

SLIB

The Portable Scheme Library
Edition 2.01, for SLIB version 2a2
April 1994

by Todd R. Eigenschink, Dave Love, and Aubrey Jaffer

Copyright © 1993, 1994 Todd R. Eigenschink and Aubrey Jaffer

Permission is granted to make and distribute verbatim copies of this manual provided the copyright notice and this permission notice are preserved on all copies.

Permission is granted to copy and distribute modified versions of this manual under the conditions for verbatim copying, provided that the entire resulting derived work is distributed under the terms of a permission notice identical to this one.

Permission is granted to copy and distribute translations of this manual into another language, under the above conditions for modified versions, except that this permission notice may be stated in a translation approved by the author.

1 Overview

SLIB is a portable Scheme library meant to provide compatibility and utility functions for all standard Scheme implementations, and fixes several implementations which are non-conforming. SLIB conforms to *Revised⁴ Report on the Algorithmic Language Scheme* and the IEEE P1178 specification. SLIB supports Unix and similar systems, VMS, and MS-DOS.

For a summary of what each file contains, see the file `README`. For a list of the features that have changed since the last SLIB release, see the file `ANNOUNCE`. For a list of the features that have changed over time, see the file `ChangeLog`.

The maintainer can be reached as `jaffer@ai.mit.edu` or Aubrey Jaffer, 84 Pleasant St., Wakefield, MA 01880-1846.

1.1 Installation

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.

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`.

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 you can modify the startup file for your Scheme implementation to load this initialization file. SLIB is then installed.

Multiple implementations of Scheme can all use the same SLIB directory. Simply configure each implementation's initialization file as outlined above.

The SCM implementation does not require any initialization file as SLIB support is already built in to SCM. See the documentation with SCM for installation instructions.

SLIB includes methods to create heap images for the VSCM and Scheme48 implementations. The instructions for creating a VSCM image are in comments in `vscm.init`. To make a Scheme48 image, `cd` to the SLIB directory and type `make slib48`. This will also create a shell script with the name `slib48` which will invoke the saved image.

1.2 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* or *Revised⁴ Report on the Algorithmic Language Scheme* to support SLIB.

`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` (See Section 6.9 [Require], page 102) from the library; this will allow the use of `provide`, `provided?`, and `require` along with the *vicinity* functions (*vicinity* functions are documented in the section on Require. See Section 6.9 [Require], page 102). The rest of the library will then be accessible in a system independent fashion.

Please mail new working configuration files to `jaffer@ai.mit.edu` so that they can be included in the SLIB distribution.

1.3 Coding Standards

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 6.9 [Require], page 102).

`require.scm` defines `*catalog*`, an association list of module names and filenames. When a new package is added to the library, an entry should be added to `require.scm`. Local packages can also be added to `*catalog*` and even shadow entries already in the table.

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)`.

Submitted code should not duplicate routines which are already in SLIB files. Use `require` to force those features to be supported in your package. Care should be taken that there are no circularities in the `requires` and `loads` 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!

Modifications

Please document your changes. A line or 2 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 `diffs` 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.

If you are considering adding behavior or functions to an SLIB package which implements standard constructs (from a Scheme *Report* or other Scheme book, *Common Lisp*, or Scheme implementations) you should also provide a version which enhances the standard version of these constructs.

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.

1.4 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.

Putting code into the Public Domain

In order to put code in the public domain you need to sign a copyright disclaimer and send it to the SLIB maintainer, Aubrey Jaffer, 84 Pleasant St., Wakefield, MA 01880-1846.

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 disclaim to Aubrey Jaffer, 84 Pleasant St., Wakefield, MA 01880-1846. An example disclaimer follows.

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 Aubrey Jaffer, 84 Pleasant St., Wakefield, MA 01880-1846.

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

Please mail the signed disclaimers to:

Aubrey Jaffer
84 Pleasant St.
Wakefield, MA 01880-1846

1.5 Manual Conventions

Things that are labeled as Functions are called for their return values. Things that are labeled as Procedures are called primarily for their side effects.

All examples throughout this text were produced using the `scm` Scheme implementation.

At the beginning of each section, there is a line that looks something like

`(require 'feature)`.

This means that, in order to use `feature`, you must include the line `(require 'feature)` somewhere in your code prior to the use of that feature. `require` will make sure that the feature is loaded.

2 Data Structures

2.1 Arrays

(require 'array)

array? *obj* [Function]

Returns **#t** if the *obj* is an array, and **#f** if not.

make-array *initial-value bound1 bound2 ...* [Function]

Creates and returns an array that has as many dimensions as there are *bounds* and fills it with *initial-value*.

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

(make-array 'foo 3 3) \equiv (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 (make-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)
⇒ F00
(define freds-center
  (make-shared-array fred (lambda (i j) (list (+ 3 i) (+ 3 j)))
                      2 2))
(array-ref freds-center 0 0)
⇒ F00
```

array-rank *obj* [Function]

Returns the number of dimensions of *obj*. If *obj* is not an array, 0 is returned.

array-shape *array* [Function]

array-shape returns a list of inclusive bounds. So:

```
(array-shape (make-array 'foo 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. So:

```
(array-dimensions (make-array 'foo 3 5))
⇒ (3 5)
```

array-in-bounds? *array index1 index2 ...* [Procedure]

Returns **#t** if its arguments would be acceptable to **array-ref**.

array-ref *array index1 index2 ...* [Function]

Returns the element at the (*index1*, *index2*) element in *array*.

array-set! *array new-value index1 index2 ...* [Procedure]

array-1d-ref *array index* [Function]

array-2d-ref *array index index* [Function]

array-3d-ref *array index index index* [Function]

array-1d-set! *array new-value index* [Procedure]

array-2d-set! *array new-value index index* [Procedure]

array-3d-set! *array new-value index index index* [Procedure]

The functions are just fast versions of **array-ref** and **array-set!** that take a fixed number of arguments, and perform no bounds checking.

If you comment out the bounds checking code, this is about as efficient as you could ask for without help from the compiler.

An exercise left to the reader: implement the rest of APL.

2.2 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*)).

2.3 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.

2.4 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 3.8 [Yasos], page 57). 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 . collections` [Procedure]

`do-elts proc . collections` [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 the 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 . collections` [Procedure]

`do-keys proc . collections` [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 . collections` [Function]

A generalization of the list-based `comlist:reduce-init` (See Section 4.2.3 [Lists as sequences], page 68) to collections which will shadow the list-based version if `(require 'collect)` follows `(require 'common-list-functions)` (See Section 4.2 [Common List Functions], page 64).

Examples:

```
(reduce + 0 (vector 1 2 3))
⇒ 6
(reduce union '() '((a b c) (b c d) (d a)))
⇒ (c b d a).
```

any? *pred . collections* [Function]
 A generalization of the list-based **some** (See Section 4.2.3 [Lists as sequences], page 68) to collections.

Example:

```
(any? odd? (list 2 3 4 5))
⇒ #t
```

every? *pred . collections* [Function]
 A generalization of the list-based **every** (See Section 4.2.3 [Lists as sequences], page 68) 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 See Section 3.8.3 [Setters], page 59, 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)
 )
) ) )

```

2.5 Dynamic Data Type

```
(require 'dynamic)
```

`make-dynamic obj`

Create and returns a new *dynamic* whose global value is *obj*.

[Function]

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.

2.6 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 3 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.

2.7 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 procedure 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) \Rightarrow 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 nationalities.

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> \Rightarrow ""
(soundex "") \Rightarrow ""

Examples from Knuth:

```
(map soundex '("Euler" "Gauss" "Hilbert" "Knuth"
               "Lloyd" "Lukasiewicz"))
 $\Rightarrow$  ("E460" "G200" "H416" "K530" "L300" "L222")

(map soundex '("Ellery" "Ghosh" "Heilbronn" "Kant"
               "Ladd" "Lissajous"))
 $\Rightarrow$  ("E460" "G200" "H416" "K530" "L300" "L222")
```

Some cases in which the algorithm fails (Knuth):

```
(map soundex '("Rogers" "Rodgers"))  $\Rightarrow$  ("R262" "R326")
```

```
(map soundex '("Sinclair" "St. Clair")) ⇒ ("S524" "S324")
```

```
(map soundex '("Tchebysheff" "Chebyshev")) ⇒ ("T212" "C121")
```

2.8 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 alphabetic 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 returned. 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}"
```

2.9 Macroless Object System

```
(require 'object)
```

This is the Macroless Object System written by Wade Humeniuk (whumeniu@datap.ca). Conceptual Tributes: Section 3.8 [Yasos], page 57, MacScheme’s %object, CLOS, Lack of R4RS macros.

2.9.1 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

An object's method associated with a generic-predicate. Returns `#t`.

2.9.2 Procedures

make-object *ancestor ...* [Function]

Returns an object. Current object implementation is a tagged vector. *ancestors* are optional and must be objects in terms of `object?`. *ancestors* methods are included in the object. Multiple *ancestors* 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.

2.9.3 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)

```

2.9.3.1 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?

```

2.9.3.2 Number Documentation

Inheritance

```
<number>::()
```

Slots

```
<number>::<x>
```

Generic Methods

```

<number>::value
<number>::set-value!

```

2.9.3.3 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)
  (define self (make-object
    (make-number 1)
    (make-description "A number which can be inverted")))
  (define <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 () (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

```

2.10 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 *parameters*, 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*.

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 *defaults parameter-list* [Function]
defaults is a list of lists 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 filled with the corresponding *default*.

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** an error 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.

getopt->parameter-list *argc argv optnames arities aliases* [Function]

Returns *argv* converted to a parameter-list. *optnames* are the parameter-names. *aliases* is a list of lists of strings and elements of *optnames*. Each of these strings which have length of 1 will be treated as an option by **getopt**. Longer strings will be treated as long-named options when **getopt** gets this capability.

getopt->arglist *argc argv optnames positions arities defaults checks aliases* [Function]

Like **getopt->parameter-list**, but converts *argv* to an argument-list as specified by *optnames*, *positions*, *arities*, *defaults*, *checks*, and *aliases*.

These **getopt** functions can be used with SLIB relational databases. For an example, See Section 2.15.7 [Database Utilities], page 34.

2.11 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.

2.12 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*.

enqueue! 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 [Procedure]

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.

2.13 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*.

record? *obj* [Function]

Returns a true value if *obj* is a record of any type and a false value otherwise. Note that **record?** may be true of any Scheme value; of course, if it returns true for some particular value, then **record-type-descriptor** is applicable to that value and returns an appropriate descriptor.

record-type-descriptor *record* [Function]

Returns a record-type descriptor representing the type of the given record. That is, for example, if the returned descriptor were passed to **record-predicate**, the resulting predicate would return a true value when passed the given record. Note that it is not necessarily the case that the returned descriptor is the one that was passed to **record-constructor** in the call that created the constructor procedure that created the given record.

record-type-name *rtd* [Function]

Returns the type-name associated with the type represented by *rtd*. The returned value is `eqv?` to the *type-name* argument given in the call to `make-record-type` that created the type represented by *rtd*.

record-type-field-names *rtd* [Function]

Returns a list of the symbols naming the fields in members of the type represented by *rtd*. The returned value is `equal?` to the field-names argument given in the call to `make-record-type` that created the type represented by *rtd*.

2.14 Base Table

A base table implementation using Scheme association lists is available as the value of the identifier `alist-table` after doing:

```
(require 'alist-table)
```

Association list base tables are suitable for small databases and support all Scheme types when temporary and readable/writeable Scheme types when saved. I hope support for other base table implementations will be added in the future.

This rest of this section documents the interface for a base table implementation from which the Section 2.15 [Relational Database], page 26, package constructs a Relational system. It will be of interest primarily to those wishing to port or write new base-table implementations.

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 implementation.

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 implementation. 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 correspond 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 *keys* 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.

for-each-key *handle procedure* [Function]

Calls *procedure* once with each *key* in the table opened in *handle* in an unspecified order. An unspecified value is returned.

map-key *handle procedure* [Function]

Returns a list of the values returned by calling *procedure* once with each *key* in the table opened in *handle* in an unspecified order.

ordered-for-each-key *handle procedure* [Function]

Calls *procedure* once with each *key* in the table opened in *handle* in the natural order for the types of the primary key fields of that table. 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**.

2.15 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 2.14 [Base Table], page 23, implementation.

2.15.1 Motivations

Most nontrivial programs contain databases: Makefiles, configure scripts, file backup, calendars, editors, source revision control, CAD systems, display managers, menu GUIs, games, parsers, debuggers, profilers, and even error reporting are all rife with databases. Coding databases is such a common activity in programming that many may not be aware of how often they do it.

A database often starts as a dispatch in a program. The author, perhaps because of the need to make the dispatch configurable, the need for correlating dispatch in other routines, or because of changes or growth, devises a data structure to contain the information, a routine for interpreting that data structure, and perhaps routines for augmenting and modifying the stored data. The dispatch must be converted into this form and tested.

The programmer may need to devise an interactive program for enabling easy examination and modification of the information contained in this database. Often, in an attempt to foster modularity and avoid delays in release, intermediate file formats for the database information are devised. It often turns out that users prefer modifying these intermediate files with a text editor to using the interactive program in order to do operations (such as global changes) not foreseen by the program's author.

In order to address this need, the conscientious software engineer may even provide a scripting language to allow users to make repetitive database changes. Users will grumble that they need to read a large manual and learn yet another programming language (even if it *almost* has language "xyz" syntax) in order to do simple configuration.

All of these facilities need to be designed, coded, debugged, documented, and supported; often causing what was very simple in concept to become a major development project.

This view of databases just outlined is somewhat the reverse of the view of the originators of the *Relational Model* of database abstraction. The relational model was devised to unify and allow interoperation of large multi-user databases running on diverse platforms. A fairly general purpose "Comprehensive Language" for database manipulations is mandated (but not specified) as part of the relational model for databases.

One aspect of the Relational Model of some importance is that the "Comprehensive Language" must be expressible in some form which can be stored in the database. This frees the programmer from having to make programs data-driven in order to use a database.

This package includes as one of its basic supported types Scheme *expressions*. This type allows expressions as defined by the Scheme standards to be stored in the database. Using `slib:eval` retrieved expressions can be evaluated (in the top-level environment). Scheme's `lambda` facilitates closure of environments, modularity, etc. so that procedures (which could not be stored directly most databases) can still be effectively retrieved. Since `slib:eval` evaluates expressions in the top-level environment, built-in and user defined procedures can be easily accessed by name.

This package's purpose is to standardize (through a common interface) database creation and usage in Scheme programs. The relational model's provision for inclusion of language expressions as data as well as the description (in tables, of course) of all of its tables assures that relational databases are powerful enough to assume the roles currently played by thousands of ad-hoc routines and data formats.

Such standardization to a relational-like model brings many benefits:

- Tables, fields, domains, and types can be dealt with by name in programs.
- The underlying database implementation can be changed (for performance or other reasons) by changing a single line of code.
- The formats of tables can be easily extended or changed without altering code.
- Consistency checks are specified as part of the table descriptions. Changes in checks need only occur in one place.
- All the configuration information which the developer wishes to group together is easily grouped, without needing to change programs aware of only some of these tables.
- Generalized report generators, interactive entry programs, and other database utilities can be part of a shared library. The burden of adding configurability to a program is greatly reduced.
- Scheme is the "comprehensive language" for these databases. Scripting for configuration no longer needs to be in a separate language with additional documentation.
- Scheme's latent types mesh well with the strict typing and logical requirements of the relational model.

- Portable formats allow easy interchange of data. The included table descriptions help prevent misinterpretation of format.

2.15.2 Creating and Opening Relational Databases

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 **requireing** 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 implementation. If the database cannot be created as specified **#f** is returned. For the fields and layout of descriptor tables, See Section 2.15.5 [Catalog Representation], page 31,

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.

2.15.3 Relational 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 implementation. If the operations completed successfully, **#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 databases which are *mutable?*.

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 2.15.5 [Catalog Representation], page 31.

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.

2.15.4 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)))

```

Operations on a single column of a table are retrieved by giving the column name as the second argument to the methods procedure. For example:

```
(define column-ids ((telephone-table-desc 'get* 'column-number)))
```

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 may not be missing.

get *key1 key2 ...* [Function]
Returns the value for the specified column of the row associated with primary keys *key1, key2 ...* if it exists, or *#f* otherwise.

get* [Function]
Returns a list of the values for the specified column for all rows in this table.

row:retrieve *key1 key2 ...* [Function]
Returns the row associated with primary keys *key1, key2 ...* if it exists, or *#f* otherwise.

row:retrieve* [Function]
Returns a list of all rows in this table.

row:remove *key1 key2 ...* [Function]
Removes and returns the row associated with primary keys *key1, key2 ...* if it exists, or *#f* otherwise.

row:remove* [Function]
Removes and returns a list of all rows in this table.

row:delete *key1 key2 ...* [Function]
Deletes the row associated with primary keys *key1, key2 ...* if it exists. The value returned is unspecified.

row:delete*	[Function]
Deletes all rows in this table. The value returned is unspecified. The descriptor table and catalog entry for this table are not affected.	
row:update row	[Function]
Adds the row, <i>row</i> , to this table. If a row for the primary key(s) specified by <i>row</i> already exists in this table, it will be overwritten. The value returned is unspecified.	
row:update* rows	[Function]
Adds each row in the list <i>rows</i> , to this table. If a row for the primary key specified by an element of <i>rows</i> already exists in this table, it will be overwritten. The value returned is unspecified.	
row:insert row	[Function]
Adds the row <i>row</i> to this table. If a row for the primary key(s) specified by <i>row</i> already exists in this table an error is signaled. The value returned is unspecified.	
row:insert* rows	[Function]
Adds each row in the list <i>rows</i> , to this table. If a row for the primary key specified by an element of <i>rows</i> already exists in this table, an error is signaled. The value returned is unspecified.	
for-each-row proc	[Function]
Calls <i>proc</i> with each <i>row</i> in this table in the natural ordering for the primary key types. <i>Real</i> relational programmers would use some least-upper-bound join for every row to get them in order; But we don't have joins yet.	
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.	

2.15.5 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>

2.15.6 Unresolved Issues

Although `rdms.scm` is not large I found it very difficult to write (six rewrites). I am not aware of any other examples of a generalized relational system (although there is little new in CS). I left out several aspects of the Relational model in order to simplify the job. The major features lacking (which might be addressed portably) are views, transaction boundaries, and protection.

Protection needs a model for specifying priveledges. Given how operations are accessed from handles it should not be difficult to restrict table accesses to those allowed for that user.

The system catalog has a field called `view-procedure`. This should allow a purely functional implementation of views. This will work but is unsatisfying for views resulting from a *selection* (subset of rows); for whole table operations it will not be possible to reduce the number of keys scanned over when the selection is specified only by an opaque procedure.

Transaction boundaries present the most intriguing area. Transaction boundaries are actually a feature of the "Comprehensive Language" of the Relational database and not of the database. Scheme would seem to provide the opportunity for an extremely clean semantics for transaction boundaries since the builtin procedures with side effects are small in number and easily identified.

These side-effect builtin procedures might all be portably redefined to versions which properly handled transactions. Compiled library routines would need to be recompiled as well. Many system extensions (`delete-file`, `system`, etc.) would also need to be redefined.

There are 2 scope issues that must be resolved for multiprocess transaction boundaries:

Process scope

The actions captured by a transaction should be only for the process which invoked the start of transaction. Although standard Scheme does not provide process primitives as such, `dynamic-wind` would provide a workable hook into process switching for many implementations.

Shared utilities with state

Some shared utilities have state which should *not* be part of a transaction. An example would be calling a pseudo-random number generator. If the success of a transaction depended on the pseudo-random number and failed, the state

of the generator would be set back. Subsequent calls would keep returning the same number and keep failing.

Pseudo-random number generators are not reentrant and so would require locks in order to operate properly in a multiprocess environment. Are all examples of utilities whose state should not part of transactions also non-reentrant? If so, perhaps suspending transaction capture for the duration of locks would fix it.

2.15.7 Database Utilities

(require 'database-utilities)

This enhancement wraps a utility layer on `relational-database` which provides:

- Automatic loading of the appropriate base-table package when opening a database.
- Automatic execution of initialization commands stored in database.
- Transparent execution of database commands stored in `*commands*` table in database.

Also included are utilities which provide:

- Data definition from Scheme lists and
- Report generation

for any SLIB relational database.

`create-database filename base-table-type` [Function]
Returns an open, nearly empty enhanced (with `*commands*` table) relational database (with base-table type `base-table-type`) associated with `filename`.

`open-database filename` [Function]
`open-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-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-database! filename` [Function]
`open-database! filename base-table-type` [Function]
Returns *mutable* open enhanced relational database ...

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
	default	expression
	documentation	string

The `arity` field can take the values:

<code>single</code>	Requires a single parameter of the specified domain.
<code>optional</code>	A single parameter of the specified domain or zero parameters is acceptable.
<code>boolean</code>	A single boolean parameter or zero parameters (in which case <code>#f</code> is substituted) is acceptable.
<code>nary</code>	Any number of parameters of the specified domain are acceptable. The argument passed to the command function is always a list of the parameters.
<code>nary1</code>	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 `indexth` field must satisfy.

The `default` field is an expression whose value is either `#f` or a procedure of no arguments which returns a parameter or parameter list as appropriate. If the expression's value is `#f` then no default is appropriate for this parameter. Note that since the `default` 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.

Invoking Commands

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 2.15.3 [Relational Database Operations], page 28, 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.

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.

defaults A list of the defaults for each parameter. Corresponds to the **defaults** 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 2.10 [Parameter lists], page 19. Here is an example of setting up a command with arguments and parsing those arguments from a *getopt* style argument list (see Section 6.5 [Getopt], page 99).

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

(define my-rdb (create-database #f 'alist-table))

(define-tables my-rdb
  '(foo-params
    *parameter-columns*
    *parameter-columns*
    ((1 first-argument single string "hithere" "first argument")
     (2 flag boolean boolean #f "a flag"))))
  '(foo-pnames
    ((name string))
    ((parameter-index uint)))
```

```

      (("l" 1)
       ("a" 2)))
'(my-commands
  ((name symbol))
  ((parameters parameter-list)
   (parameter-names parameter-name-translation)
   (procedure expression)
   (documentation string))
  ((foo
    foo-params
    foo-pnames
    (lambda (rdb) (lambda (foo aflag) (print foo aflag)))
    "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 defaults dirs aliases)
    (apply comval (getopt->arglist argc argv options positions
    arities defaults dirs aliases)))))

(define (test)
  (set! *optind* 1)
  (dbutil:serve-command-line
   my-rdb 'my-commands 'foo 4 '("dummy" "-l" "foo" "-a")))
(test)
-|
"foo" #t

```

Some commands are defined in all extended relational-databases. The are called just like Section 2.15.3 [Relational Database Operations], page 28.

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 2.15.5 [Catalog Representation], page 31,

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*.

Defining Tables

define-tables *rdb spec-0* . . . [Procedure]

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 (and returns non-#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 foreign-key domain will be created for it.

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**.

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.

The report is prepared as follows:

- `Format` (see Section 4.3 [Format], page 73) 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.

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 'database-utilities)
(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

```

2.16 Red-Black Trees

```
(require 'red-black-tree)
```

This is an implementation of Red-Black trees in Scheme.

make-rb-tree *left-rotation-field-maintainer*

[Function]

right-rotation-field-maintainer insertion-field-maintainer

deletion-field-maintainer prior?

Makes an empty Red-Black tree based on the arguments:

left-rotation-field-maintainer

right-rotation-field-maintainer

Invoked in rotations to maintain augmented fields. Args are X and Y. If you have no augmented fields that depend explicitly on the structure of the tree, make these null.

insertion-field-maintainer

Invoked in insertion to maintain fields. Invoked once, on node inserted after insertion is performed, but before rotations are performed to balance tree. May also be invoked in one case of deletion.

deletion-field-maintainer

Invoked in deletion to maintain fields. Invoked once, on node deleted after deletion is performed, but before rotations are performed to balance the tree. May be invoked on non-garbage nodes during deletion, where node is spliced out of one place and into another. The tree is always guaranteed connected from parent up.

prior? *prior?* should be a binary predicate used for totally ordering the data fields of nodes. The name *prior?* is just a mnemonic aid; it means that if the predicate is successful the first arg should go to the left of the second arg, where left is in tree fields.

rb-delete! *tree node* [Procedure]

Deletes *node* from *tree*. The node that is actually deleted may not be the one passed in, so if a resource is being maintained, what should be put back on the freelist is the *node* returned by this procedure.

rb-node-successor *node* [Function]

rb-node-predecessor *node* [Function]

Return the successor and predecessor (as determined by the *prior?* argument to **make-rb-tree**) from the tree of which *node* is a member.

rb-node-maximum *node* [Function]

rb-node-minimum *node* [Function]

Return the minimum and maximum nodes in the tree of which *node* is a member.

rb-tree-maximum *tree* [Function]

rb-tree-minimum *tree* [Function]

Return the minimum and maximum nodes in *tree*.

rb-insert! *tree node* [Procedure]

Inserts *node* in *tree*. The value returned is unspecified.

make-rb-node *data* [Function]

Makes a node (suitable for insertion with **rb-insert!**) with datum *data*.

2.17 Structures

(require 'struct) for defmacros. (require 'structure) for syntax-case macros.

defmacros which implement *records* from the book *Essentials of Programming Languages* by Daniel P. Friedman, M. Wand and C.T. Haynes. Copyright 1992 Jeff Alexander, Shinnder Lee, and Lewis Patterson

define-record *tag* (*var1 var2 ...*) [Macro]

Defines several functions pertaining to record-name *tag*: *make-tag*, *tag?*, and *tag->var1*.

variant-case *exp* (*tag* (*var1 var2 ...*) *body*) ... [Macro]

executes the following for the matching clause:

```
((lambda (var1 var ...) body)
  (tag->var1 exp)
  (tag->var2 exp) ...)
```

3 Macros

3.1 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 **defmacr** 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**.

3.1.1 Defmacroexpand

(require 'defmacroexpand)

defmacro:expand* *e* [Function]

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

3.2 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**.

3.3 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)

3.3.1 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 3.4 [Macros That Work], page 45, Section 3.5 [Syntactic Closures], page 48, and Section 3.6 [Syntax-Case Macros], page 55.

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-rules*, beginning with the leftmost *syntax-rule*. When a match is found, the macro use is transcribed 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.

3.4 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 *Revised⁴ 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)

```

3.4.1 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.

3.4.2 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.

3.5 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**.

3.5.1 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 *Revised⁴ 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**.

3.5.1.1 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.

3.5.1.2 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 (caddrr 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 (caddrr exp))
            (return (caddrr (cdr exp)))
            (step (caddrr (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)

```

```
...))))
```

3.5.1.3 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 identifiers 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)
```

3.5.1.4 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.

3.6 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 6.1 [Repl], page 97) 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).

3.6.1 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).

3.7 Fluid-Let

(require 'fluid-let)

`fluid-let` (*bindings ...*) *forms...* [Syntax]
 (`fluid-let` ((*variable init*) ...) *expression expression ...*)

The *inits* are evaluated in the current environment (in some unspecified order), the current values of the *variables* are saved, the results are assigned to the *variables*, the *expressions* are evaluated sequentially in the current environment, the *variables* 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 *variables*. 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*.

3.8 Yasos

(require 'oop) or (require 'yasos)

‘Yet Another Scheme Object System’. 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].

3.8.1 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.

3.8.2 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 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 4.3 [Format], page 73) for non-instances and prints *obj* preceded by

'#<INSTANCE>' for instances. Note that `print` is also defined in the `debug` module (See Section 6.2 [Debug], page 97) and these will conflict.

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 2.4 [Collections], page 7) may override the default in an obvious way.

3.8.3 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 Yason objects (See Section 3.8 [Yason], page 57). 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) #\0) ; generalized location with getter
foo ⇒ "Fo0"
(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.

3.8.4 Examples

```
(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

```

4 Procedures

Anything that doesn't fall neatly into any of the other categories winds up here.

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.

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"
```

logtest *j k* [Function]

(logtest *j k*) \equiv (not (zero? (logand *j k*)))

```
(logtest #b0100 #b1011) ⇒ #f
(logtest #b0100 #b0111) ⇒ #t
```

logbit? *index j* [Function]

(logbit? *index j*) \equiv (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
```

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"
```

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
```

integer-length *n* [Function]

Returns the number of bits necessary 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
```

bit-extract *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.

Example:

```
(number->string (bit-extract #b1101101010 0 4) 2)
⇒ "1010"
(number->string (bit-extract #b1101101010 4 9) 2)
⇒ "10110"
```

4.2 Common List Functions

(require 'common-list-functions)

The procedures below follow the Common LISP equivalents apart from optional arguments in some cases.

4.2.1 List construction

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* *x . y* [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

```

4.2.2 Lists as sets

`eq?` is used to test for membership by all the procedures below 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* with duplicates removed.

Example:

```

(union '(1 2 3 4) '(5 6 7 8))
⇒ (4 3 2 1 5 6 7 8)
(union '(1 2 2 1) '(3 4 1 8))
⇒ (2 3 4 1 8)

```

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 the union of 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 lst . more-lsts* [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 lst . more-lsts* [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 . lst* [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 lst* [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
```

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 *elements* 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 *elements* 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
```

4.2.3 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 2.4 [Collections], page 7) 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)
```

butlast *lst* *n*

[Function]

butlast returns all but the last *n* elements of *lst*.

Example:

```
(butlast '(1 2 3 4) 3)
⇒ (1)
(butlast '(1 2 3 4) 4)
⇒ ()
```

nthcdr *n* *lst*

[Function]

nthcdr takes *n* *cdrs* of *lst* and returns the result. Thus `(nthcdr 3 lst)` \equiv `(cddddr lst)`

Example:

```
(nthcdr 2 '(1 2 3 4))
⇒ (3 4)
(nthcdr 0 '(1 2 3 4))
⇒ (1 2 3 4)
```

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)
```


$\Rightarrow 0$

4.2.4 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 5.3 [Rev2 Procedures], page 91).

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)
 $\Rightarrow$  (a b c d e f)
x
 $\Rightarrow$  (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 **cdrs** of the list. Sometimes called **reverse!**.

Example:

```
(define foo '(a b c))
(nreverse foo)
 $\Rightarrow$  (c b a)
foo
 $\Rightarrow$  (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.

<code>delete elt lst</code>	[Procedure]
<code>delete-if pred lst</code>	[Procedure]
<code>delete-if-not pred lst</code>	[Procedure]

Destructive versions of `remove` `remove-if`, and `remove-if-not`.

Example:

```
(define lst '(foo bar baz bang))
(delete 'foo lst)
⇒ (bar baz bang)
lst
⇒ (foo bar baz bang)

(define lst '(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.

4.2.5 Non-List functions

<code>and? . args</code>	[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
```

<code>or? . args</code>	[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
```

4.3 Format

```
(require 'format)
```

4.3.1 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.

4.3.2 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 | #]

4.3.2.1 Implemented CL Format Control Directives

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

<code>~A</code>	Any (print as <code>display</code> does).
<code>~@A</code>	left pad.
<code>~mincol,colinc,minpad,padcharA</code>	full padding.
<code>~S</code>	S-expression (print as <code>write</code> does).
<code>~@S</code>	left pad.
<code>~mincol,colinc,minpad,padcharS</code>	full padding.
<code>~D</code>	Decimal.
<code>~@D</code>	print number sign always.
<code>~:D</code>	print comma separated.
<code>~mincol,padchar,commacharD</code>	padding.
<code>~X</code>	Hexadecimal.
<code>~@X</code>	print number sign always.
<code>~:X</code>	print comma separated.
<code>~mincol,padchar,commacharX</code>	padding.
<code>~O</code>	Octal.
<code>~@O</code>	print number sign always.
<code>~:O</code>	print comma separated.
<code>~mincol,padchar,commacharO</code>	padding.
<code>~B</code>	Binary.
<code>~@B</code>	print number sign always.
<code>~:B</code>	print comma separated.
<code>~mincol,padchar,commacharB</code>	padding.
<code>~nR</code>	Radix <i>n</i> .
<code>~n,mincol,padchar,commacharR</code>	padding.
<code>~@R</code>	print a number as a Roman numeral.

<code>~:R</code>	print a number as an ordinal English number.
<code>~:@R</code>	print a number as a cardinal English number.
<code>~P</code>	Plural.
<code>~@P</code>	prints <i>y</i> and <i>ies</i> .
<code>~:P</code>	as <code>~P</code> but jumps 1 argument backward.
<code>~:@P</code>	as <code>~@P</code> but jumps 1 argument backward.
<code>~C</code>	Character.
<code>~@C</code>	prints a character as the reader can understand it (i.e. <code>#\</code> prefixing).
<code>~:C</code>	prints a character as emacs does (eg. <code>^C</code> for ASCII 03).
<code>~F</code>	Fixed-format floating-point (prints a flonum like <i>mmm.nnn</i>).
	<code>~width,digits,scale,overflowchar,padcharF</code>
<code>~@F</code>	If the number is positive a plus sign is printed.
<code>~E</code>	Exponential floating-point (prints a flonum like <i>mmm.nnnEee</i>).
	<code>~width,digits,exponentdigits,scale,overflowchar,padchar,exponentcharE</code>
<code>~@E</code>	If the number is positive a plus sign is printed.
<code>~G</code>	General floating-point (prints a flonum either fixed or exponential).
	<code>~width,digits,exponentdigits,scale,overflowchar,padchar,exponentcharG</code>
<code>~@G</code>	If the number is positive a plus sign is printed.
<code>~\$</code>	Dollars floating-point (prints a flonum in fixed with signs separated).
	<code>~digits,scale,width,padchar\$</code>
<code>~@\$</code>	If the number is positive a plus sign is printed.
<code>~:@\$</code>	A sign is always printed and appears before the padding.
<code>~:\$</code>	The sign appears before the padding.
<code>~%</code>	Newline.
<code>~n%</code>	print <i>n</i> newlines.
<code>~&</code>	print newline if not at the beginning of the output line.
<code>~n&</code>	prints <code>~&</code> and then <i>n-1</i> newlines.
<code>~ </code>	Page Separator.
<code>~n </code>	print <i>n</i> page separators.
<code>~~</code>	Tilde.
<code>~n~</code>	print <i>n</i> tildes.
<code>~<newline></code>	Continuation Line.
<code>~:<newline></code>	newline is ignored, white space left.

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

4.3.2.2 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)

4.3.2.3 Extended, Replaced and Additional Control Directives

- `~mincol, padchar, commachar, commawidthD`
- `~mincol, padchar, commachar, commawidthX`
- `~mincol, padchar, commachar, commawidth0`
- `~mincol, padchar, commachar, commawidthB`
- `~n, mincol, padchar, commachar, commawidthR`
 `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.
- `~nC` 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.

4.3.2.4 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`)

4.3.2.5 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.

4.4 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)
```

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

4.6 Modular Arithmetic

(require 'modular)

extended-euclid *n1 n2* [Function]
Returns a list of 3 integers (*d x y*) such that $d = \gcd(n1, n2) = n1 * x + n2 * y$.

For all of these procedure all arguments should be exact non-negative integers such that $k1 > k2$ and $k1 > k3$. The returned value will be an exact non-negative integer less than $k1$. If all the arguments are fixnums the computation will use only fixnums.

modular:invert *k1 k2* [Function]
Returns an integer *n* such that $1 = (n * k2) \bmod k1$. If $k2$ has no inverse mod $k1$ an error is signaled.

modular:negate *k1 k2* [Function]
Returns $(-k2) \bmod k1$.

modular:+ *k1 k2 k3* [Function]
Returns $(k2 + k3) \bmod k1$.

modular:- *k1 k2 k3* [Function]
Returns $(k2 - k3) \bmod k1$.

modular:* *k1 k2 k3* [Function]
Returns $(k2 * k3) \bmod k1$.

modular:expt *k1 k2 k3* [Function]
Returns $(k2 \wedge k3) \bmod k1$.

4.7 Multi-Processing

(require 'process)

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 6.13 [System], page 106).

4.8 Object-To-String

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

object->string *obj* [Function]
 Returns the textual representation of *obj* as a string.

4.9 Plotting on Character Devices

```
(require 'charplot)
```

The plotting procedure is made available through the use of the `charplot` package. `charplot` is loaded by inserting `(require 'charplot)` before the code that uses this procedure.

charplot:height [Variable]
 The number of rows to make the plot vertically.

charplot:width [Variable]
 The number of columns to make the plot horizontally.

plot! *coords x-label y-label* [Procedure]
coords is a list of pairs of x and y coordinates. *x-label* and *y-label* are strings with which to label the x and y axes.

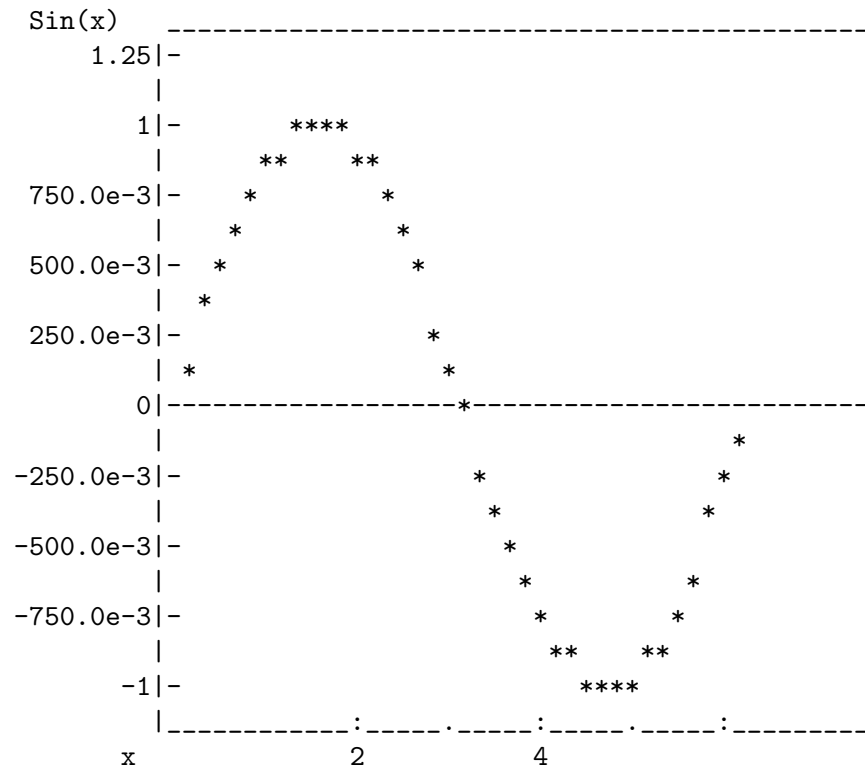
Example:

```
(require 'charplot)
(set! charplot:height 19)
(set! charplot:width 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)")
+

```



4.10 Pretty-Print

(require 'pretty-print)

`pretty-print obj` [Procedure]

`pretty-print obj port` [Procedure]

`pretty-prints 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))
```

(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-printed** to this port.

Whitespace 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")
```

4.11 Prime Factorization

`(require 'prime)`

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? p` [Function]

Returns `#f` if *p* is composite; `#t` if *p* is prime. There is a slight chance (`expt 2 (- prime:trials)`) that a composite will return `#t`.

`prime:trials` [Function]

Is the maximum number of iterations of Solovay-Strassen that will be done to test a number for primality.

`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) <)`.

4.12 Random Numbers

`(require 'random)`

`random n` [Procedure]

`random n state` [Procedure]

Accepts a positive integer or real *n* and returns a number of the same type between zero (inclusive) and *n* (exclusive). The values returned have a uniform distribution.

The optional argument *state* must be of the type produced by `(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 the `random` operation.

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.

make-random-state [Procedure]

make-random-state *state* [Procedure]

Returns a new object of type suitable for use as the value of the variable `*random-state*` and as a second argument to `random`. If argument *state* is given, a copy of it is returned. Otherwise a copy of `*random-state*` is returned.

If inexact numbers are support by the Scheme implementation, `randinex.scm` will be loaded as well. `randinex.scm` contains procedures for generating inexact distributions.

random:uniform *state* [Procedure]

Returns an uniformly distributed inexact real random number in the range between 0 and 1.

random:solid-sphere! *vect* [Procedure]

random:solid-sphere! *vect state* [Procedure]

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 $n = (\text{vector-length } vect)$, the coordinates are uniformly distributed within the unit n -sphere. The sum of the squares of the numbers is returned.

random:hollow-sphere! *vect* [Procedure]

random:hollow-sphere! *vect state* [Procedure]

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 $n = (\text{vector-length } vect)$, the coordinates are uniformly distributed over the surface of the unit n -sphere.

random:normal [Procedure]

random:normal *state* [Procedure]

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* [Procedure]

random:normal-vector! *vect state* [Procedure]

Fills *vect* with inexact real random numbers which are independent and standard normally distributed (i.e., with mean 0 and variance 1).

`random:exp` [Procedure]
`random:exp state` [Procedure]

Returns an inexact real in an exponential distribution with mean 1. For an exponential distribution with mean *u* use `(* u (random:exp))`.

4.13 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.

4.14 Standard I/O

```
(require 'stdio)
```

printf *format . args* [Procedure]

fprintf *port format . args* [Procedure]

sprintf *str format . args* [Procedure]

Note: Floating-point output is not handled yet.

stdin [Variable]

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

stdout [Variable]

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

stderr [Variable]

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

scanf *format . args* [Function]

fscanf *port format . args* [Function]

sscanf *str format . args* [Function]

Each function reads characters, interprets them according to the control string *format* argument, and returns a list of the items specified. *args* are ignored; they are allowed for compatibility with the C function of the same name. The control string 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 up to the next non-white-space character (except in cases described below).
- 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' (ell), 'h' or 'L' which is ignored, and a conversion code.

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 non-space characters; it extends to the next inappropriate character or until the field width, if specified, is exhausted. For all descriptors except 'c', white space leading an input field is ignored.

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'	A decimal integer is expected.
'o'	An octal integer is expected.
'x'	A hexadecimal integer is expected.
'f'	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'	A character is 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; 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.

The 'l', 'L' and 'h' modifiers are ignored.

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.

4.15 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.

4.16 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.

4.17 Tektronix Graphics Support

Note: The Tektronix graphics support files need more work, and are not complete.

4.17.1 Tektronix 4000 Series Graphics

The Tektronix 4000 series graphics protocol gives the user a 1024 by 1024 square drawing area. The origin is in the lower left corner of the screen. Increasing y is up and increasing x is to the right.

The graphics control codes are sent over the current-output-port and can be mixed with regular text and ANSI or other terminal control sequences.

`tek40:init` [Procedure]

`tek40:graphics` [Procedure]

`tek40:text` [Procedure]

`tek40:linetype linetype` [Procedure]

`tek40:move x y` [Procedure]

`tek40:draw x y` [Procedure]

`tek40:put-text x y str` [Procedure]

`tek40:reset` [Procedure]

4.17.2 Tektronix 4100 Series Graphics

The graphics control codes are sent over the current-output-port and can be mixed with regular text and ANSI or other terminal control sequences.

<code>tek41:init</code>	[Procedure]
<code>tek41:reset</code>	[Procedure]
<code>tek41:graphics</code>	[Procedure]
<code>tek41:move x y</code>	[Procedure]
<code>tek41:draw x y</code>	[Procedure]
<code>tek41:point x y number</code>	[Procedure]
<code>tek41:encode-x-y x y</code>	[Procedure]
<code>tek41:encode-int number</code>	[Procedure]

4.18 Tree operations

These are operations that treat lists a representations of trees.

<code>subst new old tree</code>	[Function]
<code>substq new old tree</code>	[Function]
<code>substv new old tree</code>	[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.

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))
```

<code>copy-tree tree</code>	[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
```

5 Standards Support

5.1 With-File

(require 'with-file)

with-input-from-file *file thunk* [Function]

with-output-to-file *file thunk* [Function]

Description found in R4RS.

5.2 Transcripts

(require 'transcript)

transcript-on *filename* [Function]

transcript-off *filename* [Function]

Redefines read-char, read, write-char, write, display, and newline.

5.3 Rev2 Procedures

(require 'rev2-procedures)

The procedures below were specified in the *Revised² Report on Scheme*. **N.B.:** The symbols 1+ and -1+ are not *R4RS* syntax. Scheme->C, for instance, barfs 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

$0 \leq start1 \leq end1 \leq (\text{string-length } string1)$

$0 \leq start2 \leq end1 - start1 + start2 \leq (\text{string-length } string2)$

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 indices.

substring-fill! *string start end char* [Procedure]

Fills the elements *start*–*end* of *string* with the character *char*.

string-null? *str* [Function]

$\equiv (= 0 (\text{string-length } str))$

append! . *pairs* [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 ‘?’.

`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 '())
```

`t` [Variable]

Defined as `#t`.

`nil` [Variable]

Defined as `#f`.

5.4 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]

5.5 Multi-argument / and -

(require 'mutliarg/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 . divisors` [Function]

`two-arg:- n1 n2` [Function]

The original two-argument version of `-`.

`- minuend . subtrahends` [Function]

5.6 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 . args` [Function]

5.7 Rationalize

(require 'rationalize)

The procedure `rationalize` is interesting because most programming languages do not provide anything analogous to it. For simplicity, we present an algorithm which 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. We thank Alan Bawden for contributing this algorithm.

`rationalize x e` [Function]

5.8 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*).

5.9 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.

5.10 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.

5.11 Time

The procedures **current-time**, **difftime**, and **offset-time** are supported by all implementations (SLIB provides them if feature ('current-time) is missing. **current-time** returns a *calendar time* (caltime) which can be a number or other type.

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 5.12 [CLTime], page 96. On implementations which cannot support actual times, **current-time** will increment a counter and return its value when called.

difftime *caltime1 caltime0* [Function]

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

offset-time *caltime offset* [Function]

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


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

These procedures are intended to be compatible with Posix time conversion functions.

timezone [Variable]
contains the difference, in seconds, between UTC and local standard time (for example, in the U.S. Eastern time zone (EST), *timezone* is 5*60*60). ***timezone*** is initialized by **tzset**.

tzset [Function]
initializes the **timezone** variable from the TZ environment variable. This function is automatically called by the other time conversion functions that depend on the time zone.

gmtime caltime [Function]
converts the calendar time *caltime* to a vector of integers representing the time expressed as Coordinated Universal Time (UTC).

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 by calling **tzset**. The elements of the returned vector are as follows:

0. seconds (0 - 61)
1. minutes (0 - 59)
2. hours since midnight
3. day of month
4. month (0 - 11). Note difference from **decode-universal-time**.
5. year (A.D.)
6. day of week (0 - 6)
7. day of year (0 - 365)
8. 1 for daylight savings, 0 for regular time

mktime univtime [Function]
Converts a vector of integers in Coordinated Universal Time (UTC) format to calendar time (caltime) format.

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".

ctime caltime [Function]
Equivalent to (time:asctime (time:localtime *caltime*)).

5.12 CLTime

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.)
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* [Function]
time-zone

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**.

6 Session Support

6.1 Repl

```
(require 'repl)
```

Here is a read-eval-print-loop which, given an eval, evaluates forms.

```
repl:top-level repl:eval [Procedure]
  reads, repl:evals and writes 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)
```

6.2 Debug

```
(require 'debug)
```

```
print args [Procedure]
  Print writes all its arguments, separated by spaces. Print outputs a newline at the
  end and returns the value of the last argument. Note that print is also defined in
  the yasos module (See Section 3.8 [Yasos], page 57).
```

```
qp args [Procedure]
qpn args [Procedure]
qpr args [Procedure]
  qp writes its arguments, separated by spaces, to (current-output-port). qp com-
  presses 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 uses.
```

trace-all *file* [Procedure]
 Traces (see Section 6.3 [Trace], page 98) all procedures **defined** at top-level in file *file*.

6.2.1 Breakpoints

init-debug [Function]
 Typing (**init-debug**) at top level sets up a continuation for **break**.

break *arg1* ... [Function]
 Returns from the top level continuation and pushes the continuation from which it was called on **debug:break-continuation-stack**.

continue [Function]
 Pops the topmost continuation off of **debug:break-continuation-stack** and returns **#f** to it.

6.3 Tracing

(require 'trace)

trace *proc1* ... [Macro]
 Traces the top-level named procedures given as arguments.

trace [Macro]
 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.

untrace *proc1* ... [Macro]
 Turns tracing off for its arguments.

untrace [Macro]
 With no arguments, untraces all currently traced identifiers and returns a list of these formerly traced identifiers.

The following routines are the procedures which actually do the tracing when this module is supplied by SLIB, rather than natively. If macros are not natively supported by your implementation, these might be more convenient to use.

tracef [Function]

debug:tracef [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 [Function]
debug:untracef *function* [Function]
 To untrace, type
 (set! *symbol* (untracef *symbol*))

6.4 Test

(require 'test)

test *expected fun . args* [Function]
 Compares *expected* to the result of applying *fun* to *args*. If they are not `equal?` then prints an error message. An unspecified value is returned.

report-errs [Function]
 Prints a summary of all error encountered by **test**. An unspecified value is returned.

6.5 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.scm)
(require 'debug)

(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)
```

6.6 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 'debug)
(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"
```

6.7 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 backslash (\).

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 <code>newline</code> , that <code>newline</code> is just ignored. Processing continues from the characters after the <code>newline</code> as though the backslash and <code>newline</code> were not there.
"	The characters up to the next unescaped " are taken literally, according to [R4RS] rules for literal strings (see Section “Strings” in <i>Revised(4) Scheme</i>).
(
'	One scheme expression is <code>read</code> starting with this character. The <code>read</code> expression is evaluated, converted to a string (using <code>display</code>), 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 <code>read-domannd-line</code> and backslashes before <code>newlines</code> in comments are also ignored.
---	--

6.8 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.

6.9 Require

These variables and procedures are provided by all implementations.

`*features*` [Variable]
Is a list of symbols denoting features supported in this implementation.

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 three functions if *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]

If (not (provided? *feature*)) it is loaded if *feature* is a pathname or if (assq *feature* *catalog*). Otherwise an error is signaled.

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***.

Below is a list of features that are automatically determined by **require**. For each item, (provided? '*feature*') will return **#t** if that feature is available, and **#f** if not.

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

6.10 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 *filename* [Function]
 Returns the vicinity of *filename* 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).

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*.

6.11 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 than the largest value which can be returned by **char->integer**.

most-positive-fixnum [Constant]
 The immediate integer closest to positive infinity.

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 "2a2" on scm "4e1" on unix

slib:report [Function]
 Displays the information of (slib:report-version) followed by almost all the information necessary 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 "2a2" on scm "4e1" on unix
(implementation-vicinity) is "/usr/local/src/scm/"
(library-vicinity) is "/usr/local/lib/slib/"
(scheme-file-suffix) is ".scm"
implementation *features* :
    bignum complex real rational
    inexact vicinity ed getenv
    tmpnam system 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 array-for-each
    array full-continuation char-ready? line-i/o
    i/o-extensions pipe
implementation *catalog* :
    (rev4-optional-procedures . "/usr/local/lib/slib/sc4opt")
...
```

6.12 Input/Output

These procedures are provided by all implementations.

file-exists? filename [Procedure]
 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* [Procedure]
 Deletes the file specified by *filename*. If *filename* can not be deleted, **#f** is returned. Otherwise, **#t** is returned.

tmpnam [Procedure]
 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 [Procedure]
 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 [Procedure]

output-port-width *port* [Procedure]
 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 [Procedure]

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

6.13 System

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)
```

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 separately 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.

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: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**.

Procedure and Macro Index

This is an alphabetical list of all the procedures and macros in SLIB.

—		array-map!	6
-	93	array-rank	5
-1+	92	array-ref	6
		array-set!	6
/		array-shape	5
/	93	array?	5
		asctime	95
<		ash	63
<=?	92	atom?	73
<?	92		
=			
=?	92		
>			
>=?	92		
>?	92		
1			
1+	91		
A			
add-domain	37		
add-process!	80		
add-setter	59		
adjoin	65		
adjoin-parameters!	19		
alist-associator	7		
alist-for-each	7		
alist-inquirer	7		
alist-map	7		
alist-remover	7		
and?	72		
any?	9		
append!	91		
apply	93		
array-1d-ref	6		
array-1d-set!	6		
array-2d-ref	6		
array-2d-set!	6		
array-3d-ref	6		
array-3d-set!	6		
array-dimensions	5		
array-for-each	6		
array-in-bounds?	6		
array-indexes	6		
		B	
		bit-extract	64
		break	98
		butlast	70
		C	
		call-with-dynamic-binding	11
		call-with-input-string	89
		call-with-output-string	89
		call-with-values	94
		capture-syntactic-environment	52
		cart-prod-tables	29
		chap:next-string	14
		chap:string<=?	14
		chap:string<?	14
		chap:string>=?	14
		chap:string>?	14
		check-parameters	19
		close-base	24
		close-database	29
		close-table	31
		collection?	8
		continue	98
		copy-list	64
		copy-tree	90
		create-database	28, 34
		create-report	38
		create-table	29
		create-view	29
		ctime	95
		current-error-port	106
		current-time	94

D

debug:tracef.....	98
debug:untracef.....	99
decode-universal-time.....	96
define-access-operation.....	60
define-operation.....	58
define-predicate.....	58
define-record.....	42
define-syntax.....	44
define-tables.....	38
defmacro.....	43
defmacro:eval.....	43
defmacro:expand*.....	43
defmacro:load.....	43
defmacro?.....	43
delete.....	25, 72
delete-domain.....	37
delete-file.....	106
delete-if.....	72
delete-if-not.....	72
delete-table.....	29
dequeue!.....	21
difftime.....	94
do-elts.....	8
do-keys.....	8
domain-checker.....	37
dynamic-ref.....	11
dynamic-set!.....	11
dynamic-wind.....	94
dynamic?.....	11

E

empty?.....	9
encode-universal-time.....	96
enqueue!.....	21
every.....	66
every?.....	9
extended-euclid.....	80

F

factor.....	83
file-exists?.....	105
fill-empty-parameters.....	19
find-if.....	67
fluit-let.....	57
for-each-elt.....	8
for-each-key.....	8, 25
for-each-row.....	31
force-output.....	106
format.....	73
fprintf.....	87
fscanf.....	87

G

generic-write.....	79
gentemp.....	43
get.....	30
get*.....	30
get-decoded-time.....	96
get-method.....	15
get-universal-time.....	96
getenv.....	102
getopt.....	99
getopt--.....	101
getopt->arglist.....	20
getopt->parameter-list.....	20
gmtime.....	95

H

has-duplicates?.....	68
hash.....	12
hash-associator.....	11
hash-for-each.....	12
hash-inquirer.....	11
hash-map.....	12
hash-remover.....	12
hashq.....	12
hashv.....	12
heap-extract-max!.....	20
heap-insert!.....	20
heap-length.....	20

I

identifier=?.....	55
identifier?.....	54
identity.....	106
implementation-vicinity.....	104
in-vicinity.....	104
init-debug.....	98
integer-expt.....	63
integer-length.....	63
intersection.....	65

J

jacobi-symbol.....	83
--------------------	----

K

kill-process!.....	80
kill-table.....	24

L

last	70
last-pair	92
library-vicinity	104
list*	64
list->string	92
list->vector	92
list-tail	92
localtime	95
logand	62
logbit?	62
logcount	63
logior	62
lognot	62
logtest	62
logxor	62

M

macro:eval	44, 45, 49, 56
macro:expand	44, 45, 49, 56
macro:load	44, 45, 49, 56
macroexpand	43
macroexpand-1	43
macwork:eval	45
macwork:expand	45
macwork:load	45
make-array	5
make-base	23
make-command-server	36
make-dynamic	10
make-generic-method	15
make-generic-predicate	15
make-getter	25
make-hash-table	11
make-heap	20
make-key->list	25
make-key-extractor	25
make-keyifier-1	24
make-list	64
make-list-keyifier	25
make-method!	15
make-object	15
make-parameter-list	19
make-predicate!	15
make-promise	93
make-putter	26
make-queue	21
make-random-state	84
make-rb-node	41
make-rb-tree	40
make-record-type	21
make-relational-system	28
make-shared-array	5
make-sierpinski-indexer	12
make-syntactic-closure	51
make-table	24
make-vicinity	104

map-elts	8
map-key	25
map-keys	8
member-if	66
merge	86
merge!	86
mktime	95
modular:*	80
modular:+	80
modular:-	80
modular:expt	80
modular:invert	80
modular:negate	80

N

nconc	71
notany	66
notevery	67
nreverse	71
nthcdr	70

O

object	58
object->string	81
object-with-ancestors	58
object?	15
offset-time	94
open-base	23
open-database	28, 34
open-database!	34
open-table	24, 29
operate-as	58
or?	72
ordered-for-each-key	25
output-port-height	106
output-port-width	106

P

parameter-list->arglist	20
parameter-list-expand	19
parameter-list-ref	19
plot!	81
position	68
pprint-file	82
pprint-filter-file	83
predicate->asso	7
predicate->hash	11
predicate->hash-asso	11
present?	25
pretty-print	82
prime:trials	83
prime?	83
print	58, 97
printf	87
process:schedule!	80

program-vicinity	104
project-table	29
provide	103
provided?	103

Q

qp	97
qpn	97
qpr	97
queue-empty?	21
queue-front	21
queue-pop!	21
queue-push!	21
queue-rear	21
queue?	21

R

random	83
random:exp	85
random:hollow-sphere!	84
random:normal	84
random:normal-vector!	84
random:solid-sphere!	84
random:uniform	84
rationalize	93
rb-delete!	41
rb-insert!	41
rb-node-maximum	41
rb-node-minimum	41
rb-node-predecessor	41
rb-node-successor	41
rb-tree-maximum	41
rb-tree-minimum	41
read-command	101
read-line	79
read-line!	79
record-accessor	22
record-constructor	22
record-modifier	22
record-predicate	22
record-type-descriptor	22
record-type-field-names	23
record-type-name	23
record?	22
reduce	8, 68
reduce-init	69
remove	67
remove-if	67
remove-if-not	68
remove-setter-for	59
repl:quit	97
repl:top-level	97
report-errs	99
require	103
require:feature->path	103
restrict-table	29

row:delete	30
row:delete*	30
row:insert	31
row:insert*	31
row:remove	30
row:remove*	30
row:retrieve	30
row:retrieve*	30
row:update	31
row:update*	31

S

scanf	87
set	59
set-difference	65
Setter	9
setter	59
size	9, 59
slib:error	107
slib:eval	107
slib:eval-load	107
slib:exit	107
slib:load	107
slib:load-compiled	106
slib:load-source	106
slib:report	105
slib:report-version	105
software-type	105
some	66
sort	86
sort!	87
sorted?	86
soundex	13
sprintf	87
sscanf	87
string->list	92
string-capitalize	89
string-captialize!	89
string-copy	92
string-downcase	89
string-downcase!	89
string-fill!	92
string-null?	91
string-upcase	89
string-upcase!	89
sub-vicinity	104
subst	90
substq	90
substring-fill!	91
substring-move-left!	91
substring-move-right!	91
substv	90
supported-key-type?	26
supported-type?	26
sync-base	24
syncase:eval	56
syncase:expand	56

syncase:load.....	56
synclo:eval.....	49
synclo:expand.....	49
synclo:load.....	49
syntax-rules.....	45
system.....	102

T

table-exists?.....	29
tek40:draw.....	89
tek40:graphics.....	89
tek40:init.....	89
tek40:linetype.....	89
tek40:move.....	89
tek40:put-text.....	89
tek40:reset.....	89
tek40:text.....	89
tek41:draw.....	90
tek41:encode-int.....	90
tek41:encode-x-y.....	90
tek41:graphics.....	90
tek41:init.....	90
tek41:move.....	90
tek41:point.....	90
tek41:reset.....	90
test.....	99
tmpnam.....	106
trace.....	98
trace-all.....	98
tracef.....	98

transcript-off.....	91
transcript-on.....	91
transformer.....	50
two-arg:-.....	93
two-arg:/.....	93
two-arg:apply.....	93
tzset.....	95

U

union.....	65
unmake-method!.....	15
untrace.....	98
untracef.....	99
user-vicinity.....	104

V

values.....	94
variant-case.....	42
vector->list.....	92
vector-fill!.....	92

W

with-input-from-file.....	91
with-output-to-file.....	91
write-base.....	24
write-database.....	29
write-line.....	79

Variable Index

This is an alphabetical list of all the global variables in SLIB.

*

catalog	103
features	102
modules	103
optarg	99
optind	99
qp-width	97
random-state	84
timezone	95

C

catalog-id	24
char-code-limit	104
charplot:height	81
charplot:width	81
column-domains	31
column-foreigns	31
column-names	31
column-types	31

M

most-positive-fixnum	104
----------------------	-----

N

nil	92
-----	----

P

primary-limit	31
---------------	----

S

slib:form-feed	105
slib:tab	104
stderr	87
stdin	87
stdout	87

T

t	92
---	----

Table of Contents

1	Overview	1
1.1	Installation	1
1.2	Porting	1
1.3	Coding Standards	2
1.4	Copyrights	3
1.5	Manual Conventions	4
2	Data Structures	5
2.1	Arrays	5
2.2	Array Mapping	6
2.3	Association Lists	6
2.4	Collections	7
2.5	Dynamic Data Type	10
2.6	Hash Tables	11
2.7	Hashing	12
2.8	Chapter Ordering	14
2.9	Macroless Object System	14
2.9.1	Concepts	14
2.9.2	Procedures	15
2.9.3	Examples	16
2.9.3.1	Inverter Documentation	17
2.9.3.2	Number Documentation	17
2.9.3.3	Inverter code	17
2.10	Parameter lists	19
2.11	Priority Queues	20
2.12	Queues	21
2.13	Records	21
2.14	Base Table	23
2.15	Relational Database	26
2.15.1	Motivations	26
2.15.2	Creating and Opening Relational Databases	28
2.15.3	Relational Database Operations	28
2.15.4	Table Operations	29
2.15.5	Catalog Representation	31
2.15.6	Unresolved Issues	33
2.15.7	Database Utilities	34
2.16	Red-Black Trees	40
2.17	Structures	42
3	Macros	43
3.1	Defmacro	43
3.1.1	Defmacroexpand	43

3.2	R4RS Macros	44
3.3	Macro by Example	44
3.3.1	Caveat	44
3.4	Macros That Work	45
3.4.1	Definitions	47
3.4.2	Restrictions	47
3.5	Syntactic Closures	48
3.5.1	Syntactic Closure Macro Facility	49
3.5.1.1	Terminology	49
3.5.1.2	Transformer Definition	50
3.5.1.3	Identifiers	54
3.5.1.4	Acknowledgements	55
3.6	Syntax-Case Macros	55
3.6.1	Notes	57
3.7	Fluid-Let	57
3.8	Yasos	57
3.8.1	Terms	58
3.8.2	Interface	58
3.8.3	Setters	59
3.8.4	Examples	60
4	Procedures	62
4.1	Bit-Twiddling	62
4.2	Common List Functions	64
4.2.1	List construction	64
4.2.2	Lists as sets	65
4.2.3	Lists as sequences	68
4.2.4	Destructive list operations	71
4.2.5	Non-List functions	72
4.3	Format	73
4.3.1	Format Interface	73
4.3.2	Format Specification (Format version 3.0)	73
4.3.2.1	Implemented CL Format Control Directives	74
4.3.2.2	Not Implemented CL Format Control Directives	77
4.3.2.3	Extended, Replaced and Additional Control Directives	77
4.3.2.4	Configuration Variables	77
4.3.2.5	Compatibility With Other Format Implementations	78
4.4	Generic-Write	78
4.5	Line I/O	79
4.6	Modular Arithmetic	80
4.7	Multi-Processing	80
4.8	Object-To-String	81
4.9	Plotting on Character Devices	81
4.10	Pretty-Print	82
4.11	Prime Factorization	83
4.12	Random Numbers	83

4.13	Sorting	85
4.14	Standard I/O	87
4.15	String-Case	89
4.16	String Ports	89
4.17	Tektronix Graphics Support	89
4.17.1	Tektronix 4000 Series Graphics	89
4.17.2	Tektronix 4100 Series Graphics	90
4.18	Tree operations	90
5	Standards Support	91
5.1	With-File	91
5.2	Transcripts	91
5.3	Rev2 Procedures	91
5.4	Rev4 Optional Procedures	92
5.5	Multi-argument / and -	92
5.6	Multi-argument Apply	93
5.7	Rationalize	93
5.8	Promises	93
5.9	Dynamic-Wind	93
5.10	Values	94
5.11	Time	94
5.12	CLTime	96
6	Session Support	97
6.1	Repl	97
6.2	Debug	97
6.2.1	Breakpoints	98
6.3	Tracing	98
6.4	Test	99
6.5	Getopt	99
6.6	Getopt-	101
6.7	Command Line	101
6.8	System Interface	102
6.9	Require	102
6.10	Vicinity	103
6.11	Configuration	104
6.12	Input/Output	105
6.13	System	106
	Procedure and Macro Index	108
	Variable Index	113