the minimum key and its datum. The average and worst-case times required by this operation are proportional to the logarithm of the number of associations in the tree.

These operations signal an error if the tree is empty. They could be written

```
(define (wt-tree/min tree)        (wt-tree/index tree 0))
(define (wt-tree/min-datum tree)  (wt-tree/index-datum tree 0))
(define (wt-tree/min-pair tree)   (wt-tree/index-pair tree 0))
```

wt-tree/delete-min *wt-tree* procedure+

Returns a new tree containing all of the associations in *wt-tree* except the association with the least key under the *wt-tree*'s ordering relation. An error is signalled if the tree is empty. The average and worst-case times required by this operation are proportional to the logarithm of the number of associations in the tree. This operation is equivalent to

```
(wt-tree/delete wt-tree (wt-tree/min wt-tree))
```

wt-tree/delete-min! *wt-tree* procedure+

Removes the association with the least key under the *wt-tree*'s ordering relation. An error is signalled if the tree is empty. The average and worst-case times required by this operation are proportional to the logarithm of the number of associations in the tree. This operation is equivalent to

```
(wt-tree/delete! wt-tree (wt-tree/min wt-tree))
```

6 Other Packages

6.1 Data Structures

6.1.1 Arrays

```
(require 'array)
```

array? *obj* Function

Returns `#t` if the *obj* is an array, and `#f` if not.

Note: Arrays are not disjoint from other Scheme types. Strings and vectors also satisfy `array?`. A disjoint array predicate can be written:

```
(define (strict-array? obj)
  (and (array? obj) (not (string? obj)) (not (vector? obj))))
```

array=? *array1 array2* Function

Returns `#t` if *array1* and *array2* have the same rank and shape and the corresponding elements of *array1* and *array2* are `equal?`.

```
(array=? (create-array '#(foo) 3 3) (create-array '#(foo) '(0 2) '(0 2)))
  ⇒ #t
```

create-array *prototype bound1 bound2 . . .* Function

Creates and returns an array of type *prototype* with dimensions *bound1*, *bound2*, . . . and filled with elements from *prototype*. *prototype* must be an array, vector, or string. The implementation-dependent type of the returned array will be the same as the type of *prototype*; except if that would be a vector or string with non-zero origin, in which case some variety of array will be returned.

If the *prototype* has no elements, then the initial contents of the returned array are unspecified. Otherwise, the returned array will be filled with the element at the origin of *prototype*.

These functions return a uniform array prototype enclosing the optional argument (which must be of the correct type). If the uniform-array type is supported by the implementation, then it is returned; promoting to the next larger precision type; promoting finally to vector.

ac64 *z* Function
ac64 Function

Returns a high-precision complex uniform-array prototype.

ac32 *z* Function
ac32 Function

Returns a complex uniform-array prototype.

ar64 *x* Function
ar64 Function

Returns a high-precision real uniform-array prototype.

ar32 *x* Function
ar32 Function

Returns a real uniform-array prototype.

as64 *n* Function
as64 Function

Returns an exact signed integer uniform-array prototype with at least 64 bits of precision.

as32 *n* Function
as32 Function

Returns an exact signed integer uniform-array prototype with at least 32 bits of precision.

as16 *n* Function
as16 Function

Returns an exact signed integer uniform-array prototype with at least 16 bits of precision.

as8 *n* Function
as8 Function

Returns an exact signed integer uniform-array prototype with at least 8 bits of precision.

au64 *k* Function
au64 Function

Returns an exact non-negative integer uniform-array prototype with at least 64 bits of precision.

au32 *k* Function
au32 Function

Returns an exact non-negative integer uniform-array prototype with at least 32 bits of precision.

au16 *k* Function
au16 Function

Returns an exact non-negative integer uniform-array prototype with at least 16 bits of precision.

au8 *k* Function
au8 Function

Returns an exact non-negative integer uniform-array prototype with at least 8 bits of precision.

at1 *bool* Function
at1 Function

Returns a boolean uniform-array prototype.

make-array *initial-value bound1 bound2 . . .* Function

Creates and returns an array with dimensions *bound1*, *bound2*, . . . and filled with *initial-value*.

`make-array` is a legacy function – now defined in terms of `create-array`.

```
(define (make-array initial-value . dimensions)
  (apply create-array (vector initial-value) dimensions))
```

When constructing an array, *bound* is either an inclusive range of indices expressed as a two element list, or an upper bound expressed as a single integer. So

```
(create-array '#(foo) 3 3) ≡ (make-array '#(foo) '(0 2) '(0 2))
```

make-shared-array *array mapper bound1 bound2 . . .* Function

`make-shared-array` can be used to create shared subarrays of other arrays. The *mapper* is a function that translates coordinates in the new array into coordinates in the old array. A *mapper* must be linear, and its range must stay within the bounds of the old array, but it can be otherwise arbitrary. A simple example:

```
(define fred (create-array '#(#f) 8 8))
(define freds-diagonal
  (make-shared-array fred (lambda (i) (list i i)) 8))
(array-set! freds-diagonal 'foo 3)
(array-ref fred 3 3)
   ⇒ FOO
(define freds-center
  (make-shared-array fred (lambda (i j) (list (+ 3 i) (+ 3 j)))
                     2 2))
(array-ref freds-center 0 0)
   ⇒ FOO
```

array-rank *obj* Function

Returns the number of dimensions of *obj*. If *obj* is not an array, 0 is returned.

array-shape *array* Function

Returns a list of inclusive bounds.

```
(array-shape (create-array '#() 3 5))
   ⇒ ((0 2) (0 4))
```

array-dimensions *array* Function

`array-dimensions` is similar to `array-shape` but replaces elements with a 0 minimum with one greater than the maximum.

```
(array-dimensions (create-array '#() 3 5))
   ⇒ (3 5)
```

array-in-bounds? *array index1 index2 . . .* Function

Returns `#t` if its arguments would be acceptable to `array-ref`.

array-ref *array index1 index2 . . .* Function

Returns the (*index1*, *index2*, . . .) element of *array*.

array-set! *array obj index1 index2 . . .* Function

Stores *obj* in the (*index1*, *index2*, . . .) element of *array*. The value returned by `array-set!` is unspecified.

6.1.2 Subarrays

```
(require 'subarray)
```

subarray *array select . . .* Function

selects a subset of an array. For *array* of rank n, there must be at least n *selects* arguments. For 0 <= j < n, *selectsj* is either an integer, a list of two integers within the range for the jth index, or #f.

When *selectsj* is a list of two integers, then the jth index is restricted to that subrange in the returned array.

When *selectsj* is #f, then the full range of the jth index is accessible in the returned array. An elided argument is equivalent to #f.

When *selectsj* is an integer, then the rank of the returned array is less than *array*, and only elements whose jth index equals *selectsj* are shared.

```
> (define ra '#2A((a b c) (d e f)))
#<unspecified>
> (subarray ra 0 #f)
#1A(a b c)
> (subarray ra 1 #f)
#1A(d e f)
> (subarray ra #f 1)
#1A(b e)
> (subarray ra '(0 1) #f)
#2A((a b c) (d e f))
> (subarray ra #f '(0 1))
#2A((a b) (d e))
> (subarray ra #f '(1 2))
#2A((b c) (e f))
```

subarray0 *array select . . .* Function

Behaves like subarray, but aligns the returned array origin to 0 . . . .

array-align *array coord . . .* Function

Returns an array shared with *array* but with a different origin. The *coords* are the exact integer coordinates of the new origin. Indexes corresponding to missing or #f coordinates are not realigned.

For example:

```
(define ra2 (create-array '#(5) '(5 9) '(-4 0)))
(array-shape ra2)                          ⇒ ((5 9) (-4 0))
(array-shape (array-align ra2 0 0))        ⇒ ((0 4) (0 4))
(array-shape (array-align ra2 0))          ⇒ ((0 4) (-4 0))
(array-shape (array-align ra2))            ⇒ ((5 9) (-4 0))
(array-shape (array-align ra2 0 #f))       ⇒ ((0 4) (-4 0))
(array-shape (array-align ra2 #f 0))       ⇒ ((5 9) (0 4))
```

array-trim *array trim . . .* Function

Returns a subarray sharing contents with *array* except for slices removed from either side of each dimension. Each of the *trims* is an exact integer indicating how much to trim. A positive *s* trims the data from the lower end and reduces the upper bound of the result; a negative *s* trims from the upper end and increases the lower bound.

For example:

```
(array-trim '#(0 1 2 3 4) 1)  ⇒ #1A(1 2 3 4) ;; shape is ((0 3))
(array-trim '#(0 1 2 3 4) -1) ⇒ #1A(0 1 2 3) ;; shape is ((1 4))

(require 'array-for-each)
(define (centered-difference ra)
  (array-map - (array-trim ra 1) (array-trim ra -1)))
(define (forward-difference ra)
  (array-map - (array-trim ra 1) ra))
(define (backward-difference ra)
  (array-map - ra (array-trim ra -1)))

(centered-difference '#(0 1 3 5 9 22))
  ⇒ #1A(3 4 6 17) ;;shape is ((1 4))
(backward-difference '#(0 1 3 5 9 22))
  ⇒ #1A(1 2 2 4 13) ;; shape is ((1 5))
(forward-difference '#(0 1 3 5 9 22))
  ⇒ #(1 2 2 4 13)  ;; shape is ((0 4))
```

6.1.3 Array Mapping

```
(require 'array-for-each)
```

array-map! *array0 proc array1 . . .* Function

array1, . . . must have the same number of dimensions as *array0* and have a range for each index which includes the range for the corresponding index in *array0*. *proc* is applied to each tuple of elements of *array1* . . . and the result is stored as the corresponding element in *array0*. The value returned is unspecified. The order of application is unspecified.

array-for-each *proc array0 . . .* Function

proc is applied to each tuple of elements of *array0* . . . in row-major order. The value returned is unspecified.

array-indexes *array* Function

Returns an array of lists of indexes for *array* such that, if *li* is a list of indexes for which *array* is defined, (equal? *li* (apply array-ref (array-indexes *array*) *li*)).

array-index-map! *array proc* Function

applies *proc* to the indices of each element of *array* in turn, storing the result in the corresponding element. The value returned and the order of application are unspecified.

One can implement *array-indexes* as

```
(define (array-indexes array)
    (let ((ra (apply create-array '#() (array-shape array))))
      (array-index-map! ra (lambda x x))
      ra))
```

Another example:

```
(define (apl:index-generator n)
    (let ((v (make-vector n 1)))
      (array-index-map! v (lambda (i) i))
      v))
```

array-copy! *source destination* Function

Copies every element from vector or array *source* to the corresponding element of *destination*. *destination* must have the same rank as *source*, and be at least as large in each dimension. The order of copying is unspecified.

6.1.4 Association Lists

```
(require 'alist)
```

Alist functions provide utilities for treating a list of key-value pairs as an associative database. These functions take an equality predicate, *pred*, as an argument. This predicate should be repeatable, symmetric, and transitive.

Alist functions can be used with a secondary index method such as hash tables for improved performance.

predicate->asso *pred* Function

Returns an *association function* (like `assq`, `assv`, or `assoc`) corresponding to *pred*. The returned function returns a key-value pair whose key is `pred`-equal to its first argument or `#f` if no key in the alist is *pred*-equal to the first argument.

alist-inquirer *pred* Function

Returns a procedure of 2 arguments, *alist* and *key*, which returns the value associated with *key* in *alist* or `#f` if *key* does not appear in *alist*.

alist-associator *pred* Function

Returns a procedure of 3 arguments, *alist*, *key*, and *value*, which returns an alist with *key* and *value* associated. Any previous value associated with *key* will be lost. This returned procedure may or may not have side effects on its *alist* argument. An example of correct usage is:

```
(define put (alist-associator string-ci=?))
(define alist '())
(set! alist (put alist "Foo" 9))
```

alist-remover *pred* Function

Returns a procedure of 2 arguments, *alist* and *key*, which returns an alist with an association whose *key* is key removed. This returned procedure may or may not have side effects on its *alist* argument. An example of correct usage is:

```
(define rem (alist-remover string-ci=?))
(set! alist (rem alist "foo"))
```

alist-map *proc alist* Function

Returns a new association list formed by mapping *proc* over the keys and values of *alist*. *proc* must be a function of 2 arguments which returns the new value part.

alist-for-each *proc alist* Function

Applies *proc* to each pair of keys and values of *alist*. *proc* must be a function of 2 arguments. The returned value is unspecified.

6.1.5 Byte

```
(require 'byte)
```

Some algorithms are expressed in terms of arrays of small integers. Using Scheme strings to implement these arrays is not portable vis-a-vis the correspondence between integers and characters and non-ascii character sets. These functions abstract the notion of a *byte*.

byte-ref *bytes k* Function

k must be a valid index of *bytes*. `byte-ref` returns byte *k* of *bytes* using zero-origin indexing.

byte-set! *bytes k byte* Procedure

k must be a valid index of *bytes*%, and *byte* must be a small integer. `Byte-set!` stores *byte* in element *k* of *bytes* and returns an unspecified value.

make-bytes *k* Function
make-bytes *k byte* Function

`Make-bytes` returns a newly allocated byte-array of length *k*. If *byte* is given, then all elements of the byte-array are initialized to *byte*, otherwise the contents of the byte-array are unspecified.

bytes-length *bytes* Function

`bytes-length` returns length of byte-array *bytes*.

bytes *byte ...* Function

Returns a newly allocated byte-array composed of the arguments.

bytes->list *bytes* Function
list->bytes *bytes* Function

`Bytes->list` returns a newly allocated list of the bytes that make up the given byte-array. `List->bytes` returns a newly allocated byte-array formed from the small integers in the list *bytes*. `Bytes->list` and `list->bytes` are inverses so far as `equal?` is concerned.

Input and output of bytes should be with ports opened in *binary* mode (see Section 1.5.4 [Input/Output], page 8). Calling `open-file` with 'rb or 'wb modes argument will return a binary port if the Scheme implementation supports it.

write-byte *byte* Function
write-byte *byte port* Function

Writes the byte *byte* (not an external representation of the byte) to the given *port* and returns an unspecified value. The *port* argument may be omitted, in which case it defaults to the value returned by `current-output-port`.

read-byte Function
read-byte *port* Function

Returns the next byte available from the input *port*, updating the *port* to point to the following byte. If no more bytes are available, an end of file object is returned. *Port* may be omitted, in which case it defaults to the value returned by `current-input-port`.

6.1.6 MAT-File Format

```
(require 'matfile)
```

http://www.mathworks.com/access/helpdesk/help/pdf_doc/matlab/matfile_format.pdf

This package reads MAT-File Format version 4 (MATLAB) binary data files. MAT-files written from big-endian or little-endian computers having IEEE format numbers are currently supported. Support for files written from VAX or Cray machines could also be added.

The numeric and text matrix types handled; support for *sparse* matrices awaits a sample file.

matfile:read *filename* Function

filename should be a string naming an existing file containing a MATLAB Version 4 MAT-File. The `matfile:read` procedure reads matrices from the file and returns a list of the results; a list of the name string and array for each matrix.

matfile:load *filename* Function

filename should be a string naming an existing file containing a MATLAB Version 4 MAT-File. The `matfile:load` procedure reads matrices from the file and defines the `string-ci->symbol` for each matrix to its corresponding array. `matfile:load` returns a list of the symbols defined.

6.1.7 Portable Image Files

```
(require 'pnm)
```

pnm:type-dimensions *path* Function

The string *path* must name a *portable bitmap graphics* file. `pnm:type-dimensions` returns a list of 4 items:

1. A symbol describing the type of the file named by *path*.
2. The image width in pixels.
3. The image height in pixels.
4. The maximum value of pixels assume in the file.

The current set of file-type symbols is:

pbm pbm-raw	Black-and-White image; pixel values are 0 or 1.
pgm pgm-raw	Gray (monochrome) image; pixel values are from 0 to *maxval* specified in file header.
ppm ppm-raw	RGB (full color) image; red, green, and blue interleaved pixel values are from 0 to *maxval*

pnm:image-file->array *path array* Function

Reads the *portable bitmap graphics* file named by *path* into *array*. *array* must be the correct size and type for *path*. *array* is returned.

pnm:image-file->array *path* Function

`pnm:image-file->array` creates and returns an array with the *portable bitmap graphics* file named by *path* read into it.

pnm:array-write *type array maxval path* Procedure

Writes the contents of *array* to a *type* image file named *path*. The file will have pixel values between 0 and *maxval*, which must be compatible with *type*. For '`pbm`' files, *maxval* must be '`1`'.

6.1.8 Collections

```
(require 'collect)
```

Routines for managing collections. Collections are aggregate data structures supporting iteration over their elements, similar to the Dylan(TM) language, but with a different interface. They have *elements* indexed by corresponding *keys*, although the keys may be implicit (as with lists).

New types of collections may be defined as YASOS objects (see Section 2.8 [Yasos], page 28). They must support the following operations:

- `(collection?` *self*`)` (always returns `#t`);
- `(size` *self*`)` returns the number of elements in the collection;
- `(print` *self port*`)` is a specialized print operation for the collection which prints a suitable representation on the given *port* or returns it as a string if *port* is `#t`;
- `(gen-elts` *self*`)` returns a thunk which on successive invocations yields elements of *self* in order or gives an error if it is invoked more than `(size` *self*`)` times;
- `(gen-keys` *self*`)` is like `gen-elts`, but yields the collection's keys in order.

They might support specialized `for-each-key` and `for-each-elt` operations.

collection? *obj* Function

A predicate, true initially of lists, vectors and strings. New sorts of collections must answer `#t` to `collection?`.

map-elts *proc collection1 . . .* Procedure
do-elts *proc collection1 . . .* Procedure

proc is a procedure taking as many arguments as there are *collections* (at least one). The *collections* are iterated over in their natural order and *proc* is applied to the elements yielded by each iteration in turn. The order in which the arguments are supplied corresponds to te order in which the *collections* appear. `do-elts` is used when only side-effects of *proc* are of interest and its return value is unspecified. `map-elts` returns a collection (actually a vector) of the results of the applications of *proc*.

Example:

```
(map-elts + (list 1 2 3) (vector 1 2 3))
   ⇒ #(2 4 6)
```

map-keys *proc collection1 . . .* Procedure
do-keys *proc collection1 . . .* Procedure

These are analogous to `map-elts` and `do-elts`, but each iteration is over the *collections*' *keys* rather than their elements.

Example:

```
(map-keys + (list 1 2 3) (vector 1 2 3))
   ⇒ #(0 2 4)
```

for-each-key *collection proc* Procedure
for-each-elt *collection proc* Procedure

These are like `do-keys` and `do-elts` but only for a single collection; they are potentially more efficient.

reduce *proc seed collection1 . . .* Function

A generalization of the list-based `comlist:reduce-init` (see Section 6.2.1.3 [Lists as sequences], page 182) to collections which will shadow the list-based version if `(require 'collect)` follows `(require 'common-list-functions)` (see Section 6.2.1 [Common List Functions], page 177).

Examples:

```
(reduce + 0 (vector 1 2 3))
   ⇒ 6
(reduce union '() '((a b c) (b c d) (d a)))
   ⇒ (c b d a).
```

any? *pred collection1 . . .* Function

A generalization of the list-based `some` (see Section 6.2.1.3 [Lists as sequences], page 182) to collections.

Example:

```
(any? odd? (list 2 3 4 5))
   ⇒ #t
```

every? *pred collection1 . . .* Function

A generalization of the list-based `every` (see Section 6.2.1.3 [Lists as sequences], page 182) to collections.

Example:

```
(every? collection? '((1 2) #(1 2)))
   ⇒ #t
```

empty? *collection* Function

Returns `#t` iff there are no elements in *collection*.

`(empty?` *collection*`)` ≡ `(zero? (size` *collection*`))`

size *collection* Function

Returns the number of elements in *collection*.

Setter *list-ref* Function

See Section 2.8.3 [Setters], page 29 for a definition of *setter*. N.B. `(setter list-ref)` doesn't work properly for element 0 of a list.

Here is a sample collection: `simple-table` which is also a `table`.

```
(define-predicate TABLE?)
(define-operation (LOOKUP table key failure-object))
(define-operation (ASSOCIATE! table key value)) ;; returns key
(define-operation (REMOVE! table key))          ;; returns value

(define (MAKE-SIMPLE-TABLE)
  (let ( (table (list)) )
    (object
     ;; table behaviors
     ((TABLE? self) #t)
     ((SIZE self) (size table))
     ((PRINT self port) (format port "#<SIMPLE-TABLE>"))
     ((LOOKUP self key failure-object)
      (cond
       ((assq key table) => cdr)
       (else failure-object)
       ))
     ((ASSOCIATE! self key value)
      (cond
       ((assq key table)
        => (lambda (bucket) (set-cdr! bucket value) key))
       (else
        (set! table (cons (cons key value) table))
        key)
       ))
     ((REMOVE! self key);; returns old value
      (cond
       ((null? table) (slib:error "TABLE:REMOVE! Key not found: " key))
       ((eq? key (caar table))
        (let ( (value (cdar table)) )
          (set! table (cdr table))
          value)
        )
       (else
        (let loop ( (last table) (this (cdr table)) )
          (cond
           ((null? this)
            (slib:error "TABLE:REMOVE! Key not found: " key))
           ((eq? key (caar this))
            (let ( (value (cdar this)) )
              (set-cdr! last (cdr this))
              value)
            )
           (else
            (loop (cdr last) (cdr this)))
           ) ) )
       ))
     ;; collection behaviors
     ((COLLECTION? self) #t)
     ((GEN-KEYS self) (collect:list-gen-elts (map car table)))
     ((GEN-ELTS self) (collect:list-gen-elts (map cdr table)))
     ((FOR-EACH-KEY self proc)
      (for-each (lambda (bucket) (proc (car bucket))) table)
      )
     ((FOR-EACH-ELT self proc)
      (for-each (lambda (bucket) (proc (cdr bucket))) table)
      )
     ) ) )
```

6.1.9 Dynamic Data Type

```
(require 'dynamic)
```

make-dynamic *obj* Function

Create and returns a new *dynamic* whose global value is *obj*.

dynamic? *obj* Function

Returns true if and only if *obj* is a dynamic. No object satisfying `dynamic?` satisfies any of the other standard type predicates.

dynamic-ref *dyn* Function

Return the value of the given dynamic in the current dynamic environment.

dynamic-set! *dyn obj* Procedure

Change the value of the given dynamic to *obj* in the current dynamic environment. The returned value is unspecified.

call-with-dynamic-binding *dyn obj thunk* Function

Invoke and return the value of the given thunk in a new, nested dynamic environment in which the given dynamic has been bound to a new location whose initial contents are the value *obj*. This dynamic environment has precisely the same extent as the invocation of the thunk and is thus captured by continuations created within that invocation and re-established by those continuations when they are invoked.

The `dynamic-bind` macro is not implemented.

6.1.10 Hash Tables

```
(require 'hash-table)
```

predicate->hash *pred* Function

Returns a hash function (like `hashq`, `hashv`, or `hash`) corresponding to the equality predicate *pred*. *pred* should be `eq?`, `eqv?`, `equal?`, `=`, `char=?`, `char-ci=?`, `string=?`, or `string-ci=?`.

A hash table is a vector of association lists.

make-hash-table *k* Function

Returns a vector of *k* empty (association) lists.

Hash table functions provide utilities for an associative database. These functions take an equality predicate, *pred*, as an argument. *pred* should be `eq?`, `eqv?`, `equal?`, `=`, `char=?`, `char-ci=?`, `string=?`, or `string-ci=?`.

predicate->hash-asso *pred* Function

Returns a hash association function of 2 arguments, *key* and *hashtab*, corresponding to *pred*. The returned function returns a key-value pair whose key is *pred*-equal to its first argument or `#f` if no key in *hashtab* is *pred*-equal to the first argument.

hash-inquirer *pred* Function

Returns a procedure of 2 arguments, *hashtab* and *key*, which returns the value associated with *key* in *hashtab* or `#f` if *key* does not appear in *hashtab*.

hash-associator *pred* Function

Returns a procedure of 3 arguments, *hashtab*, *key*, and *value*, which modifies *hashtab* so that *key* and *value* associated. Any previous value associated with *key* will be lost.

hash-remover *pred* Function

Returns a procedure of 2 arguments, *hashtab* and *key*, which modifies *hashtab* so that the association whose key is *key* is removed.

hash-map *proc hash-table* Function

Returns a new hash table formed by mapping *proc* over the keys and values of *hash-table*. *proc* must be a function of 2 arguments which returns the new value part.

hash-for-each *proc hash-table* Function

Applies *proc* to each pair of keys and values of *hash-table*. *proc* must be a function of 2 arguments. The returned value is unspecified.

6.1.11 Hashing

```
(require 'hash)
```

These hashing functions are for use in quickly classifying objects. Hash tables use these functions.

hashq *obj k* Function
hashv *obj k* Function
hash *obj k* Function

Returns an exact non-negative integer less than *k*. For each non-negative integer less than *k* there are arguments *obj* for which the hashing functions applied to *obj* and *k* returns that integer.

For `hashq`, `(eq? obj1 obj2)` implies `(= (hashq obj1 k) (hashq obj2))`.

For `hashv`, `(eqv? obj1 obj2)` implies `(= (hashv obj1 k) (hashv obj2))`.

For `hash`, `(equal? obj1 obj2)` implies `(= (hash obj1 k) (hash obj2))`.

`hash`, `hashv`, and `hashq` return in time bounded by a constant. Notice that items having the same `hash` implies the items have the same `hashv` implies the items have the same `hashq`.

```
(require 'sierpinski)
```

make-sierpinski-indexer *max-coordinate* Function

Returns a procedure (eg hash-function) of 2 numeric arguments which preserves *nearness* in its mapping from NxN to N.

max-coordinate is the maximum coordinate (a positive integer) of a population of points. The returned procedures is a function that takes the x and y coordinates of a point, (non-negative integers) and returns an integer corresponding to the relative position of that point along a Sierpinski curve. (You can think of this as computing a (pseudo-) inverse of the Sierpinski spacefilling curve.)

Example use: Make an indexer (hash-function) for integer points lying in square of integer grid points [0,99]x[0,99]:

```
(define space-key (make-sierpinski-indexer 100))
```

Now let's compute the index of some points:

```
(space-key 24 78)               ⇒ 9206
(space-key 23 80)               ⇒ 9172
```

Note that locations (24, 78) and (23, 80) are near in index and therefore, because the Sierpinski spacefilling curve is continuous, we know they must also be near in the plane. Nearness in the plane does not, however, necessarily correspond to nearness in index, although it *tends* to be so.

Example applications:

- Sort points by Sierpinski index to get heuristic solution to *travelling salesman problem*. For details of performance, see L. Platzman and J. Bartholdi, "Spacefilling curves and the Euclidean travelling salesman problem", JACM 36(4):719–737 (October 1989) and references therein.
- Use Sierpinski index as key by which to store 2-dimensional data in a 1-dimensional data structure (such as a table). Then locations that are near each other in 2-d space will tend to be near each other in 1-d data structure; and locations that are near in 1-d data structure will be near in 2-d space. This can significantly speed retrieval from secondary storage because contiguous regions

in the plane will tend to correspond to contiguous regions in secondary storage. (This is a standard technique for managing CAD/CAM or geographic data.)

```
(require 'soundex)
```

soundex *name* Function

Computes the *soundex* hash of *name*. Returns a string of an initial letter and up to three digits between 0 and 6. Soundex supposedly has the property that names that sound similar in normal English pronunciation tend to map to the same key.

Soundex was a classic algorithm used for manual filing of personal records before the advent of computers. It performs adequately for English names but has trouble with other languages.

See Knuth, Vol. 3 *Sorting and searching*, pp 391–2

To manage unusual inputs, `soundex` omits all non-alphabetic characters. Consequently, in this implementation:

```
(soundex <string of blanks>)    ⇒ ""
(soundex "")                    ⇒ ""
```

Examples from Knuth:

```
(map soundex '("Euler" "Gauss" "Hilbert" "Knuth"
                         "Lloyd" "Lukasiewicz"))
        ⇒ ("E460" "G200" "H416" "K530" "L300" "L222")

(map soundex '("Ellery" "Ghosh" "Heilbronn" "Kant"
                          "Ladd" "Lissajous"))
        ⇒ ("E460" "G200" "H416" "K530" "L300" "L222")
```

Some cases in which the algorithm fails (Knuth):

```
(map soundex '("Rogers" "Rodgers"))     ⇒ ("R262" "R326")

(map soundex '("Sinclair" "St. Clair")) ⇒ ("S524" "S324")

(map soundex '("Tchebysheff" "Chebyshev")) ⇒ ("T212" "C121")
```

6.1.12 Macroless Object System

```
(require 'object)
```

This is the Macroless Object System written by Wade Humeniuk (whumeniu@datap.ca). Conceptual Tributes: Section 2.8 [Yasos], page 28, MacScheme's %object, CLOS, Lack of R4RS macros.

6.1.13 Concepts

OBJECT An object is an ordered association-list (by `eq?`) of methods (procedures). Methods can be added (`make-method!`), deleted (`unmake-method!`) and retrieved (`get-method`). Objects may inherit methods from other objects. The object binds to the environment it was created in, allowing closures to be used to hide private procedures and data.

GENERIC-METHOD
: A generic-method associates (in terms of `eq?`) object's method. This allows scheme function style to be used for objects. The calling scheme for using a generic method is `(generic-method object param1 param2 ...)`.

METHOD
: A method is a procedure that exists in the object. To use a method get-method must be called to look-up the method. Generic methods implement the get-method functionality. Methods may be added to an object associated with any scheme obj in terms of eq?

GENERIC-PREDICATE
: A generic method that returns a boolean value for any scheme obj.

PREDICATE
: A object's method asscociated with a generic-predicate. Returns `#t`.

6.1.14 Procedures

make-object *ancestor . . .* Function

Returns an object. Current object implementation is a tagged vector. *ancestor*s are optional and must be objects in terms of object?. *ancestor*s methods are included in the object. Multiple *ancestor*s might associate the same generic-method with a method. In this case the method of the *ancestor* first appearing in the list is the one returned by `get-method`.

object? *obj* Function

Returns boolean value whether *obj* was created by make-object.

make-generic-method *exception-procedure* Function

Returns a procedure which be associated with an object's methods. If *exception-procedure* is specified then it is used to process non-objects.

make-generic-predicate Function

Returns a boolean procedure for any scheme object.

make-method! *object generic-method method* Function

Associates *method* to the *generic-method* in the object. The *method* overrides any previous association with the *generic-method* within the object. Using `unmake-method!` will restore the object's previous association with the *generic-method*. *method* must be a procedure.

make-predicate! *object generic-preciate* Function

Makes a predicate method associated with the *generic-predicate*.

unmake-method! *object generic-method* Function

Removes an object's association with a *generic-method* .

get-method *object generic-method* Function

Returns the object's method associated (if any) with the *generic-method*. If no associated method exists an error is flagged.

6.1.15 Examples

```
(require 'object)

(define instantiate (make-generic-method))

(define (make-instance-object . ancestors)
  (define self (apply make-object
                      (map (lambda (obj) (instantiate obj)) ancestors)))
  (make-method! self instantiate (lambda (self) self))
  self)

(define who (make-generic-method))
(define imigrate! (make-generic-method))
(define emigrate! (make-generic-method))
(define describe (make-generic-method))
(define name (make-generic-method))
(define address (make-generic-method))
(define members (make-generic-method))

(define society
  (let ()
    (define self (make-instance-object))
    (define population '())
    (make-method! self imigrate!
                  (lambda (new-person)
                    (if (not (eq? new-person self))
                        (set! population (cons new-person population)))))
    (make-method! self emigrate!
                  (lambda (person)
                    (if (not (eq? person self))
                        (set! population
                              (comlist:remove-if (lambda (member)
                                                   (eq? member person))
                                                 population)))))
    (make-method! self describe
                  (lambda (self)
                    (map (lambda (person) (describe person)) population)))
    (make-method! self who
                  (lambda (self) (map (lambda (person) (name person))
                                      population)))
    (make-method! self members (lambda (self) population))
```

```
      self))

  (define (make-person %name %address)
    (define self (make-instance-object society))
    (make-method! self name (lambda (self) %name))
    (make-method! self address (lambda (self) %address))
    (make-method! self who (lambda (self) (name self)))
    (make-method! self instantiate
                  (lambda (self)
                    (make-person (string-append (name self) "-son-of")
                                 %address)))
    (make-method! self describe
                  (lambda (self) (list (name self) (address self))))
    (imigrate! self)
    self)
```

6.1.15.1 Inverter Documentation

Inheritance:

```
<inverter>::(<number> <description>)
```

Generic-methods

```
<inverter>::value      ⇒ <number>::value
<inverter>::set-value! ⇒ <number>::set-value!
<inverter>::describe   ⇒ <description>::describe
<inverter>::help
<inverter>::invert
<inverter>::inverter?
```

6.1.15.2 Number Documention

Inheritance

```
<number>::()
```

Slots

```
<number>::<x>
```

Generic Methods

```
<number>::value
<number>::set-value!
```

6.1.15.3 Inverter code

```
  (require 'object)

  (define value (make-generic-method (lambda (val) val)))
  (define set-value! (make-generic-method))
  (define invert (make-generic-method
                  (lambda (val)
                    (if (number? val)
                        (/ 1 val)
                        (error "Method not supported:" val)))))
  (define noop (make-generic-method))
  (define inverter? (make-generic-predicate))
  (define describe (make-generic-method))
  (define help (make-generic-method))

  (define (make-number x)
    (define self (make-object))
    (make-method! self value (lambda (this) x))
    (make-method! self set-value!
                  (lambda (this new-value) (set! x new-value)))
    self)

  (define (make-description str)
    (define self (make-object))
    (make-method! self describe (lambda (this) str))
    (make-method! self help (lambda (this) "Help not available"))
    self)

  (define (make-inverter)
    (let* ((self (make-object
                  (make-number 1)
                  (make-description "A number which can be inverted")))
           (<value> (get-method self value)))
      (make-method! self invert (lambda (self) (/ 1 (<value> self))))
      (make-predicate! self inverter?)
      (unmake-method! self help)
      (make-method! self help
                    (lambda (self)
                      (display "Inverter Methods:") (newline)
                      (display "  (value inverter) ==> n") (newline)))
      self))

  ;;;; Try it out

  (define invert! (make-generic-method))

  (define x (make-inverter))

  (make-method! x invert! (lambda (x) (set-value! x (/ 1 (value x)))))

  (value x)                       ⇒ 1
  (set-value! x 33)               ⇒ undefined
  (invert! x)                     ⇒ undefined
  (value x)                       ⇒ 1/33

  (unmake-method! x invert!)      ⇒ undefined
```

```
(invert! x)                    error  ERROR: Method not supported: x
```

6.1.16 Priority Queues

```
(require 'priority-queue)
```

make-heap *pred<?* Function
: Returns a binary heap suitable which can be used for priority queue operations.

heap-length *heap* Function
: Returns the number of elements in *heap*.

heap-insert! *heap item* Procedure
: Inserts *item* into *heap*. *item* can be inserted multiple times. The value returned is unspecified.

heap-extract-max! *heap* Function
: Returns the item which is larger than all others according to the *pred<?* argument to `make-heap`. If there are no items in *heap*, an error is signaled.

The algorithm for priority queues was taken from *Introduction to Algorithms* by T. Cormen, C. Leiserson, R. Rivest. 1989 MIT Press.

6.1.17 Queues

```
(require 'queue)
```

A *queue* is a list where elements can be added to both the front and rear, and removed from the front (i.e., they are what are often called *dequeues*). A queue may also be used like a stack.

make-queue Function
: Returns a new, empty queue.

queue? *obj* Function
: Returns `#t` if *obj* is a queue.

queue-empty? *q* Function
: Returns `#t` if the queue *q* is empty.

queue-push! *q datum* Procedure
: Adds *datum* to the front of queue *q*.

enquque! *q datum* Procedure

Adds *datum* to the rear of queue *q*.

All of the following functions raise an error if the queue *q* is empty.

queue-front *q* Function

Returns the datum at the front of the queue *q*.

queue-rear *q* Function

Returns the datum at the rear of the queue *q*.

queue-pop! *q* Prcoedure

dequeue! *q* Procedure

Both of these procedures remove and return the datum at the front of the queue. `queue-pop!` is used to suggest that the queue is being used like a stack.

6.1.18 Records

```
(require 'record)
```

The Record package provides a facility for user to define their own record data types.

make-record-type *type-name field-names* Function

Returns a *record-type descriptor*, a value representing a new data type disjoint from all others. The *type-name* argument must be a string, but is only used for debugging purposes (such as the printed representation of a record of the new type). The *field-names* argument is a list of symbols naming the *fields* of a record of the new type. It is an error if the list contains any duplicates. It is unspecified how record-type descriptors are represented.

record-constructor *rtd* [*field-names*] Function

Returns a procedure for constructing new members of the type represented by *rtd*. The returned procedure accepts exactly as many arguments as there are symbols in the given list, *field-names*; these are used, in order, as the initial values of those fields in a new record, which is returned by the constructor procedure. The values of any fields not named in that list are unspecified. The *field-names* argument defaults to the list of field names in the call to `make-record-type` that created the type represented by *rtd*; if the *field-names* argument is provided, it is an error if it contains any duplicates or any symbols not in the default list.

record-predicate *rtd* Function

Returns a procedure for testing membership in the type represented by *rtd*. The returned procedure accepts exactly one argument and returns a true value if the argument is a member of the indicated record type; it returns a false value otherwise.

record-accessor *rtd field-name* Function

Returns a procedure for reading the value of a particular field of a member of the type represented by *rtd*. The returned procedure accepts exactly one argument which must be a record of the appropriate type; it returns the current value of the field named by the symbol *field-name* in that record. The symbol *field-name* must be a member of the list of field-names in the call to `make-record-type` that created the type represented by *rtd*.

record-modifier *rtd field-name* Function

Returns a procedure for writing the value of a particular field of a member of the type represented by *rtd*. The returned procedure accepts exactly two arguments: first, a record of the appropriate type, and second, an arbitrary Scheme value; it modifies the field named by the symbol *field-name* in that record to contain the given value. The returned value of the modifier procedure is unspecified. The symbol *field-name* must be a member of the list of field-names in the call to `make-record-type` that created the type represented by *rtd*.

In May of 1996, as a product of discussion on the `rrrs-authors` mailing list, I rewrote '`record.scm`' to portably implement type disjointness for record data types.

As long as an implementation's procedures are opaque and the `record` code is loaded before other programs, this will give disjoint record types which are unforgeable and incorruptible by R4RS procedures.

As a consequence, the procedures `record?`, `record-type-descriptor`, `record-type-name`.and `record-type-field-names` are no longer supported.

6.2 Sorting and Searching

6.2.1 Common List Functions

```
(require 'common-list-functions)
```

The procedures below follow the Common LISP equivalents apart from optional arguments in some cases.

6.2.1.1 List construction

make-list *k* Function
make-list *k init* Function

`make-list` creates and returns a list of *k* elements. If *init* is included, all elements in the list are initialized to *init*.

Example:

```
(make-list 3)
   ⇒ (#<unspecified> #<unspecified> #<unspecified>)
(make-list 5 'foo)
   ⇒ (foo foo foo foo foo)
```

list* *obj1 obj2 . . .* Function

Works like `list` except that the cdr of the last pair is the last argument unless there is only one argument, when the result is just that argument. Sometimes called `cons*`. E.g.:

```
(list* 1)
   ⇒ 1
(list* 1 2 3)
   ⇒ (1 2 . 3)
(list* 1 2 '(3 4))
   ⇒ (1 2 3 4)
(list* args '())
   ≡ (list args)
```

copy-list *lst* Function

`copy-list` makes a copy of *lst* using new pairs and returns it. Only the top level of the list is copied, i.e., pairs forming elements of the copied list remain `eq?` to the corresponding elements of the original; the copy is, however, not `eq?` to the original, but is `equal?` to it.

Example:

```
(copy-list '(foo foo foo))
   ⇒ (foo foo foo)
(define q '(foo bar baz bang))
(define p q)
(eq? p q)
   ⇒ #t
(define r (copy-list q))
(eq? q r)
   ⇒ #f
(equal? q r)
   ⇒ #t
(define bar '(bar))
(eq? bar (car (copy-list (list bar 'foo))))
⇒ #t
```

6.2.1.2 Lists as sets

`eqv?` is used to test for membership by procedures which treat lists as sets.

adjoin *e l* Function

`adjoin` returns the adjoint of the element *e* and the list *l*. That is, if *e* is in *l*, `adjoin` returns *l*, otherwise, it returns `(cons` *e l*`)`.

Example:

```
(adjoin 'baz '(bar baz bang))
   ⇒ (bar baz bang)
(adjoin 'foo '(bar baz bang))
   ⇒ (foo bar baz bang)
```

union *l1 l2* Function

`union` returns the combination of *l1* and *l2*. Duplicates between *l1* and *l2* are culled. Duplicates within *l1* or within *l2* may or may not be removed.

Example:

```
(union '(1 2 3 4) '(5 6 7 8))
   ⇒ (8 7 6 5 1 2 3 4)
(union '(1 2 3 4) '(3 4 5 6))
   ⇒ (6 5 1 2 3 4)
```

intersection *l1 l2* Function

`intersection` returns all elements that are in both *l1* and *l2*.

Example:

```
(intersection '(1 2 3 4) '(3 4 5 6))
   ⇒ (3 4)
(intersection '(1 2 3 4) '(5 6 7 8))
   ⇒ ()
```

set-difference *l1 l2* Function

`set-difference` returns all elements that are in *l1* but not in *l2*.

Example:

```
(set-difference '(1 2 3 4) '(3 4 5 6))
   ⇒ (1 2)
(set-difference '(1 2 3 4) '(1 2 3 4 5 6))
   ⇒ ()
```

member-if *pred lst* Function

`member-if` returns *lst* if `(`*pred element*`)` is `#t` for any *element* in *lst*. Returns `#f` if *pred* does not apply to any *element* in *lst*.

Example:

```
(member-if vector? '(1 2 3 4))
   ⇒ #f
(member-if number? '(1 2 3 4))
   ⇒ (1 2 3 4)
```

some *pred lst1 lst2 ...* Function

pred is a boolean function of as many arguments as there are list arguments to `some` i.e., *lst* plus any optional arguments. *pred* is applied to successive elements of the list arguments in order. `some` returns `#t` as soon as one of these applications returns `#t`, and is `#f` if none returns `#t`. All the lists should have the same length.

Example:

```
(some odd? '(1 2 3 4))
   ⇒ #t

(some odd? '(2 4 6 8))
   ⇒ #f

(some > '(2 3) '(1 4))
   ⇒ #f
```

every *pred lst1 lst2 . . .* Function

`every` is analogous to `some` except it returns `#t` if every application of *pred* is `#t` and `#f` otherwise.

Example:

```
(every even? '(1 2 3 4))
   ⇒ #f

(every even? '(2 4 6 8))
   ⇒ #t

(every > '(2 3) '(1 4))
   ⇒ #f
```

notany *pred lst1 . . .* Function

`notany` is analogous to `some` but returns `#t` if no application of *pred* returns `#t` or `#f` as soon as any one does.

notevery *pred lst1 . . .* Function

`notevery` is analogous to `some` but returns `#t` as soon as an application of *pred* returns `#f`, and `#f` otherwise.

Example:

```
(notevery even? '(1 2 3 4))
   ⇒ #t

(notevery even? '(2 4 6 8))
   ⇒ #f
```

list-of?? *predicate* Function

Returns a predicate which returns true if its argument is a list every element of which satisfies *predicate*.

list-of?? *predicate low-bound high-bound* Function

low-bound and *high-bound* are non-negative integers. `list-of??` returns a predicate which returns true if its argument is a list of length between *low-bound* and *high-bound* (inclusive); every element of which satisfies *predicate*.

list-of?? *predicate bound* Function

bound is an integer. If *bound* is negative, `list-of??` returns a predicate which returns true if its argument is a list of length greater than `(- `*bound*`)`; every element of which satisfies *predicate*. Otherwise, `list-of??` returns a predicate which returns true if its argument is a list of length less than or equal to *bound*; every element of which satisfies *predicate*.

find-if *pred lst* Function

`find-if` searches for the first *element* in *lst* such that `(`*pred element*`)` returns `#t`. If it finds any such *element* in *lst*, *element* is returned. Otherwise, `#f` is returned.

Example:

```
(find-if number? '(foo 1 bar 2))
   ⇒ 1

(find-if number? '(foo bar baz bang))
   ⇒ #f

(find-if symbol? '(1 2 foo bar))
   ⇒ foo
```

remove *elt lst* Function

`remove` removes all occurrences of *elt* from *lst* using `eqv?` to test for equality and returns everything that's left. N.B.: other implementations (Chez, Scheme->C and T, at least) use `equal?` as the equality test.

Example:

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

(remove 'foo '(bar baz bang))
   ⇒ (bar baz bang)
```

remove-if *pred lst* Function

`remove-if` removes all *element*s from *lst* where `(`*pred element*`)` is `#t` and returns everything that's left.

Example:

```
(remove-if number? '(1 2 3 4))
   ⇒ ()

(remove-if even? '(1 2 3 4 5 6 7 8))
   ⇒ (1 3 5 7)
```

remove-if-not *pred lst* Function

`remove-if-not` removes all *element*s from *lst* for which `(`*pred element*`)` is `#f` and returns everything that's left.

Example:

```
(remove-if-not number? '(foo bar baz))
   ⇒ ()
(remove-if-not odd? '(1 2 3 4 5 6 7 8))
   ⇒ (1 3 5 7)
```

has-duplicates? *lst* Function

returns `#t` if 2 members of *lst* are `equal?`, `#f` otherwise.

Example:

```
(has-duplicates? '(1 2 3 4))
   ⇒ #f

(has-duplicates? '(2 4 3 4))
   ⇒ #t
```

The procedure `remove-duplicates` uses `member` (rather than `memv`).

remove-duplicates *lst* Function

returns a copy of *lst* with its duplicate members removed. Elements are considered duplicate if they are `equal?`.

Example:

```
(remove-duplicates '(1 2 3 4))
   ⇒ (1 2 3 4)

(remove-duplicates '(2 4 3 4))
   ⇒ (2 4 3)
```

6.2.1.3 Lists as sequences

position *obj lst* Function

`position` returns the 0-based position of *obj* in *lst*, or `#f` if *obj* does not occur in *lst*.

Example:

```
(position 'foo '(foo bar baz bang))
   ⇒ 0
(position 'baz '(foo bar baz bang))
   ⇒ 2
(position 'oops '(foo bar baz bang))
   ⇒ #f
```

reduce *p lst* Function

`reduce` combines all the elements of a sequence using a binary operation (the combination is left-associative). For example, using `+`, one can add up all the elements. `reduce` allows you to apply a function which accepts only two arguments to more than 2 objects. Functional programmers usually refer to this as *foldl*. `collect:reduce` (see Section 6.1.8 [Collections], page 163) provides a version of `collect` generalized to collections.

Example:

```
(reduce + '(1 2 3 4))
   ⇒ 10
(define (bad-sum . l) (reduce + l))
(bad-sum 1 2 3 4)
   ≡ (reduce + (1 2 3 4))
   ≡ (+ (+ (+ 1 2) 3) 4)
⇒ 10
(bad-sum)
   ≡ (reduce + ())
   ⇒ ()
(reduce string-append '("hello" "cruel" "world"))
   ≡ (string-append (string-append "hello" "cruel") "world")
   ⇒ "hellocruelworld"
(reduce anything '())
   ⇒ ()
(reduce anything '(x))
   ⇒ x
```

What follows is a rather non-standard implementation of `reverse` in terms of `reduce` and a combinator elsewhere called *C*.

```
;;; Contributed by Jussi Piitulainen (jpiitula @ ling.helsinki.fi)

(define commute
  (lambda (f)
    (lambda (x y)
      (f y x))))

(define reverse
  (lambda (args)
    (reduce-init (commute cons) '() args)))
```

reduce-init *p init lst* Function

`reduce-init` is the same as reduce, except that it implicitly inserts *init* at the start of the list. `reduce-init` is preferred if you want to handle the null list, the one-element, and lists with two or more elements consistently. It is common to use the operator's idempotent as the initializer. Functional programmers usually call this *foldl*.

Example:

```
(define (sum . l) (reduce-init + 0 l))
(sum 1 2 3 4)
   ≡ (reduce-init + 0 (1 2 3 4))
   ≡ (+ (+ (+ (+ 0 1) 2) 3) 4)
   ⇒ 10
(sum)
   ≡ (reduce-init + 0 '())
   ⇒ 0
```

```
(reduce-init string-append "@" '("hello" "cruel" "world"))
≡
(string-append (string-append (string-append "@" "hello")
                              "cruel")
               "world")
⇒ "@hellocruelworld"
```

Given a differentiation of 2 arguments, `diff`, the following will differentiate by any number of variables.

```
(define (diff* exp . vars)
  (reduce-init diff exp vars))
```

Example:

```
;;; Real-world example:  Insertion sort using reduce-init.

(define (insert l item)
  (if (null? l)
      (list item)
      (if (< (car l) item)
          (cons (car l) (insert (cdr l) item))
          (cons item l))))
(define (insertion-sort l) (reduce-init insert '() l))

(insertion-sort '(3 1 4 1 5)
   ≡ (reduce-init insert () (3 1 4 1 5))
   ≡ (insert (insert (insert (insert (insert () 3) 1) 4) 1) 5)
   ≡ (insert (insert (insert (insert (3)) 1) 4) 1) 5)
   ≡ (insert (insert (insert (1 3) 4) 1) 5)
   ≡ (insert (insert (1 3 4) 1) 5)
   ≡ (insert (1 1 3 4) 5)
   ⇒ (1 1 3 4 5)
```

last *lst n* Function

`last` returns the last *n* elements of *lst*. *n* must be a non-negative integer.

Example:

```
(last '(foo bar baz bang) 2)
   ⇒ (baz bang)
(last '(1 2 3) 0)
   ⇒ 0
```

butlast *lst n* Function

`butlast` returns all but the last *n* elements of *lst*.

Example:

```
(butlast '(a b c d) 3)
   ⇒ (a)
(butlast '(a b c d) 4)
```

```
            ⇒ ()
```

`last` and `butlast` split a list into two parts when given identical arugments.

```
(last '(a b c d e) 2)
   ⇒ (d e)
(butlast '(a b c d e) 2)
   ⇒ (a b c)
```

nthcdr *n lst* Function

`nthcdr` takes *n* `cdr`s of *lst* and returns the result. Thus `(nthcdr 3` *lst*`)` ≡ `(cdddr` *lst*`)`

Example:

```
(nthcdr 2 '(a b c d))
   ⇒ (c d)
(nthcdr 0 '(a b c d))
   ⇒ (a b c d)
```

butnthcdr *n lst* Function

`butnthcdr` returns all but the nthcdr *n* elements of *lst*.

Example:

```
(butnthcdr 3 '(a b c d))
   ⇒ (a b c)
(butnthcdr 4 '(a b c d))
   ⇒ (a b c d)
```

`nthcdr` and `butnthcdr` split a list into two parts when given identical arugments.

```
(nthcdr 2 '(a b c d e))
   ⇒ (c d e)
(butnthcdr 2 '(a b c d e))
   ⇒ (a b)
```

6.2.1.4 Destructive list operations

These procedures may mutate the list they operate on, but any such mutation is undefined.

nconc *args* Procedure

`nconc` destructively concatenates its arguments. (Compare this with `append`, which copies arguments rather than destroying them.) Sometimes called `append!` (see Section 6.4.4 [Rev2 Procedures], page 201).

Example: You want to find the subsets of a set. Here's the obvious way:

```
(define (subsets set)
  (if (null? set)
      '(())
```

```
                (append (mapcar (lambda (sub) (cons (car set) sub))
                                (subsets (cdr set)))
                        (subsets (cdr set)))))
```

But that does way more consing than you need. Instead, you could replace the `append` with `nconc`, since you don't have any need for all the intermediate results.

Example:

```
(define x '(a b c))
(define y '(d e f))
(nconc x y)
   ⇒ (a b c d e f)
x
   ⇒ (a b c d e f)
```

`nconc` is the same as `append!` in '`sc2.scm`'.

nreverse *lst* Procedure

`nreverse` reverses the order of elements in *lst* by mutating `cdr`s of the list. Sometimes called `reverse!`.

Example:

```
(define foo '(a b c))
(nreverse foo)
   ⇒ (c b a)
foo
   ⇒ (a)
```

Some people have been confused about how to use `nreverse`, thinking that it doesn't return a value. It needs to be pointed out that

```
(set! lst (nreverse lst))
```

is the proper usage, not

```
(nreverse lst)
```

The example should suffice to show why this is the case.

delete *elt lst* Procedure
delete-if *pred lst* Procedure
delete-if-not *pred lst* Procedure

Destructive versions of `remove` `remove-if`, and `remove-if-not`.

Example:

```
(define lst (list 'foo 'bar 'baz 'bang))
(delete 'foo lst)
   ⇒ (bar baz bang)
lst
   ⇒ (foo bar baz bang)

(define lst (list 1 2 3 4 5 6 7 8 9))
(delete-if odd? lst)
   ⇒ (2 4 6 8)
```

```
lst
   ⇒ (1 2 4 6 8)
```

Some people have been confused about how to use `delete`, `delete-if`, and `delete-if`, thinking that they don't return a value. It needs to be pointed out that

```
(set! lst (delete el lst))
```

is the proper usage, not

```
(delete el lst)
```

The examples should suffice to show why this is the case.

6.2.1.5 Non-List functions

and? *arg1 . . .* Function

`and?` checks to see if all its arguments are true. If they are, `and?` returns `#t`, otherwise, `#f`. (In contrast to `and`, this is a function, so all arguments are always evaluated and in an unspecified order.)

Example:

```
(and? 1 2 3)
   ⇒ #t
(and #f 1 2)
   ⇒ #f
```

or? *arg1 . . .* Function

`or?` checks to see if any of its arguments are true. If any is true, `or?` returns `#t`, and `#f` otherwise. (To `or` as `and?` is to `and`.)

Example:

```
(or? 1 2 #f)
   ⇒ #t
(or? #f #f #f)
   ⇒ #f
```

atom? *object* Function

Returns `#t` if *object* is not a pair and `#f` if it is pair. (Called `atom` in Common LISP.)

```
(atom? 1)
   ⇒ #t
(atom? '(1 2))
   ⇒ #f
(atom? #(1 2))   ; dubious!
   ⇒ #t
```

6.2.2 Tree operations

```
(require 'tree)
```

These are operations that treat lists a representations of trees.

subst *new old tree* Function
subst *new old tree equ?* Function
substq *new old tree* Function
substv *new old tree* Function

`subst` makes a copy of *tree*, substituting *new* for every subtree or leaf of *tree* which is `equal?` to *old* and returns a modified tree. The original *tree* is unchanged, but may share parts with the result.

`substq` and `substv` are similar, but test against *old* using `eq?` and `eqv?` respectively. If `subst` is called with a fourth argument, *equ?* is the equality predicate.

Examples:

```
(substq 'tempest 'hurricane '(shakespeare wrote (the hurricane)))
   ⇒ (shakespeare wrote (the tempest))
(substq 'foo '() '(shakespeare wrote (twelfth night)))
   ⇒ (shakespeare wrote (twelfth night . foo) . foo)
(subst '(a . cons) '(old . pair)
       '((old . spice) ((old . shoes) old . pair) (old . pair)))
   ⇒ ((old . spice) ((old . shoes) a . cons) (a . cons))
```

copy-tree *tree* Function

Makes a copy of the nested list structure *tree* using new pairs and returns it. All levels are copied, so that none of the pairs in the tree are `eq?` to the original ones – only the leaves are.

Example:

```
(define bar '(bar))
(copy-tree (list bar 'foo))
   ⇒ ((bar) foo)
(eq? bar (car (copy-tree (list bar 'foo))))
   ⇒ #f
```

6.2.3 Chapter Ordering

```
(require 'chapter-order)
```

The '`chap:`' functions deal with strings which are ordered like chapter numbers (or letters) in a book. Each section of the string consists of consecutive numeric or consecutive aphabetic characters of like case.

chap:string<? *string1 string2* Function

Returns #t if the first non-matching run of alphabetic upper-case or the first non-matching run of alphabetic lower-case or the first non-matching run of numeric characters of *string1* is `string<?` than the corresponding non-matching run of characters of *string2*.

```
(chap:string<? "a.9" "a.10")                    ⇒ #t
(chap:string<? "4c" "4aa")                      ⇒ #t
(chap:string<? "Revised^{3.99}" "Revised^{4}")  ⇒ #t
```

chap:string>? *string1 string2* Function
chap:string<=? *string1 string2* Function
chap:string>=? *string1 string2* Function

Implement the corresponding chapter-order predicates.

chap:next-string *string* Function

Returns the next string in the *chapter order*. If *string* has no alphabetic or numeric characters, `(string-append` *string* `"0")` is returnd. The argument to chap:next-string will always be `chap:string<?` than the result.

```
(chap:next-string "a.9")              ⇒ "a.10"
(chap:next-string "4c")               ⇒ "4d"
(chap:next-string "4z")               ⇒ "4aa"
(chap:next-string "Revised^{4}")      ⇒ "Revised^{5}"
```

6.2.4 Sorting

`(require 'sort)`

Many Scheme systems provide some kind of sorting functions. They do not, however, always provide the *same* sorting functions, and those that I have had the opportunity to test provided inefficient ones (a common blunder is to use quicksort which does not perform well).

Because `sort` and `sort!` are not in the standard, there is very little agreement about what these functions look like. For example, Dybvig says that Chez Scheme provides

```
(merge predicate list1 list2)
(merge! predicate list1 list2)
(sort predicate list)
(sort! predicate list)
```

while MIT Scheme 7.1, following Common LISP, offers unstable

```
(sort list predicate)
```

TI PC Scheme offers

```
(sort! list/vector predicate?)
```

and Elk offers

```
(sort list/vector predicate?)
(sort! list/vector predicate?)
```

Here is a comprehensive catalogue of the variations I have found.

1. Both `sort` and `sort!` may be provided.
2. `sort` may be provided without `sort!`.
3. `sort!` may be provided without `sort`.

4. Neither may be provided.
5. The sequence argument may be either a list or a vector.
6. The sequence argument may only be a list.
7. The sequence argument may only be a vector.
8. The comparison function may be expected to behave like `<`.
9. The comparison function may be expected to behave like `<=`.
10. The interface may be `(sort predicate? sequence)`.
11. The interface may be `(sort sequence predicate?)`.
12. The interface may be `(sort sequence &optional (predicate? <))`.
13. The sort may be stable.
14. The sort may be unstable.

All of this variation really does not help anybody. A nice simple merge sort is both stable and fast (quite a lot faster than *quick* sort).

I am providing this source code with no restrictions at all on its use (but please retain D.H.D.Warren's credit for the original idea). You may have to rename some of these functions in order to use them in a system which already provides incompatible or inferior sorts. For each of the functions, only the top-level define needs to be edited to do that.

I could have given these functions names which would not clash with any Scheme that I know of, but I would like to encourage implementors to converge on a single interface, and this may serve as a hint. The argument order for all functions has been chosen to be as close to Common LISP as made sense, in order to avoid NIH-itis.

Each of the five functions has a required *last* parameter which is a comparison function. A comparison function `f` is a function of 2 arguments which acts like `<`. For example,

```
(not (f x x))
(and (f x y) (f y z)) ≡ (f x z)
```

The standard functions `<`, `>`, `char<?`, `char>?`, `char-ci<?`, `char-ci>?`, `string<?`, `string>?`, `string-ci<?`, and `string-ci>?` are suitable for use as comparison functions. Think of `(less? x y)` as saying when `x` must *not* precede `y`.

sorted? *sequence less?* Function

Returns `#t` when the sequence argument is in non-decreasing order according to *less?* (that is, there is no adjacent pair `... x y ...` for which `(less? y x)`).

Returns `#f` when the sequence contains at least one out-of-order pair. It is an error if the sequence is neither a list nor a vector.

merge *list1 list2 less?* Function

This merges two lists, producing a completely new list as result. I gave serious consideration to producing a Common-LISP-compatible version. However, Common LISP's `sort` is our `sort!` (well, in fact Common LISP's `stable-sort` is our `sort!`, merge sort is *fast* as well as stable!) so adapting CL code to Scheme takes a bit of work anyway. I did, however, appeal to CL to determine the *order* of the arguments.

merge! *list1 list2 less?* Procedure

Merges two lists, re-using the pairs of *list1* and *list2* to build the result. If the code is compiled, and *less?* constructs no new pairs, no pairs at all will be allocated. The first pair of the result will be either the first pair of *list1* or the first pair of *list2*, but you can't predict which.

The code of `merge` and `merge!` could have been quite a bit simpler, but they have been coded to reduce the amount of work done per iteration. (For example, we only have one `null?` test per iteration.)

sort *sequence less?* Function

Accepts either a list or a vector, and returns a new sequence which is sorted. The new sequence is the same type as the input. Always `(sorted? (sort sequence less?) less?)`. The original sequence is not altered in any way. The new sequence shares its *elements* with the old one; no elements are copied.

sort! *sequence less?* Procedure

Returns its sorted result in the original boxes. If the original sequence is a list, no new storage is allocated at all. If the original sequence is a vector, the sorted elements are put back in the same vector.

Some people have been confused about how to use `sort!`, thinking that it doesn't return a value. It needs to be pointed out that

```
(set! slist (sort! slist <))
```

is the proper usage, not

```
(sort! slist <)
```

Note that these functions do *not* accept a CL-style '`:key`' argument. A simple device for obtaining the same expressiveness is to define

```
(define (keyed less? key)
  (lambda (x y) (less? (key x) (key y))))
```

and then, when you would have written

```
(sort a-sequence #'my-less :key #'my-key)
```

in Common LISP, just write

```
(sort! a-sequence (keyed my-less? my-key))
```

in Scheme.

6.2.5 Topological Sort

`(require 'topological-sort)` or `(require 'tsort)`

The algorithm is inspired by Cormen, Leiserson and Rivest (1990) *Introduction to Algorithms*, chapter 23.

tsort *dag pred* Function
topological-sort *dag pred* Function

where

dag is a list of sublists. The car of each sublist is a vertex. The cdr is the adjacency list of that vertex, i.e. a list of all vertices to which there exists an edge from the car vertex.

pred is one of `eq?`, `eqv?`, `equal?`, `=`, `char=?`, `char-ci=?`, `string=?`, or `string-ci=?`.

Sort the directed acyclic graph *dag* so that for every edge from vertex *u* to *v*, *u* will come before *v* in the resulting list of vertices.

Time complexity: O (|V| + |E|)

Example (from Cormen):

> Prof. Bumstead topologically sorts his clothing when getting dressed. The first argument to 'tsort' describes which garments he needs to put on before others. (For example, Prof Bumstead needs to put on his shirt before he puts on his tie or his belt.) 'tsort' gives the correct order of dressing:

```
(require 'tsort)
(tsort '((shirt tie belt)
         (tie jacket)
         (belt jacket)
         (watch)
         (pants shoes belt)
         (undershorts pants shoes)
         (socks shoes))
       eq?)
⇒
(socks undershorts pants shoes watch shirt belt tie jacket)
```

6.2.6 String Search

```
(require 'string-search)
```

string-index *string char* Procedure
string-index-ci *string char* Procedure

Returns the index of the first occurence of *char* within *string*, or `#f` if the *string* does not contain a character *char*.

string-reverse-index *string char* Procedure
string-reverse-index-ci *string char* Procedure

Returns the index of the last occurence of *char* within *string*, or `#f` if the *string* does not contain a character *char*.

substring? *pattern string* procedure
substring-ci? *pattern string* procedure

Searches *string* to see if some substring of *string* is equal to *pattern*. `substring?` returns the index of the first character of the first substring of *string* that is equal to *pattern*; or `#f` if *string* does not contain *pattern*.

```
(substring? "rat" "pirate") ⇒  2
(substring? "rat" "outrage") ⇒  #f
(substring? "" any-string) ⇒  0
```

find-string-from-port? *str in-port max-no-chars* Procedure

Looks for a string *str* within the first *max-no-chars* chars of the input port *in-port*.

find-string-from-port? *str in-port* Procedure

When called with two arguments, the search span is limited by the end of the input stream.

find-string-from-port? *str in-port char* Procedure

Searches up to the first occurrence of character *char* in *str*.

find-string-from-port? *str in-port proc* Procedure

Searches up to the first occurrence of the procedure *proc* returning non-false when called with a character (from *in-port*) argument.

When the *str* is found, `find-string-from-port?` returns the number of characters it has read from the port, and the port is set to read the first char after that (that is, after the *str*) The function returns `#f` when the *str* isn't found.

`find-string-from-port?` reads the port *strictly* sequentially, and does not perform any buffering. So `find-string-from-port?` can be used even if the *in-port* is open to a pipe or other communication channel.

string-subst *txt old1 new1 ...* Function

Returns a copy of string *txt* with all occurrences of string *old1* in *txt* replaced with *new1*, *old2* replaced with *new2* ....

6.2.7 Sequence Comparison

```
(require 'diff)
```

This package implements the algorithm:

If the items being sequenced are text lines, then the computed edit-list is equivalent to the output of the *diff* utility program. If the items being sequenced are words, then it is like the lesser known *spiff* program.

The values returned by `diff:edit-length` can be used to gauge the degree of match between two sequences.

I believe that this algorithm is currently the fastest for these tasks, but genome sequencing applications fuel extensive research in this area.

diff:longest-common-subsequence *array1 array2* `=`*?* Function
diff:longest-common-subsequence *array1 array2* Function

array1 and *array2* are one-dimensional arrays. The procedure *=?* is used to compare sequence tokens for equality. *=?* defaults to `eqv?`. `diff:longest-common-subsequence` returns a one-dimensional array of length `(quotient (- (+ len1 len2) (fp:edit-length` *array1 array2*`)) 2)` holding the longest sequence common to both *array*s.

diff:edits *array1 array2* `=`*?* Function
diff:edits *array1 array2* Function

array1 and *array2* are one-dimensional arrays. The procedure *=?* is used to compare sequence tokens for equality. *=?* defaults to `eqv?`. `diff:edits` returns a list of length `(fp:edit-length` *array1 array2*`)` composed of a shortest sequence of edits transformaing *array1* to *array2*.

Each edit is a list of an integer and a symbol:

(*j* insert)	Inserts `(array-ref` *array1 j*`)` into the sequence.
(*k* delete)	Deletes `(array-ref` *array2 k*`)` from the sequence.

diff:edit-length *array1 array2* `=`*?* Function
diff:edit-length *array1 array2* Function

array1 and *array2* are one-dimensional arrays. The procedure *=?* is used to compare sequence tokens for equality. *=?* defaults to `eqv?`. `diff:edit-length` returns the length of the shortest sequence of edits transformaing *array1* to *array2*.

```
(diff:longest-common-subsequence '#(f g h i e j c k l m)
                                 '#(f g e h i j k p q r l m))
                                 ⇒ #(f g h i j k l m)

(diff:edit-length '#(f g h i e j c k l m)
                  '#(f g e h i j k p q r l m))
⇒ 6

(pretty-print (diff:edits '#(f g h i e j c k l m)
                          '#(f g e h i j k p q r l m)))
⊣
((3 insert)                     ; e
 (4 delete)                     ; c
 (6 delete)                     ; h
 (7 insert)                     ; p
 (8 insert)                     ; q
 (9 insert))                    ; r
```

6.3 Procedures

Anything that doesn't fall neatly into any of the other categories winds up here.

6.3.1 Type Coercion

```
(require 'coerce)
```

type-of *obj* Function

Returns a symbol name for the type of *obj*.

coerce *obj result-type* Function

Converts and returns *obj* of type `char`, `number`, `string`, `symbol`, `list`, or `vector` to *result-type* (which must be one of these symbols).

6.3.2 String-Case

```
(require 'string-case)
```

string-upcase *str* Procedure
string-downcase *str* Procedure
string-capitalize *str* Procedure

The obvious string conversion routines. These are non-destructive.

string-upcase! *str* Function
string-downcase! *str* Function
string-captialize! *str* Function

The destructive versions of the functions above.

string-ci->symbol *str* Function

Converts string *str* to a symbol having the same case as if the symbol had been `read`.

symbol-append *obj1 . . .* Function

Converts *obj1 . . .* to strings, appends them, and converts to a symbol which is returned. Strings and numbers are converted to read's symbol case; the case of symbol characters is not changed. #f is converted to the empty string (symbol).

StudlyCapsExpand *str delimiter* Function
StudlyCapsExpand *str* Function

delimiter must be a string or character. If absent, *delimiter* defaults to '-'. `StudlyCapsExpand` returns a copy of *str* where *delimiter* is inserted between each lower-case character immediately followed by an upper-case character; and between two upper-case characters immediately followed by a lower-case character.

```
(StudlyCapsExpand "aX" " ")   ⇒ "a X"
(StudlyCapsExpand "aX" "..")  ⇒ "a..X"
(StudlyCapsExpand "AX")       ⇒ "AX"
(StudlyCapsExpand "Ax")       ⇒ "Ax"
```

```
(StudlyCapsExpand "AXLE")     ⇒ "AXLE"
(StudlyCapsExpand "aAXACz")   ⇒ "a-AXA-Cz"
(StudlyCapsExpand "AaXACz")   ⇒ "Aa-XA-Cz"
(StudlyCapsExpand "AAaXACz")  ⇒ "A-Aa-XA-Cz"
(StudlyCapsExpand "AAaXAC")   ⇒ "A-Aa-XAC"
```

6.3.3 String Ports

```
(require 'string-port)
```

call-with-output-string *proc* Procedure

proc must be a procedure of one argument. This procedure calls *proc* with one argument: a (newly created) output port. When the function returns, the string composed of the characters written into the port is returned.

call-with-input-string *string proc* Procedure

proc must be a procedure of one argument. This procedure calls *proc* with one argument: an (newly created) input port from which *string*'s contents may be read. When *proc* returns, the port is closed and the value yielded by the procedure *proc* is returned.

6.3.4 Line I/O

```
(require 'line-i/o)
```

read-line Function
read-line *port* Function

Returns a string of the characters up to, but not including a newline or end of file, updating *port* to point to the character following the newline. If no characters are available, an end of file object is returned. The *port* argument may be omitted, in which case it defaults to the value returned by `current-input-port`.

read-line! *string* Function
read-line! *string port* Function

Fills *string* with characters up to, but not including a newline or end of file, updating the *port* to point to the last character read or following the newline if it was read. If no characters are available, an end of file object is returned. If a newline or end of file was found, the number of characters read is returned. Otherwise, `#f` is returned. The *port* argument may be omitted, in which case it defaults to the value returned by `current-input-port`.

write-line *string* Function
write-line *string port* Function

Writes *string* followed by a newline to the given *port* and returns an unspecified value. The *Port* argument may be omitted, in which case it defaults to the value returned by `current-input-port`.

display-file *path* Function
display-file *path port* Function

Displays the contents of the file named by *path* to *port*. The *port* argument may be ommited, in which case it defaults to the value returned by `current-output-port`.

6.3.5 Multi-Processing

```
(require 'process)
```

This module implements asynchronous (non-polled) time-sliced multi-processing in the SCM Scheme implementation using procedures `alarm` and `alarm-interrupt`. Until this is ported to another implementation, consider it an example of writing schedulers in Scheme.

add-process! *proc* Procedure

Adds proc, which must be a procedure (or continuation) capable of accepting accepting one argument, to the `process:queue`. The value returned is unspecified. The argument to *proc* should be ignored. If *proc* returns, the process is killed.

process:schedule! Procedure

Saves the current process on `process:queue` and runs the next process from `process:queue`. The value returned is unspecified.

kill-process! Procedure

Kills the current process and runs the next process from `process:queue`. If there are no more processes on `process:queue`, `(slib:exit)` is called (see Section 1.5.5 [System], page 9).

6.3.6 Metric Units

```
(require 'metric-units)
```

`http://swissnet.ai.mit.edu/~jaffer/MIXF.html`

Metric Interchange Format is a character string encoding for numerical values and units which:

- is unambiguous in all locales;
- uses only [TOG] "Portable Character Set" characters matching "Basic Latin" characters in Plane 0 of the Universal Character Set [UCS];
- is transparent to [UTF-7] and [UTF-8] UCS transformation formats;

- is human readable and writable;
- is machine readable and writable;
- incorporates SI prefixes and units;
- incorporates [ISO 6093] numbers; and
- incorporates [IEC 60027-2] binary prefixes.

In the expression for the value of a quantity, the unit symbol is placed after the numerical value. A dot (PERIOD, '`.`') is placed between the numerical value and the unit symbol.

Within a compound unit, each of the base and derived symbols can optionally have an attached SI prefix.

Unit symbols formed from other unit symbols by multiplication are indicated by means of a dot (PERIOD, '`.`') placed between them.

Unit symbols formed from other unit symbols by division are indicated by means of a SOLIDUS ('`/`') or negative exponents. The SOLIDUS must not be repeated in the same compound unit unless contained within a parenthesized subexpression.

The grouping formed by a prefix symbol attached to a unit symbol constitutes a new inseparable symbol (forming a multiple or submultiple of the unit concerned) which can be raised to a positive or negative power and which can be combined with other unit symbols to form compound unit symbols.

The grouping formed by surrounding compound unit symbols with parentheses ('`(`' and '`)`') constitutes a new inseparable symbol which can be raised to a positive or negative power and which can be combined with other unit symbols to form compound unit symbols.

Compound prefix symbols, that is, prefix symbols formed by the juxtaposition of two or more prefix symbols, are not permitted.

Prefix symbols are not used with the time-related unit symbols min (minute), h (hour), d (day). No prefix symbol may be used with dB (decibel). Only submultiple prefix symbols may be used with the unit symbols L (liter), Np (neper), o (degree), oC (degree Celsius), rad (radian), and sr (steradian). Submultiple prefix symbols may not be used with the unit symbols t (metric ton), r (revolution), or Bd (baud).

A unit exponent follows the unit, separated by a CIRCUMFLEX ('`^`'). Exponents may be positive or negative. Fractional exponents must be parenthesized.

6.3.6.1 SI Prefixes

Factor	Name	Symbol	Factor	Name	Symbol
1e24	yotta	Y	1e-1	deci	d
1e21	zetta	Z	1e-2	centi	c
1e18	exa	E	1e-3	milli	m
1e15	peta	P	1e-6	micro	u
1e12	tera	T	1e-9	nano	n
1e9	giga	G	1e-12	pico	p
1e6	mega	M	1e-15	femto	f
1e3	kilo	k	1e-18	atto	a

1e2	hecto	h	|	1e-21	zepto	z
1e1	deka	da	|	1e-24	yocto	y

6.3.6.2 Binary Prefixes

These binary prefixes are valid only with the units B (byte) and bit. However, decimal prefixes can also be used with bit; and decimal multiple (not submultiple) prefixes can also be used with B (byte).

Factor	(power-of-2)	Name	Symbol
1.152921504606846976e18	(2^60)	exbi	Ei
1.125899906842624e15	(2^50)	pebi	Pi
1.099511627776e12	(2^40)	tebi	Ti
1.073741824e9	(2^30)	gibi	Gi
1.048576e6	(2^20)	mebi	Mi
1.024e3	(2^10)	kibi	Ki

6.3.6.3 Unit Symbols

Type of Quantity	Name	Symbol	Equivalent
time	second	s	
time	minute	min	= 60.s
time	hour	h	= 60.min
time	day	d	= 24.h
frequency	hertz	Hz	s^-1
signaling rate	baud	Bd	s^-1
length	meter	m	
volume	liter	L	dm^3
plane angle	radian	rad	
solid angle	steradian	sr	rad^2
plane angle	revolution	* r	= 6.283185307179586.rad
plane angle	degree	* o	= 2.777777777777778e-3.r
information capacity	bit	bit	
information capacity	byte, octet	B	= 8.bit
mass	gram	g	
mass	ton	t	Mg
mass	unified atomic mass unit	u	= 1.66053873e-27.kg
amount of substance	mole	mol	
catalytic activity	katal	kat	mol/s
thermodynamic temperature	kelvin	K	
centigrade temperature	degree Celsius	oC	
luminous intensity	candela	cd	
luminous flux	lumen	lm	cd.sr
illuminance	lux	lx	lm/m^2
force	newton	N	m.kg.s^-2
pressure, stress	pascal	Pa	N/m^2
energy, work, heat	joule	J	N.m
energy	electronvolt	eV	= 1.602176462e-19.J

power, radiant flux	watt		W	J/s
logarithm of power ratio	neper		Np	
logarithm of power ratio	decibel	*	dB	= 0.1151293.Np
electric current	ampere		A	
electric charge	coulomb		C	s.A
electric potential, EMF	volt		V	W/A
capacitance	farad		F	C/V
electric resistance	ohm		Ohm	V/A
electric conductance	siemens		S	A/V
magnetic flux	weber		Wb	V.s
magnetic flux density	tesla		T	Wb/m^2
inductance	henry		H	Wb/A
radionuclide activity	becquerel		Bq	s^-1
absorbed dose energy	gray		Gy	m^2.s^-2
dose equivalent	sievert		Sv	m^2.s^-2

* The formulas are:

- r/rad = 8 * atan(1)
- o/r = 1 / 360
- db/Np = ln(10) / 20

si:conversion-factor *to-unit from-unit* Function

If the strings *from-unit* and *to-unit* express valid unit expressions for quantities of the same unit-dimensions, then the value returned by `si:conversion-factor` will be such that multiplying a numerical value expressed in *from-unit*s by the returned conversion factor yields the numerical value expressed in *to-unit*s.

Otherwise, `si:conversion-factor` returns:

-3	if neither *from-unit* nor *to-unit* is a syntactically valid unit.
-2	if *from-unit* is not a syntactically valid unit.
-1	if *to-unit* is not a syntactically valid unit.
0	if linear conversion (by a factor) is not possible.

```
(si:conversion-factor "km/s" "m/s" ) ⇒ 0.001
(si:conversion-factor "N"    "m/s" ) ⇒ 0
(si:conversion-factor "moC"  "oC"  ) ⇒ 1000
(si:conversion-factor "mK"   "oC"  ) ⇒ 0
(si:conversion-factor "rad"  "o"   ) ⇒ 0.0174533
(si:conversion-factor "K"    "o"   ) ⇒ 0
(si:conversion-factor "K"    "K"   ) ⇒ 1
(si:conversion-factor "oK"   "oK"  ) ⇒ -3
(si:conversion-factor ""     "s/s" ) ⇒ 1
(si:conversion-factor "km/h" "mph" ) ⇒ -2
```

6.4 Standards Support

6.4.1 RnRS

The `r2rs`, `r3rs`, `r4rs`, and `r5rs` features attempt to provide procedures and macros to bring a Scheme implementation to the desired version of Scheme.

r2rs Feature

Requires features implementing procedures and optional procedures specified by *Revised^2 Report on the Algorithmic Language Scheme*; namely `rev3-procedures` and `rev2-procedures`.

r3rs Feature

Requires features implementing procedures and optional procedures specified by *Revised^3 Report on the Algorithmic Language Scheme*; namely `rev3-procedures`.

Note: SLIB already mandates the `r3rs` procedures which can be portably implemented in `r4rs` implementations.

r4rs Feature

Requires features implementing procedures and optional procedures specified by *Revised^4 Report on the Algorithmic Language Scheme*; namely `rev4-optional-procedures`.

r5rs Feature

Requires features implementing procedures and optional procedures specified by *Revised^5 Report on the Algorithmic Language Scheme*; namely `values`, `macro`, and `eval`.

6.4.2 With-File

```
(require 'with-file)
```

with-input-from-file *file thunk* Function
with-output-to-file *file thunk* Function

Description found in R4RS.

6.4.3 Transcripts

```
(require 'transcript)
```

transcript-on *filename* Function
transcript-off *filename* Function

Redefines `read-char`, `read`, `write-char`, `write`, `display`, and `newline`.

6.4.4 Rev2 Procedures

`(require 'rev2-procedures)`

The procedures below were specified in the *Revised^2 Report on Scheme*. **N.B.**: The symbols `1+` and `-1+` are not *R4RS* syntax. Scheme->C, for instance, chokes on this module.

substring-move-left! *string1 start1 end1 string2 start2* Procedure
substring-move-right! *string1 start1 end1 string2 start2* Procedure

string1 and *string2* must be a strings, and *start1*, *start2* and *end1* must be exact integers satisfying

0 <= *start1* <= *end1* <= (string-length *string1*)
0 <= *start2* <= *end1* - *start1* + *start2* <= (string-length *string2*)

`substring-move-left!` and `substring-move-right!` store characters of *string1* beginning with index *start1* (inclusive) and ending with index *end1* (exclusive) into *string2* beginning with index *start2* (inclusive).

`substring-move-left!` stores characters in time order of increasing indices. `substring-move-right!` stores characters in time order of increasing indeces.

substring-fill! *string start end char* Procedure

Fills the elements *start*–*end* of *string* with the character *char*.

string-null? *str* Function

≡ `(= 0 (string-length` *str*`))`

append! *pair1 . . .* Procedure

Destructively appends its arguments. Equivalent to `nconc`.

1+ *n* Function

Adds 1 to *n*.

-1+ *n* Function

Subtracts 1 from *n*.

<? Function
<=? Function
=? Function
>? Function
>=? Function

These are equivalent to the procedures of the same name but without the trailing '`?`'.

6.4.5 Rev4 Optional Procedures

```
(require 'rev4-optional-procedures)
```

For the specification of these optional procedures, See section "Standard procedures" in *Revised(4) Scheme*.

list-tail *l p* Function

string->list *s* Function

list->string *l* Function

string-copy Function

string-fill! *s obj* Procedure

list->vector *l* Function

vector->list *s* Function

vector-fill! *s obj* Procedure

6.4.6 Multi-argument / and -

```
(require 'multiarg/and-)
```

For the specification of these optional forms, See section "Numerical operations" in *Revised(4) Scheme*. The `two-arg:`* forms are only defined if the implementation does not support the many-argument forms.

two-arg:/ *n1 n2* Function

The original two-argument version of `/`.

/ *dividend divisor1 . . .* Function

two-arg:- *n1 n2* Function

The original two-argument version of `-`.

- *minuend subtrahend1 . . .* Function

6.4.7 Multi-argument Apply

```
(require 'multiarg-apply)
```

For the specification of this optional form, See section "Control features" in *Revised(4) Scheme*.

two-arg:apply *proc l* Function

The implementation's native `apply`. Only defined for implementations which don't support the many-argument version.

apply *proc arg1 . . .* Function

6.4.8 Rationalize

```
(require 'rationalize)
```

The procedure *rationalize* is interesting because most programming languages do not provide anything analogous to it. Thanks to Alan Bawden for contributing this algorithm.

rationalize *x y* Function

Computes the correct result for exact arguments (provided the implementation supports exact rational numbers of unlimited precision); and produces a reasonable answer for inexact arguments when inexact arithmetic is implemented using floating-point.

`Rationalize` has limited use in implementations lacking exact (non-integer) rational numbers. The following procedures return a list of the numerator and denominator.

find-ratio *x y* Function

`find-ratio` returns the list of the *simplest* numerator and denominator whose quotient differs from *x* by no more than *y*.

```
(find-ratio 3/97 .0001)          ⇒ (3 97)
(find-ratio 3/97 .001)           ⇒ (1 32)
```

find-ratio-between *x y* Function

`find-ratio-between` returns the list of the *simplest* numerator and denominator between *x* and *y*.

```
(find-ratio-between 2/7 3/5)     ⇒ (1 2)
(find-ratio-between -3/5 -2/7)   ⇒ (-1 2)
```

6.4.9 Promises

```
(require 'promise)
```

make-promise *proc* Function

Change occurrences of `(delay` *expression*`)` to `(make-promise (lambda ()` *expression*`))` and `(define force promise:force)` to implement promises if your implementation doesn't support them (see section "Control features" in *Revised(4) Scheme*).

6.4.10 Dynamic-Wind

```
(require 'dynamic-wind)
```

This facility is a generalization of Common LISP `unwind-protect`, designed to take into account the fact that continuations produced by `call-with-current-continuation` may be reentered.

dynamic-wind *thunk1 thunk2 thunk3* Procedure

The arguments *thunk1*, *thunk2*, and *thunk3* must all be procedures of no arguments (thunks).

`dynamic-wind` calls *thunk1*, *thunk2*, and then *thunk3*. The value returned by *thunk2* is returned as the result of `dynamic-wind`. *thunk3* is also called just before control leaves the dynamic context of *thunk2* by calling a continuation created outside that context. Furthermore, *thunk1* is called before reentering the dynamic context of *thunk2* by calling a continuation created inside that context. (Control is inside the context of *thunk2* if *thunk2* is on the current return stack).

Warning: There is no provision for dealing with errors or interrupts. If an error or interrupt occurs while using `dynamic-wind`, the dynamic environment will be that in effect at the time of the error or interrupt.

6.4.11 Eval

```
(require 'eval)
```

eval *expression environment-specifier* Function

Evaluates *expression* in the specified environment and returns its value. *Expression* must be a valid Scheme expression represented as data, and *environment-specifier* must be a value returned by one of the three procedures described below. Implementations may extend `eval` to allow non-expression programs (definitions) as the first argument and to allow other values as environments, with the restriction that `eval` is not allowed to create new bindings in the environments associated with `null-environment` or `scheme-report-environment`.

```
(eval '(* 7 3) (scheme-report-environment 5))
                                                     ⇒  21

(let ((f (eval '(lambda (f x) (f x x))
               (null-environment))))
  (f + 10))
                                                     ⇒  20
```

scheme-report-environment *version* Function
null-environment *version* Function
null-environment Function

Version must be an exact non-negative integer *n* corresponding to a version of one of the Revised^*n* Reports on Scheme. `Scheme-report-environment` returns a specifier for an environment that contains the set of bindings specified in the corresponding report that the implementation supports. `Null-environment` returns a specifier for an environment that contains only the (syntactic) bindings for all the syntactic keywords defined in the given version of the report.

Not all versions may be available in all implementations at all times. However, an implementation that conforms to version *n* of the Revised^*n* Reports on Scheme must accept version *n*. An error is signalled if the specified version is not available.

The effect of assigning (through the use of `eval`) a variable bound in a `scheme-report-environment` (for example `car`) is unspecified. Thus the environments specified by `scheme-report-environment` may be immutable.

interaction-environment Function

This optional procedure returns a specifier for the environment that contains implementation-defined bindings, typically a superset of those listed in the report. The intent is that this procedure will return the environment in which the implementation would evaluate expressions dynamically typed by the user.

Here are some more `eval` examples:

```
(require 'eval)
⇒ #<unspecified>
(define car 'volvo)
⇒ #<unspecified>
car
⇒ volvo
(eval 'car (interaction-environment))
⇒ volvo
(eval 'car (scheme-report-environment 5))
⇒ #<primitive-procedure car>
(eval '(eval 'car (interaction-environment))
      (scheme-report-environment 5))
⇒ volvo
(eval '(eval '(set! car 'buick) (interaction-environment))
      (scheme-report-environment 5))
⇒ #<unspecified>
```

```
car
⇒ buick
(eval 'car (scheme-report-environment 5))
⇒ #<primitive-procedure car>
(eval '(eval 'car (interaction-environment))
      (scheme-report-environment 5))
⇒ buick
```

6.4.12 Values

```
(require 'values)
```

values *obj ...* Function

`values` takes any number of arguments, and passes (returns) them to its continuation.

call-with-values *thunk proc* Function

thunk must be a procedure of no arguments, and *proc* must be a procedure. `call-with-values` calls *thunk* with a continuation that, when passed some values, calls *proc* with those values as arguments.

Except for continuations created by the `call-with-values` procedure, all continuations take exactly one value, as now; the effect of passing no value or more than one value to continuations that were not created by the `call-with-values` procedure is unspecified.

6.4.13 SRFI

```
(require 'srfi)
```

Implements *Scheme Request For Implementation* (SRFI) as described at http://srfi.schemers.org/

The Copyright terms of each SRFI states:

"However, this document itself may not be modified in any way, ..."

Therefore, the specification of SRFI constructs must not be quoted without including the complete SRFI document containing discussion and a sample implementation program.

cond-expand *<clause1> <clause2> ...* Macro

Syntax: Each <clause> should be of the form

`(<feature> <expression1> ...)`

where <feature> is a boolean expression composed of symbols and 'and', 'or', and 'not' of boolean expressions. The last <clause> may be an "else clause," which has the form

`(else <expression1> <expression2> ...).`

The first clause whose feature expression is satisfied is expanded. If no feature expression is satisfied and there is no else clause, an error is signaled.

SLIB `cond-expand` is an extension of SRFI-0, http://srfi.schemers.org/srfi-0/srfi-0.html .

6.4.13.1 SRFI-1

```
(require 'srfi-1)
```

Implements the *SRFI-1 list-processing library* as described at http://srfi.schemers.org/srfi-1/srfi-

Constructors

xcons *d a* Function
: `(define (xcons d a) (cons a d))`.

list-tabulate *len proc* Function
: Returns a list of length *len*. Element *i* is `(`*proc i*`)` for 0 `<=` *i* `<` *len*.

cons* *obj1 obj2* Function

iota *count start step* Function
iota *count start* Function
iota *count* Function
: Returns a list of *count* numbers: (*start*, *start*+*step*, . . . , *start*+(*count*-1)**step*).

circular-list *obj1 obj2 . . .* Function
: Returns a circular list of *obj1*, *obj2*, ...

Predicates

proper-list? *obj* Function

circular-list? *x* Function

dotted-list? *obj* Function

null-list? *obj* Function

not-pair? *obj* Function

list= `=`*pred list . . .* Function

Selectors

first *pair* Function
fifth *obj* Function
sixth *obj* Function
seventh *obj* Function
eighth *obj* Function
ninth *obj* Function
tenth *obj* Function

car+cdr *pair* Function

take *lst k* Function
drop *lst k* Function

take-right *lst k* Function

split-at *lst k* Function

last *lst* Function

(car (last-pair lst))

Miscellaneous

length+ *obj* Function

concatenate *lists* Function
concatenate! *lists* Function

reverse! *lst* Function

append-reverse *rev-head tail* Function
append-reverse! *rev-head tail* Function

zip *list1 list2 . . .* Function

unzip1 *lst* Function
unzip2 *lst* Function
unzip3 *lst* Function
unzip4 *lst* Function
unzip5 *lst* Function

count *pred list1 list2 . . .* Function

Fold and Unfold

Filtering and Partitioning

Searching

find *pred list* Function

find-tail *pred list* Function

member *obj list pred* Function
member *obj list* Function

`member` returns the first sublist of *list* whose car is *obj*, where the sublists of *list* are the non-empty lists returned by `(list-tail` *list k*`)` for *k* less than the length of *list*. If *obj* does not occur in *list*, then `#f` (not the empty list) is returned. The procedure *pred* is used for testing equality. If *pred* is not provided, '`equal?`' is used.

Deleting

Association lists

assoc *obj alist pred* Function
assoc *obj alist* Function

alist (for "association list") must be a list of pairs. These procedures find the first pair in *alist* whose car field is *obj*, and returns that pair. If no pair in *alist* has *obj* as its car, then `#f` (not the empty list) is returned. The procedure *pred* is used for testing equality. If *pred* is not provided, '`equal?`' is used.

Set operations

6.5 Session Support

6.5.1 Repl

```
(require 'repl)
```

Here is a read-eval-print-loop which, given an eval, evaluates forms.

repl:top-level *repl:eval* Procedure

`read`s, `repl:eval`s and `write`s expressions from `(current-input-port)` to `(current-output-port)` until an end-of-file is encountered. `load`, `slib:eval`, `slib:error`, and `repl:quit` dynamically bound during `repl:top-level`.

repl:quit Procedure

Exits from the invocation of `repl:top-level`.

The `repl:` procedures establish, as much as is possible to do portably, a top level environment supporting macros. `repl:top-level` uses `dynamic-wind` to catch error conditions and interrupts. If your implementation supports this you are all set.

Otherwise, if there is some way your implementation can catch error conditions and interrupts, then have them call `slib:error`. It will display its arguments and reenter `repl:top-level`. `slib:error` dynamically bound by `repl:top-level`.

To have your top level loop always use macros, add any interrupt catching lines and the following lines to your Scheme init file:

```
(require 'macro)
(require 'repl)
(repl:top-level macro:eval)
```

6.5.2 Quick Print

```
(require 'qp)
```

When displaying error messages and warnings, it is paramount that the output generated for circular lists and large data structures be limited. This section supplies a procedure to do this. It could be much improved.

> Notice that the neccessity for truncating output eliminates Common-Lisp's Section 3.2 [Format], page 39 from consideration; even when variables `*print-level*` and `*print-level*` are set, huge strings and bit-vectors are *not* limited.

qp *arg1 . . .* Procedure
qpn *arg1 . . .* Procedure
qpr *arg1 . . .* Procedure

`qp` writes its arguments, separated by spaces, to `(current-output-port)`. `qp` compresses printing by substituting '`...`' for substructure it does not have sufficient room to print. `qpn` is like `qp` but outputs a newline before returning. `qpr` is like `qpn` except that it returns its last argument.

qp-width Variable

`*qp-width*` is the largest number of characters that `qp` should use.

6.5.3 Debug

```
(require 'debug)
```

Requiring `debug` automatically requires `trace` and `break`.

An application with its own datatypes may want to substitute its own printer for `qp`. This example shows how to do this:

```
(define qpn (lambda args) ...)
(provide 'qp)
(require 'debug)
```

trace-all *file* . . . Procedure

Traces (see Section 6.5.5 [Trace], page 211) all procedures `define`d at top-level in '`file`' . . . . **track-all** *file* . . .

Tracks (see Section 6.5.5 [Trace], page 211) all procedures `define`d at top-level in '`file`' . . . . **stack-all** *file* . . .

Stacks (see Section 6.5.5 [Trace], page 211) all procedures `define`d at top-level in '`file`' . . . .

break-all *file* . . . Procedure

Breakpoints (see Section 6.5.4 [Breakpoints], page 210) all procedures `define`d at top-level in '`file`' . . . .

6.5.4 Breakpoints

```
(require 'break)
```

init-debug Function

If your Scheme implementation does not support `break` or `abort`, a message will appear when you `(require 'break)` or `(require 'debug)` telling you to type `(init-debug)`. This is in order to establish a top-level continuation. Typing `(init-debug)` at top level sets up a continuation for `break`.

breakpoint *arg1* . . . Function

Returns from the top level continuation and pushes the continuation from which it was called on a continuation stack.

continue Function

Pops the topmost continuation off of the continuation stack and returns an unspecified value to it.

continue *arg1* . . . Function

Pops the topmost continuation off of the continuation stack and returns *arg1* . . . to it.

break *proc1* . . . Macro

Redefines the top-level named procedures given as arguments so that `breakpoint` is called before calling *proc1* . . . . **break**

With no arguments, makes sure that all the currently broken identifiers are broken (even if those identifiers have been redefined) and returns a list of the broken identifiers.

unbreak *proc1 ...* Macro

Turns breakpoints off for its arguments. **unbreak**

With no arguments, unbreaks all currently broken identifiers and returns a list of these formerly broken identifiers.

These are *procedures* for breaking. If defmacros are not natively supported by your implementation, these might be more convenient to use.

breakf *proc* Function

breakf *proc name* Function

To break, type

```
(set! symbol (breakf symbol))
```

or

```
(set! symbol (breakf symbol 'symbol))
```

or

```
(define symbol (breakf function))
```

or

```
(define symbol (breakf function 'symbol))
```

unbreakf *proc* Function

To unbreak, type

```
(set! symbol (unbreakf symbol))
```

6.5.5 Tracing

```
(require 'trace)
```

This feature provides three ways to monitor procedure invocations:

stack Pushes the procedure-name when the procedure is called; pops when it returns.

track Pushes the procedure-name and arguments when the procedure is called; pops when it returns.

trace Pushes the procedure-name and prints '`CALL` *procedure-name arg1* `...`' when the procdure is called; pops and prints '`RETN` *procedure-name value*' when the procedure returns.

debug:max-count Variable

If a traced procedure calls itself or untraced procedures which call it, stack, track, and trace will limit the number of stack pushes to *debug:max-count*.

print-call-stack Function

print-call-stack *port* Function

Prints the call-stack to *port* or the current-error-port.

trace *proc1 ...* Macro
Traces the top-level named procedures given as arguments. **trace**
With no arguments, makes sure that all the currently traced identifiers are traced (even if those identifiers have been redefined) and returns a list of the traced identifiers.

track *proc1 ...* Macro
Traces the top-level named procedures given as arguments. **track**
With no arguments, makes sure that all the currently tracked identifiers are tracked (even if those identifiers have been redefined) and returns a list of the tracked identifiers.

stack *proc1 ...* Macro
Traces the top-level named procedures given as arguments. **stack**
With no arguments, makes sure that all the currently stacked identifiers are stacked (even if those identifiers have been redefined) and returns a list of the stacked identifiers.

untrace *proc1 ...* Macro
Turns tracing, tracking, and off for its arguments. **untrace**
With no arguments, untraces all currently traced identifiers and returns a list of these formerly traced identifiers.

untrack *proc1 ...* Macro
Turns tracing, tracking, and off for its arguments. **untrack**
With no arguments, untracks all currently tracked identifiers and returns a list of these formerly tracked identifiers.

unstack *proc1 ...* Macro
Turns tracing, stacking, and off for its arguments. **unstack**
With no arguments, unstacks all currently stacked identifiers and returns a list of these formerly stacked identifiers.

These are *procedures* for tracing. If defmacros are not natively supported by your implementation, these might be more convenient to use.

tracef *proc* Function
tracef *proc name* Function
To trace, type

```
(set! symbol (tracef symbol))
```

or

```
(set! symbol (tracef symbol 'symbol))
```

or

```
(define symbol (tracef function))
```

or

```
(define symbol (tracef function 'symbol))
```

untracef *proc* Function

Removes tracing, tracking, or stacking for *proc*. To untrace, type

```
(set! symbol (untracef symbol))
```

6.5.6 System Interface

If `(provided? 'getenv)`:

getenv *name* Function

Looks up *name*, a string, in the program environment. If *name* is found a string of its value is returned. Otherwise, `#f` is returned.

If `(provided? 'system)`:

system *command-string* Function

Executes the *command-string* on the computer and returns the integer status code.

If `system` is provided by the Scheme implementation, the *net-clients* package provides interfaces to common network client programs like FTP, mail, and Netscape.

```
(require 'net-clients)
```

call-with-tmpnam *proc* Function
call-with-tmpnam *proc k* Function

Calls *proc* with *k* arguments, strings returned by successive calls to `tmpnam`. If *proc* returns, then any files named by the arguments to *proc* are deleted automatically and the value(s) yielded by the *proc* is(are) returned. *k* may be ommited, in which case it defaults to `1`.

user-email-address Function

`user-email-address` returns a string of the form '`username@hostname`'. If this e-mail address cannot be obtained, #f is returned.

current-directory Function

`current-directory` returns a string containing the absolute file name representing the current working directory. If this string cannot be obtained, #f is returned.

If `current-directory` cannot be supported by the platform, the value of `current-directory` is #f.

make-directory *name* Function

Creates a sub-directory *name* of the current-directory. If successful, `make-directory` returns #t; otherwise #f.

null-directory? *file-name* Function

Returns #t if changing directory to *file-name* makes the current working directory the same as it is before changing directory; otherwise returns #f.

absolute-path? *file-name* Function

Returns #t if *file-name* is a fully specified pathname (does not depend on the current working directory); otherwise returns #f.

glob-pattern? *str* Function

Returns #t if the string *str* contains characters used for specifying glob patterns, namely '`*`', '`?`', or '`[`'.

parse-ftp-address *uri* Function

Returns a list of the decoded FTP *uri*; or #f if indecipherable. FTP *Uniform Resource Locator*, *ange-ftp*, and *getit* formats are handled. The returned list has four elements which are strings or #f:

0. username
1. password
2. remote-site
3. remote-directory

path->uri *path* Function

Returns a URI-string for *path* on the local host.

browse-url-netscape *url* Function

If a '`netscape`' browser is running, `browse-url-netscape` causes the browser to display the page specified by string *url* and returns #t.

If the browser is not running, `browse-url-netscape` runs '`netscape`' with the argument *url*. If the browser starts as a background job, `browse-url-netscape` returns #t immediately; if the browser starts as a foreground job, then `browse-url-netscape` returns #t when the browser exits; otherwise it returns #f.

6.6 Extra-SLIB Packages

Several Scheme packages have been written using SLIB. There are several reasons why a package might not be included in the SLIB distribution:

- Because it requires special hardware or software which is not universal.
- Because it is large and of limited interest to most Scheme users.
- Because it has copying terms different enough from the other SLIB packages that its inclusion would cause confusion.
- Because it is an application program, rather than a library module.
- Because I have been too busy to integrate it.

Once an optional package is installed (and an entry added to `*catalog*`, the `require` mechanism allows it to be called up and used as easily as any other SLIB package. Some optional packages (for which `*catalog*` already has entries) available from SLIB sites are:

SLIB-PSD is a portable debugger for Scheme (requires emacs editor).

http://swissnet.ai.mit.edu/ftpdir/scm/slib-psd1-3.tar.gz

swissnet.ai.mit.edu:/pub/scm/slib-psd1-3.tar.gz

ftp.maths.tcd.ie:pub/bosullvn/jacal/slib-psd1-3.tar.gz

ftp.cs.indiana.edu:/pub/scheme-repository/utl/slib-psd1-3.tar.gz

With PSD, you can run a Scheme program in an Emacs buffer, set breakpoints, single step evaluation and access and modify the program's variables. It works by instrumenting the original source code, so it should run with any R4RS compliant Scheme. It has been tested with SCM, Elk 1.5, and the sci interpreter in the Scheme->C system, but should work with other Schemes with a minimal amount of porting, if at all. Includes documentation and user's manual. Written by Pertti Kellom\"aki, pk @ cs.tut.fi. The Lisp Pointers article describing PSD (Lisp Pointers VI(1):15-23, January-March 1993) is available as http://www.cs.tut.fi/staff/pk/scheme/psd/article/article.html

SCHELOG

is an embedding of Prolog in Scheme. http://www.cs.rice.edu/CS/PLT/packages/schelog/

JFILTER is a Scheme program which converts text among the JIS, EUC, and Shift-JIS Japanese character sets. http://www.sci.toyama-u.ac.jp/~iwao/Scheme/Jfilter/index.html

7 About SLIB

7.1 Installation

There are four parts to installation:

- Unpack the SLIB distribution.
- Configure the Scheme implementation(s) to locate the SLIB directory.
- Arrange for Scheme implementation to load its SLIB initialization file.
- Build the SLIB catalog for the Scheme implementation.

7.1.1 Unpacking the SLIB Distribution

If the SLIB distribution is a Linux RPM, it will create the SLIB directory '`/usr/share/slib`'.

If the SLIB distribution is a ZIP file, unzip the distribution to create the SLIB directory. Locate this '`slib`' directory either in your home directory (if only you will use this SLIB installation); or put it in a location where libraries reside on your system. On unix systems this might be '`/usr/share/slib`', '`/usr/local/lib/slib`', or '`/usr/lib/slib`'. If you know where SLIB should go on other platforms, please inform agj `@` alum.mit.edu.

7.1.2 Configure Scheme Implementation to Locate SLIB

If the Scheme implementation supports `getenv`, then the value of the shell environment variable *SCHEME_LIBRARY_PATH* will be used for `(library-vicinity)` if it is defined. Currently, Chez, Elk, MITScheme, scheme->c, VSCM, and SCM support `getenv`. Scheme48 supports `getenv` but does not use it for determining `library-vicinity`. (That is done from the Makefile.)

The `(library-vicinity)` can also be specified from the SLIB initialization file or by implementation-specific means.

7.1.3 Loading SLIB Initialization File

Check the manifest in '`README`' to find a configuration file for your Scheme implementation. Initialization files for most IEEE P1178 compliant Scheme Implementations are included with this distribution.

You should check the definitions of `software-type`, `scheme-implementation-version`, `implementation-vicinity`, and `library-vicinity` in the initialization file. There are comments in the file for how to configure it.

Once this is done, modify the startup file for your Scheme implementation to `load` this initialization file.

7.1.4 Build New SLIB Catalog for Implementation

When SLIB is first used from an implementation, a file named '`slibcat`' is written to the `implementation-vicinity` for that implementation. Because users may lack permission to write in `implementation-vicinity`, it is good practice to build the new catalog when installing SLIB.

To build (or rebuild) the catalog, start the Scheme implementation (with SLIB), then:

```
(require 'new-catalog)
```

The catalog also supports color-name dictionaries. With an SLIB-installed scheme implementation, type:

```
(require 'color-names)
(make-slib-color-name-db)
(require 'new-catalog)
(slib:exit)
```

7.1.5 Implementation-specific Instructions

Multiple implementations of Scheme can all use the same SLIB directory. Simply configure each implementation's initialization file as outlined above.

SCM Implementation

The SCM implementation does not require any initialization file as SLIB support is already built into SCM. See the documentation with SCM for installation instructions.

VSCM Implementation

From: Matthias Blume <blume @ cs.Princeton.EDU>
Date: Tue, 1 Mar 1994 11:42:31 -0500

Disclaimer: The code below is only a quick hack. If I find some time to spare I might get around to make some more things work.

You have to provide '`vscm.init`' as an explicit command line argument. Since this is not very nice I would recommend the following installation procedure:

1. run scheme
2. `(load "vscm.init")`
3. `(slib:dump "dumpfile")`
4. mv dumpfile place-where-vscm-standard-bootfile-resides e.g. mv dumpfile /usr/local/vscm/lib/sc
boot (In this case vscm should have been compiled with flag -DDEFAULT_BOOTFILE='"/usr/lo
boot"'. See Makefile (definition of DDP) for details.)

Scheme48 Implementation

To make a Scheme48 image for an installation under `<prefix>`,

1. `cd` to the SLIB directory
2. type `make prefix=<prefix> slib48`.
3. To install the image, type `make prefix=<prefix> install48`. This will also create a shell script with the name `slib48` which will invoke the saved image.

PLT Scheme Implementation
DrScheme Implementation
MzScheme Implementation

The '`init.ss`' file in the _slibinit_ collection is an SLIB initialization file.

To use SLIB in MzScheme, set the *SCHEME_LIBRARY_PATH* environment variable to the installed SLIB location; then invoke MzScheme thus:

```
mzscheme -L init.ss slibinit
```

MIT Scheme Implementation

```
scheme -load ${SCHEME_LIBRARY_PATH}mitscheme.init
```

Guile Implementation

```
guile -l ${SCHEME_LIBRARY_PATH}guile.init
```

7.2 Porting

If there is no initialization file for your Scheme implementation, you will have to create one. Your Scheme implementation must be largely compliant with *IEEE Std 1178-1990*, *Revised^4 Report on the Algorithmic Language Scheme*, or *Revised^5 Report on the Algorithmic Language Scheme* in order to support SLIB.[1]

'`Template.scm`' is an example configuration file. The comments inside will direct you on how to customize it to reflect your system. Give your new initialization file the implementation's name with '`.init`' appended. For instance, if you were porting `foo-scheme` then the initialization file might be called '`foo.init`'.

Your customized version should then be loaded as part of your scheme implementation's initialization. It will load '`require.scm`' from the library; this will allow the use of `provide`, `provided?`, and `require` along with the *vicinity* functions (these functions are documented in the section Section 1.5.1 [Require], page 4). The rest of the library will then be accessible in a system independent fashion.

Please mail new working configuration files to `agj @ alum.mit.edu` so that they can be included in the SLIB distribution.

7.3 Coding Guidelines

All library packages are written in IEEE P1178 Scheme and assume that a configuration file and '`require.scm`' package have already been loaded. Other versions of Scheme can be supported in library packages as well by using, for example, `(provided? 'rev3-report)` or `(require 'rev3-report)` (see Section 1.5.1 [Require], page 4).

[1] If you are porting a *Revised^3 Report on the Algorithmic Language Scheme* implementation, then you will need to finish writing '`sc4sc3.scm`' and `load` it from your initialization file.

The module name and '`:`' should prefix each symbol defined in the package. Definitions for external use should then be exported by having `(define foo module-name:foo)`.

Code submitted for inclusion in SLIB should not duplicate routines already in SLIB files. Use `require` to force those library routines to be used by your package. Care should be taken that there are no circularities in the `require`s and `load`s between the library packages.

Documentation should be provided in Emacs Texinfo format if possible, But documentation must be provided.

Your package will be released sooner with SLIB if you send me a file which tests your code. Please run this test *before* you send me the code!

7.3.1 Modifications

Please document your changes. A line or two for '`ChangeLog`' is sufficient for simple fixes or extensions. Look at the format of '`ChangeLog`' to see what information is desired. Please send me `diff` files from the latest SLIB distribution (remember to send `diff`s of '`slib.texi`' and '`ChangeLog`'). This makes for less email traffic and makes it easier for me to integrate when more than one person is changing a file (this happens a lot with '`slib.texi`' and '`*.init`' files).

If someone else wrote a package you want to significantly modify, please try to contact the author, who may be working on a new version. This will insure against wasting effort on obsolete versions.

Please *do not* reformat the source code with your favorite beautifier, make 10 fixes, and send me the resulting source code. I do not have the time to fish through 10000 diffs to find your 10 real fixes.

7.4 Copyrights

This section has instructions for SLIB authors regarding copyrights.

Each package in SLIB must either be in the public domain, or come with a statement of terms permitting users to copy, redistribute and modify it. The comments at the beginning of '`require.scm`' and '`macwork.scm`' illustrate copyright and appropriate terms.

If your code or changes amount to less than about 10 lines, you do not need to add your copyright or send a disclaimer.

7.4.1 Putting code into the Public Domain

In order to put code in the public domain you should sign a copyright disclaimer and send it to the SLIB maintainer. Contact agj @ alum.mit.edu for the address to mail the disclaimer to.

> I, *name*, hereby affirm that I have placed the software package *name* in the public domain.

> I affirm that I am the sole author and sole copyright holder for the software package, that I have the right to place this software package in the public domain, and that I will do nothing to undermine this status in the future.
>
> *signature and date*

This wording assumes that you are the sole author. If you are not the sole author, the wording needs to be different. If you don't want to be bothered with sending a letter every time you release or modify a module, make your letter say that it also applies to your future revisions of that module.

Make sure no employer has any claim to the copyright on the work you are submitting. If there is any doubt, create a copyright disclaimer and have your employer sign it. Mail the signed disclaimer to the SLIB maintainer. Contact agj @ alum.mit.edu for the address to mail the disclaimer to. An example disclaimer follows.

7.4.2 Explicit copying terms

If you submit more than about 10 lines of code which you are not placing into the Public Domain (by sending me a disclaimer) you need to:

- Arrange that your name appears in a copyright line for the appropriate year. Multiple copyright lines are acceptable.
- With your copyright line, specify any terms you require to be different from those already in the file.
- Make sure no employer has any claim to the copyright on the work you are submitting. If there is any doubt, create a copyright disclaimer and have your employer sign it. Mail the signed disclaim to the SLIB maintainer. Contact agj @ alum.mit.edu for the address to mail the disclaimer to.

7.4.3 Example: Company Copyright Disclaimer

This disclaimer should be signed by a vice president or general manager of the company. If you can't get at them, anyone else authorized to license out software produced there will do. Here is a sample wording:

> *employer* Corporation hereby disclaims all copyright interest in the program *program* written by *name*.
>
> *employer* Corporation affirms that it has no other intellectual property interest that would undermine this release, and will do nothing to undermine it in the future.
>
> *signature and date*,
> *name*, *title*, *employer* Corporation

Procedure and Macro Index

This is an alphabetical list of all the procedures and macros in SLIB.

Variable Index

This is an alphabetical list of all the global variables in SLIB.

Concept and Feature Index

Table of Contents