LispPad Library Reference

Version 2.0.1

2024-02-05

Matthias Zenger

Table of contents

1 Introduction

1.1 Overview

LispPad is an integrated development environment for Scheme on macOS. LispPad's Scheme interpreter is based on LispKit, a R7RS-compliant implementation of Scheme which comes with a large number of pre-packaged Scheme libraries. There is also a version of LispPad for iOS, called LispPad Go, which includes almost the same set of libraries.

This document is a reference manual for the core Scheme libraries coming with LispKit and LispPad. The LispPad homepage provides access to frequently updated online documentation.

1.2 Further reading

There are several books which can be recommended for learning Scheme and related topics:

- "The Scheme Programming Language" by R. Kent Dybvig provides a comprehensive introduction into Scheme based on R6RS. It discusses several advanced topics and covers many Scheme libraries.
- "Simply Scheme: Introducing Computer Science" by Brian Harvey and Matthew Wright introduces Scheme slowly to beginners.
- "Structure and Interpretation of Computer Programs" by Harold Abelson and Gerald Jay Syssman is the ultimate book teaching Computer Science, all in Scheme. The book covers a broad range of Computer Science topics and should be standing on every Scheme programmers desk.
- "Essentials of Programming Languages" by Daniel P. Friedman and Mitchell Wand provides a deep understanding of the essential concepts of programming languages and uses Scheme as the language to implement the concepts.

1.3 Acknowledgments

Some of this documentation is derived from existing Scheme language specifications, such as the R5RS, the R6RS, and the R7RS standards. In recent years, these standards evolved using the SRFI process, which provides access to a large number of standardized Scheme components and libraries.

The following people have contributed over the last 20 years to the evolution, standardization, and documentation of Scheme: R. Kelsey, W. Clinger, J. Rees, H. Abelson, N. I. Adams IV, D. H. Bartley, G. Brooks, W. Corcoran-Mathe, R. K. Dybvig, D. P. Friedman, R. Halstead, C. Hanson, C. T. Haynes, E. Kohlbecker, D. Oxley, M. Nieper-Wißkirchen, K. M. Pitman, M. Sperber, M. Flatt, A. v. Straaten, A. Shinn, J. Cowan, A. A. Gleckler, S. Ganz, A. W. Hsu, B. Lucier, E. Medernach, A. Radul, J. T. Read, D. Rush, B. L. Russel, O. Shivers, A. Snell-Pym, and G. J. Sussman.

2 LispKit Archive Tar

Library `(lispkit archive tar)` provides an API for creating and managing tar archives. `(lispkit archive tar)` manages tar archives fully in memory, i.e. they are created and mutated in memory and can be saved and loaded from disk. A tar archive is made up of file objects, also called *tar entries*, which consist of the actual file data, compressed or uncompressed, as well as metadata, such as creation and modification date and file permissions.

2.1 Constructors

(make-tar-archive) procedure
(make-tar-archive *bytevector*)
(make-tar-archive *bytevector start*)
(make-tar-archive *bytevector start end*)

Returns a new tar archive. If no argument is provided, a new, empty tar archive object is created. Otherwise, a tar archive is created from the binary data provided by *bytevector* between indices *start* and *end*.

If is an error if *bytevector*, between *start* and *end*, is not encoded using a supported tar encoding. At this point, the basic tar format and POSIX encodings *ustar* and *pax* are supported. If *end* is not provided, it is assumed to be the length of *bytevector*. If *start* is not provided, it is assumed to be 0.

2.2 Reading and writing archives

(load-tar-archive *filepath*) procedure
Loads the tar file at path *filepath* and stores its content in a new tar archive object in memory. This new object is returned by `load-tar-archive`. It is an error if the file at the given file path is not in a supported tar file format. Supported tar file formats are the basic tar format as well as the two POSIX encodings *ustar* and *pax*.

(save-tar-archive *archive*) procedure
(save-tar-archive *archive filepath*)

Saves the given tar *archive* at path *filepath* using the *pax* encoding. If the archive was previously loaded from disk, then it is possible to omit parameter *filepath* and overwrite the previously loaded archive at the same location on disk.

2.3 Properties of archives

tar-archive-type-tag object
Symbol representing the `tar-archive` type. The `type-for` procedure of library `(lispkit type)` returns this symbol for all tar archive objects.

(tar-archive? *obj*) procedure

Returns `#t` if *obj* is a tar archive object; otherwise `#f` is returned.

(tar-archive-path *archive*) procedure

Returns the file path of *archive* or `#f` if *archive* was not loaded from disk, i.e. it was created via `make-tar-archive`.

(tar-archive-bytevector *archive*) procedure

Procedure `tar-archive-bytevector` returns *archive* encoded using the *pax* format into a bytevector. This bytevector can be written to disk or used to create an in-memory copy of the tar archive via `make-tar-archive`.

(tar-entry-count *archive*) procedure

Returns the number of entries in tar *archive*.

(tar-entries *archive*) procedure
(tar-entries *archive prefix*)

Returns a list of file paths for all entries of tar *archive* which have *prefix* as their file path prefix. If *prefix* is not given, the file paths of all entries are being returned as a list.

2.4 Introspecting entries

Entries in zip archives are referred to via their relative file path in the archive. All procedures that provide information about a zip archive entry therefore expect two arguments: the zip archive and a file path.

(tar-entry-exists? *archive path*) procedure
(tar-entry-exists? *archive path prefix?*)

Returns `#t` if *archive* contains an entry for *path* (string) if *prefix?* is `#f`, or *archive* contains an entry whose path is a prefix of *path* if *prefix?* is `#t`.

(tar-entry-file? *archive path*) procedure

Returns `#t` if *archive* contains an entry for the given path and this entry is a file. If *path* does not refer to a valid entry in *archive*, the procedure `tar-entry-file?` fails with an error.

(tar-entry-directory? *archive path*) procedure

Returns `#t` if *archive* contains an entry for the given path and this entry is a directory. If *path* does not refer to a valid entry in *archive*, the procedure `tar-entry-directory?` fails with an error.

(tar-entry-symlink? *archive path*) procedure

Returns `#t` if *archive* contains an entry for the given path and this entry is a symbolic link. If *path* does not refer to a valid entry in *archive*, the procedure `tar-entry-symlink?` fails with an error.

(tar-entry-linked *archive path*) procedure

If *archive* contains an entry for the given *path* and this entry is a symbolic link, then procedure `tar-entry-linked` returns the path of the linked file as a string, otherwise `#f` is returned. If *path* does not refer to a valid entry in *archive*, the procedure `tar-entry-linked` fails with an error.

(tar-entry-creation-date *archive path*) procedure

Returns the creation date of the entry to which *path* refers to in *archive*. If there is no entry in *archive* for the given *path*, `#f` gets returned.

(tar-entry-modification-date *archive path*) procedure

Returns the modification date of the entry to which *path* refers to in *archive*. If there is no entry in *archive* for the given *path*, `#f` gets returned.

(tar-entry-permissions *archive path*) procedure

Returns the access permissions of the entry to which *path* refers to in *archive* as a fixnum (Unix style). If there is no entry in *archive* for the given *path*, `#f` gets returned.

Permission numbers can be created by selecting from the following attributes and summing up the values: *Read by owner* (400), *Write by owner* (200), *Execute by owner* (100), *Read by group* (040), *Write by group* (020), *Execute by group* (010), *Read by others* (004), *Write by others* (002), *Execute by others* (001).

(tar-entry-size *archive path*) procedure

Returns the size of the entry in bytes to which *path* refers to in *archive*. If there is no entry in *archive* for the given *path*, `#f` gets returned.

(tar-entry-data *archive path*) procedure

Returns the data of the entry to which *path* refers to in *archive* as a bytevector. If there is no entry in *archive* for the given *path*, `#f` gets returned.

2.5 Adding and removing entries

(add-tar-entry! *archive path base*) procedure
(add-tar-entry! *archive path base recursive?*)

Adds the file, directory, or symbolic link at *path* relative to path *base* to the given tar *archive*. The corresponding entry in *archive* is identified via *path*. If the entry is a directory and parameter *recursive?* is true, all entries in the directory are added to *archive* as well.

(set-tar-entry! *archive path content*) procedure
(set-tar-entry! *archive path content mdate*)
(set-tar-entry! *archive path content mdate permissions*)

Adds or overwrites the entry in *archive* at *path*. Parameter *content* determines whether the new entry is a file, directory, or symbolic link. If *content* is a bytevector, the new entry is a file containing the data of the bytevector. If *content* is a string, it represents a path of a symbolic link. For `()`, the new entry is a directory.

The creation date of the new entry is the current date, the modification date is *mdate* or the current date if *mdate* is not provided. The permissions are *permissions* or an entry type specific default (`644` for files, and `755` otherwise).

(delete-tar-entry! *archive path*) procedure
(delete-tar-entry! *archive path prefix?*)

Deletes the entry at *path* from *archive*. If argument *prefix?* is true, *path* is interpreted as a prefix and all entries with prefix *path* are deleted.

2.6 Extracting entries

(extract-tar-entry *archive path base*) procedure
(extract-tar-entry *archive path base prefix?*)

Extracts the entry at *path* in *archive* and stores it on the file system at *path* relative to path *base*. If *prefix?* is true, all entries with prefix *path* are extracted.

(get-tar-entry *archive path*) procedure

Returns a representation of the entry at *path* in *archive*. If the entry is a file, `get-tar-entry` returns the content of this file as a bytevector. If the entry is a symbolic link, then the target of the link is returned as a string. If the entry is a directory, `()` is returned. The procedure fails if the entry does not exist.

3 LispKit Archive Zip

Library `(lispkit archive zip)` provides an API for creating and managing zip archives. Zip archives are either persisted on the file system, or they are created in-memory. Zip archives can be opened either in read-only or read-write mode. They allow either files or in-memory data (in the form of bytevectors) to be included. Such *zip entries* are either a file, a directory, or a symbolic link. In an archive, files are stored in either compressed or uncompressed form.

3.1 Constructors

(make-zip-archive) procedure
(make-zip-archive *bvec*)
(make-zip-archive *bvec mutable?*)

Procedure `make-zip-archive` creates an in-memory zip archive. If bytevector *bvec* is provided, the zip archive is created from the given binary data, otherwise, a new empty zip archive is returned. For a zip archive created from a bytevector, parameter *mutable?* determines if it is a read-only or read-write zip archive. In the latter case, *mutable?* has to be set to `#t`, the default is `#f`.

(create-zip-archive *path*) procedure
Creates a new empty read-write zip archive at the given file path.

(open-zip-archive *path*) procedure
(open-zip-archive *path mutable?*)

Opens a zip archive at the given file path. By default, the zip archive is opened in read-only mode, unless *mutable?* is set to `#t`.

3.2 Properties of archives

zip-archive-type-tag object
Symbol representing the `zip-archive` type. The `type-for` procedure of library `(lispkit type)` returns this symbol for all zip archive objects.

(zip-archive? *obj*) procedure
Returns `#t` if *obj* refers to a zip archive, otherwise `#f` is being returned.

(zip-archive-mutable? *archive*) procedure
Returns `#t` if the given zip archive is mutable, i.e. opened in read-write mode, `#f` otherwise.

(zip-archive-path *archive*) procedure
Procedure `zip-archive-path` returns the file path at which *archive* is being persisted. If *archive* is a in-memory zip archive, then `#f` is returned.

(zip-archive-bytevector *archive*) procedure
Procedure `zip-archive-bytevector` returns *archive* as a bytevector. This bytevector can be written to disk or used to create a in-memory copy of the zip archive.

3.3 Introspecting entries

Entries in zip archives are referred to via their relative file path in the archive. All procedures that provide information about a zip archive entry therefore expect two arguments: the zip archive and a file path.

(zip-entry-count *archive*) procedure
Returns the number of entries in *archive*.

(zip-entries *archive*) procedure
Returns a list of file paths for all entries of zip archive *archive*.

(zip-entry-exists? *archive path*) procedure
Returns `#t` if *archive* contains an entry with the given file path.

(zip-entry-compressed? *archive path*) procedure
Returns `#t` if *archive* contains an entry for the given file path and this entry is stored in compressed form. If *path* does not refer to a valid entry in *archive*, the procedure `zip-entry-compressed?` fails with an error.

(zip-entry-file? *archive path*) procedure
Returns `#t` if *archive* contains an entry for the given file path and this entry is a file. If *path* does not refer to a valid entry in *archive*, the procedure `zip-entry-file?` fails with an error.

(zip-entry-directory? *archive path*) procedure
Returns `#t` if *archive* contains an entry for the given file path and this entry is a directory. If *path* does not refer to a valid entry in *archive*, the procedure `zip-entry-directory?` fails with an error.

(zip-entry-symlink? *archive path*) procedure
Returns `#t` if *archive* contains an entry for the given file path and this entry is a symbolic link. If *path* does not refer to a valid entry in *archive*, the procedure `zip-entry-directory?` fails with an error.

(zip-entry-compressed-size *archive path*) procedure
Returns the size of the compressed file for the entry at the given path in bytes. If *path* does not refer to a valid entry in *archive*, the procedure `zip-entry-compressed-size` fails with an error.

(zip-entry-uncompressed-size *archive path*) procedure
Returns the size of the uncompressed file for the entry at the given path in bytes. If *path* does not refer to a valid entry in *archive*, the procedure `zip-entry-uncompressed-size` fails with an error.

3.4 Adding and removing entries

(add-zip-entry *archive path base*) procedure
(add-zip-entry *archive path base compressed?*)

Adds the file, directory, or symbolic link at *path* relative to path *base* to the given zip *archive*. The corresponding entry in *archive* is identified via *path*. The file is stored in uncompressed form if *compressed?* is set to `#f`. The default for *compressed?* is `#t`.

(write-zip-entry *archive path bvec*) procedure
(write-zip-entry *archive path bvec compressed?*)
(write-zip-entry *archive path bvec compressed? time*)

Adds a new file entry to *archive* at *path* based on the content of bytevector *bvec*. The entry is stored in uncompressed form if *compressed?* is set to `#f`. The default for *compressed?* is `#t`. *time* is a `date-time` object as defined by library `(lispkit date-time)` which defines the modification time of the new entry.

(delete-zip-entry *archive path*) procedure

Deletes the entry at *path* from *archive*. Procedure `delete-zip-entry` fails if the entry does not exist or if the archive is opened in read-only mode.

3.5 Extracting entries

(extract-zip-entry *archive path base*) procedure

Extracts the entry at *path* in *archive* and stores it on the file system at *path* relative to path *base*.

(read-zip-entry *archive path*) procedure

Returns the file entry at *path* in *archive* in form of a bytevector. Procedure `read-zip-entry` fails if the entry does not exist or if the entry is not a file entry.

4 LispKit Base

Library `(lispkit base)` aggregates all exported values, parameter objects, and functions from the following libraries and re-exports them.

- `(lispkit box)`
- `(lispkit bytevector)`
- `(lispkit char)`
- `(lispkit control)`
- `(lispkit core)`
- `(lispkit dynamic)`
- `(lispkit hashtable)`
- `(lispkit list)`
- `(lispkit math)`
- `(lispkit port)`
- `(lispkit record)`
- `(lispkit string)`
- `(lispkit system)`
- `(lispkit type)`
- `(lispkit vector)`

5 LispKit Bitset

Library `(lispkit bitset)` implements bit sets of arbitrary size. Bit sets are mutable objects. The API of library `(lispkit bitset)` provides functionality to create, to inspect, to compose, and to mutate bit sets efficiently.

(bitset? *obj*) procedure

Returns `#t` if *obj* is a bit set, `#f` otherwise.

(bitset *i* ...) procedure

Returns a new bit set with bits *i* ... set. Each *i* is a fixnum referring to one bit in the bit set by its ordinality.

(list->bitset *list*) procedure

Returns a new bit set with bits specified by *list*. Each element in *list* is a fixnum referring to one bit in the bit set by its ordinality.

(fixnum->bitset *x*) procedure

Returns a bit set with all bits set that are set in fixnum *x*.

(bitset-copy *bs*) procedure

Returns a copy of bit set *bs*.

(bitset-size *bs*) procedure

Returns the number of bits set in bit set *bs*.

(bitset-next *bs*) procedure
(bitset-next *bs i*)

Returns the next bit set in *bs* following bit *i*. If *i* is not provided, the first bit set in *bs* is returned.

(bitset-empty? *bs*) procedure

Returns `#t` if bit set *bs* is empty, `#f` otherwise.

(bitset-disjoint? *bs1 bs2*) procedure

Returns `#t` if bit sets *bs1* and *bs2* are disjoint, `#f` otherwise.

(bitset-subset? *bs1 bs2*) procedure

Returns `#t` if bit set *bs2* is a subset of bit set *bs2*, `#f` otherwise.

(bitset-contains? *bs i* ...) procedure

Returns `#t` if all bits *i* ... are set in bit set *bs*, `#f` otherwise.

(bitset-adjoin! *bs i* ...) procedure

Inserts the bits *i* ... into bit set *bs*. `bitset-adjoin!` returns *bs*.

(bitset-adjoin-all! *bs list*) procedure

Inserts all the bits specified by *list* into bit set *bs*. Each element in *list* is a fixnum referring to one bit in the bit set by its ordinality. `bitset-adjoin-all!` returns *bs*.

(bitset-delete! *bs i* ...) procedure

Removes the bits *i* ... from bit set *bs*. `bitset-delete!` returns *bs*.

(bitset-delete-all! *bs list*) procedure

Removes all the bits specified by *list* from bit set *bs*. Each element in *list* is a fixnum referring to one bit in the bit set by its ordinality. `bitset-delete-all!` returns *bs*.

(bitset-union! *bs bs1 ...*) procedure

Computes the union of bit sets *bs*, *bs1* ... and stores the result in *bs*. `bitset-union!` returns *bs*.

(bitset-intersection! *bs bs1 ...*) procedure

Computes the intersection of bit sets *bs*, *bs1* ... and stores the result in *bs*. `bitset-intersection!` returns *bs*.

(bitset-difference! *bs bs1 ...*) procedure

Computes the difference between bit sets *bs* and *bs1* ... and stores the result in *bs*. `bitset-difference!` returns *bs*.

(bitset-xor! *bs bs1 ...*) procedure

Computes the exclusive disjunction of bit sets *bs*, *bs1* ... and stores the result in *bs*. `bitset-xor!` returns *bs*.

(bitset=? *bs bs1 ...*) procedure

Returns `#t` if the bit sets *bs*, *bs1* ... are all equal, i.e. have the same bits set. Otherwise `#f` is returned.

(bitset<? *bs bs1 ...*) procedure

Returns `#t` if *bs* is a proper subset of *bs1*, and *bs1* is a proper subset of *bs2*, etc. Otherwise `#f` is returned.

(bitset>? *bs bs1 ...*) procedure

Returns `#t` if *bs* is a proper superset of *bs1*, and *bs1* is a proper superset of *bs2*, etc. Otherwise `#f` is returned.

(bitset<=? *bs bs1 ...*) procedure

Returns `#t` if *bs* is a subset of *bs1*, and *bs1* is a subset of *bs2*, etc. Otherwise `#f` is returned.

(bitset>=? *bs bs1 ...*) procedure

Returns `#t` if *bs* is a superset of *bs1*, and *bs1* is a superset of *bs2*, etc. Otherwise `#f` is returned.

(bitset->list *bs*) procedure

Returns a list of all bits set in *bs*.

(bitset->fixnum *bs*) procedure

Returns a fixnum with all bits set which are also set in bit set *bs*. If *bs* includes bits that cannot be represented by a fixnum, then `bitset->fixnum` returns `#f`.

(bitset-for-each *proc bs*) procedure

Invokes *proc* on each bit set in *bs* in increasing ordinal order.

(bitset-fold *proc z bs*) procedure

The current state is initialized to z, and *proc* is invoked on each bit of *bs* in increasing ordinal order and the current state, setting the current state to the result. The algorithm is repeated until all the bits of *bs* have been processed. Then the current state is returned.

(bitset-any? *pred bs*) procedure

Returns `#t` if any application of *pred* to the bits of *bs* returns true, and `#f` otherwise.

(bitset-every? *pred bs*) procedure

Returns `#t` if every application of *pred* to the bits of *bs* returns true, and `#f` otherwise.

(bitset-filter *pred bs*) procedure

Returns a new bit set containing the bits from *bs* that satisfy *pred*.

(bitset-filter! *pred bs*) procedure

Removes all bits from *bs* for which *pred* returns `#f`.

6 LispKit Box

LispKit is a R7RS-compliant implementation with one exception: pairs are immutable. This library provides implementations of basic mutable data structures with reference semantics: mutable multi-place buffers, also called *boxes*, and mutable pairs. The difference between a two-place box and a mutable pair is that a mutable pair allows mutations of the two elements independent of each other.

6.1 Boxes

(box? *obj*) procedure

Returns `#t` if *obj* is a box; `#f` otherwise.

(box *obj* ...) procedure

Returns a new box object that contains the objects *obj* ....

(unbox *box*) procedure

Returns the current contents of *box*. If multiple values have been stored in the box, `unbox` will return multiple values. This procedure fails if *box* is not referring to a box.

(set-box! *box obj* ...) procedure

Sets the content of *box* to objects *obj* .... This procedure fails if *box* is not referring to a box.

(update-box! *box proc*) procedure

Invokes *proc* with the content of *box* and stores the result of this function invocation in *box*. `update-box!` is implemented like this:

```
(define (update-box! box proc)
  (set-box! box (apply-with-values proc (unbox box))))
```

6.2 Mutable pairs

(mpair? *obj*) procedure

Returns `#t` if v is a mutable pair (mpair); `#f` otherwise.

(mcons *car cdr*) procedure

Returns a new mutable pair whose first element is set to *car* and whose second element is set to *cdr*.

(mcar *mpair*) procedure

Returns the first element of the mutable pair *mpair*.

(mcdr *mpair*) procedure

Returns the second element of the mutable pair *mpair*.

(set-mcar! *mpair obj*) procedure

Sets the first element of the mutable pair *mpair* to *obj*.

(set-mcdr! *mpair obj*) procedure

Sets the second element of the mutable pair *mpair* to *obj*.

7 LispKit Bytevector

Bytevectors represent blocks of binary data. They are fixed-length sequences of bytes, where a *byte* is a fixnum in the range from 0 to 255 inclusive. A bytevector is typically more space-efficient than a vector containing the same values.

The *length* of a bytevector is the number of elements that it contains. The length is a non-negative integer that is fixed when the bytevector is created. The *valid indexes* of a bytevector are the exact non-negative integers less than the length of the bytevector, starting at index zero as with vectors.

Bytevectors are written using the notation `#u8(byte ...)` . For example, a bytevector of length 3 containing the byte 0 in element 0, the byte 10 in element 1, and the byte 5 in element 2 can be written as follows: `#u8(0 10 5)`. Bytevector constants are self-evaluating, so they do not need to be quoted.

7.1 Basic

(bytevector? *obj*) procedure

Returns `#t` if *obj* is a bytevector; otherwise, `#f` is returned.

(bytevector *byte* ...) procedure

Returns a newly allocated bytevector containing its arguments as bytes in the given order.

```
(bytevector 1 3 5 1 3 5)  ⇒  #u8(1 3 5 1 3 5)
(bytevector)              ⇒  #u8()
```

(make-bytevector *k*) procedure
(make-bytevector *k byte*)

The `make-bytevector` procedure returns a newly allocated bytevector of length *k*. If *byte* is given, then all elements of the bytevector are initialized to *byte*, otherwise the contents of each element are unspecified.

```
(make-bytevector 3 12)  ⇒  #u8(12 12 12)
```

(bytevector=? *bytevector* ...) procedure

Returns `#t` if all *bytevector* ... contain the same sequence of bytes, otherwise `#f` is returned.

(bytevector-length *bytevector*) procedure

Returns the length of bytevector in bytes as an exact integer.

(bytevector-u8-ref *bytevector k*) procedure

Returns the *k*-th byte of *bytevector*. It is an error if *k* is not a valid index of bytevector.

```
(bytevector-u8-ref #u8(1 1 2 3 5 8 13 21) 5)  ⇒  8
```

(bytevector-u8-set! *bytevector k byte*) procedure

Stores *byte* as the *k*-th byte of bytevector. It is an error if *k* is not a valid index of bytevector.

```
(let ((bv (bytevector 1 2 3 4)))
  (bytevector-u8-set! bv 1 3)
  bv)
⇒  #u8(1 3 3 4)
```

(bytevector-copy *bytevector*) procedure
(bytevector-copy *bytevector start*)
(bytevector-copy *bytevector start end*)

Returns a newly allocated bytevector containing the bytes in *bytevector* between *start* and *end*. If *end* is not provided, it is assumed to be the length of *bytevector*. If *start* is not provided, it is assumed to be 0.

```
(define a #u8(1 2 3 4 5))
(bytevector-copy a 2 4))   ⇒  #u8(3 4)
```

(bytevector-copy! *to at from*) procedure
(bytevector-copy! *to at from start*)
(bytevector-copy! *to at from start end*)

Copies the bytes of bytevector *from* between *start* and *end* to bytevector *to*, starting at *at*. The order in which bytes are copied is unspecified, except that if the source and destination overlap, copying takes place as if the source is first copied into a temporary bytevector and then into the destination. This can be achieved without allocating storage by making sure to copy in the correct direction in such circumstances.

It is an error if *at* is less than zero or greater than the length of *to*. It is also an error if `(- (bytevector-length to) at)` is less than `(- end start)`.

```
(define a (bytevector 1 2 3 4 5))
(define b (bytevector 10 20 30 40 50))
(bytevector-copy! b 1 a 0 2)
b  ⇒  #u8(10 1 2 40 50)
```

(bytevector-append *bytevector* ...) procedure
Returns a newly allocated bytevector whose elements are the concatenation of the elements in the given bytevectors.

```
(bytevector-append #u8(0 1 2) #u8(3 4 5))
  ⇒  #u8(0 1 2 3 4 5)
```

7.2 Input/Output

(read-binary-file *path*) procedure
Reads the file at *path* and stores its content in a new bytevector which gets returned by `read-binary-file`.

(write-binary-file *path bytevector*) procedure
(write-binary-file *path bytevector start*)
(write-binary-file *path bytevector start end*)

Writes the bytes of *bytevector* between *start* and *end* into a new binary file at *path*. If *end* is not provided, it is assumed to be the length of *bytevector*. If *start* is not provided, it is assumed to be 0.

7.3 Compression

(bytevector-deflate *bytevector*) procedure
(bytevector-deflate *bytevector start*)
(bytevector-deflate *bytevector start end*)

`bytevector-deflate` encodes *bytevector* between *start* and *end* using the *Deflate* data compression alogrithm returning a new compressed bytevector. If *end* is not provided, it is assumed to be the length of *bytevector*. If *start* is not provided, it is assumed to be 0.

(bytevector-inflate *bytevector*) procedure
(bytevector-inflate *bytevector start*)
(bytevector-inflate *bytevector start end*)

`bytevector-inflate` assumes *bytevector* is encoded using the *Deflate* data compression alogrithm between *start* and *end*. The procedure returns a corresponding new decoded bytevector.

If is an error if *bytevector*, between *start* and *end*, is not encoded using *Deflate*. If *end* is not provided, it is assumed to be the length of *bytevector*. If *start* is not provided, it is assumed to be 0.

(bytevector-zip *bytevector*) procedure
(bytevector-zip *bytevector start*)
(bytevector-zip *bytevector start end*)

`bytevector-zip` encodes *bytevector* between *start* and *end* using the *Deflate* data compression alogrithm returning a new compressed bytevector which is using a *zlib* wrapper. If *end* is not provided, it is assumed to be the length of *bytevector*. If *start* is not provided, it is assumed to be 0.

(bytevector-unzip *bytevector*) procedure
(bytevector-unzip *bytevector start*)
(bytevector-unzip *bytevector start end*)

`bytevector-unzip` assumes *bytevector* is using a *zlib* wrapper for data encoded using the *Deflate* data compression alogrithm between *start* and *end*. The procedure returns a corresponding new decoded bytevector.

If is an error if *bytevector*, between *start* and *end*, is not encoded using *Deflate* or is not using the *zlib* wrapper format. If *end* is not provided, it is assumed to be the length of *bytevector*. If *start* is not provided, it is assumed to be 0.

(bytevector-gzip *bytevector*) procedure
(bytevector-gzip *bytevector start*)
(bytevector-gzip *bytevector start end*)

`bytevector-gzip` encodes *bytevector* between *start* and *end* using the *Deflate* data compression alogrithm returning a new compressed bytevector which is using a *gzip* wrapper. If *end* is not provided, it is assumed to be the length of *bytevector*. If *start* is not provided, it is assumed to be 0.

(bytevector-gunzip *bytevector*) procedure
(bytevector-gunzip *bytevector start*)
(bytevector-gunzip *bytevector start end*)

`bytevector-gunzip` assumes *bytevector* is using a *gzip* wrapper for data encoded using the *Deflate* data compression alogrithm between *start* and *end*. The procedure returns a corresponding new decoded bytevector.

If is an error if *bytevector*, between *start* and *end*, is not encoded using *Deflate* or is not using the *gzip* wrapper format. If *end* is not provided, it is assumed to be the length of *bytevector*. If *start* is not provided, it is assumed to be 0.

7.4 Advanced

(utf8->string *bytevector*) procedure
(utf8->string *bytevector start*)
(utf8->string *bytevector start end*)
(string->utf8 *string*)
(string->utf8 *string start*)
(string->utf8 *string start end*)

These procedures translate between strings and bytevectors that encode those strings using the UTF-8 encoding. The `utf8->string` procedure decodes the bytes of a *bytevector* between *start* and *end* and returns the corresponding string. The `string->utf8` procedure encodes the characters of a *string* between *start* and *end* and returns the corresponding bytevector.

It is an error for *bytevector* to contain invalid UTF-8 byte sequences.

```
(utf8->string #u8(#x41))  ⇒  "A"
(string->utf8 "λ")        ⇒  #u8(#xCE #xBB)
```

(bytevector->base64 *bytevector*) procedure
(bytevector->base64 *bytevector start*)
(bytevector->base64 *bytevector start end*)

`bytevector->base64` encodes *bytevector* between *start* and *end* as a string consisting of ASCII characters using the *Base64* encoding scheme. If *end* is not provided, it is assumed to be the length of *bytevector*. If *start* is not provided, it is assumed to be 0.

(base64->bytevector *str*) procedure
(base64->bytevector *str start*)
(base64->bytevector *str start end*)

`base64->bytevector` assumes string *str* is encoded using *Base64* between *start* and *end* and returns a corresponding new decoded bytevector.

If is an error if *str* between *start* and *end* is not a valid *Base64*-encoded string. If *end* is not provided, it is assumed to be the length of *str*. If *start* is not provided, it is assumed to be 0.

(bytevector->hex *bytevector*) procedure
(bytevector->hex *bytevector start*)
(bytevector->hex *bytevector start end*)

Returns a string representation of *bytevector* in which every byte between *start* and *end* is represented by two characters denoting the value of the byte in hexadecimal form. The characters representing the individual bytes are concatenated such that a bytevector is represented by a hex string of length *end* - *start*. If *end* is not provided, it is assumed to be the length of *bytevector*. If *start* is not provided, it is assumed to be 0.

```
(bytevector->hex #u8(7 8 9 10 11 12)) ⇒ "0708090a0b0c"
```

(hex->bytevector *str*) procedure
(hex->bytevector *str start*)
(hex->bytevector *str start end*)

Returns a bytevector for a given hex string between *start* and *end*. Such strings encode every byte with two characters representing the value of the byte in hexadecimal form.

If is an error if *str* between *start* and *end* is not a valid hex string. If *end* is not provided, it is assumed to be the length of *str*. If *start* is not provided, it is assumed to be 0.

```
(hex->bytevector "1718090a0b0c") ⇒ #u8(23 24 9 10 11 12)
```

(bytevector-adler32 ***bytevector*****)** procedure
(bytevector-adler32 ***bytevector start*****)**
(bytevector-adler32 ***bytevector start end*****)**

`bytevector-adler32` computes the Adler32 checksum for *bytevector* between *start* and *end*. If *end* is not provided, it is assumed to be the length of *bytevector*. If *start* is not provided, it is assumed to be 0.

(bytevector-crc32 ***bytevector*****)** procedure
(bytevector-crc32 ***bytevector start*****)**
(bytevector-crc32 ***bytevector start end*****)**

`bytevector-crc32` computes the CRC32 checksum for *bytevector* between *start* and *end*. If *end* is not provided, it is assumed to be the length of *bytevector*. If *start* is not provided, it is assumed to be 0.

8 LispKit Char

Characters are objects that represent printed characters such as letters and digits. In LispKit, characters are UTF-16 code units.

Character literals are written using the notation `#\` *character*, or `#\` *character-name*, or `#\x` *hex-scalar-value*. Characters written using this `#\` notation are self-evaluating, i.e. they do not have to be quoted.

The following standard character names are supported by LispKit:

- `#\alarm` : U+0007
- `#\backspace` : U+0008
- `#\delete` : U+007F
- `#\escape` : U+001B
- `#\newline` : the linefeed character U+000A
- `#\null` : the null character U+0000
- `#\return` : the return character U+000D
- `#\space` : the space character U+0020
- `#\tab` : the tab character U+0009

Here are some examples using the `#\` notation:

- `#\m` : lowercase letter 'm'
- `#\M` : uppercase letter 'M'
- `#\(` : left parenthesis ')'
- `#\\` : backslash '\'
- `#\ ` : space character ' '
- `#\x03BB` : the lambda character

Case is significant in `#\` *character*, and in `#\` *character-name*, but not in `#\x` *hex-scalar-value*. If *character* in `#\` *character* is alphabetic, then any character immediately following *character* cannot be one that can appear in an identifier. This rule resolves the ambiguous case where, for example, the sequence of characters `#\space` could be taken to be either a representation of the space character or a representation of the character `#\s` followed by a representation of the symbol `pace`.

Some of the procedures that operate on characters ignore the difference between upper case and lower case. The procedures that ignore case have "-ci" (for "case insensitive") embedded in their names.

8.1 Predicates

(char? *obj*) procedure

Returns `#t` if *obj* is a character, otherwise returns `#f`.

(char=? *char* ...) procedure
(char<? *char* ...)
(char>? *char* ...)
(char<=? *char* ...)
(char>=? *char* ...)

These procedures return `#t` if the results of passing their arguments to `char->integer` are respectively equal, monotonically increasing, monotonically decreasing, monotonically non-decreasing, or monotonically non-increasing. These predicates are transitive.

(char-ci=? *char* ...) procedure
(char-ci<? *char* ...)
(char-ci>? *char* ...)
(char-ci<=? *char* ...)
(char-ci>=? *char* ...)

These procedures are similar to `char=?` etc., but they treat upper case and lower case letters as the same. For example, `(char-ci=? #\A #\a)` returns `#t`. Specifically, these procedures behave as if `char-foldcase` were applied to their arguments before they were compared.

(char-alphabetic? *char*) procedure
Procedure `char-alphabetic?` returns `#t` if its argument is a alphabetic character, otherwise it returns `#f`. Note that many Unicode characters are alphabetic but neither upper nor lower case.

(char-numeric? *char*) procedure
Procedure `char-numeric?` returns `#t` if its argument is a numeric character, otherwise it returns `#f` .

(char-whitespace? *char*) procedure
Procedure `char-whitespace?` returns `#t` if its argument is a whitespace character, otherwise it returns `#f`.

(char-upper-case? *char*) procedure
Procedure `char-upper-case??` returns `#t` if its argument is an upper-case character, otherwise it returns `#f`.

(char-lower-case? *char*) procedure
Procedure `char-lower-case?` returns `#t` if its argument is a lower-case character, otherwise it returns `#f`.

8.2 Transforming characters

(char-upcase *char*) procedure
The `char-upcase` procedure, given an argument that is the lowercase part of a Unicode casing pair, returns the uppercase member of the pair, provided that both characters are supported by LispKit. Note that language-sensitive casing pairs are not used. If the argument is not the lowercase member of such a pair, it is returned.

(char-downcase *char*) procedure
The `char-downcase` procedure, given an argument that is the uppercase part of a Unicode casing pair, returns the lowercase member of the pair, provided that both characters are supported by LispKit. If the argument is not the uppercase member of such a pair, it is returned.

(char-foldcase *char*) procedure
The `char-foldcase` procedure applies the Unicode simple case-folding algorithm to its argument and returns the result. Note that language-sensitive folding is not used. If the argument is an uppercase letter, the result will be either a lowercase letter or the same as the argument if the lowercase letter does not exist or is not supported by LispKit.

8.3 Converting characters

(digit-value *char*) procedure

Procedure `digit-value` returns the numeric value (0 to 9) of its argument if it is a numeric digit (that is, if `char-numeric?` returns `#t`), or `#f` on any other character.

```
(digit-value #\3)      ⇒  3
(digit-value #\x0EA6)  ⇒  #f
```

(char->integer *char*) procedure
(integer->char *n*)

Given a Unicode character, `char->integer` returns an exact integer between 0 and #xD7FF or between #xE000 and #x10FFFF which is equal to the Unicode scalar value of that character. Given a non-Unicode character, it returns an exact integer greater than #x10FFFF.

Given an exact integer that is the value returned by a character when `char->integer` is applied to it, `integer->char` returns that character.

(char-name *char*) procedure
(char-name *char encoded?*)

If character *char* has a corresponding named XML entity, then procedure `char-name` returns the name of this entity. Otherwise, `char-name` returns `#f`. If parameter *encoded* is set to `#t`, then the name is returned including the full entity encoding.

```
(char-name #\<)     ⇒  "LT"
(char-name #\&)     ⇒  "AMP"
(char-name #\")     ⇒  "quot"
(char-name #\a)     ⇒  #f
(char-name #\> #t)  ⇒  "&gt;"
```

9 LispKit Char-Set

Library `(lispkit char-set)` implements efficient means to represent and manipulate sets of characters. Its design is based on SRFI 14 but the implementation is specific to the definition of characters in LispKit; i.e. library `(lispkit char-set)` assumes that characters are UTF-16 code units.

As opposed to SRFI 14, it can be assumed that the update procedures ending with "!" are mutating the corresponding character set. This means that clients of these procedures may rely on these procedures performing their functionality in terms of side effects.

In the procedure specifications below, the following conventions are used:

- A *cs* parameter is a character set.
- A *s* parameter is a string.
- A *char* parameter is a character.
- A *char-list* parameter is a list of characters.
- A *pred* parameter is a unary character predicate procedure, returning either `#t` or `#f` when applied to a character.
- An *obj* parameter may be any value at all.

Passing values to procedures with these parameters that do not satisfy these types is an error.

Unless otherwise noted in the specification of a procedure, procedures always return character sets that are distinct from the parameter character sets (unless the procedure mutates a character set and its name ends with "!"). For example, `char-set-adjoin` is guaranteed to provide a fresh character set, even if it is not given any character parameters.

Library `(lispkit char-set)` supports both mutable as well as immutable character sets. Character sets are assumed to be mutable unless they are explicitly specified to be immutable.

9.1 Constants

char-set:lower-case object
char-set:upper-case
char-set:title-case
char-set:letter
char-set:digit
char-set:letter+digit
char-set:graphic
char-set:printing
char-set:whitespace
char-set:newlines
char-set:iso-control
char-set:punctuation
char-set:symbol
char-set:hex-digit
char-set:blank

char-set:ascii
char-set:empty
char-set:full

Library `(lispkit char-set)` predefines these frequently used immutable character sets.

Note that there may be characters in `char-set:letter` that are neither upper or lower case. The `char-set:whitespaces` character set contains whitespace and newline characters. `char-set:blanks` only contains whitespace (i.e. "blank") characters. `char-set:newlines` only contains newline characters.

9.2 Predicates

(char-set? *obj*) procedure
Returns `#t` if *obj* is a character set, otherwise returns `#f`.

(char-set-empty? *cs*) procedure
Returns `#t` if the character set *cs* does not contain any characters, otherwise returns `#f`.

(char-set=? *cs* ...) procedure
Returns `#t` if all the provided character sets *cs* ... contain the exact same characters; returns `#f` otherwise. For both corner cases, `(char-set=?)` and `(char-set=? cs)`, `char-set=?` returns `#t`.

(char-set<=? *cs* ...) procedure
Returns `#t` if every character set *cs-i* is a subset of character set *cs-i+1*; returns `#f` otherwise. For both corner cases, `(char-set<=?)` and `(char-set<=? cs)`, `char-set<=?` returns `#t`.

(char-set-disjoint? *cs1 cs2*) procedure
Returns `#t` if character sets *cs1* and *cs2* are disjoint, i.e. they do not share a single character; returns `#f` otherwise.

(char-set-contains? *cs char*) procedure
Returns `#t` if character *char* is contained in character set *cs*; returns `#f` otherwise.

(char-set-every? *pred cs*) procedure
(char-set-any? *pred cs*)

The `char-set-every?` procedure returns `#t` if predicate *pred* returns `#t` for every character in the character set *cs*. Likewise, `char-set-any?` applies *pred* to every character in character set *cs*, and returns `#t` if there is at least one character for which *pred* returns `#t`. If no character produces a `#t` value, it returns `#f`. The order in which these procedures sequence through the elements of *cs* is not specified.

9.3 Constructors

(char-set *char* ...) procedure
Return a newly allocated mutable character set containing the given characters.

(immutable-char-set *char* ...) procedure
Return a newly allocated immutable character set containing the given characters.

(char-set-copy *cs*) procedure
(char-set-copy *cs mutable?*)

Returns a newly allocated copy of the character set *cs*. The copy is mutable by default unless parameter *mutable?* is provided and set to `#f`.

(list->char-set *char-list*) procedure
(list->char-set *char-list base-cs*)

Return a newly allocated mutable character set containing the characters in the list of characters *char-list*. If character set *base-cs* is provided, the characters from *base-cs* are added to it as well.

(string->char-set *s*) procedure
(string->char-set *s base-cs*)

Return a newly allocated mutable character set containing the characters of the string *s*. If character set *base-cs* is provided, the characters from *base-cs* are added to it as well.

(ucs-range->char-set *lower upper*) procedure
(ucs-range->char-set *lower upper base-cs*)
(ucs-range->char-set *lower upper limit base-cs*)

Returns a newly allocated mutable character set containing every character whose ISO/IEC 10646 UCS-4 code lies in the half-open range *[lower,upper)*. *lower* and *upper* are exact non-negative integers where *lower* <= *upper* <= *limit* is required to hold. *limit* is either an exact non-negative integer specifying the maximum upper limit, or it is `#t` which specifies the maximum UTF-16 code unit value. If *limit* is not provided, a very large default is assumed (equivalent to *limit* being `#f`).

This signature is compatible with the SRFI 16 specification which states that if the requested range includes unassigned UCS values, these are silently ignored. If the requested range includes "private" or "user space" codes, these are passed through transparently. If any code from the requested range specifies a valid, assigned UCS character that has no corresponding representative in the implementation's character type, then

1. an error is raised if *limit* is `#t`, and
2. the code is ignored if *limit* is `#f` (the default).

If character set *base-cs* is provided, the characters of *base-cs* are included in the newly allocated mutable character set.

(char-set-filter *pred cs*) procedure
(char-set-filter *pred cs base-cs*)

Returns a new character set containing every character *c* in character set *cs* such that *(pred c)* returns true. If character set *base-cs* is provided, the characters specified by *base-cs* are added to it.

(->char-set *x*) procedure
Coerces object *x* into a character set. *x* may be a string, character or character set. A string is converted to the set of its constituent characters; a character is converted to a singleton character set; a character set is returned as is. This procedure is intended for use by other procedures that want to provide "user-friendly", wide-spectrum interfaces to their clients.

9.4 Querying character sets

(char-set-size *cs*) procedure
Returns the number of elements in character set *cs*.

(char-set-count *pred cs*) procedure
Apply *pred* to the characters of character set *cs*, and return the number of characters that caused the predicate to return `#t`.

(char-set->list *cs*) procedure
This procedure returns a list of the characters of character set *cs*. The order in which *cs*'s characters appear in the list is not defined, and may be different from one call to another.

(char-set->string *cs*) procedure

This procedure returns a string containing the characters of character set *cs*. The order in which *cs*'s characters appear in the string is not defined, and may be different from one call to another.

(char-set-hash *cs*) procedure
(char-set-hash *cs bound*)

Compute a hash value for the character set *cs*. *bound* is a non-negative exact integer specifying the range of the hash function. A positive value restricts the return value to the range *[0, bound)*. If *bound* is either zero or not given, a default value is used, chosen to be as large as it is efficiently practical.

9.5 Character set algebra

(char-set-adjoin *cs char* ...) procedure

Return a newly allocated mutable copy of *cs* into which the characters *char* ... were inserted.

(char-set-delete *cs char* ...) procedure

Return a newly allocated mutable copy of *cs* from which the characters *char* ... were removed.

(char-set-complement *cs*) procedure

Return a newly allocated character set containing all characters that are not contained in *cs*.

(char-set-union *cs* ...) procedure
(char-set-intersection *cs* ...)
(char-set-difference *cs* ...)
(char-set-xor *cs* ...)
(char-set-diff+intersection *cs1 cs2* ...)

These procedures implement set complement, union, intersection, difference, and exclusive-or for character sets. The union, intersection and xor operations are n-ary. The difference function is also n-ary, associates to the left (that is, it computes the difference between its first argument and the union of all the other arguments), and requires at least one argument.

Boundary cases:

```
(char-set-union)          ⇒  char-set:empty
(char-set-intersection)   ⇒  char-set:full
(char-set-xor)            ⇒  char-set:empty
(char-set-difference cs)  ⇒  cs
```

`char-set-diff+intersection` returns both the difference and the intersection of the arguments, i.e. it partitions its first parameter. It is equivalent to `(values (char-set-difference cs1 cs2 ...) (char-set-intersection cs1 (char-set-union cs2 ...)))` but can be implemented more efficiently.

(char-set-filter *pred cs*) procedure
(char-set-filter *pred cs base-cs*)

Returns a new character set containing every character *c* in *cs* such that *(pred c)* returns `#t`. If character set *base-cs* is provided, the characters specified by *pred* are added to a copy of it.

9.6 Mutating character sets

(char-set-adjoin! *cs char* ...) procedure

Insert the characters *char* ... into the character set *cs*.

(char-set-delete! *cs char …*) procedure

Remove the characters *char* … from the character set *cs*.

(char-set-complement! *cs*) procedure

Complement the character set *cs* by including all characters that were not contained in *cs* previously and by removing all previously contained characters.

(char-set-union! *cs1 cs2 …*) procedure
(char-set-intersection! *cs1 cs2 …*)
(char-set-difference! *cs1 cs2 …*)
(char-set-xor! *cs1 cs2 …*)
(char-set-diff+intersection! *cs1 cs2 cs3 …*)

These are update variants of the set-algebra functions mutating the first character set *cs1* instead of creating a new one. `char-set-diff+intersection!` will perform a side-effect on both of its two required parameters *cs1* and *cs2*.

(char-set-filter! *pred cs base-cs*) procedure

Adds every character *c* in *cs* for which *(pred c)* returns `#t` to the given character set `base-cs`.

(list->char-set! *char-list base-cs*) procedure

Add the characters from the character list *char-list* to character set *base-cs* and return the mutated character set *base-cs*.

(string->char-set! *s base-cs*) procedure

Add the characters of the string *s* to character set *base-cs* and return the mutated character set *base-cs*.

(ucs-range->char-set! *lower upper base-cs*) procedure
(ucs-range->char-set! *lower upper limit base-cs*)

Mutates the mutable character set *base-cs* including every character whose ISO/IEC 10646 UCS-4 code lies in the half-open range *[lower,upper)*. *lower* and *upper* are exact non-negative integers where *lower* <= *upper* <= *limit* is required to hold. *limit* is either an exact non-negative integer specifying the maximum upper limit, or it is `#t` which specifies the maximum UTF-16 code unit value. If *limit* is not provided, a very large default is assumed (equivalent to *limit* being `#f`).

(char-set-unfold! *f p g seed base-cs*) procedure

This is a fundamental constructor for character sets.

- *g* is used to generate a series of "seed" values from the initial seed: seed, (g seed), (g2 seed), (g3 seed), …
- *p* tells us when to stop by returning `#t` when applied to one of these seed values.
- *f* maps each seed value to a character. These characters are added to the base character set *base-cs* to form the result. `char-set-unfold!` adds the characters by mutating *base-cs* as a side effect.

9.7 Iterating over character sets

(char-set-cursor *cs*) procedure
(char-set-ref *cs cursor*)
(char-set-cursor-next *cs cursor*)
(end-of-char-set? *cursor*)

Cursors are a low-level facility for iterating over the characters in a character set *cs*. A cursor is a value that indexes a character in a character set. `char-set-cursor` produces a new cursor for a given character set. The set element indexed by the cursor is fetched with `char-set-ref`. A cursor index is incremented with `char-set-cursor-next`; in this way, code can step through every character in a character set. Stepping a cursor "past the end" of a character set produces a cursor that answers true to `end-of-char-set?`. It is an error to pass such a cursor to `char-set-ref` or to `char-set-cursor-next`.

A cursor value may not be used in conjunction with a different character set; if it is passed to `char-set-ref` or `char-set-cursor-next` with a character set other than the one used to create it, the results and effects are undefined. These primitives are necessary to export an iteration facility for character sets to loop macros.

```
(define cs (char-set #\G #\a #\T #\e #\c #\h))

;; Collect elts of CS into a list.
(let lp ((cur (char-set-cursor cs)) (ans '()))
  (if (end-of-char-set? cur) ans
      (lp (char-set-cursor-next cs cur)
          (cons (char-set-ref cs cur) ans))))
  ⇒ (#\G #\T #\a #\c #\e #\h)

;; Equivalently, using a list unfold (from SRFI 1):
(unfold-right end-of-char-set?
              (curry char-set-ref cs)
              (curry char-set-cursor-next cs)
              (char-set-cursor cs))
  ⇒ (#\G #\T #\a #\c #\e #\h)
```

(char-set-fold *kons knil cs*) procedure

This is the fundamental iterator for character sets. Applies the function *kons* across the character set *cs* using initial state value *knil*. That is, if *cs* is the empty set, the procedure returns *knil*. Otherwise, some element *c* of *cs* is chosen; let *cs'* be the remaining, unchosen characters. The procedure returns `(char-set-fold kons (kons c knil) cs')`.

```
; CHAR-SET-MEMBERS
(lambda (cs) (char-set-fold cons '() cs))
; CHAR-SET-SIZE
(lambda (cs) (char-set-fold (lambda (c i) (+ i 1)) 0 cs))
; How many vowels in the char set?
(lambda (cs)
  (char-set-fold (lambda (c i) (if (vowel? c) (+ i 1) i)) 0 cs))
```

(char-set-unfold *f p g seed*) procedure
(char-set-unfold *f p g seed base-cs*)

This is a fundamental constructor for character sets.

- *g* is used to generate a series of "seed" values from the initial seed: seed, (g seed), (g2 seed), (g3 seed), ...
- *p* tells us when to stop, when it returns `#t` when applied to one of these seed values.
- *f* maps each seed value to a character. These characters are added to a mutable copy of the base character set *base-cs* to form the result; *base-cs* defaults to an empty set.

More precisely, the following definitions hold, ignoring the optional-argument issues:

```
(define (char-set-unfold p f g seed base-cs)
  (char-set-unfold! p f g seed (char-set-copy base-cs)))

(define (char-set-unfold! p f g seed base-cs)
  (let lp ((seed seed) (cs base-cs))
    (if (p seed) cs                          ; P says we are done
        (lp (g seed)                         ; Loop on (G SEED)
            (char-set-adjoin! cs (f seed)))))) ; Add (F SEED) to set
```

Examples:

```
(port->char-set p) = (char-set-unfold eof-object?
                                      values
                                      (lambda (x) (read-char p))
                                      (read-char p))
(list->char-set lis) = (char-set-unfold null? car cdr lis)
```

(char-set-for-each *proc cs*) procedure

Apply procedure *proc* to each character in the character set *cs*. Note that the order in which proc is applied to the characters in the set is not specified, and may even change from one procedure application to another.

(char-set-map *proc cs*) procedure

proc is a procedure mapping characters to characters. It will be applied to all the characters in the character set *cs*, and the results will be collected in a newly allocated mutable character set which will be returned by `char-set-map`.

10 LispKit Combinator

Library `(lispkit combinator)` defines abstractions for *combinator-style programming*. It provides means to create and compose functions.

(const *c* ...) procedure

Returns a function accepting any number of arguments and returning the values `c ...`.

(flip *f*) procedure

Takes a function with two parameters and returns an equivalent function where the two parameters are swapped.

```
(define snoc (flip cons))
(snoc (snoc (snoc '() 3) 2) 1)  ⇒  (1 2 3)
```

(negate *f*) procedure

Returns a function which invokes `f` and returns the logical negation.

```
(define gvector-has-elements? (negate gvector-empty?))
(gvector-has-elements? #g(1 2 3))  ⇒  #t
```

(partial f *arg* ...) procedure

Applies arguments *arg* ... partially to *f* and returns a new function accepting the remaining arguments. For a function `(f a1 a2 a3 ... an)`, `(partial f a1 a2)` will return a function `(lambda (a3 ... an) (f a1 a2 a3 ... an))`.

(compose *f* ...) procedure

Composes the given functions *f* ... such that `((compose f1 f2 ... fn) x)` is equivalent to `(f1 (f2 (... (fn x))))`. `compose` supports functions returning multiple arguments.

(o *f* ...) procedure

Composes the given functions *f* ... such that `((o f1 f2 ... fn) x)` is equivalent to `(f1 (f2 (... (fn x))))`. `o` is a more efficient version of `compose` which only works if the involved functions only return a single argument. `compose` is more general and supports functions returning multiple arguments.

(conjoin *f* ...) procedure

Returns a function invoking all functions *f* ... and combining the results with `and`. `((conjoin f1 f2 ...) x ...)` is equivalent to `(and (f1 x ...) (f2 x ...) ...)`.

(disjoin *f* ...) procedure

Returns a function invoking all functions *f* ... and combining the results with `or`. `((disjoin f1 f2 ...) x ...)` is equivalent to `(or (f1 x ...) (f2 x ...) ...)`.

(list-of? *f*) procedure

Returns a predicate which takes a list as its argument and returns `#t` if for every element *x* of the list *(f x)* returns true.

(each *f* ...) procedure

Returns a function which applies the functions `f` ... each individually to its arguments in the given order, returning the result of the last function application.

(cut *f*) syntax
(cut *f* <...>)
(cut *f arg* ...)
(cut *f arg* ... <...>)

Special form `cut` transforms an expression *(f arg ...)* into a lambda expression with as many formal variables as there are slots `<>` in the expression *(f arg ...)*. The body of the resulting lambda expression calls procedure *f* with arguments *arg* ... in the order they appear. In case there is a rest symbol `<...>` at the end, the resulting procedure is of variable arity, and the body calls *f* with all arguments provided to the actual call of the specialized procedure.

```
(cut cons (+ a 1) <>)   ⇒  (lambda (x2) (cons (+ a 1) x2))
(cut list 1 <> 3 <> 5)  ⇒  (lambda (x2 x4) (list 1 x2 3 x4 5))
(cut list 1 <> 3 <...>) ⇒  (lambda (x2 . xs) (apply list 1 x2 3 xs))
```

(cute *f*) syntax
(cute *f* <...>)
(cute *f arg* ...)
(cute *f arg* ... <...>)

Special form `cute` is similar to `cut`, except that it first binds new variables to the result of evaluating the non-slot expressions (in an unspecific order) and then substituting the variables for the non-slot expressions. In effect, `cut` evaluates non-slot expressions at the time the resulting procedure is called, whereas `cute` evaluates the non-slot expressions at the time the procedure is constructed.

```
(cute cons (+ a 1) <>)
⇒  (let ((a1 (+ a 1))) (lambda (x2) (cons a1 x2)))
```

(Y *f*) procedure
Y combinator for computing a fixed point of a function *f*. This is a value that is mapped to itself.

```
; factorial function
(define fac
  (Y (lambda (r)
       (lambda (x) (if (< x 2) 1 (* x (r (- x 1))))))))
; fibonacci numbers
(define fib
  (Y (lambda (f)
       (lambda (x)
         (if (< x 2) x (+ (f (- x 1)) (f (- x 2))))))))
```

11 LispKit Comparator

Comparators bundle a type test predicate, an equality predicate, an optional ordering predicate, and an optional hash function into a single object. By packaging these procedures together, they can be treated as a single item for use in the implementation of data structures that typically rely on a consistent combination of such functions.

Library `(lispkit comparator)` implements a large part of the API of SRFI 128 and thus, can be used as a drop-in replacement for the core functionality of library `(srfi 128)`. A few procedures and objects from SRFI 162 were adopted as well.

11.1 Comparator objects

Comparators are objects of a distinct type which bundle procedures together that are useful for comparing two objects in a total order. It is an error, if any of the procedures have side effects. There are four procedures in the bundle:

- The *type test predicate* returns `#t` if its argument has the correct type to be passed as an argument to the other three procedures, and `#f` otherwise.
- The *equality predicate* returns `#t` if the two objects are the same in the sense of the comparator, and `#f` otherwise. It is the programmer's responsibility to ensure that it is reflexive, symmetric, transitive, and can handle any arguments that satisfy the type test predicate.
- The *ordering predicate* returns `#t` if the first object precedes the second in a total order, and `#f` otherwise. Note that if it is true, the equality predicate must be false. It is the programmer's responsibility to ensure that it is irreflexive, antisymmetric, transitive, and can handle any arguments that satisfy the type test predicate.
- The *hash function* takes an object and returns an exact non-negative integer. It is the programmer's responsibility to ensure that it can handle any argument that satisfies the type test predicate, and that it returns the same value on two objects if the equality predicate says they are the same (but not necessarily the converse).

It is also the programmer's responsibility to ensure that all four procedures provide the same result whenever they are applied to the same objects in the sense of `eqv?`, unless the objects have been mutated since the last invocation.

Comparator objects are not applicable to circular structure, or to objects containing any of these. Attempts to pass any such objects to any procedure defined here, or to any procedure that is part of a comparator defined here, has undefined behavior.

11.2 Predicates

(comparator? *obj*) procedure

Returns `#t` if obj is a comparator, and `#f` otherwise.

(comparator-ordered? *cmp*) procedure

Returns `#t` if comparator *cmp* has a supplied ordering predicate, and `#f` otherwise.

(comparator-hashable? *cmp*) procedure

Returns `#t` if comparator *cmp* has a supplied hash function, and `#f` otherwise.

11.3 Constructors

(make-comparator *test equality ordering hash*) procedure

Returns a comparator which bundles the *test*, *equality*, *ordering*, and *hash* procedures provided as arguments to `make-comparator`. If *ordering* or *hash* is `#f`, a procedure is provided that signals an error on application. The predicates `comparator-ordered?` and `comparator-hashable?` will return `#f` in the respective cases.

Here are calls on `make-comparator` that will return useful comparators for standard Scheme types:

- `(make-comparator boolean? boolean=? (lambda (x y) (and (not x) y)) boolean-hash)` will return a comparator for booleans, expressing the ordering `#f < #t` and the standard hash function for booleans
- `(make-comparator real? = < (lambda (x) (exact (abs x))))` will return a comparator expressing the natural ordering of real numbers and a plausible hash function
- `(make-comparator string? string=? string<? string-hash)` will return a comparator expressing the implementation's ordering of strings and the standard hash function
- `(make-comparator string? string-ci=? string-ci<? string-ci-hash)` will return a comparator expressing the implementation's case-insensitive ordering of strings and the standard case-insensitive hash function

(make-pair-comparator *car-comparator cdr-comparator*) procedure

This procedure returns comparators whose functions behave as follows:

- The type test returns `#t` if its argument is a pair, if the car satisfies the type test predicate of *car-comparator*, and the cdr satisfies the type test predicate of *cdr-comparator*
- The equality function returns `#t` if the cars are equal according to *car-comparator* and the cdrs are equal according to *cdr-comparator*, and `#f` otherwise. The ordering function first compares the cars of its pairs using the equality predicate of *car-comparator*. If they are not equal, then the ordering predicate of *car-comparator* is applied to the cars and its value is returned. Otherwise, the predicate compares the cdrs using the equality predicate of *cdr-comparator*. If they are not equal, then the ordering predicate of *cdr-comparator* is applied to the cdrs and its value is returned
- The hash function computes the hash values of the car and the cdr using the hash functions of *car-comparator* and *cdr-comparator* respectively and then hashes them together in an implementation-defined way

(make-list-comparator *element-comparator type-test empty? head tail*) procedure

This procedure returns comparators whose functions behave as follows:

- The type test returns `#t` if its argument satisfies *type-test* and the elements satisfy the type test predicate of *element-comparator*
- The total order defined by the equality and ordering functions is *lexicographic*. It is defined as follows:
  - The empty sequence, as determined by calling *empty?*, compares equal to itself
  - The empty sequence compares less than any non-empty sequence
  - Two non-empty sequences are compared by calling the *head* procedure on each. If the heads are not equal when compared using *element-comparator*, the result is the result of that comparison. Otherwise, the results of calling the *tail* procedure are compared recursively.
- The hash function computes the hash values of the elements using the hash function of *element-comparator* and then hashes them together in an implementation-defined way

(make-vector-comparator ***element-comparator type-test length ref*****)** procedure

This procedure returns comparators whose functions behave as follows:

- The type test returns `#t` if its argument satisfies *type-test* and the elements satisfy the type test predicate of *element-comparator*.
- The equality predicate returns `#t` if both of the following tests are satisfied in order: the lengths of the vectors are the same in the sense of `=`, and the elements of the vectors are the same in the sense of the equality predicate of *element-comparator*.
- The ordering predicate returns `#t` if the results of applying *length* to the first vector is less than the result of applying *length* to the second vector. If the lengths are equal, then the elements are examined pairwise using the ordering predicate of *element-comparator*. If any pair of elements returns `#t`, then that is the result of the list comparator's ordering predicate; otherwise the result is `#f`
- The hash function computes the hash values of the elements using the hash function of *element-comparator* and then hashes them together in an implementation-defined way

Here is an example, which returns a comparator for byte vectors:

```
(make-vector-comparator
  (make-comparator exact-integer? = < number-hash)
  bytevector?
  bytevector-length
  bytevector-u8-ref)
```

11.4 Default comparators

eq-comparator object
eqv-comparator
equal-comparator

These objects implement comparators whose functions behave as follows:

- The type test returns `#t` in all cases
- The equality functions are `eq?`, `eqv?`, and `equal?` respectively
- The ordering function is implementation-defined, except that it must conform to the rules for ordering functions. It may signal an error instead.
- The hash functions are `eq-hash`, `eqv-hash`, and `equal-hash` respectively

boolean-comparator object

`boolean-comparator` is defined as follows:

```
(make-comparator boolean? boolean=? (lambda (x y) (and (not x) y)) boolean-hash))
```

real-comparator object

`real-comparator` is defined as follows:

```
(make-comparator real? = < number-hash))
```

char-comparator object

`char-comparator` is defined as follows:

```
(make-comparator char? char=? char<? char-hash))
```

char-ci-comparator object

`char-ci-comparator` is defined as follows:

```
(make-comparator char? char-ci=? char-ci<? char-ci-hash))
```

string-comparator object

`string-comparator` is defined as follows:

```
(make-comparator string? string=? string<? string-hash))
```

string-ci-comparator object

`string-ci-comparator` is defined as follows:

```
(make-comparator string? string-ci=? string-ci<? string-ci-hash))
```

11.5 Accessors and invokers

(comparator-type-test-predicate *cmp*) procedure

Returns the type test predicate of comparator *cmp*.

(comparator-equality-predicate *cmp*) procedure

Returns the equality predicate of comparator *cmp*.

(comparator-ordering-predicate *cmp*) procedure

Returns the ordering predicate of comparator *cmp*.

(comparator-hash-function *cmp*) procedure

Returns the hash function of comparator *cmp*.

(comparator-test-type *cmp obj*) procedure

Invokes the type test predicate of comparator *cmp* on *obj* and returns what it returns. This procedure is convenient than `comparator-type-test-predicate`, but less efficient when the predicate is called repeatedly.

(comparator-check-type *cmp obj*) procedure

Invokes the type test predicate of comparator *cmp* on *obj* and returns `#t` if it returns `#t`, but signals an error otherwise. This procedure is more convenient than `comparator-type-test-predicate`, but less efficient when the predicate is called repeatedly.

(comparator-hash *cmp obj*) procedure

Invokes the hash function of comparator *cmp* on *obj* and returns what it returns. This procedure is more convenient than comparator-hash-function, but less efficient when the function is called repeatedly.

11.6 Comparison predicates

(=? *cmp object1 object2 object3* ...) procedure
(<? *cmp object1 object2 object3* ...)
(>? *cmp object1 object2 object3* ...)
(<=? *cmp object1 object2 object3* ...)
(>=? *cmp object1 object2 object3* ...)

These procedures are analogous to the number, character, and string comparison predicates of Scheme. They allow the convenient use of comparators to handle variable data types.

These procedures apply the equality and ordering predicates of comparator *cmp* to the *objects* as follows. If the specified relation returns `#t` for all *objecti* and *objectj* where *n* is the number of objects and $1 <= i < j <= n$, then the procedures return `#t`, but otherwise `#f`. Because the relations are transitive, it suffices to compare each object with its successor. The order in which the values are compared is unspecified.

(comparator-max *cmp obj1 obj2 ...*) procedure
(comparator-min *cmp obj1 obj2 ...*)
(comparator-max-in-list *cmp list*)
(comparator-min-in-list *cmp list*)

These procedures are analogous to `min` and `max` respectively, but may be applied to any orderable objects, not just to real numbers. They apply the ordering procedure of comparator *cmp* to the objects *obj1 ...* to find and return a minimal or maximal object. The order in which the values are compared is unspecified. If two objects are equal in the sense of the comparator *cmp*, either may be returned.

The `-in-list` versions of the procedures accept a single list argument.

11.7 Syntax

(comparator-if<=> *obj1 obj2 less-than equal-to greater-than*) syntax
(comparator-if<=> *cmp obj1 obj2 less-than equal-to greater-than*)

It is an error unless comparator *cmp* evaluates to a comparator and *obj1* and *obj2* evaluate to objects that the comparator can handle. If the ordering predicate returns `#t` when applied to the values of *obj1* and *obj2* in that order, then expression *less-than* is evaluated and its value is returned. If the equality predicate returns `#t` when applied in the same way, then expression *equal-to* is evaluated and its value is returned. If neither returns `#t`, expression *greater-than* is evaluated and its value is returned.

If *cmp* is omitted, `equal-comparator` is used as a default.

(if<=> *obj1 obj2 less-than equal-to greater-than*) syntax

This special form is equivalent to `(comparator-if<=> obj1 obj2 less-than equal-to greater-than)`, i.e. it uses the predicates provided by `equal-comparator` to determine whether expression *less-than*, *equal-to*, or *greater-than* gets evaluated and its value returned.

This documentation was derived from the SRFI 128 and the SRFI 162 specifications by John Cowan.

12 LispKit Control

12.1 Sequencing

(begin *expr* … *exprn*) syntax

`begin` evaluates *expr*, …, *exprn* sequentially from left to right. The values of the last expression *exprn* are returned. This form is typically used to sequence side effects such as assignments or input and output.

12.2 Conditionals

(if *test consequent*) syntax
(if *test consequent alternate*)

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

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

(when *test consequent* …) syntax

The *test* expression is evaluated, and if it evaluates to a true value, the expressions *consequent* … are evaluated in order. The result of the `when` expression is the value to which the last *consequent* expression evaluates or *void* if *test* evaluates to false.

```
(when (= 1 1.0)
  (display "1")
  (display "2"))  ⇒ (void), prints: 12
```

(unless *test alternate* …) syntax

The *test* is evaluated, and if it evaluates to false, the expressions *alternate* … are evaluated in order. The result of the `unless` expression is the value to which the last *consequent* expression evaluates or *void* if *test* evaluates to a true value.

```
(unless (= 1 1.0)
  (display "1")
  (display "2"))  ⇒ (void), prints nothing
```

(cond *clause1 clause2* …) syntax

Clauses like *clause1* and *clause2* take one of two forms, either

- `(_test expr1 ..._)`, or
- `(_test_ => _expr_)`

The last clause in a `cond` expression can be an “else clause”, which has the form

- `(else _expr1 expr2 ..._)`

A `cond` expression is evaluated by evaluating the *test* expressions of successive clauses in order until one of them evaluates to a true value. When a *test* expression evaluates to a true value, the remaining expressions in its clause are evaluated in order, and the results of the last expression are returned as the results of the entire `cond` expression.

If the selected *clause* contains only the *test* and no expressions, then the value of the *test* expression is returned as the result of the `cond` expression. If the selected clause uses the `=>` alternate form, then the expression is evaluated. It is an error if its value is not a procedure that accepts one argument. This procedure is then called on the value of the *test* and the values returned by this procedure are returned by the `cond` expression.

If all tests evaluate to `#f`, and there is no `else` clause, then the result of the conditional expression is *void*. If there is an `else` clause, then its expressions are evaluated in order, and the values of the last one are returned.

```
(cond ((> 3 2) 'greater)
      ((< 3 2) 'less))      ⇒ greater
(cond ((> 3 3) 'greater)
      ((< 3 3) 'less)
      (else 'equal))        ⇒ equal
(cond ((assv 'b '((a 1) (b 2))) => cadr)
      (else #f))            ⇒ 2
```

(case *key clause1 clause2* ...) syntax

key can be any expression. Each clause *clause1, clause2,* ... has the form:

- `((_datum1 ..._) _expr1 expr2 ..._)`

where each *datum* is an external representation of some object. It is an error if any of the datums are the same anywhere in the expression. Alternatively, a clause can be of the form:

- `((_datum1 ..._) => _expr_)`

The last clause in a `case` expression can be an “else clause”, which has one of the following forms:

- `(else _expr1 expr2 ..._)`, or
- `(else => _expr_)`

A case expression is evaluated as follows. Expression *key* is evaluated and its result is compared against each *datum*. If the result of evaluating *key* is the same, in the sense of `eqv?` , to a *datum*, then the expressions in the corresponding clause are evaluated in order and the results of the last expression in the clause are returned as the results of the `case` expression.

If the result of evaluating *key* is different from every *datum*, then if there is an `else` clause, its expressions are evaluated and the results of the last are the results of the `case` expression. Otherwise, the result of the `case` expression is *void*.

If the selected *clause* or `else` clause uses the `=>` alternate form, then the expression is evaluated. It is an error, if its value is not a procedure accepting one argument. This procedure is then called on the value of the *key* and the values returned by this procedure are returned by the `case` expression.

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

```
  ((a) 'a)
  ((b) 'b))              ⇒ (void)
(case (car '(c d))
  ((a e i o u) 'vowel)
  ((w y) 'semivowel)
  (else => (lambda (x) x)))   ⇒ c
```

12.3 Local bindings

The binding constructs `let`, `let*`, `letrec`, `letrec*`, `let-values`, and `let*-values` give Scheme a block structure. The syntax of the first four constructs is identical, but they differ in the regions they establish for their variable bindings. In a `let` expression, the initial values are computed before any of the variables become bound. In a `let*` expression, the bindings and evaluations are performed sequentially. While in `letrec` and `letrec*` expressions, all the bindings are in effect while their initial values are being computed, thus allowing mutually recursive definitions. The `let-values` and `let*-values` constructs are analogous to `let` and `let*` respectively, but are designed to handle multiple-valued expressions, binding different identifiers to the returned values.

(let *bindings body*) syntax

bindings has the form `((` *variable init* `) ...)`, where each *init* is an expression, and *body* is a sequence of zero or more definitions followed by a sequence of one or more expressions. It is an error for *variable* to appear more than once in the list of variables being bound.

All *init* expressions are evaluated in the current environment, the variables are bound to fresh locations holding the results, the *body* is evaluated in the extended environment, and the values of the last expression of *body* are returned. Each binding of a *variable* has *body* as its scope.

```
(let ((x 2) (y 3))
  (* x y))            ⇒ 6
(let ((x 2) (y 3))
  (let ((x 7)
        (z (+ x y)))
    (* z x)))         ⇒ 35
```

(let* *bindings body*) syntax

bindings has the form `((` *variable init* `) ...)`, where each *init* is an expression, and *body* is a sequence of zero or more definitions followed by a sequence of one or more expressions.

The `let*` binding construct is similar to let, but the bindings are performed sequentially from left to right, and the region of a binding indicated by `(` *variable init* `)` is that part of the `let*` expression to the right of the binding. Thus, the second binding is done in an environment in which the first binding is visible, and so on. The variables need not be distinct.

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

(letrec *bindings body*) syntax

bindings has the form `((` *variable init* `) ...)`, where each *init* is an expression, and *body* is a sequence of zero or more definitions followed by a sequence of one or more expressions. It is an error for *variable* to appear more than once in the list of variables being bound.

The variables are bound to fresh locations holding unspecified values, the *init* expressions are evaluated in the resulting environment, each *variable* is assigned to the result of the corresponding *init* expression, the *body* is evaluated in the resulting environment, and the values of the last expression in *body* are returned. Each binding of a *variable* has the entire `letrec` expression as its scope, making it possible to define mutually recursive procedures.

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

One restriction of `letrec` is very important: if it is not possible to evaluate each *init* expression without assigning or referring to the value of any *variable*, it is an error. The restriction is necessary because `letrec` is defined in terms of a procedure call where a `lambda` expression binds the variables to the values of the *init* expressions. In the most common uses of `letrec`, all the *init* expressions are lambda expressions and the restriction is satisfied automatically.

(letrec* *bindings body*) syntax

bindings has the form `((` *variable init* `) ...)`, where each *init* is an expression, and *body* is a sequence of zero or more definitions followed by a sequence of one or more expressions. It is an error for *variable* to appear more than once in the list of variables being bound.

The variables are bound to fresh locations, each *variable* is assigned in left-to-right order to the result of evaluating the corresponding *init* expression, the *body* is evaluated in the resulting environment, and the values of the last expression in *body* are returned. Despite the left-to-right evaluation and assignment order, each binding of a *variable* has the entire `letrec*` expression as its region, making it possible to define mutually recursive procedures. If it is not possible to evaluate each *init* expression without assigning or referring to the value of the corresponding *variable* or the *variable* of any of the bindings that follow it in *bindings*, it is an error. Another restriction is that it is an error to invoke the continuation of an *init* expression more than once.

```
(letrec* ((p (lambda (x)
               (+ 1 (q (- x 1)))))
          (q (lambda (y)
               (if (zero? y) 0 (+ 1 (p (- y 1))))))
          (x (p 5))
          (y x))
  y)  ⇒  5
```

(let-values *bindings body*) syntax

bindings has the form `((` *formals init* `) ...)`, where each *formals* is a list of variables, *init* is an expression, and *body* is zero or more definitions followed by a sequence of one or more expressions. It is an error for a variable to appear more than once in *formals*.

The *init* expressions are evaluated in the current environment as if by invoking `call-with-values`, and the variables occurring in list *formals* are bound to fresh locations holding the values returned by the *init* expressions, where the *formals* are matched to the return values in the same way that the *formals* in a lambda expression are matched to the arguments in a procedure call. Then, *body* is evaluated in the extended environment, and the values of the last expression of *body* are returned. Each binding of a variable has *body* as its scope.

It is an error if the variables in list *formals* do not match the number of values returned by the corresponding *init* expression.

```
(let-values (((root rem) (exact-integer-sqrt 32)))
  (* root rem))  ⇒  35
```

(let*-values *bindings body*) syntax

bindings has the form `((formals init) ...)`, where each *formals* is a list of variables, *init* is an expression, and *body* is zero or more definitions followed by a sequence of one or more expressions. It is an error for a variable to appear more than once.

The `let*-values` construct is similar to `let-values`, but the *init* expressions are evaluated and bindings created sequentially from left to right, with the region of the bindings of each variable in *formals* including the *init* expressions to its right as well as *body*. Thus the second *init* expression is evaluated in an environment in which the first set of bindings is visible and initialized, and so on.

```
(let ((a 'a)
      (b 'b)
      (x 'x)
      (y 'y))
  (let*-values (((a b) (values x y))
                ((x y) (values a b)))
    (list a b x y)))
⇒  (x y x y)
```

(letrec-values *bindings body*) syntax

bindings has the form `((formals init) ...)`, where each *formals* is a list of variables, *init* is an expression, and *body* is zero or more definitions followed by a sequence of one or more expressions. It is an error for a variable to appear more than once.

First, the variables of the *formals* lists are bound to fresh locations holding unspecified values. Then, the *init* expressions are evaluated in the current environment as if by invoking `call-with-values`, where the *formals* are matched to the return values in the same way that the *formals* in a lambda expression are matched to the arguments in a procedure call. Finally, *body* is evaluated in the resulting environment, and the values of the last expression in *body* are returned. Each binding of a *variable* has the entire `letrec-values` expression as its scope, making it possible to define mutually recursive procedures.

```
(letrec-values
  (((a)   (lambda (n)
            (if (zero? n) #t (odd? (- n 1)))))
   ((b c) (values
            (lambda (n) (if (zero? n) #f (even? (- n 1))))
            a)))
  (list (a 1972) (b 1972) (c 1972)))
⇒  (#t #f #t)
```

(let-optionals *args* (*arg* ... (*var default*) ...) *body* ...) syntax
(let-optionals *args* (*arg* ... (*var default*) ... . *rest*) *body* ...)

This binding construct can be used to handle optional arguments of procedures. *args* refers to the rest parameter list of a procedure or lambda expression. `let-optionals` binds the variables *arg* ... to the arguments available in *args*. It is an error if there are not sufficient elements in *args*. Then, the variables *var*, ... are bound to the remaining elements available in list *args*, or to *default*, ... if there are not enough elements available in *args*. Variables are bound in parallel, i.e. all *default* expressions are evaluated in the current environment in which the new variables are not bound yet. Then, *body* is evaluated in the extended environment including all variable definitions of `let-optionals`, and the values of the last expression of *body* are returned. Each binding of a variable has *body* as its scope.

```
(let-optionals '("zero" "one" "two")
               (zero (one 1) (two 2) (three 3))
  (list zero one two three))
⇒  ("zero" "one" "two" 3)
```

(let*-optionals *args* (*arg* ... (*var default*) ...) *body* ...) syntax
(let*-optionals *args* (*arg* ... (*var default*) ... . *rest*) *body* ...)

The `let*-optionals` construct is similar to `let-optionals`, but the *default* expressions are evaluated and bindings created sequentially from left to right, with the scope of the bindings of each variable including the *default* expressions to its right as well as *body*. Thus, the second *default* expression is evaluated in an environment in which the first binding is visible and initialized, and so on.

```
(let*-optionals '(0 10 20)
                (zero
                 (one (+ zero 1))
                 (two (+ zero one 1))
                 (three (+ two 1)))
  (list zero one two three))  ⇒  (0 10 20 21)
```

(let-keywords *args* (*binding* ...) *body* ...) syntax
binding has one of two forms:

- (*var default*) , and
- (*var keyword default*)

where *var* is a variable, *keyword* is a symbol, and *default* is an expression. It is an error for a variable *var* to appear more than once.

This binding construct can be used to handle keyword arguments of procedures. *args* refers to the rest parameter list of a procedure or lambda expression. `let-keywords` binds the variables *var*, ... by name, i.e. by searching in *args* for the keyword argument. If an optional *keyword* is provided, it is used as the name of the keyword to search for, otherwise, *var* is used, appending `:`. If the keyword is not found in *args*, *var* is bound to *default*.

Variables are bound in parallel, i.e. all *default* expressions are evaluated in the current environment in which the new variables are not bound yet. Then, *body* is evaluated in the extended environment including all variable definitions of `let-keywords`, and the values of the last expression of *body* are returned. Each binding of a variable has *body* as its scope.

```
(define (make-person . args)
  (let-keywords args ((name "J. Doe")
                      (age 0)
                      (occupation job: 'unknown))
    (list name age occupation)))
(make-person)                     ⇒  ("J. Doe" 0 unknown)
(make-person 'name: "M. Zenger")  ⇒  ("M. Zenger" 0 unknown)
(make-person 'age: 31 'job: 'eng) ⇒  ("J. Doe" 31 eng)
```

(let*-keywords *args* (*binding* ...) *body* ...) syntax
binding has one of two forms:

- (*var default*) , and
- (*var keyword default*)

where *var* is a variable, *keyword* is a symbol, and *default* is an expression. It is an error for a variable *var* to appear more than once.

The `let*-keywords` construct is similar to `let-keywords`, but the *default* expressions are evaluated and bindings created sequentially from left to right, with the scope of the bindings of each variable including the *default* expressions to its right as well as *body*. Thus the second *default* expression is evaluated in an environment in which the first binding is visible and initialized, and so on.

12.4 Local syntax bindings

The `let-syntax` and `letrec-syntax` binding constructs are analogous to `let` and `letrec`, but they bind syntactic keywords to macro transformers instead of binding variables to locations that contain values. Syntactic keywords can also be bound globally or locally with `define-syntax`.

(let-syntax *bindings body*) syntax

bindings has the form `((` *keyword transformer* `) ...)`. Each *keyword* is an identifier, each *transformer* is an instance of `syntax-rules`, and *body* is a sequence of one or more definitions followed by one or more expressions. It is an error for a *keyword* to appear more than once in the list of keywords being bound.

body is expanded in the syntactic environment obtained by extending the syntactic environment of the `let-syntax` expression with macros whose keywords are the *keyword* symbols bound to the specified transformers. Each binding of a *keyword* has *body* as its scope.

```
(let-syntax
  ((given-that (syntax-rules ()
                 ((_ test stmt1 stmt2 ...)
                   (if test
                       (begin stmt1 stmt2 ...))))))
  (let ((if #t))
    (given-that if (set! if 'now))
    if))
⇒ now
(let ((x 'outer))
  (let-syntax ((m (syntax-rules () ((m) x))))
    (let ((x 'inner))
      (m))))
⇒ outer
```

(letrec-syntax *bindings body*) syntax

bindings has the form `((` *keyword transformer* `) ...)`. Each *keyword* is an identifier, each *transformer* is an instance of `syntax-rules`, and *body* is a sequence of one or more definitions followed by one or more expressions. It is an error for a *keyword* to appear more than once in the list of keywords being bound.

body is expanded in the syntactic environment obtained by extending the syntactic environment of the `letrec-syntax` expression with macros whose keywords are the keywords, bound to the specified transformers. Each binding of a *keyword* symbol has the *transformer* as well as the *body* within its scope, so the transformers can transcribe expressions into uses of the macros introduced by the `letrec-syntax` expression.

```
(letrec-syntax
  ((my-or (syntax-rules ()
            ((my-or) #f)
            ((my-or e) e)
            ((my-or e1 e2 ...)
              (let ((temp e1))
                (if temp temp (my-or e2 ...)))))))
  (let ((x #f)
        (y 7)
        (temp 8)
        (let odd?)
        (if even?))
    (my-or x (let temp) (if y) y)))
⇒ 7
```

12.5 Iteration

(do ((*variable init step*) ...)
(*test res* ...)
***command* ...)** syntax

A `do` expression is an iteration construct. It specifies a set of variables to be bound, how they are to be initialized at the start, and how they are to be updated on each iteration. When a termination condition *test* is met (i.e. it evaluates to `#t`), the loop exits after evaluating the *res* expressions.

A `do` expression is evaluated as follows: The *init* expressions are evaluated, the variables are bound to fresh locations, the results of the *init* expressions are stored in the bindings of the variables, and then the iteration phase begins.

Each iteration begins by evaluating *test*. If the result is false, then the *command* expressions are evaluated in order, the *step* expressions are evaluated in some unspecified order, the variables are bound to fresh locations, the results of the *step* expressions are stored in the bindings of the variables, and the next iteration begins.

If *test* evaluates to `#t`, then the *res* expressions are evaluated from left to right and the values of the last *res* expression are returned. If no *res* expressions are present, then the `do` expression evaluates to void.

The scope of the binding of a variable consists of the entire `do` expression except for the *init* expressions. It is an error for a variable to appear more than once in the list of `do` variables. A *step* can be omitted, in which case the effect is the same as if `(` *variable init variable* `)` had been written instead of `(` *variable init* `)`.

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

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

13 LispKit Core

Library `(lispkit core)` provides a foundational API for

- primitive procedures,
- definitions (including an import mechanism),
- procedures (including support for optional arguments and tagged procedures),
- delayed execution,
- multiple return values,
- symbols,
- booleans,
- environments,
- syntax errors, and
- loading source files and support for conditional compilation.

13.1 Primitives

(eval *expr*) procedure
(eval *expr env*)

If *expr* is an expression, it is evaluated in the specified environment *env* and its values are returned. If it is a definition, the specified identifiers are defined in the specified environment, provided the environment is not immutable. Should *env* not be provided, the global interaction environment is used.

(apply *proc arg1 ... args*) procedure
The `apply` procedure calls *proc* with the elements of the list `(append (list arg1 ...) args)` as the actual arguments.

(equal? *obj1 obj2*) procedure
The `equal?` procedure, when applied to pairs, vectors, strings and bytevectors, recursively compares them, returning `#t` when the unfoldings of its arguments into possibly infinite trees are equal (in the sense of `equal?`) as ordered trees, and `#f` otherwise. It returns the same as `eqv?` when applied to booleans, symbols, numbers, characters, ports, procedures, and the empty list. If two objects are `eqv?` , they must be `equal?` as well. In all other cases, `equal?` may return either `#t` or `#f` . Even if its arguments are circular data structures, `equal?` must always terminate. As a rule of thumb, objects are generally `equal?` if they print the same.

(eqv? *obj1 obj2*) procedure
The `eqv?` procedure defines a useful equivalence relation on objects. It returns `#t` if *obj1* and *obj2* are regarded as the same object.

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

- *obj1* and *obj2* are both `#t` or both `#f`
- *obj1* and *obj2* are both symbols and are the same symbol according to the `symbol=?` procedure
- *obj1* and *obj2* are both exact numbers and are numerically equal in the sense of `=`
- *obj1* and *obj2* are both inexact numbers such that they are numerically equal in the sense of `=` , and they yield the same results in the sense of `eqv?` when passed as arguments to any other procedure

that can be defined as a finite composition of Scheme's standard arithmetic procedures, provided it does not result in a NaN value
- *obj1* and *obj2* are both characters and are the same character according to the `char=?` procedure
- *obj1* and *obj2* are both the empty list
- *obj1* and *obj2* are both a pair and `car` and `cdr` of both pairs are the same in the sense of `eqv?`
- *obj1* and *obj2* are ports, vectors, hashtables, bytevectors, records, or strings that denote the same location in the store
- *obj1* and *obj2* are procedures whose location tags are equal

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

- *obj1* and *obj2* are of different types
- one of *obj1* and *obj2* is `#t` but the other is `#f`
- *obj1* and *obj2* are symbols but are not the same symbol according to the `symbol=?` procedure
- one of *obj1* and *obj2* is an exact number but the other is an inexact number
- *obj1* and *obj2* are both exact numbers and are numerically unequal in the sense of `=`
- *obj1* and *obj2* are both inexact numbers such that either they are numerically unequal in the sense of `=`, or they do not yield the same results in the sense of `eqv?` when passed as arguments to any other procedure that can be defined as a finite composition of Scheme's standard arithmetic procedures, provided it does not result in a NaN value. As an exception, the behavior of `eqv?` is unspecified when both *obj1* and *obj2* are NaN.
- *obj1* and *obj2* are characters for which the `char=?` procedure returns `#f`
- one of *obj1* and *obj2* is the empty list but the other is not
- *obj1* and *obj2* are both a pair but either `car` or `cdr` of both pairs are not the same in the sense of `eqv?`
- *obj1* and *obj2* are ports, vectors, hashtables, bytevectors, records, or strings that denote distinct locations
- *obj1* and *obj2* are procedures that would behave differently (i.e. return different values or have different side effects) for some arguments

(eq? *obj1 obj2*) procedure

The `eq?` procedure is similar to `eqv?` except that in some cases it is capable of discerning distinctions finer than those detectable by `eqv?`. It always returns `#f` when `eqv?` also would, but returns `#f` in some cases where `eqv?` would return `#t`. On symbols, booleans, the empty list, pairs, and records, and also on non-empty strings, vectors, and bytevectors, `eq?` and `eqv?` are guaranteed to have the same behavior.

(quote *datum*) syntax

`(quote` *datum* `)` evaluates to *datum*. *datum* can be any external representation of a LispKit object. This notation is used to include literal constants in LispKit code. `(quote` *datum* `)` can be abbreviated as '*datum*. The two notations are equivalent in all respects. Numerical constants, string constants, character constants, vector constants, bytevector constants, and boolean constants evaluate to themselves. They need not be quoted.

(quasiquote *template*) syntax

Quasiquote expressions are useful for constructing a list or vector structure when some but not all of the desired structure is known in advance. If no commas appear within *template*, the result of evaluating `(quasiquote` *template* `)` is equivalent to the result of evaluating `(quote` *template* `)`. If a comma appears within *template*, however, the expression following the comma is evaluated ("unquoted") and its result is inserted into the structure instead of the comma and the expression. If a comma appears followed without intervening whitespace by `@`, then it is an error if the following expression does not evaluate to a list; the opening and closing parentheses of the list are then "stripped away" and the elements of the list are inserted in place of the `,@` expression sequence. `,@` normally appears only within a list or vector.

Quasiquote expressions can be nested. Substitutions are made only for unquoted components appearing at the same nesting level as the outermost quasiquote. The nesting level increases by one inside each successive quasiquotation, and decreases by one inside each unquotation. Comma corresponds to form `unquote`, `,@` corresponds to form `unquote-splicing`.

13.2 Definitions

(define *var expr*) syntax
(define *var expr doc*)
(define (*f arg* …) *expr* …)
(define (*f arg* …) *doc expr* …)

`define` is used to define variables. At the outermost level of a program, a definition `(define` *var expr* `)` has essentially the same effect as the assignment expression `(set!` *var expr* `)` if variable *var* is bound to a non-syntax value. However, if *var* is not bound, or is a syntactic keyword, then the definition will bind *var* to a new location before performing the assignment, whereas it would be an error to perform a `set!` on an unbound variable.

The form `(define (` *f arg* … `)` *expr* `)` defines a function *f* with arguments *arg* … and body *expr*. It is equivalent to `(define` *f* `(lambda (` *arg* … `)` *expr* `))`.

The parameter *doc* is a string literal defining documentation for variable *var*. It can be accessed, for instance, by using the procedure `environment-documentation` on the environment in which the variable was bound.

```
(define pi 3.141 "documentation for `pi`")
(environment-documentation (interaction-environment) 'pi)
⇒ "documentation for `pi`"
```

(define-values *var expr*) syntax
(define-values (*var* …) *expr*)
(define-values (*var doc* …) *expr*)

`define-values` creates multiple definitions *var* … from a single expression *expr* returning multiple values. It is allowed wherever `define` is allowed.

expr is evaluated, and the variables *var* … are bound to the return values in the same way that the formal arguments in a `lambda` expression are matched to the actual arguments in a procedure call.

It is an error if a variable *var* appears more than once in *var* ….

```
(define-values (x y) (integer-sqrt 17))
(list x y)                          ⇒ (4 1)
(define-values vs (values 'a 'b 'c))
vs                                  ⇒ (a b c)
```

The parameter *doc* is an optional string literal defining documentation for variable *var*. It directly follows the identifier name.

(define-syntax *keyword transformer*) syntax
(define-syntax *keyword doc transformer*)

Syntax definitions have the form `(define-syntax` *keyword transformer* `)`. *keyword* is an identifier, and *transformer* is an instance of `syntax-rules`. Like variable definitions, syntax definitions can appear at the outermost level or nested within a body.

If the `define-syntax` occurs at the outermost level, then the global syntactic environment is extended by binding the *keyword* to the specified *transformer*, but previous expansions of any global binding for *keyword* remain unchanged. Otherwise, it is an internal syntax definition, and is local to the "body" in which it is defined. Any use of a syntax keyword before its corresponding definition is an error.

Macros can expand into definitions in any context that permits them. However, it is an error for a definition to define an identifier whose binding has to be known in order to determine the meaning of the definition itself, or of any preceding definition that belongs to the same group of internal definitions. Similarly, it is an error for an internal definition to define an identifier whose binding has to be known in order to determine the boundary between the internal definitions and the expressions of the body it belongs to.

Here is an example defining syntax for `while` loops. `while` evaluates the body of the loop as long as the predicate is true.

```
(define-syntax while
  (syntax-rules ()
    ((_ pred body ...)
      (let loop () (when pred body ... (loop))))))
```

The parameter *doc* is an optional string literal defining documentation for the keyword *var*:

```
(define-syntax kwote "alternative to quote"
  (syntax-rules ()
    ((kwote exp)
      (quote exp))))
```

(syntax-rules (*literal* ...) *rule* ...) syntax
(syntax-rules *ellipsis* (*literal* ...) *rule* ...)

A *transformer spec* has one of the two forms listed above. It is an error if any of the *literal* ..., or the *ellipsis* symbol in the second form, is not an identifier. It is also an error if syntax rules *rule* are not of the form

- (*pattern template*) .

The *pattern* in a *rule* is a list pattern whose first element is an identifier. In general, a *pattern* is either an identifier, a constant, or one of the following:

- (*pattern* ...)
- (*pattern pattern* ... . *pattern*)
- (*pattern* ... *pattern ellipsis pattern* ...) (*pattern* ... *pattern ellipsis pattern* ... . *pattern*)
- #(*pattern* ...)
- #(*pattern* ... *pattern ellipsis pattern* ...)

A *template* is either an identifier, a constant, or one of the following:

- (*element* ...)
- (*element element* ... . *template*) (*ellipsis template*)
- #(*element* ...)

where an *element* is a *template* optionally followed by an *ellipsis*. An *ellipsis* is the identifier specified in the second form of `syntax-rules`, or the default identifier `...` (three consecutive periods) otherwise.

Here is an example showcasing how `when` can be defined in terms of `if`:

```
(define-syntax when
  (syntax-rules ()
    ((_ c e ...)
      (if c (begin e ...)))))
```

(define-library (*name* …) *declaration* …) syntax

A library definition takes the following form: `(define-library (` *name* … `)` *declaration* … `)`. `(` *name* … `)` is a list whose members are identifiers and exact non-negative integers. It is used to identify the library uniquely when importing from other programs or libraries. It is inadvisable, but not an error, for identifiers in library names to contain any of the characters `|`, `\`, `?`, `*`, `<`, `"`, `:`, `>`, `+`, `[`, `]`, `/`.

A *declaration* is any of:

- `(export` *exportspec* … `)`
- `(export-mutable` *exportspec* … `)`
- `(import` *importset* … `)`
- `(begin` *statement* … `)`
- `(include` *filename* … `)`
- `(include-ci` *filename* … `)`
- `(include-library-declarations` *filename* … `)`
- `(cond-expand` *clause* … `)`

An export declaration specifies a list of identifiers which can be made visible to other libraries or programs. An *exportspec* takes one of the following forms:

- *ident*
- `(rename` *ident1 ident2* `)`

In an *exportspec*, an identifier *ident* names a single binding defined within or imported into the library, where the external name for the export is the same as the name of the binding within the library. A `rename` spec exports the binding defined within or imported into the library and named by *ident1* in each `(` *ident1 ident2* `)` pairing, using *ident2* as the external name. A regular `export` statement exports bindings in a immutable fashion, not allowing binding changes outside of the library. `export-mutable` is a LispKit-specific variant which allows library-external mutations of the exported bindings.

An `import` declaration provides a way to import the identifiers exported by another library. It has the same syntax and semantics as an `import` declaration used in a program or at the read-eval-print loop.

The `begin`, `include`, and `include-ci` declarations are used to specify the body of the library. They have the same syntax and semantics as the corresponding expression types.

The `include-library-declarations` declaration is similar to `include` except that the contents of the file are spliced directly into the current library definition. This can be used, for example, to share the same `export` declaration among multiple libraries as a simple form of library interface.

The `cond-expand` declaration has the same syntax and semantics as the `cond-expand` expression type, except that it expands to spliced-in library declarations rather than expressions enclosed in `begin`.

(set! *var expr*) syntax

Procedure `set!` is used to assign values to variables. *expr* is evaluated, and the resulting value is stored in the location to which variable *var* is bound. It is an error if *var* is not bound either in some region enclosing the `set!` expression or else globally. The result of the `set!` expression is unspecified.

13.3 Importing definitions

(import *importset* …) syntax

An `import` declaration provides a way to import identifiers exported by a library. Each *importset* names a set of bindings from a library and possibly specifies local names for the imported bindings. It takes one of the following forms:

- *libraryname*

- `(only` *importset identifier* ... `)`
- `(except` *importset identifier* ... `)`
- `(prefix` *importset identifier* `)`
- `(rename` *importset* `(` *ifrom ito* `)` ... `)`

In the first form, all of the identifiers in the named library's export clauses are imported with the same names (or the exported names if exported with `rename`). The additional *importset* forms modify this set as follows:

- `only` produces a subset of the given *importset* including only the listed identifiers (after any renaming). It is an error if any of the listed identifiers are not found in the original set.
- `except` produces a subset of the given *importset*, excluding the listed identifiers (after any renaming). It is an error if any of the listed identifiers are not found in the original set.
- `rename` modifies the given *importset*, replacing each instance of *ifrom* with *ito*. It is an error if any of the listed identifiers are not found in the original set.
- `prefix` automatically renames all identifiers in the given *importset*, prefixing each with the specified identifier.

In a program or library declaration, it is an error to import the same identifier more than once with different bindings, or to redefine or mutate an imported binding with a definition or with `set!`, or to refer to an identifier before it is imported. However, a read-eval-print loop will permit these actions.

13.4 Symbols

(symbol? *obj*) procedure
Returns `#t` if *obj* is a symbol, otherwise returns `#f`.

(symbol-interned? *obj*) procedure
Returns `#t` if *obj* is an interned symbol, otherwise returns `#f`.

(gensym) procedure
(gensym *str*)

Returns a new (fresh) symbol whose name consists of prefix *str* followed by a number. If *str* is not provided, "g" is used as a prefix.

(symbol=? *sym* ...) procedure
Returns `#t` if all the arguments are symbols and all have the same names in the sense of `string=?`.

(string->symbol *str*) procedure
Returns the symbol whose name is string *str*. This procedure can create symbols with names containing special characters that would require escaping when written, but does not interpret escapes in its input.

(string->uninterned-symbol *str*) procedure
Returns a new uninterned symbol whose name is *str*. This procedure can create symbols with names containing special characters that would require escaping when written, but does not interpret escapes in its input.

(symbol->string *sym*) procedure
Returns the name of symbol *sym* as a string, but without adding escapes.

13.5 Booleans

The standard boolean objects for true and false are written as `#t` and `#f`. Alternatively, they can be written `#true` and `#false`, respectively. What really matters, though, are the objects that the Scheme conditional expressions (`if` , `cond` , `and` , `or` , `when` , `unless` , `do`) treat as true or false. The phrase a "true value" (or sometimes just "true") means any object treated as true by the conditional expressions, and the phrase "a false value" (or "false") means any object treated as false by the conditional expressions.

Of all the Scheme values, only `#f` counts as false in conditional expressions. All other Scheme values, including `#t` , count as true. Boolean literals evaluate to themselves, so they do not need to be quoted in programs.

(boolean? *obj*) procedure

The `boolean?` predicate returns `#t` if *obj* is either `#t` or `#f` and returns `#f` otherwise.

```
(boolean? #f)    ⇒  #t
(boolean? 0)     ⇒  #f
(boolean? '())   ⇒  #f
```

(boolean=? *obj1 obj2 ...*) procedure

Returns `#t` if all the arguments are booleans and all are `#t` or all are `#f` .

(and *test ...*) syntax

The *test* ... expressions are evaluated from left to right, and if any expression evaluates to `#f` , then `#f` is returned. Any remaining expressions are not evaluated. If all the expressions evaluate to true values, the values of the last expression are returned. If there are no expressions, then `#t` is returned.

```
(and (= 2 2) (> 2 1))  ⇒  #t
(and (= 2 2) (< 2 1))  ⇒  #f
(and 12 'c '(f g))     ⇒  (f g)
(and)                  ⇒  #t
```

(or *test ...*) syntax

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

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

(not *obj*) procedure

The `not` procedure returns `#t` if *obj* is false, and returns `#f` otherwise.

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

(opt *pred obj*) procedure
(opt *pred obj failval*)

The `opt` procedure returns *failval* if *obj* is `#f` . If *obj* is not `#f` , `opt` applies predicate *pred* to *obj* and returns the result of this function application. If *failval* is not provided, `#t` is used as a default. This function is useful to verify a given predicate *pred* for an optional value *obj*.

13.6 Procedures

(procedure? *obj*) procedure

Returns `#t` if *obj* is a procedure; otherwise, it returns `#f`.

(thunk? *obj*) procedure

Returns `#t` if *obj* is a procedure which accepts zero arguments; otherwise, it returns `#f`.

(procedure-of-arity? *obj n*) procedure

Returns `#t` if *obj* is a procedure that accepts *n* arguments; otherwise, it returns `#f`. This is equivalent to:

```
(and (procedure? obj)
     (procedure-arity-includes? obj n))
```

(lambda *(arg1 ...) expr ...*) syntax
(lambda *(arg1 ... . rest) expr ...*)
(lambda *rest expr ...*)
(λ *(arg1 ...) expr ...*)
(λ *(arg1 ... . rest) expr ...*)
(λ *rest expr ...*)

A lambda expression evaluates to a procedure. The environment in effect when the lambda expression was evaluated is remembered as part of the procedure. When the procedure is later called with some actual arguments, the environment in which the lambda expression was evaluated will be extended by binding the variables in the formal argument list *arg1* ... to fresh locations, and the corresponding actual argument values will be stored in those locations. Next, the expressions in the body of the lambda expression will be evaluated sequentially in the extended environment. The results of the last expression in the body will be returned as the results of the procedure call.

(case-lambda *(formals expr ...)* ...) syntax
(case-λ *(formals expr ...)* ...)

A case-lambda expression evaluates to a procedure that accepts a variable number of arguments and is lexically scoped in the same manner as a procedure resulting from a lambda expression. When the procedure is called, the first clause for which the arguments agree with *formals* is selected, where agreement is specified as for *formals* of a lambda expression. The variables of *formals* are bound to fresh locations, the values of the arguments are stored in those locations, the expressions in the body are evaluated in the extended environment, and the results of the last expression in the body is returned as the results of the procedure call. It is an error for the arguments not to agree with *formals* of any clause.

Here is an example showing how to use `case-lambda` for defining a simple accumulator:

```
(define (make-accumulator n)
  (case-lambda
    (()  n)
    ((m) (set! n (+ n m)) n)))
(define a (make-accumulator 1))
(a)                              ⇒ 1
(a 5)                            ⇒ 6
(a)                              ⇒ 6
```

(thunk *expr ...*) syntax

Returns a procedure accepting no arguments and evaluating *expr* ..., returning the result of the last expression being evaluated as the result of a procedure call. `(thunk expr ...)` is equivalent to `(lambda () expr ...)`.

(thunk* *expr* ...) syntax

Returns a procedure accepting an arbitrary amount of arguments and evaluating *expr* ..., returning the result of the last expression being evaluated as the result of a procedure call. `(thunk* expr ...)` is equivalent to `(lambda args expr ...)`.

(procedure-name *proc*) procedure

Returns the name of procedure *proc* as a string, or `#f` if *proc* does not have a name.

(procedure-arity *proc*) procedure

Returns a value representing the arity of procedure *proc*, or returns `#f` if no arity information is available for *proc*.

If `procedure-arity` returns a fixnum *k*, then procedure *proc* accepts exactly *k* arguments and applying *proc* to some number of arguments other than *k* will result in an arity error.

If `procedure-arity` returns an "arity-at-least" object *a*, then procedure *proc* accepts `(arity-at-least-value a)` or more arguments and applying *proc* to some number of arguments less than `(arity-at-least-value a)` will result in an arity error.

If `procedure-arity` returns a list, then procedure *proc* accepts any of the arities described by the elements of the list. Applying *proc* to some number of arguments not described by an element of the list will result in an arity error.

(procedure-arity-range *proc*) procedure

Returns the smallest arity range in form of a pair *(min . max)* such that if *proc* is provided *n* arguments with $n < min$ or $n > max$, an arity error gets raised.

```
(procedure-arity-range (lambda () 3))      ⇒ (0 . 0)
(procedure-arity-range (lambda (x) x))     ⇒ (1 . 1)
(procedure-arity-range (lambda x x))       ⇒ (0 . #f)
(procedure-arity-range (lambda (x . y) x)) ⇒ (1 . #f)
```

(procedure-arity-includes? *proc k*) procedure

Returns `#t` if procedure *proc* can accept *k* arguments and `#f` otherwise. If this procedure returns `#f`, applying *proc* to *k* arguments will result in an arity error.

(arity-at-least? *obj*) procedure

Returns `#t` if *obj* is an "arity-at-least" object and `#f` otherwise.

(arity-at-least-value *obj*) procedure

Returns a fixnum denoting the minimum number of arguments required by the given "arity-at-least" object *obj*.

13.7 Procedures with optional arguments

(opt-lambda *(arg1 ... arg1 bind1 ... bindm) expr ...*) syntax
(opt-lambda *(arg1 ... argn bind1 ... bindm . rest) expr ...*)
(opt-lambda *rest expr ...*)

An `opt-lambda` expression evaluates to a procedure and is lexically scoped in the same manner as a procedure resulting from a `lambda` expression. When the procedure is later called with actual arguments, the variables are bound to fresh locations, the values of the corresponding arguments are stored in those locations, the body *expr* ... is evaluated in the extended environment, and the result of the last body expression is returned as the result of the procedure call.

Formal arguments *argi* are required arguments. Arguments *bindi* are optional. They are of the form `(var init)`, with *var* being a symbol and *init* an initialization expression which gets evaluated and assigned to *var* if the argument is not provided when the procedure is called. It is a syntax violation if the same variable appears more than once among the variables.

A procedure created with the first syntax of `opt-formals` takes at least *n* arguments and at most $n + m$ arguments. A procedure created with the second syntax of `opt-formals` takes *n* or more arguments. If the procedure is called with fewer than $n + m$ (but at least *n* arguments), the missing actual arguments are substituted by the values resulting from evaluating the corresponding _init_s. The corresponding _init_s are evaluated in an unspecified order in the lexical environment of the `opt-lambda` expression when the procedure is called.

(opt*-lambda *(arg1 … arg1 bind1 … bindm) expr …*) syntax
(opt*-lambda *(arg1 … argn bind1 … bindm . rest) expr …*)
(opt*-lambda *rest expr …*)

Similar to syntax `opt-lambda` except that the initializers of optional arguments *bindi* corresponding to missing actual arguments are evaluated sequentially from left to right. The region of the binding of a variable is that part of the `opt*-lambda` expression to the right of it or its binding.

(define-optionals (*f arg … bind …*) *expr …*) syntax
(define-optionals (*f arg … bind … . rest*) *expr …*)

`define-optionals` is operationally equivalent to `(define f (opt-lambda (arg ... bind ...) expr ...))` or `(define f (opt-lambda (arg ... bind ... . rest) expr ...))` if the second syntactical form is used.

(define-optionals* (*f arg … bind …*) *expr …*) syntax
(define-optionals* (*f arg … bind … . rest*) *expr …*)

`define-optionals*` is operationally equivalent to `(define f (opt*-lambda (arg ... bind ...) expr ...))` or `(define f (opt*-lambda (arg ... bind ... . rest) expr ...))` if the second syntactical form is used.

13.8 Tagged procedures

LispKit allows a data object to be associated with a procedure. Such data objects are called *tags*, procedures with an associated tag are called *tagged procedures*. The tag of a procedure is immutable and defined at the time a procedure gets created. It is defined at procedure creation time and can later be retrieved without calling the procedure.

(procedure/tag? *obj*) procedure
Returns `#t` if *obj* is a tagged procedure and `#f` otherwise. Procedures are tagged procedures if they were created either via `lambda/tag` or `case-lambda/tag`.

(procedure-tag *proc*) procedure
Returns the tag of the tagged procedure *proc*. It is an error if *proc* is not a tagged procedure.

(lambda/tag *tag (arg1 …) expr …*) syntax
(lambda/tag *tag (arg1 … . rest) expr …*)
(lambda/tag *tag rest expr …*)

A `lambda/tag` expression evaluates to a tagged procedure. First, *tag* is evaluated to obtain the tag value for the procedure. Then, the tagged procedure itself gets created for the given formal arguments. The procedure is lexically scoped in the same manner as a procedure resulting from a `lambda` expression. When the procedure is called, it behaves as if constructed by a `lambda` expression with the same formal arguments and body.

(case-lambda/tag *tag (formals expr ...) ...*) syntax

A `case-lambda/tag` expression evaluates to a tagged procedure. First, *tag* is evaluated to obtain the tag value for the procedure. Then, the tagged procedure itself gets created, accepting a variable number of arguments. It is lexically scoped in the same manner as a procedure resulting from a `lambda` expression. When the procedure is called, it behaves as if it was constructed by a `case-lambda` expression with the same formal arguments and bodies.

13.9 Delayed execution

LispKit provides *promises* to delay the execution of code. Built on top of *promises* are *streams*. Streams are similar to lists, except that the tail of a stream is not computed until it is de-referenced. This allows streams to be used to represent infinitely long lists. Library `(lispkit core)` only defines procedures for *streams* equivalent to *promises*. Library `(lispkit stream)` provides all the list-like functionality.

(promise? *obj*) procedure

The `promise?` procedure returns `#t` if argument *obj* is a promise, and `#f` otherwise.

(make-promise *obj*) procedure
(eager *obj*)

The `make-promise` procedure returns a promise which, when forced, will return *obj*. It is similar to `delay`, but does not delay its argument: it is a procedure rather than syntax. If *obj* is already a promise, it is returned. `eager` represents the same procedure like `make-promise`.

(delay *expr*) syntax

The `delay` construct is used together with the procedure `force` to implement lazy evaluation or "call by need". `(delay expr)` returns an object called a promise which, at some point in the future, can be asked (by the `force` procedure) to evaluate *expr*, and deliver the resulting value.

(delay-force *expr*) syntax
(lazy *expr*)

The expression `(delay-force expr)` is conceptually similar to `(delay (force expr))`, with the difference that forcing the result of `delay-force` will in effect result in a tail call to `(force expr)`, while forcing the result of `(delay (force expr))` might not. Thus iterative lazy algorithms that might result in a long series of chains of delay and force can be rewritten using `delay-force` to prevent consuming unbounded space during evaluation. `lazy` represents the same procedure like `delay-force`.

(force *promise*) procedure

The `force` procedure forces the value of a promise created by `delay`, `delay-force`, or `make-promise`. If no value has been computed for the promise, then a value is computed and returned. The value of the promise must be cached (or "memoized") so that if it is forced a second time, the previously computed value is returned. Consequently, a delayed expression is evaluated using the parameter values and exception handler of the call to `force` which first requested its value. If *promise* is not a promise, it may be returned unchanged.

```
(force (delay (+ 1 2)))       ⇒  3

(let ((p (delay (+ 1 2))))
  (list (force p) (force p)))  ⇒ (3 3)

(define integers
  (letrec ((next (lambda (n)
                   (delay (cons n (next (+ n 1)))))))
    (next 0)))
```

```
(define head
  (lambda (stream) (car (force stream))))
(define tail
  (lambda (stream) (cdr (force stream))))

(head (tail (tail integers)))  ⇒  2
```

The following example is a mechanical transformation of a lazy stream-filtering algorithm into Scheme. Each call to a constructor is wrapped in `delay`, and each argument passed to a deconstructor is wrapped in `force`. The use of `(delay-force ...)` instead of `(delay (force ...))` around the body of the procedure ensures that an ever-growing sequence of pending promises does not exhaust available storage, because `force` will, in effect, force such sequences iteratively.

```
(define (stream-filter p? s)
  (delay-force
    (if (null? (force s))
        (delay '())
        (let ((h (car (force s)))
              (t (cdr (force s))))
          (if (p? h)
              (delay (cons h (stream-filter p? t)))
              (stream-filter p? t))))))

(head (tail (tail (stream-filter odd? integers))))  ⇒  5
```

The following examples are not intended to illustrate good programming style, as `delay`, `force`, and `delay-force` are mainly intended for programs written in the functional style. However, they do illustrate the property that only one value is computed for a promise, no matter how many times it is forced.

```
(define count 0)
(define p
  (delay (begin (set! count (+ count 1))
                (if (> count x) count (force p)))))
(define x 5)
p                              ⇒  a promise
(force p)                      ⇒  6
p                              ⇒  a promise
(begin (set! x 10) (force p))  ⇒  6
```

(stream? *obj*) procedure

The `stream?` procedure returns `#t` if argument *obj* is a stream, and `#f` otherwise. If *obj* is a stream, `stream?` does not force its promise. If `(stream? obj)` is `#t`, then one of `(stream-null? obj)` and `(stream-pair? obj)` will be `#t` and the other will be `#f`; if `(stream? obj)` is `#f`, both `(stream-null? obj)` and `(stream-pair? obj)` will be `#f`.

(make-stream *obj*) procedure
(stream-eager *obj*)

The `make-stream` procedure returns a stream which, when forced, will return *obj*. It is similar to `stream-delay`, but does not delay its argument: it is a procedure rather than syntax. If *obj* is already a stream, it is returned. `stream-eager` represents the same procedure like `make-stream`.

(stream-delay *expr*) syntax

The `stream-delay` syntax is used together with procedure `stream-force` to implement lazy evaluation or "call by need". `(stream-delay expr)` returns an object called a stream which, at some point in the future, can be asked (by the `stream-force` procedure) to evaluate *expr*, and deliver the resulting value.

(stream-delay-force *expr*) syntax
(stream-lazy *expr*)

The expression `(stream-delay-force expr)` is conceptually similar to `(stream-delay (stream-force expr))`, with the difference that forcing the result of `stream-delay-force` will in effect result in a tail call to `(stream-force expr)`, while forcing the result of `(stream-delay (stream-force expr))` might not. Thus iterative lazy algorithms that might result in a long series of chains of delay and force can be rewritten using `stream-delay-force` to prevent consuming unbounded space during evaluation. `stream-lazy` represents the same procedure like `stream-delay-force`.

13.10 Multiple values

(values *obj* ...) procedure
Delivers all of its arguments to its continuation. The `values` procedure might be defined as follows:

```
(define (values . things)
  (call-with-current-continuation
    (lambda (cont) (apply cont things))))
```

(call-with-values *producer consumer*) procedure
Calls its *producer* argument with no arguments and a continuation that, when passed some values, calls the *consumer* procedure with those values as arguments. The continuation for the call to *consumer* is the continuation of the call to `call-with-values`.

```
(call-with-values
  (lambda () (values 4 5))
  (lambda (a b) b))
⇒  5
(call-with-values * -)
⇒  -1
```

(apply-with-values *proc vals*) procedure
`apply-with-values` calls procedure *proc* with *vals* as its arguments and returns the corresponding result. *vals* might refer to multiple values created via procedure `values`. This is a LispKit-specific procedure that relies on multiple return values being represented by a container object.

13.11 Environments

Environments are first-class objects which associate identifiers (symbols) with values. Environments are used implicitly by the LispKit compiler and runtime system, but library `(lispkit core)` also provides an API allowing systems to manipulate and use environments programmatically.

For instance, when a top-level variable gets created with `define`, the name/value association for that variable is added to the "top-level" environment. The LispKit compiler implicitly creates environments other than the top-level environment, for example, when compiling and executing libraries.

There are several types of bindings that can occur within an environment. A *variable binding* associates a value with an identifier. This is the most common type of binding. In addition to variable bindings, environments can have *keyword bindings*. A keyword binding associates an identifier with a macro transformer (usually via `syntax-rules`). There are also *unassigned* bindings referring to bindings without a known value.

(environment? *obj*) procedure

Returns `#t` if *obj* is an environment. Otherwise, it returns `#f`.

(interaction-environment? *obj*) procedure

Returns `#t` if *obj* is an interaction environment, i.e. a mutable environment in which expressions entered by the user into a read-eval-print loop are being evaluated. Otherwise, procedure `interaction-environment?` returns `#f`.

(custom-environment? *obj*) procedure

Returns `#t` if *obj* is a custom environment, i.e. an environment that was programmatically constructed. Otherwise, predicate `custom-environment?` returns `#f`.

(the-environment) syntax

Special form `the-environment` returns the current top-level environment. If there is none, `the-environment` returns `#f`.

Here is an example how one can print the names bound at compile-time:

```
(define-library (foo)
  (import (only (lispkit core) the-environment environment-bound-names)
          (only (lispkit port) display newline))
  (begin
    (display "bound = ")
    (display (environment-bound-names (the-environment)))
    (newline)))
(import (foo))
⇒
bound = (display the-environment newline environment-bound-names)
```

(environment *list1* ...) procedure

This procedure returns an environment that results by starting with an empty environment and then importing each list, considered as an import set, into it. The bindings of the environment represented by the specifier are immutable, as is the environment itself.

(environment-bound-names *env*) procedure

Returns a list of the symbols that are bound by environment *env*.

(environment-bindings *env*) procedure

Returns a list of the bindings of environment *env*. Each element of this list takes one of two forms: the form *(name)* indicates that *name* is bound but unassigned, while *(name obj)* indicates that *name* is bound to value *obj*.

(environment-bound? *env ident*) procedure

Returns `#t` if symbol *ident* is bound in environment *env*; otherwise returns `#f`.

(environment-lookup *env ident*) procedure

Returns the value to which symbol *ident* is bound in environment *env*. This procedure throws an error if *ident* is not bound to a value in *env*.

(environment-assignable? *env ident*) procedure

Symbol *ident* must be bound in environment *env*. Procedure `environment-assignable?` returns `#t` if the binding of *ident* may be modified.

(environment-assign! *env ident obj*) procedure

Symbol *ident* must be bound in environment *env* and must be assignable. Procedure `environment-assign!` modifies the binding to have *obj* as its value.

(environment-definable? *env ident*) procedure

Predicate `environment-definable?` returns `#t` if symbol *ident* is definable in environment *env*, and

`#f` otherwise. Currently, interaction environments and custom environments allow for identifiers to be defined. For all other types of environments, this predicate returns `#f` independent of *ident*.

(environment-define *env ident obj*) procedure
Defines *ident* to be bound to *obj* in environment *env*. This procedure signals an error if *ident* is not definable in *env*.

(environment-define-syntax *env ident transf*) procedure
Defines *ident* to be a keyword bound to macro transformer *transf* (typically expressed in terms of `syntax-rules`) in environment *env*. This procedure signals an error if *ident* is not definable in environment *env*.

(environment-import *env ident importset*) procedure
Imports the identifiers exported by a library and specified via an import set *importset* into the environment *env*. The procedure fails if the type of environment does not allow identifiers to be defined programmatically.

(environment-documentation *env ident*) procedure
Returns the documentation associated with the identifier *ident* in environment *env* as a string. This procedure returns `#f` if *ident* is not associated with any documentation.

(environment-assign-documentation! *env ident str*) procedure
Assigns the documentation string *str* to identifier *ident* in environment *env*.

(scheme-report-environment *version*) procedure
If *version* is equal to 5, corresponding to R5RS, *scheme-report-environment* returns an environment that contains only the bindings defined in the R5RS library.

(null-environment *version*) procedure
If *version* is equal to 5, corresponding to R5RS, the *null-environment* procedure returns an environment that contains only the bindings for all syntactic keywords defined in the R5RS library.

(interaction-environment) procedure
This procedure returns a mutable environment which is the environment in which expressions entered by the user into a read-eval-print loop are evaluated. This is typically a superset of bindings from *(lispkit base)*.

13.12 Loading source files

(load *filename*) procedure
(load *filename environment*)

`load` reads a source file specified by *filename* and executes it in the given *environment*. If no environment is specified, the current *interaction environment* is used, which can be accessed via `(interaction-environment)`. Execution of the file consists of reading expressions and definitions from the file, compiling them, and evaluating them sequentially in the environment. `load` returns the result of evaluating the last expression or definition from the file. During compilation, the special form `source-directory` can be used to access the directory in which the executed file is located.

It is an error if *filename* is not a string. If *filename* is not an absolute file path, LispKit will try to find the file in a predefined set of directories, such as the default search path. If no file name suffix, also called *path extension*, is provided, the system will try to determine the right suffix. For instance, `(load "Prelude")` will find the prelude file, determine its suffix and load and execute the file.

(load-program *filename*) procedure
`load-program` reads a source file specified by *filename* and executes it in a new empty *environment*. Execution of the file consists of reading expressions and definitions from the file, compiling them, and

evaluating them sequentially. `load-program` returns the result of evaluating the last expression or definition from the file.

It is an error if *filename* is not a string. If *filename* is not an absolute file path, LispKit will try to find the file in a predefined set of directories, such as the default search path. If no file name suffix is provided, the system will try to determine the right suffix.

13.13 Conditional and inclusion compilation

(cond-expand *ce-clause1 ce-clause2* …) syntax

The `cond-expand` expression type provides a way to statically expand different expressions depending on the implementation. A *ce-clause* takes the following form:

(*featurerequirement expression* …)

The last clause can be an "else clause," which has the form:

(else *expression* …)

A *featurerequirement* takes one of the following forms: - *featureidentifier* - `(library` *name* `)` - `(and` *featurerequirement* … `)` - `(or` *featurerequirement* … `)` - `(not` *featurerequirement* `)`

LispKit maintains a list of feature identifiers which are present, as well as a list of libraries which can be imported. The value of a *featurerequirement* is determined by replacing each *featureidentifier* and `(library` *name* `)` with `#t`, and all other feature identifiers and library names with `#f`, then evaluating the resulting expression as a Scheme boolean expression under the normal interpretation of `and`, `or`, and `not`.

A `cond-expand` is then expanded by evaluating the *featurerequirements* of successive *ce-clause* in order until one of them returns `#t`. When a true clause is found, the corresponding *expression* … are expanded to a `begin`, and the remaining clauses are ignored. If none of the listed *featurerequirement* evaluates to `#t`, then if there is an "else" clause, its *expression* … are included. Otherwise, the behavior of the `cond-expand` is unspecified. Unlike `cond`, `cond-expand` does not depend on the value of any variables. The exact features provided are defined by the implementation, its environment and host platform.

LispKit supports the following *featureidentifier*:

- `lispkit`
- `r7rs`
- `ratios`
- `complex`
- `syntax-rules`
- `little-endian`
- `big-endian`
- `dynamic-loading`
- `modules`
- `32bit`
- `64bit`
- `macos`
- `macosx`
- `ios`
- `linux`
- `i386`
- `x86-64`
- `arm64`

- arm

(include *str1 str2 ...*) syntax
(include-ci *str1 str2 ...*)

Both `include` and `include-ci` take one or more filenames expressed as string literals, apply an implementation-specific algorithm to find corresponding files, read the contents of the files in the specified order as if by repeated applications of read, and effectively replace the `include` or `include-ci` expression with a `begin` expression containing what was read from the files. The difference between the two is that `include-ci` reads each file as if it began with the `#!fold-case` directive, while `include` does not.

13.14 Syntax errors

(syntax-error *message args ...*) syntax
`syntax-error` behaves similarly to `error` except that implementations with an expansion pass separate from evaluation should signal an error as soon as `syntax-error` is expanded. This can be used as a `syntax-rules` *template* for a *pattern* that is an invalid use of the macro, which can provide more descriptive error messages.

message is a string literal, and *args* ... are arbitrary expressions providing additional information. Applications cannot count on being able to catch syntax errors with exception handlers or guards.

```
(define-syntax simple-let
  (syntax-rules ()
    ((_ (head ... ((x . y) val) . tail) body1 body2 ...)
      (syntax-error "expected an identifier but got" (x . y)))
    ((_ ((name val) ...) body1 body2 ...)
      ((lambda (name ...) body1 body2 ...) val ...))))
```

13.15 Utilities

(void) procedure
Performs no operation and returns nothing. This is often useful as a placeholder or whenever a no-op statement is needed.

(void? *obj*) procedure
Returns `#t` if *obj* is the “void” value (i.e. no value); returns `#f` otherwise.

(identity *obj*) procedure
The identity function is always returning its argument *obj*.

14 LispKit Crypto

Library `(lispkit crypto)` provides an API for cryptographic operations, including message digests, the creation and management of secure keys, and cryptographic algorithms for encrypting, decrypting, signing, and verifying messages.

The following sample code illustrates how `(lispkit crypto)` can be used to implement public key cryptography:

```
;; Define a UTF-encoded message
(define msg (string->utf8 "This is a secret message!"))
;; Create private and public keys
(define privkey (make-private-key 'rsa))
(define pubkey (public-key privkey))
;; Encrypt and decrypt the message
(define encr (encrypt pubkey 'rsa-encryption-pkcs1 msg))
(define decr (decrypt privkey 'rsa-encryption-pkcs1 encr))
(utf8->string decr)
⇒ "This is a secret message!"
```

14.1 Hash functions

Library `(lispkit crypto)` provides a number of cryptographic hash functions, sometimes also called *message digest functions*. A cryptographic hash function maps variable-length, potentially long sequences of bytes to fixed length, relatively short hashes, also called *digests*.

(md5 *bytevector*) procedure
(md5 *bytevector start*)
(md5 *bytevector start end*)

Implementation of the MD5 message-digest algorithm. Computes a 128-bit hash for the bytes of *bytevector* between index *start* (including) and *end* (excluding) and returns the result as a bytevector. If *end* is not provided, it is assumed to be the length of *bytevector*. If *start* is not provided, it is assumed to be 0.

(sha1 *bytevector*) procedure
(sha1 *bytevector start*)
(sha1 *bytevector start end*)

Implementation of the "Secure Hash Algorithm 1". Computes a 160-bit hash for the bytes of *bytevector* between index *start* (including) and *end* (excluding) and returns the result as a bytevector. If *end* is not provided, it is assumed to be the length of *bytevector*. If *start* is not provided, it is assumed to be 0.

(sha256 *bytevector*) procedure
(sha256 *bytevector start*)
(sha256 *bytevector start end*)

Implementation of the "Secure Hash Algorithm 2" with a 256-bit digest. Computes a 256-bit hash for the bytes of *bytevector* between index *start* (including) and *end* (excluding) and returns the result as a bytevector. If *end* is not provided, it is assumed to be the length of *bytevector*. If *start* is not provided, it is assumed to be 0.

(sha384 *bytevector*) procedure
(sha384 *bytevector start*)
(sha384 *bytevector start end*)

Implementation of the "Secure Hash Algorithm 2" with a 384-bit digest. Computes a 384-bit hash for the bytes of *bytevector* between index *start* (including) and *end* (excluding) and returns the result as a bytevector. If *end* is not provided, it is assumed to be the length of *bytevector*. If *start* is not provided, it is assumed to be 0.

(sha512 *bytevector*) procedure
(sha512 *bytevector start*)
(sha512 *bytevector start end*)

Implementation of the "Secure Hash Algorithm 2" with a 512-bit digest. Computes a 512-bit hash for the bytes of *bytevector* between index *start* (including) and *end* (excluding) and returns the result as a bytevector. If *end* is not provided, it is assumed to be the length of *bytevector*. If *start* is not provided, it is assumed to be 0.

14.2 Secure keys

secure-key-type-tag object
Symbol representing the `secure-key` type. The `type-for` procedure of library `(lispkit type)` returns this symbol for all secure key objects.

(make-private-key *cs*) procedure
(make-private-key *cs size*)
(make-private-key *cs size tag*)

Creates a new private key for the cryptographic system *cs* and returns it as a `secure-key` object. Cryptographic systems are identified via symbols. Currently only `rsa` is supported as value for *cs*. *size* defines the size of the key in bits; default is 1024. *tag* is an optional string identifying the application generating the key. If *tag* is not provided, a random UUID string is being used as application tag.

```
(make-private-key 'rsa 2048 "myapp")
⇒ #<secure-key 600000b26ce0: rsa private 2048>
```

(public-key *key*) procedure
Returns a public key for the given private *key* as a `secure-key` object.

```
(define privkey (make-private-key 'rsa 1024 "demo app"))
(define pubkey (public-key privkey))
(display (secure-key->string pubkey)) ⇒
-----BEGIN RSA PUBLIC KEY-----
MIGJAoGBAPh4qcCprhnrCelHVzhvlzVBxe62qDwes3IrMNcvKAYnqgVpSvUN+RNI
AWcQPVEiIPWxMz0/75mT3jFukysGMg6LdFzjslmtgvVutUM7vqkaliGIiBp92QBa
iXjZpD33YxQvKTp7F8hv4sJwgVz4junkM11X7Wnw8R6+1l4fYCbvAgMBAAE=
-----END RSA PUBLIC KEY-----
```

(secure-key? *obj*) procedure
Returns `#t` if *obj* is a secure key; otherwise `#f` is returned.

(secure-key-type? *key cs*) procedure
Returns `#t` if the secure *key* is suitable to be used with the cryptographic system *cs*; otherwise `#f` is returned. Cryptographic systems are identified via symbols. Currently only `rsa` is supported as value for argument *cs*.

(secure-key-private? ***key*****)** procedure

Returns `#t` if the secure *key* is a private key; otherwise `#f` is returned.

(secure-key-public? ***key*****)** procedure

Returns `#t` if the secure *key* is a public key; otherwise `#f` is returned.

(secure-key-can-encrypt? ***key*****)** procedure
(secure-key-can-encrypt? ***key algo*****)**

Returns `#t` if secure *key* can be used to encrypt messages via algorithm *algo*, otherwise `#f` is returned. *algo* is a symbol identifying the encryption algorithm (see section on Crypto Algorithms). If *algo* is not provided, `secure-key-can-encrypt?` returns `#t` if *key* can be used to encrypt messages in general, independent of a concrete algorithm.

(secure-key-can-decrypt? ***key*****)** procedure
(secure-key-can-decrypt? ***key algo*****)**

Returns `#t` if secure *key* can be used to decrypt messages via algorithm *algo*, otherwise `#f` is returned. *algo* is a symbol identifying the encryption algorithm (see section on Crypto Algorithms). If *algo* is not provided, `secure-key-can-decrypt?` returns `#t` if *key* can be used to decrypt messages in general, independent of a concrete algorithm.

(secure-key-can-sign? ***key*****)** procedure
(secure-key-can-sign? ***key algo*****)**

Returns `#t` if secure *key* can be used to sign messages via algorithm *algo*, otherwise `#f` is returned. *algo* is a symbol identifying the encryption algorithm (see section on Crypto Algorithms). If *algo* is not provided, `secure-key-can-sign?` returns `#t` if *key* can be used to sign messages in general, independent of a concrete algorithm.

(secure-key-can-verify? ***key*****)** procedure
(secure-key-can-verify? ***key algo*****)**

Returns `#t` if secure *key* can be used to verify messages via algorithm *algo*, otherwise `#f` is returned. *algo* is a symbol identifying the encryption algorithm (see section on Crypto Algorithms). If *algo* is not provided, `secure-key-can-verify?` returns `#t` if *key* can be used to verify messages in general, independent of a concrete algorithm.

(secure-key-size ***key*****)** procedure
(secure-key-size ***key effective*****)**

Returns the size of the given secure *key* in bits. If *effective* is `#f` or omitted, the total number of bits in the secure key are returned, otherwise, the effective number of bits used by this secure key are returned.

(secure-key-block-size ***key*****)** procedure

Returns the block length associated with the given secure *key* in bytes. If the secure key is an RSA key, for instance, this is the size of the modulus.

(secure-key-attributes ***key*****)** procedure

Returns the attributes associated with the given secure *key* as an association list. Each attribute is represented by a cons whose car is the key of an attribute and whose cdr is the corresponding value.

```
(secure-key-attributes (make-private-key 'rsa))
⇒ ((unwp . 1) (priv . 0) (sens . 0) (extr . 1) (vrfy . 0)
   (encr . 0) (drve . 1) (modi . 0) (sign . 1) (vyrc . 0)
   (next . 0) (type . "42") (bsiz . 1024) (kcls . "1")
   (asen . 0) (esiz . 1024) (decr . 1) (wrap . 0)
   (class . "keys") (snrc . 0) (perm . 1))
```

(secure-key=? *key0 key* ...) procedure

Defines an identity relationship for secure keys. `secure-key=?` returns `#t` if each *key* ... is the same key object as *key0*, otherwise `#f` gets returned.

(secure-key-data=? *key* ...) procedure

Defines an equivalence relationship for secure keys based on the equivalence of a serialized representation of a secure key. `secure-key-data=?` returns `#t` if each *key* ... is representing an equivalent key with the same serialized external representation as *key0*. Otherwise, `#f` gets returned.

(secure-key->bytevector *key*) procedure

Serializes the given secure *key* into a new bytevector and returns the bytevector.

(bytevector->private-key *cs bytevector*) procedure
(bytevector->private-key *cs bytevector start*)
(bytevector->private-key *cs bytevector start end*)

Turns a serialized representation of a secure key in *bytevector* between indices *start* (inclusive) and *end* (exclusive) into a new private key of the given cryptographic system *cs*. Cryptographic systems are identified via symbols. Currently only `rsa` is supported as value for *cs*. If *end* is not provided, it is assumed to be the length of *bytevector*. If *start* is not provided, it is assumed to be 0.

(bytevector->public-key *obj*) procedure

Turns a serialized representation of a secure key in *bytevector* between indices *start* (inclusive) and *end* (exclusive) into a new public key of the given cryptographic system *cs*. Cryptographic systems are identified via symbols. Currently only `rsa` is supported as value for *cs*. If *end* is not provided, it is assumed to be the length of *bytevector*. If *start* is not provided, it is assumed to be 0.

(secure-key->string *key*) procedure

Returns a PEM (Privacy Encoded Mail)-encoded string representation of the given secure *key*.

```
(define priv (make-private-key 'rsa 1024 "test"))
(define pub (public-key priv))
(define str (secure-key->string pub))
(display str)
⇒
-----BEGIN RSA PUBLIC KEY-----
MIGJAoGBALz04JfvuKHf0tKE6rYGasoebv4T54m88OR0tJ0Bb+7gUWkX8DPWEy/y
Y0m6QOUK0nvpCvNdvZq7dW2Pjnwd4Cwy9lCUGj+MTSrwqL8fM3FvbLGI6lAAPqYb
S/T9zcG/YnNSmB/A6o3EcfYi/nT0u83t6bmSwa0SHNoOQ110fm6jAgMBAAE=
-----END RSA PUBLIC KEY-----
```

(string->secure-key *str*) procedure

Turns a PEM (Privacy Encoded Mail)-encoded representation of a secure key *str* (a string) into a new secure key object and returns this object. An error is raised if it is not possible to extract a secure key supported by this library.

14.3 Crypto algorithms

In library `(lispkit crypto)`, cryptographic algorithms for encrypting, decrypting, signing, and verifying messages are identified via interned symbols. The following cryptographic algorithms, identified via the listed symbols, are supported:

RSA Encryption

- `rsa-encryption-raw`
- `rsa-encryption-pkcs1`

RSA Encryption OAEP

- `rsa-encryption-oaep-sha1`
- `rsa-encryption-oaep-sha256`
- `rsa-encryption-oaep-sha384`
- `rsa-encryption-oaep-sha512`

RSA Encryption OAEP AESGCM

- `rsa-encryption-oaep-sha1-aesgcm`
- `rsa-encryption-oaep-sha256-aesgcm`
- `rsa-encryption-oaep-sha384-aesgcm`
- `rsa-encryption-oaep-sha512-aesgcm`

RSA Signature Raw

- `rsa-signature-raw`

RSA Signature Digest PKCS1v15

- `rsa-signature-digest-pkcs1v15-raw`
- `rsa-signature-digest-pkcs1v15-sha1`
- `rsa-signature-digest-pkcs1v15-sha256`
- `rsa-signature-digest-pkcs1v15-sha384`
- `rsa-signature-digest-pkcs1v15-sha512`

RSA Signature Message PKCS1v15

- `rsa-signature-message-pkcs1v15-sha1`
- `rsa-signature-message-pkcs1v15-sha256`
- `rsa-signature-message-pkcs1v15-sha384`
- `rsa-signature-message-pkcs1v15-sha512`

RSA Signature Digest PSS

- `rsa-signature-digest-pss-sha1`
- `rsa-signature-digest-pss-sha256`
- `rsa-signature-digest-pss-sha384`
- `rsa-signature-digest-pss-sha512`

RSA Signature Message PSS

- `rsa-signature-message-pss-sha1`
- `rsa-signature-message-pss-sha256`
- `rsa-signature-message-pss-sha384`
- `rsa-signature-message-pss-sha512`

(encrypt *key algo bytevector*) procedure
(encrypt *key algo bytevector start*)
(encrypt *key algo bytevector start end*)

Encrypts a message represented by *bytevector* between indices *start* (inclusive) and *end* (exclusive) using the given secure *key* and encryption algorithm *algo*. The encrypted message is returned as a new bytevector.

algo is a symbol identifying the encryption algorithm. It has to identify a suitable algorithm for usage by the `encrypt` procedure, otherwise an error is signaled. If argument *end* is not provided, it is assumed to be the length of *bytevector*. If *start* is not provided, it is assumed to be 0.

(decrypt *key algo bytevector*) procedure
(decrypt *key algo bytevector start*)
(decrypt *key algo bytevector start end*)

Decrypts an encrypted message represented by *bytevector* between indices *start* (inclusive) and *end* (exclusive) using the given secure *key* and encryption algorithm *algo*. The decrypted message is returned as a new bytevector.

algo is a symbol identifying the encryption algorithm. If *end* is not provided, it is assumed to be the length of *bytevector*. If *start* is not provided, it is assumed to be 0.

(sign *key algo bytevector*) procedure
(sign *key algo bytevector start*)
(sign *key algo bytevector start end*)

Signs a message represented by *bytevector* between indices *start* (inclusive) and *end* (exclusive) using the given secure *key* and signature algorithm *algo*. The signature of the message is returned as a new bytevector.

algo is a symbol identifying the signature algorithm. If *end* is not provided, it is assumed to be the length of *bytevector*. If *start* is not provided, it is assumed to be 0.

(verify *key algo bytevector*) procedure
(verify *key algo bytevector start*)
(verify *key algo bytevector start end*)

Verifies a signature represented by *bytevector* between indices *start* (inclusive) and *end* (exclusive) using the given secure *key* and signature algorithm *algo*. `verify` returns `#t` if the signature could be verified, otherwise `#f` is returned.

algo is a symbol identifying the signature algorithm. If *end* is not provided, it is assumed to be the length of *bytevector*. If *start* is not provided, it is assumed to be 0.

15 LispKit CSV

Library `(lispkit csv)` provides a simple API for reading and writing structured data in CSV format from a text file. The API provides two different levels of abstraction: reading and writing at

1. line-level (lower-level API), and
2. record-level (higher-level API).

A text file in CSV format typically has the following structure:

```
"First name", "Last name", "Birth date"
Steve, Jobs, 1955-02-24
Bill, Gates, "1955-10-28"
"Jeff", "Bezos", "1964-01-12"
```

The first line is called the *header*. It defines the names and the order of the columns. Columns are separated by a *separator* character (which is `,` in the example above). The *column names* can optionally be wrapped by a *quotation* character, which is needed if the name contains, for instance, the separator character.

Each following line defines one data record which provides values for the columns defined in the header. The values are again separated by the *separator* character and they may be optionally wrapped by the *quotation* character. If a value is wrapped with a quotation character, the same character can be used within the value if it is escaped. The quotation character can be escaped by a sequence of two quotation characters (e.g. if `"` is used as a quotation character, `""` encodes a single `"` character within the string value).

The client of the API decides how to handle inconsistencies between the lines, e.g. if lines have too few or too many values.

15.1 CSV ports

Both levels use a *CSV port* to configure the textual input/output port, the separator and quotation character.

(csv-port? *obj*) procedure
Returns `#t` if *obj* is a CSV port; returns `#f` otherwise.

(csv-input-port? *obj*) procedure
Returns `#t` if *obj* is a CSV port for reading data; returns `#f` otherwise.

(csv-output-port? *obj*) procedure
Returns `#t` if *obj* is a CSV port for writing data; returns `#f` otherwise.

(make-csv-port) procedure
(make-csv-port *tport*)
(make-csv-port *tport sep*)
(make-csv-port *tport sep quote*)

Returns a new CSV port for reading or writing data via an underlying textual port *tport*. If *tport* is an output port, the CSV port can be used for writing. If *tport* is an input port, the CSV port can be used for reading. The default for *tport* is the current input port `current-input-port` exported from library `(lispkit port)`.

The separation character used by the CSV port is *sep*, the quotation character is *quote*. The default for *sep* is `#\,` and for *quote* the default is `#\"`.

(csv-base-port *csvp*) procedure
Returns the textual port on which the CSV port *csvp* is based on.

(csv-separator *csvp*) procedure
Returns the separation character used by the CSV port *csvp*.

(csv-quotechar *csvp*) procedure
Returns the quotation character used by the CSV port *csvp*.

15.2 Line-level API

The line-level API provides means to read a whole CSV file via `csv-read` and write data in CSV format via `csv-write`.

(csv-read *csvp*) procedure
(csv-read *csvp readheader?*)

Reads from CSV port *csvp* first the header, if `readheader?` is set to `#t`, and then all the lines until the end of the input is reached. Procedure `csv-read` returns two values: the header line (a list of strings representing the column names), and a vector of all data lines, which itself are lists of strings representing the individual field values. The default for *readheader?* is `#t`. If *readheader?* is set to `#f`, the first result of `csv-read` is always `#f`.

(csv-write *csvp header lines*) procedure
Writes to CSV port *csvp* first the *header* (a list of strings representing the column names) unless *header* is set to `#f`. Then procedure `csv-write` writes each line of `lines`. `lines` is a vector of lists representing the individual field values in string form.

15.3 Record-level API

The higher level API has a notion of records. The default representation for records are association lists. The functions for reading and writing records are `csv-read-records` and `csv-write-records`:

(csv-read-records *csvp*) procedure
(csv-read-records *csvp make-col*)
(csv-read-records *csvp make-col make-record*)

Reads from CSV port *csvp* first the header and then all the data lines until the end of the input is reached. Header names (strings) are mapped via procedure *make-col* into *column identifiers* or *column factories* (i.e. procedures which take one argument, a column value, and they return either a representation of this column if the value is valid, or `#f` if the column value is invalid). With `make-record` a list of *column identifiers* and *column factories* as well as a list of column values (strings) of a data line are mapped into a record. Procedure `csv-read-records` returns a vector of records.

The default *make-col* procedure is `make-symbol-column`. The default `make-record` function is `make-alist-record/excess`.

(csv-write-records *csvp header records*) procedure
(csv-write-records *csvp header records col->str*)
(csv-write-records *csvp header records col->str field->str*)

First writes the header to CSV port *csvp* by mapping `header`, which is a list of column identifiers. to a list of header names using procedure *col->str*. Then, `csv-write-records` writes all the records from the vector *records* by mapping each record to a data line. This is done by applying *field->str* to all column identifiers for the record. *field->str* takes two arguments: a column identifier and the record.

The default implementation for procedure *col->str* is `symbol->string`. The default implementation for procedure *field->str* is `alist-field->string`.

(make-symbol-column *str*) procedure
Returns a symbol representing the trimmed string *str*. If the trimmed string is empty, `make-symbol-column` returns `#t`. This procedure can be used for creating column identifers out of column names in procedure `csv-read-records`.

(make-alist-record *cols fields*) procedure
Returns a new record given a list of column identifiers or column factories (i.e. procedures which take one argument, a column value, and they return either a representation of this column if the value is valid, or `#f` if the column value is invalid) *cols*, and a list of column values *fields*.

This procedure represents records as association lists, iterating through all *cols* and *fields* values. If there are more *fields* values than *cols* expressions, than they are skipped. If there are more *cols* expressions than *fields* values, `#f` is used as a default for missing *fields* values. If a *cols* expression is a procedure, the association entry gets created by calling the procedure with the corresponding *fields* value. For all other *cols* expression types, a pair is created with the *cols* expression being the car and the *fields* value being the cdr.

(make-alist-record/excess *?*) procedure
Returns a new record given a list of column identifiers or column factories (i.e. procedures which take one argument, a column value, and they return either a representation of this column if the value is valid, or `#f` if the column value is invalid) *cols*, and a list of column values *fields*.

This procedure represents records as association lists, iterating through all *cols* and *fields* values. If there are more *fields* values than *cols* expressions, than `#f` is used as a default *cols* expression. If there are more *cols* expressions than *fields* values, `#f` is used as a default for missing *fields* values. If a *cols* expression is a procedure, the association entry gets created by calling the procedure with the corresponding *fields* value. For all other *cols* expression types, a pair is created with the *cols* expression being the car and the *fields* value being the cdr.

(alist-field->string *record col*) procedure
Returns the column value of column *col* from association list *record*. `alist-field->string` assumes that *record* is an association list whose values are strings. This is how the procedure is defined:

```
(define (alist-field->string record column)
  (cdr (assq column record)))
```

16 LispKit Datatype

Library `(lispkit datatype)` implements algebraic datatypes for LispKit. It provides the following functionality:

- `define-datatype` creates a new algebraic datatype consisting of a type test predicate and a number of variants. Each variant implicitly defines a constructor and a pattern.
- `define-pattern` introduces a new pattern and constructor for an existing datatype variant.
- `match` matches a value of an algebraic datatype against patterns, binding pattern variables and executing the code of the first case whose pattern matches the value.

16.1 Usage

Here is an example of a datatype defining a tree for storing and finding elements:

```
(define-datatype tree tree?
  (empty)
  (node left element right) where (and (tree? left) (tree? right)))
```

The datatype `tree` defines a predicate `tree?` for checking whether a value is of type `tree`. In addition, it defines two variants with corresponding constructors `empty` and `node` for creating values of type `tree`. Variant `node` defines an invariant that prevents nodes from being constructed unless `left` and `right` are also trees.

The following line defines a new tree:

```
(define t1 (node (empty) 4 (node (empty) 7 (empty))))
```

Using `match`, values like `t1` can be deconstructed using pattern matching. The following function `elements` shows how to collect all elements from a tree in a list:

```
(define (elements tree)
  (match tree
    ((empty)       ())
    ((node l e r)  (append (elements l) (list e) (elements r)))))
```

`match` is a special form which takes a value of an algebraic datatype and matches it against a list of cases. Each case defines a pattern and a sequence of statements which get executed if the pattern matches the value.

Cases can also optionally define a guard which is a boolean expression that gets executed if the pattern of the case matches a value. Only if the guard evaluates to true, the statements of the case get executed. Otherwise, pattern matching continues. The following function `insert` demonstrates this functionality:

```
(define (insert tree x)
  (match tree
    ((empty)
      (node (empty) x (empty)))
    ((node l e r) where (< x e)
      (node (insert l x) e r))
    ((node l e r)
      (node l e (insert r x)))))
```

A new tree `t2`, with two new elements inserted, can be created like this:

```
(define t2 (insert (insert t1 2) 9))
```

If a pattern is used frequently containing a lot of boilerplate, it is possible to define a shortcut using the `define-pattern` syntax:

```
(define-pattern (single x)
  (node (empty) x (empty)))
```

With this declaration, it is possible to use `single` in patterns. The following example also shows that it is possible to use `else` for defining a fallback case, if no other pattern is matching.

```
(match t
  ((empty) #f)
  ((single x) x)
  (else (error "two many elements")))
```

`single` can also be used as a constructor for creating trees with a single element:

```
(single 6)
```

An advanced feature of `match` is the usage of pattern alternatives in a single case of a `match` construct. This can be achieved using the `or` form on the top level of a pattern:

```
(define (has-many-elements tree)
  (match tree
    ((or (empty) (single _)) #f)
    (else #t)))
```

The underscore in the `(single _)` subpattern is a wildcard that matches every value and that does not bind a new variable.

16.2 API

(define-datatype *type (constr arg ...) ...*) syntax
(define-datatype *type pred (constr arg ...) ...*)
(define-datatype *type pred (constr arg ...)* where *condition ...*)

Defines a new datatype with a given number of datatype variants. The definition requires the symbol *type* denoting the new type, an optional symbol *pred* which gets bound to a type test function for testing whether a value is an instance of this type, and a list of constructors of the form *(constr arg1 arg2 ...)*

where *constr* is the constructor and *arg1*, *arg2*, … are parameter names of the constructor. A constructor can be annotated with an invariant for defining requirements the parameters need to meet. This is done via clause `where` *expr* succeeding the constructor declaration. *condition* is a boolean expression which gets checked when the constructor gets invoked.

(define-pattern *(constr arg …) (impl expr …)*) syntax

Defines a new pattern *(constr arg …)* which specializes an existing pattern *(impl expr …)*. Such custom patterns can be used in pattern matching expressions as well as constructors for defining values of an algebraic datatype.

(match *expr case* …) syntax
(match *expr case* … (else *stat* …))

`match` provides a mechanism for decomposing values of algebraic datatypes via pattern matching. A `match` construct takes a value *expr* to pattern match on, as well as a sequence of cases. Each case consists of pattern alternatives, an optional guard, and a sequence of statements:

```
case      =  `(` patterns stat ... `)`
          |  `(` patterns `where` condition stat ... `)`
patterns  =  pattern
          |  `(` `or` pattern ... `)`
pattern   =  '_'                                ; wildcard
          |  var                                ; variable
          |  `#t`                               ; literal boolean (true)
          |  `#f`                               ; literal boolean (false)
          |  string                             ; literal string
          |  number                             ; literal number
          |  character                          ; literal character
          |  vector                             ; literal vector
          |  `'` expr                           ; constant expression
          |  `,` expr                           ; value (result of evaluating expr)
          |  pattern `as` var                   ; pattern bound to variable
          |  `(` `list` pattern ... `)`         ; list pattern
          |  `(` `list` pattern ... `.` var `)` ; list pattern with rest
          |  `(` `list` pattern ... `.` `_` `)` ; list pattern with unbound rest
          |  `(` constr pattern ... `)`         ; variant pattern
```

`match` iterates through the cases and executes the sequence of statements *stat* … of the first case whose pattern is matching `expr` and whose guard *condition* evaluates to true. The value returned by this sequence of statements is returned by `match`.

17 LispKit Date-Time

Library `(lispkit date-time)` provides functionality for handling time zones, dates, and times. Time zones are represented by string identifiers referring to the region and corresponding city, e.g. `"America/Los_Angeles"`. Dates and times are represented via `date-time` data structures. These encapsulate the following components:

- *time zone*: the time zone of the date
- *date*: the date consisting of its year, month, and day
- *time*: the time on *date* consisting of the hour (>= 0, < 24), the minute (>= 0, < 60), the second (>= 60, <60), and the nano second.

The library uses a floating-point representation of seconds since 00:00 UTC on January 1, 1970, as a means to refer to specific points in time independent of timezones. This means that, for instance, for comparing date-times with each other, a user would have to convert them to seconds and then compare the seconds instead. Here is an example:

```
(define initial-time (date-time "Europe/Zurich"))
(define later-time (date-time "GMT"))
(date-time< initial-time later-time)
  ⇒ #t
; the following line is equivalent:
(< (date-time->seconds initial-time) (date-time->seconds later-time))
  ⇒ #t
```

For now, `(lispkit date-time)` assumes all dates are based on the Gregorian calendar, independent of the settings at the operating system-level.

17.1 Time zones

Time zones are represented by string identifiers referring to the region and corresponding city, e.g. `"America/Los_Angeles"`. Procedure `timezones` returns a list of all supported time zone identifiers. Each time zone has a locale-specific name and an offset in seconds from Greenwich Mean Time. Some time zones also have an abbreviation which can be used as an alternative way to identify a time zone.

(timezones) procedure
(timezones *filter*)

Returns a list of string identifiers for all supported time zones. If *filter* is provided, it can either be set to `#f`, in which case a list of abbreviations is returned instead, or it is a string, and only time zone identifiers which contain *filter* are returned.

```
(timezones #f)
⇒ ("CEST" "GST" "NZDT" "BRST" "WEST" "AST" "MSD" "CDT" "WIT" "MSK" "COT" "IST" "EST" "BST"
 ↪ "CLST" "NDT" "TRT" "EET" "IRST" "EDT" "BRT" "ICT" "CST" "AKST" "BDT" "PHT" "SGT" "WET"
 ↪ "ART" "CLT" "CAT" "UTC" "EEST" "ADT" "JST" "HST" "PET" "MST" "NST" "NZST" "GMT" "MDT" "PKT"
 ↪ "WAT" "HKT" "AKDT" "KST" "PST" "CET" "PDT" "EAT")
```

(timezone? *obj*) procedure

Returns `#t` if *obj* is a valid time zone identifier or time zone abbreviation; returns `#f` otherwise.

(timezone) procedure
(timezone *ident*)

Returns the identifier for the time zone specified by *ident*. *ident* can either be an identifier, an abbreviation or a GMT offset as a floating-point number or integer. If *ident* does not refer to a supported time zone, procedure `timezone` will fail.

(timezone-name *tz*) procedure
(timezone-name *tz locale*)
(timezone-name *tz locale format*)

Returns a locale-specific name for time zone *tz*. If *locale* is not specified, the current locale defined at the operating-system level is used. *format* specifies the name format. It can have one of the following symbolic values:

- `standard`
- `standard-short`
- `dst`
- `dst-short`
- `generic`
- `generic-short`

(timezone-abbreviation *tz*) procedure

Returns a string representing a time zone abbreviation for *tz*; e.g. `"PDT"`. If the time zone *tz* does not have an abbreviation, this function returns `#f`.

(timezone-gmt-offset *tz*) procedure

Returns the difference in seconds between time zone *tz* and Greenwich Mean Time. The difference is returned as a floating-point number (since seconds are always represented as such by this library).

17.2 Time stamps

Time stamps, i.e. discreet points in time, are represented as floating-point numbers corresponding to the number of seconds since 00:00 UTC on January 1, 1970.

(current-seconds) procedure

Returns a floating-point number representing the number of seconds since 00:00 UTC on January 1, 1970.

(seconds->date-time *secs*) procedure
(seconds->date-time *secs tz*)

Converts the given number of seconds *secs* into date-time format for the given time zone *tz*. *secs* is a floating-point number. It is interpreted as the number of seconds since 00:00 UTC on January 1, 1970. *secs* is negative if the date-time is earlier than 00:00 UTC on January 1, 1970. If *tz* is missing, the current, operating-system defined time zone is used.

(date-time->seconds *dtime*) procedure

Returns a floating-point number representing the number of seconds since 00:00 UTC on January 1, 1970 for the given date-time object *dtime*.

17.3 Date-times

(date-time? *obj*) procedure

Returns `#t` if *obj* is a date-time object; returns `#f` otherwise.

(date-time) procedure
(date-time *year month day*)
(date-time *year month day hour*)
(date-time *year month day hour min*)
(date-time *year month day hour min sec*)
(date-time *year month day hour min sec nano*)
(date-time *tz*)
(date-time *tz year month day*)
(date-time *tz year month day hour*)
(date-time *tz year month day hour min*)
(date-time *tz year month day hour min sec*)
(date-time *tz year month day hour min sec nano*)

Constructs a date-time representation out of the given date time components. *tz* is the only string argument; it is referring to a time zone. All other arguments are numeric arguments. This procedure returns a date-time object for the specified time at the given date. If no date components are provided as arguments, procedure `date-time` returns a date-time for the current date and time.

(week->date-time *year week*) procedure
(week->date-time *year week wday*)
(week->date-time *year week wday hour*)
(week->date-time *year week wday hour min*)
(week->date-time *year week wday hour min sec*)
(week->date-time *year week wday hour min sec nano*)
(week->date-time *tz year week*) **(week->date-time *tz year week wday*)**
(week->date-time *tz year week wday hour*)
(week->date-time *tz year week wday hour min*)
(week->date-time *tz year week wday hour min sec*)
(week->date-time *tz year week wday hour min sec nano*)

Constructs a date-time representation out of the given date time components. *tz* is the only string argument; it is referring to a time zone. All other arguments are numeric arguments. Argument *wday* specifies the week day in the given week. Week days are given numbers from 1 (= Monday) to 7 (= Sunday). This procedure returns a date-time object for the specified time at the given date.

The difference to `date-time` is that this procedure does not refer to a month and day. It rather refers to the week number as well as the weekday within this specified week number.

(date-time-in-timezone *dtime*) procedure
(date-time-in-timezone *dtime tzone*)

Constructs a date-time representation of the same point in time like *dtime*, but in a potentially different time zone *tzone*. If *tzone* is not given, the default time zone specified by the user in the operating system will be used.

(string->date-time *str*) procedure
(string->date-time *str tz*)
(string->date-time *str tz locale*)
(string->date-time *str tz locale format*)

Extracts a date and time from the given string *str* in the time zone *tz*, or the current time zone if *tz* is omitted. The format of the string representation is defined in terms of *locale* and *format*. *format* can have three different forms:

1. Combined format identifier for date and time: parsing is based on the settings of the operating system. *format* is one of the following symbols: `none`, `short`, `medium`, `long`, or `full`.
2. Separate format identifiers for date and time: `date-time` parsing is based on the settings of the operating system, but the format for dates and times is specified separately. *format* is a list of the form `(` *dateformat timeformat* `)` where both *dateformat* and *timeformat* are one of the 5 symbols listed under 1. This makes it possible, for instance, to just parse a date (without time) in string form to a date-time object, e.g. by using `(short none)` as *format*.
3. Custom format specifier: `date-time` parsing is based on a custom format string. *format* is a string using the following characters as placeholders. Repetitions of the placeholder characters are used to specify the width and format of the field.
   - `y` : Year
   - `M` : Month
   - `d` : Day
   - `H` : Hour (12 hours)
   - `h` : Hour (24 hours)
   - `m` : Minute
   - `s` : Second
   - `S` : Micro second
   - `Z` : Time zone
   - `a` : AM/PM
   - `E` : Weekday

Here are a few examples:

```
EEEE, MMM d, yyyy        ~~>  Thursday, Feb 8, 1973
dd/MM/yyyy               ~~>  08/02/1973
dd-MM-yyyy HH:mm         ~~>  08-02-1973 17:01
MMM d, h:mm a            ~~>  Thu 8, 2:11 AM
yyyy-MM-dd'T'HH:mm:ssZ   ~~>  1973-08-02T17:01:31+0000
HH:mm:ss.SSS             ~~>  11:02:19.213
```

(date-time->string *dtime*) procedure
(date-time->string *dtime locale*)
(date-time->string *dtime locale format*)

Returns a string representation of the date-time object *dtime*. The format of the string is defined in terms of *locale* and *format*. *format* can have three different forms (just like for `string->date-time`):

1. Combined format identifier for date and time: formatting is based on the settings of the operating system. *format* is one of the following symbols: `none`, `short`, `medium`, `long`, or `full`.
2. Separate format identifiers for date and time: `date-time` formatting is based on the settings of the operating system, but the format for dates and times is specified separately. *format* is a list of the form `(` *dateformat timeformat* `)` where both *dateformat* and *timeformat* are one of the 5 symbols listed under 1. This makes it possible, for instance, to just output a date (without time) in string form, e.g. by using `(short none)` as *format*.
3. Custom format specifier: `date-time` formatting is based on a custom format string. *format* is a string using the following characters as placeholders. Repetitions of the placeholder characters are used to specify the width and format of the field.
   - `y` : Year
   - `M` : Month

- `d` : Day
- `H` : Hour (12 hours)
- `h` : Hour (24 hours)
- `m` : Minute
- `s` : Second
- `S` : Micro second
- `Z` : Time zone
- `a` : AM/PM
- `E` : Weekday

Here are a few examples:

```
EEEE, MMM d, yyyy        ~~>  Thursday, Feb 8, 1973
dd/MM/yyyy               ~~>  08/02/1973
dd-MM-yyyy HH:mm         ~~>  08-02-1973 17:01
MMM d, h:mm a            ~~>  Thu 8, 2:11 AM
yyyy-MM-dd'T'HH:mm:ssZ   ~~>  1973-08-02T17:01:31+0000
HH:mm:ss.SSS             ~~>  11:02:19.213
```

(date-time->iso8601-string *dtime*) procedure
Returns a string representation of the date-time object *dtime* in ISO 8601 format.

(date-time-timezone *dtime*) procedure
Returns the time zone of *dtime*.

(date-time-year *dtime*) procedure
Returns the year of *dtime*.

(date-time-month *dtime*) procedure
Returns the month of *dtime*.

(date-time-day *dtime*) procedure
Returns the day of *dtime*.

(date-time-hour *dtime*) procedure
Returns the hour of *dtime*.

(date-time-minute *dtime*) procedure
Returns the minute of *dtime*.

(date-time-second *dtime*) procedure
Returns the second of *dtime*.

(date-time-nano *dtime*) procedure
Returns the nano-second of *dtime*.

(date-time-weekday *dtime*) procedure
Returns the week day of *dtime*. Week days are represented as fixnums where 1 is Monday, 2 is Tuesday, …, and 7 is Sunday.

(date-time-week *dtime*) procedure
Returns the week number of *dtime* according to the ISO-8601 standard. Based on this standard, weeks start on Monday. The first week of the year is the week that contains that year's first Thursday.

(date-time-dst-offset *dtime*) procedure
Returns the daylight saving time offset of *dtime* in seconds related to GMT. If daylight savings time is not active, `date-time-dst-offset` returns `0.0`. The result is always a floating-point number.

(date-time-hash *dtime*) procedure
Returns a hash code for the given date-time object. This hash code can be used in combination with both `date-time=?` and `date-time-same?`.

17.4 Date-time predicates

(date-time-same? *dtime1 dtime2*) procedure

Returns `#t` if date-time *dtime1* and *dtime2* have the same timezone and refer to the same point in time, i.e. `(date-time->seconds dtime1)` and `(date-time->seconds dtime2)` are equals.

```
(define d1 (date-time 'CET))
(define d2 (date-time-in-timezone d1 'PST))
(date-time-same? d1 d1)  ⇒  #t
(date-time-same? d1 d2)  ⇒  #f
(date-time=? d1 d2)      ⇒  #t
```

(date-time=? *dtime1 dtime2*) procedure

Returns `#t` if date-time *dtime1* and *dtime2* specify the same point in time, i.e. `(date-time->seconds dtime1)` and `(date-time->seconds dtime2)` are equals.

```
(define d1 (date-time 'CET))
(define d2 (date-time-in-timezone d1 'PST))
(date-time=? d1 d2)                ⇒  #t
(date-time=? d1 (date-time 'CET))  ⇒  #f
```

(date-time<? *dtime1 dtime2*) procedure

Returns `#t` if date-time *dtime1* specifies an earlier point in time compared to *dtime2*, i.e. `(date-time->seconds dtime1)` is less than `(date-time->seconds dtime2)`.

(date-time>? *dtime1 dtime2*) procedure

Returns `#t` if date-time *dtime1* specifies a later point in time compared to *dtime2*, i.e. `(date-time->seconds dtime1)` is greater than `(date-time->seconds dtime2)`.

(date-time<=? *dtime1 dtime2*) procedure

Returns `#t` if date-time *dtime1* specifies an earlier or equal point in time compared to *dtime2*, i.e. `(date-time->seconds dtime1)` is less than or equal to `(date-time->seconds dtime2)`.

(date-time>=? *dtime1 dtime2*) procedure

Returns `#t` if date-time *dtime1* specifies a later or equal point in time compared to *dtime2*, i.e. `(date-time->seconds dtime1)` is greater than or equal to `(date-time->seconds dtime2)`.

(date-time-has-dst? *dtime*) procedure

Returns `#t` if daylight saving time is active for *dtime*; returns `#f` otherwise.

17.5 Date-time operations

(date-time-add *dtime days*) procedure
(date-time-add *dtime days hrs*)
(date-time-add *dtime days hrs min*)
(date-time-add *dtime days hrs min sec*)
(date-time-add *dtime days hrs min sec nano*)

Compute a new date-time from adding *days*, *hrs*, *min*, *sec*, and *nano* (all fixnums) to the given date-time *dtime*. The resulting date-time is using the same timezone like *dtime*.

(date-time-add-seconds *dtime sec*) procedure

Compute a new date-time from adding the number of seconds *sec* (a flonum) to the given date-time *dtime*.

(date-time-diff-seconds *dtime1 dtime2*) procedure

Computes the difference between *dtime2* and *dtime1* as a number of seconds (a flonum).

(next-dst-transition *dtime*) procedure

Returns the date and time when the next daylight savings time transition takes place after *dtime*. `next-dst-transition` returns `#f` if there is no daylight savings time for the time zone of *dtime*.

18 LispKit Debug

Library `(lispkit debug)` provides utilities for debugging code. Available are procedures for measuring execution latencies, for tracing procedure calls, for expanding macros, for disassembling code, as well as for inspecting the execution environment.

18.1 Timing execution

(time *expr*) syntax

`time` compiles *expr* and executes it. The form displays the time it took to execute *expr* as a side-effect. It returns the result of executing *expr*.

(time-values *expr*) syntax

`time-values` executes *expr*. If *expr* evaluates to *n* values `x1, ..., xn`, `time-values` returns *n + 1* values `t, x1, ..., xn` where `t` is the time it takes to evaluate *expr*.

18.2 Tracing procedure calls

(trace-calls) procedure
(trace-calls *level*)

This function is used to enable/disable call tracing. When call tracing is enabled, all function calls that are executed by the virtual machine are being printed to the console. Call tracing operates at three levels:

- `0` : Call tracing is switched off
- `1` : Call tracing is enabled only for procedures for which it is enabled (via function `set-procedure-trace!`)
- `2` : Call tracing is switched on for all procedures (independent of procedure-level tracing being enabled or disabled)

`(trace-calls n)` will set call tracing to level `n`. If the level is ommitted, `trace-calls` will return the current call tracing level.

For instance, if call tracing is enabled via `(trace-calls 2)`, executing `(fib 3)` will print the following call trace.

```
> (define (fib n)
    (if (< n 2) n (+ (fib (- n 1)) (fib (- n 2)))))
> (trace-calls 2)
> (fib 2)
  ↘ (fib 2) in <repl>
    → (< 2 2) in fib
    ← #f from <
    → (- 2 1) in fib
    ← 1 from -
    → (fib 1) in fib
      → (< 1 2) in fib
```

```
      ← #t from < in fib
    ← 1 from fib in fib
    → (- 2 2) in fib
    ← 0 from -
    → (fib 0) in fib
      → (< 0 2) in fib
      ← #t from < in fib
    ← 0 from fib in fib
  ↘ (+ 1 0) in fib
  ← 1 from fib
1
```

Function invocations are prefixed with `→`, or `↘` if it's a tail call. The value returned by a function call is prefixed with `←`.

(procedure-trace? ***proc*****)** procedure

Returns `#f` if procedure-level call tracing is disabled for *proc*, `#t` otherwise.

(set-procedure-trace! ***proc trace?*****)** procedure

Enables procedure-level call tracing for procedure *proc* if *trace?* is set to `#t`. It disables call tracing for *proc* if *trace?* is `#f`.

18.3 Macro expansion

(quote-expanded ***expr*****)** syntax

`quote-expanded` is syntax for macro-expanding expression *expr* in the current syntactical environment. Macro-expansion is applied consecutively as long as the top-level can be expanded further.

```
(quote-expanded (assert (+ 1 2)))
⇒  (if (not (+ 1 2))
       (assertion (quote (+ 1 2))))
```

(quote-expanded-1 ***expr*****)** syntax

`quote-expanded-1` is syntax for macro-expanding expression *expr* in the current syntactical environment. Macro-expansion is applied at most once, even if the top-level can be expanded further.

```
(quote-expanded-1 (for x in '(1 2 3) (display x)))
⇒  (dolist (x (quote (1 2 3))) (display x))
```

(macroexpand ***expr*****)** procedure
(macroexpand ***expr env*****)**

Procedure `macroexpand` applies macro-expansion to the expression *expr* in the environment *env* as long as the expression on its top-level can be expanded further. If *env* is not provided, the current interaction environment is used.

```
(macroexpand
  '(dotimes (x (+ 2 2)) (display x) (newline)))
⇒  (do ((maxvar (+ 2 2))
        (x 0 (fx1+ x)))
       ((fx>= x maxvar))
     (display x)
     (newline))
```

(macroexpand-1 *expr*)
(macroexpand-1 *expr env*)

procedure

Procedure `macroexpand-1` applies macro-expansion to the expression *expr* in the environment *env* at most once. The resulting expression might therefore only be partially expanded at the top-level. If *env* is not provided, the current interaction environment is used.

```
(macroexpand-1 '(for x in '(1 2 3) (display x)))
⇒  (dolist (x (quote (1 2 3))) (display x))

(macroexpand-1
  (macroexpand-1
    '(for x in '(1 2 3) (display x))))
⇒  (let ((x (quote ()))
         (ys (quote (1 2 3))))
     (if (null? ys)
         (void)
         (do ((xs ys (cdr xs)))
             ((null? xs))
           (set! x (car xs))
           (display x))))
```

18.4 Disassembling code

(compile *expr*)
(compile *expr env*)

procedure

Compiles expression *expr* in environment *env* and displays the disassembled code. If *env* is not given, the current interaction environment is used. This is what is being printed when executing `(compile '(do ((i 0 (fx1+ i)))((fx> i 10))(display i)(newline)))`:

```
CONSTANTS:
    0: #<procedure display>
    1: #<procedure newline>
INSTRUCTIONS:
    0: alloc 1
    1: push_fixnum 0
    2: make_local_variable 0
    3: push_local_value 0
    4: push_fixnum 10
    5: fx_gt
    6: branch_if 14                  ;; jump to 20
    7: make_frame
    8: push_constant 0               ;; #<procedure display>
    9: push_local_value 0
   10: call 1
   11: pop
   12: make_frame
   13: push_constant 1               ;; #<procedure newline>
   14: call 0
   15: pop
   16: push_local_value 0
   17: fx_inc
   18: set_local_value 0
   19: branch -16                    ;; jump to 3
   20: push_void
   21: reset 0, 1
   22: return
```

(disassemble *proc*) procedure

Disassembles procedure *proc* and prints out the code. This is what is being printed when executing `(disassemble caddr)`:

```
CONSTANTS:
INSTRUCTIONS:
    0: assert_arg_count 1
    1: push_global 426
    2: make_frame
    3: push_global 431
    4: push_local 0
    5: call 1
    6: tail_call 1
```

18.5 Execution environment

(gc) procedure

Force garbage collection to be performed.

(available-symbols) procedure

Returns a list of all symbols that have been used so far.

(loaded-libraries) procedure

Returns a list of all libraries that have been loaded so far.

```
> (loaded-libraries)
((lispkit draw) (lispkit base) (lispkit port) (lispkit control) (lispkit type) (lispkit list)
 ↪ (lispkit string) (lispkit math) (lispkit date-time) (lispkit dynamic) (lispkit char-set)
 ↪ (lispkit bytevector) (lispkit char) (lispkit vector) (lispkit regexp) (lispkit record)
 ↪ (lispkit hashtable) (lispkit system) (lispkit core) (lispkit gvector) (lispkit box))
```

(loaded-sources) procedure

Returns a list of all sources that have been loaded.

(environment-info) procedure

Prints out debug information about the current execution environment (mostly relevant for developing LispKit).

(stack-size) procedure

Returns the number of elements that are currently on the stack.

(call-stack-procedures) procedure

Returns a list of procedures that are currently in process of being executed in the current thread.

(call-stack-trace) procedure

Returns a list of procedure calls that are currently in process of being executed in the current thread. The result is a list of lists, where each element corresponds to an active procedure call (with the given parameters, where this can be reconstructed).

(set-max-call-stack! *n*) procedure

When exceptions and errors are created, a call stack trace is attached to them. Since these can be quite large, call stack traces are capped at the top-most *n* entries. *n* can be at most 1000, default is *20*.

(internal-call-stack) procedure

Returns a list of strings, each representing a native function that is currently being executed internally.

19 LispKit Disjoint-Set

Library `(lispkit disjoint-set)` implements disjoint sets, a mutable union-find data structure that tracks a set of elements partitioned into disjoint subsets. Disjoint sets are based on hashtables and require the definition of an equality and a hash function.

disjoint-set-type-tag object

Symbol representing the `disjoint-set` type. The `type-for` procedure of library `(lispkit type)` returns this symbol for all disjoint set objects.

(disjoint-set? *obj*) procedure

Returns `#t` if *obj* is a disjoint set object; otherwise `#f` is returned.

(make-eq-disjoint-set) procedure

Returns a new empty disjoint set using `eq` as equality and `eq-hash` as hash function.

(make-eqv-disjoint-set) procedure

Returns a new empty disjoint set using `eqv` as equality and `eqv-hash` as hash function.

(make-disjoint-set *comparator*) procedure
(make-disjoint-set *hash eql*)

Returns a new empty disjoint set using *eql* as equality and *hash* as hash function. Instead of providing two functions, a new disjoint set can also be created based on a *comparator*.

(disjoint-set-make *dset x*) procedure

Adds a new singleton set *x* to *dset* if element *x* does not exist already in disjoint set *dset*.

(disjoint-set-find *dset x*) procedure
(disjoint-set-find *dset x default*)

Looks up element *x* in *dset* and returns the set in which *x* is currently contained. Returns *default* if element *x* is not found. If *default* is not provided, `disjoint-set-find` uses `#f` instead.

(disjoint-set-union *dset x y*) procedure

Unifies the sets containing *x* and *y* in disjoint set *dset*.

(disjoint-set-size *dset*) procedure

Returns the number of sets in *dset*.

20 LispKit Draw

Library `(lispkit draw)` provides an API for creating *drawings*. A drawing is defined in terms of a sequence of instructions for drawing *shapes* and *images*. Drawings can be composed and saved as a PDF. It is also possible to draw a drawing into a *bitmap* and save it in formats like PNG, JPG, or TIFF. A *bitmap* is a special *image* that is not based on vector graphics. Bitmaps have a size expressed in points and a resolution expressed in pixels per inch (ppi).

Both drawings and shapes are based on a coordinate system whose zero point is in the upper left corner of a plane. The x and y axis extend to the right and down. Coordinates and dimensions are always expressed in terms of floating-point numbers.

20.1 Drawings

Drawings are mutable objects created via `make-drawing`. The functions listed in this section change the state of a drawing object and they persist drawing instructions defining the drawing. For most functions, the drawing is an optional argument. If it is not provided, the function applies to the drawing provided by the `current-drawing` parammeter object.

current-drawing [parameter object]
Defines the *current drawing*, which is used as a default by all functions for which the drawing argument is optional. If there is no current drawing, this parameter is set to `#f`.

(drawing? *obj*) [procedure]
Returns `#t` if *obj* is a drawing. Otherwise, it returns `#f`.

(make-drawing) [procedure]
Returns a new, empty drawing. A drawing consists of a sequence of drawing instructions and drawing state consisting of the following components:

- Stroke color (set via `set-color`)
- Fill color (set via `fill-color`)
- Shadow (set via `set-shadow` and `remove-shadow`)
- Transformation (add transformation via `enable-transformation` and remove via `disable-transformation`)

(copy-drawing *drawing*) [procedure]
Returns a copy of the given drawing.

(set-color *color*) [procedure]
(set-color *color drawing*)

Sets the *stroke color* for the given drawing, or `current-drawing` if the drawing argument is not provided.

(set-fill-color *color*) [procedure]
(set-fill-color *color drawing*)

Sets the *fill color* for the given drawing, or `current-drawing` if the drawing argument is not provided.

(set-line-width *width*) procedure
(set-line-width *width drawing*)

Sets the default *stroke width* for the given drawing, or `current-drawing` if the drawing argument is not provided.

(set-shadow *color size blur-radius*) procedure
(set-shadow *color size blur-radius drawing*)

Defines a shadow for the given drawing, or `current-drawing` if the drawing argument is not provided. *color* is the color of the shade, *blur-radius* defines the radius for bluring the shadow.

(remove-shadow) procedure
(remove-shadow *drawing*)

Removes shadow for the subsequent drawing instructions of the given drawing, or `current-drawing` if the drawing argument is missing.

(enable-transformation *tf*) procedure
(enable-transformation *tf drawing*)

Enables the transformation *tf* for subsequent drawing instructions of the given drawing, or `current-drawing` if the drawing argument is missing. Each drawing maintains an active affine transformation for shifting, rotating, and scaling the coordinate systems of subsequent drawing instructions.

(disable-transformation *tf*) procedure
(disable-transformation *tf drawing*)

Disables the transformation *tf* for subsequent drawing instructions of the given drawing, or `current-drawing` if the drawing argument is missing.

(draw *shape*) procedure
(draw *shape width*)
(draw *shape width drawing*)

Draws *shape* with a given stroke *width* into the drawing specified via *drawing* or parameter object `current-drawing` if *drawing* is not provided. The default for *width*, in case it is not provided, is set via *set-line-width*. The stroke is drawn in the current stroke color of the drawing.

(draw-dashed *shape lengths phase*) procedure
(draw-dashed *shape lengths phase width*)
(draw-dashed *shape lengths phase width drawing*)

Draws *shape* with a dashed stroke of width *width* into the drawing specified via *drawing* or parameter object `current-drawing` if *drawing* is not provided. `1.0` is the default for *width* in case it is not provided. *lengths* specifies an alternating list of dash/space lengths. *phase* determines the start of the dash/space pattern. The dashed stroke is drawn in the current stroke color of the drawing.

(fill *shape*) procedure
(fill *shape drawing*)

Fills *shape* with the current *fill color* in the drawing specified via *drawing* or parameter object `current-drawing` if *drawing* is not provided.

(fill-gradient *shape colors*) procedure
(fill-gradient *shape colors spec*)
(fill-gradient *shape colors spec drawing*)

Fills *shape* with a gradient in the drawing specified via argument *drawing* or parameter object `current-drawing` if *drawing* is not provided. The gradient is specified in terms of a list of *colors* and argument *spec*. *spec* can either be a number or a point. If *spec* is a number, this number determines an angle for a linear gradient. If *spec* is a point, it is the center of a radial gradient.

(draw-line *start end*) procedure
(draw-line *start end drawing*)

Draws a line between point *start* and point *end* in the drawing specified via argument *drawing* or parameter object `current-drawing`, if *drawing* is not provided. The line is drawn in the default *stroke width* and the current *stroke color*.

(draw-rect *rect*) procedure
(draw-rect *rect drawing*)

Draws a rectangular given by *rect* in the drawing specified via argument *drawing* or parameter object `current-drawing`, if *drawing* is not provided. The rectangular is drawn in the default *stroke width* and the current *stroke color*.

(fill-rect *rect*) procedure
(fill-rect *rect drawing*)

Fills a rectangular given by *rect* with the current *fill color* in the drawing specified via argument *drawing* or parameter object `current-drawing`, if *drawing* is not provided.

(draw-ellipse *rect*) procedure
(draw-ellipse *rect drawing*)

Draws an ellipse into the rectangular *rect* in the drawing specified via argument *drawing* or parameter object `current-drawing`, if *drawing* is not provided. The ellipse is drawn in the default *stroke width* and the current *stroke color*.

(fill-ellipse *rect*) procedure
(fill-ellipse *rect drawing*)

Fills an ellipse given by rectangular *rect* with the current *fill color* in the drawing specified via argument *drawing* or parameter object `current-drawing`, if *drawing* is not provided.

(draw-text *str location font*) procedure
(draw-text *str location font color*)
(draw-text *str location font color drawing*)

Draws string *str* at *location* in the given font and color in the drawing specified by argument *drawing* or parameter object `current-drawing` if *drawing* is not provided. *location* is either the left, top-most point at which the string is drawn, or it is a rect specifying a bounding box. *color* specifies the text color. If it is not provided, the text is drawn in black.

(text-size *str*) procedure
(text-size *str font*)
(text-size *str font dimensions*)

Returns a size object describing the width and height needed to draw string *str* using *font* in a space constrained by *dimensions*. *dimensions* is either a size object specifying the maximum width and height, or it is a number constraining the width only, assuming infinite hight. If *dimensions* is omitted, the maximum width and height is infinity.

(draw-styled-text *txt location*) procedure
(draw-styled-text *txt location drawing*)

Draws styled text *txt* at *location* in the drawing specified by argument *drawing* or parameter object `current-drawing` if *drawing* is not provided. *location* is either the left, top-most point at which the styled text is drawn, or it is a rect specifying a bounding box.

(styled-text-size *txt*) procedure
(styled-text-size *txt dimensions*)

Returns a size object describing the width and height needed to draw styled text *txt* in a space constrained by *dimensions*. *dimensions* is either a size object specifying the maximum width and height, or it is a number constraining the width only, assuming infinite hight. If *dimensions* is omitted, the maximum width and height is infinity.

(draw-html *html location*) procedure
(draw-html *html location drawing*)

Draws a string *html* containing HTML source code at *location* in the drawing specified by argument *drawing* or parameter object `current-drawing` if *drawing* is not provided. *location* is either the left, top-most point at which the HTML is drawn, or it is a rect specifying a bounding box.

(html-size *html*) procedure
(html-size *html dimensions*)

Returns a size object describing the width and height needed to render the HTML in string *html* in a space constrained by *dimensions*. *dimensions* is either a size object specifying the maximum width and height, or it is a number constraining the width only, assuming infinite hight. If *dimensions* is omitted, the maximum width and height is infinity.

(draw-image *image location*) procedure
(draw-image *image location opacity*)
(draw-image *image location opacity composition*)
(draw-image *image location opacity composition drawing*)

Draws image *image* at *location* with the given *opacity* and *composition* method. The image is drawn in the drawing specified by argument *drawing* or parameter object `current-drawing` if *drawing* is not provided. *location* is either the left, top-most point at which the image is drawn, or it is a rect specifying a bounding box for the image. *composition* is a floating-point number between 0.0 (= transparent) and 1.0 (= completely not transparent) with 1.0 being the default. *composition* refers to a symbol specifying a composition method. The following methods are supported (the source is the image, the destination is the drawing):

- `clear` : Transparency everywhere.
- `copy` : The source image (default).
- `multiply` : The source color is multiplied by the destination color.
- `overlay` : Source colors overlay the destination.
- `source-over` : The source image wherever it is opaque, and the destination elsewhere.
- `source-in` : The source image wherever both images are opaque, and transparent elsewhere.
- `source-out` : The source image wherever it is opaque and the destination is transparent, and transparent elsewhere.
- `source-atop` : The source image wherever both source and destination are opaque, the destination wherever it is opaque but the source image is transparent, and transparent elsewhere.
- `destination-over` : The destination wherever it is opaque, and the source image elsewhere.
- `destination-in` : The destination wherever both images are opaque, and transparent elsewhere.
- `destination-out` : The destination wherever it is opaque and the source image is transparent, and transparent elsewhere.
- `destination-atop` : The destination wherever both image and destination are opaque, the source image wherever it is opaque and the destination is transparent, and transparent elsehwere.

(draw-drawing *other*) procedure
(draw-drawing *other drawing*)

Draws drawing *other* into the drawing specified by argument *drawing* or parameter object `current-drawing` if *drawing* is not provided. This function can be used to compose drawings.

(clip-drawing *other clippingshape*) procedure
(clip-drawing *other clippingshape drawing*)

Draws drawing *other* into the drawing specified by argument *drawing* or parameter object `current-drawing` if *drawing* is not provided. This function clips the drawing using shape *clippingshape*; i.e. only parts within *clippingshape* are drawn.

(inline-drawing *other*) procedure
(inline-drawing *other drawing*)

Draws drawing *other* into the drawing specified by argument *drawing* or parameter object `current-drawing` if *drawing* is not provided. This function can be used to compose drawings in a way such that the drawing instructions from *other* are inlined into *drawing*.

(save-drawing *path drawing size*) procedure
(save-drawing *path drawing size title*)
(save-drawing *path drawing size title author*)

Saves *drawing* into a PDF file at the given filepath *path*. *size* is a size specifying the width and height of the PDF page containing the drawing in points; i.e. the media box of the page is `(rect zero-point` *size* `)`. *title* and *author* are optional strings defining the title and author metadata for the generated PDF file.

(save-drawings *path pages*) procedure
(save-drawings *path pages title*)
(save-drawings *path pages title author*)

Saves a list of pages into a PDF file at the given filepath *path*. A page is defined in terms of a list of two elements (*drawing size*), where *drawing* is a drawing for that page and *size* is a media box for the page. *title* and *author* are optional strings defining the title and author metadata for the generated PDF file.

(drawing *body* ...) syntax
Creates a new empty drawing, binds parameter object `current-drawing` to it and executes the body statements in the dynamic scope of this binding. This special form returns the new drawing.

(with-drawing *drawing body* ...) syntax
Binds parameter object `current-drawing` to *drawing* and executes the body statements in the dynamic scope of this binding. This special form returns the result of the last statement in the body.

(transform *tf body* ...) syntax
This form is used in the context of drawing into `current-drawing`. It enables the transformation *tf*, executes the statements in the body and disables the transformation again.

20.2 Shapes

Shapes are mutable objects created via a number of constructors, including `make-shape`, `copy-shape`, `line`, `polygon`, `rectangular`, `circle`, `oval`, `arc`, and `glyphs`. Besides the constructors, functions like `move-to`, `line-to` and `curve-to` are used to extend a shape. For those functions, the affected shape is an optional argument. If it is not provided, the function applies to the shape defined by the `current-shape` parammeter object.

current-shape parameter object
Defines the *current shape*, which is used as a default by all functions for which the shape argument is optional. If there is no current shape, this parameter is set to `#f`.

(shape? *obj*) procedure
Returns `#t` if *obj* is a shape. Otherwise, it returns `#f`.

(make-shape) procedure
(make-shape *prototype*)

Returns a new shape object. If argument *prototype* is provided, the new shape object will inherit from shape *prototype*; i.e. the new shape's definition will extend the definition of shape *prototype*.

(copy-shape *shape*) procedure
Returns a copy of *shape*.

(line *start end*) procedure
Retuns a new line shape. *start* and *end* are the start and end points of the line.

(polygon *point ...*) procedure
Returns a new polygon shape. The polygon is defined in terms of a sequence of points.

(rectangle *point size*) procedure
(rectangle *point size radius*)
(rectangle *point size xradius yradius*)

Returns a new rectangular shape. The rectangle is defined in terms of the left, topmost point and a *size* defining both width and height of the rectangle. The optional *radius*, *xradius* and *yradius* arguments are used to create a rounded rectangular whose rounded edges are defined in terms of an x and y-radius. If only one radius is provided, it defines both x and y-radius.

(circle *point radius*) procedure
Returns a new circle shape. The circle is defined in terms of a center point and a radius.

(oval *point size*) procedure
Returns a new oval shape. The oval is defined in terms of a rectangle whose left, topmost point is provided as argument *point*, and whose width and height are given via argument *size*.

(arc *point radius start end*) procedure
(arc *point radius start end clockwise*)

Returns a new arc shape. The arc is defined via the arguments *point*, *radius*, *start* , *end* and optionally *clockwise*. *point* is the starting point of the arc, *radius* defines the radius of the arc, *start* is a starting angle in radians, and *end* is the end angle in radians. *clockwise* is a boolean argument defining whether the arc is drawn clockwise or counter-clockwise. The default is `#t` .

(glyphs *str point size font*) procedure
Returns a new "glyphs" shape. This is a shape defined by a string *str* rendered in the given size and font at a given point. *font* is a font object, *size* is the font size in points, and *point* are the start coordinates where the glyphs are drawn.

(transform-shape *shape tf*) procedure
Returns a new shape derived from *shape* by applying transformation *tf*.

(flip-shape *shape*) procedure
(flip-shape *shape box*)
(flip-shape *shape box orientation*)

Returns a new shape by flipping/mirroring *shape* within *box*. *box* is a rect. If it is not provided, the bounding box of *shape* is used as a default. Argument *orientation* is a symbol defining along which axis the shape is flipped. Supported are `horizontal` , `vertical` , and `mirror` . Default is `vertical` .

(interpolate *points*) procedure
(interpolate *points closed*)
(interpolate *points closed alpha*)
(interpolate *points closed alpha method*)

Returns a shape interpolating a list of points. *closed* is an optional boolean argument specifying whether the shape is closed. The default for *closed* is `#f`. *alpha* is an interpolation parameter in the range [0.0,1.0]; default is 0.33. *method* specifies the interpolation method via a symbol. The following two methods are supported: `hermite` and `catmull-rom`; default is `hermite`.

(move-to *point*) procedure
(move-to *point shape*)

Sets the "current point" to *point* for the shape specified by argument *shape* or parameter object `current-shape` if *shape* is not provided.

(line-to *point* …) procedure
(line-to *point* … *shape*)

Creates a line from the "current point" to *point* for the shape specified by argument *shape* or parameter object `current-shape` if *shape* is not provided. *point* becomes the new "current point".

(curve-to *point cntrl1 cntrl2*) procedure
(curve-to *point cntrl1 cntrl2 shape*)

Creates a curve from the "current point" to *point* for the shape specified by argument *shape* or parameter object `current-shape` if *shape* is not provided. *cntrl1* and *cntrl2* are control points defining tangets shaping the curve at the start and end points.

(relative-move-to *point*) procedure
(relative-move-to *point shape*)

This function is equivalent to `move-to` with the exception that *point* is relative to the "current point".

(relative-line-to *point* …) procedure
(relative-line-to *point* … *shape*)

This function is equivalent to `line-to` with the exception that *point* is relative to the "current point".

(relative-curve-to *point cntrl1 cntrl2*) procedure
(relative-curve-to *point cntrl1 cntrl2 shape*)

This function is equivalent to `curve-to` with the exception that *point*, *cntrl1* and *cntrl2* are relative to the "current point".

(add-shape *other*) procedure
(add-shape *other shape*)

Adds shape *other* to the shape specified by argument *shape* or parameter object `current-shape` if *shape* is not provided. This function is typically used to compose shapes.

(shape-bounds *shape*) procedure
Returns the bounding box for the given shape as a rect.

(shape *body* …) syntax
Creates a new empty shape, binds parameter object `current-shape` to it and executes the body statements in the dynamic scope of this binding. This special form returns the new drawing.

(with-shape *shape body* …) syntax
Binds parameter object `current-shape` to *shape* and executes the body statements in the dynamic scope of this binding. This special form returns the result of the last statement in the body.

20.3 Images

Images are objects representing immutable pictures of mutable size and metadata. Images are either loaded from image files or they are created from drawings. Images are either vector-based or bitmap-based. The current image API only allows loading vector-based images from PDF files. Bitmap-based images, on the other hand, can be loaded from PNG, JPG, GIF, etc. image files or they are created by drawing a drawing object into an empty bitmap. Bitmap-based images optionally have mutable EXIF data.

(image? *obj*) procedure
Returns `#t` if *obj* is an image. Otherwise, it returns `#f`.

(load-image *path*) procedure
Loads an image from the file at *path* and returns the corresponding image object.

(load-image-asset *path type*) procedure
(load-image-asset *path type dir*)

Loads an image from the file at the given relative file *path* and returns the corresponding image object. *type* refers to the default suffix of the file to load (e.g. `"png"` for PNG images).

`load-image-asset` constructs a relative file path in the following way (assuming *path* does not have a suffix already):

dir/path.type

where *dir* is `"Images"` if it is not provided as a parameter. It then searches the asset paths in their given order for a file matching this relative file path. Once the first matching file is found, the file is loaded as an image file and the image gets returned by `load-image-asset`. It is an error if no matching image was found.

(image-size *image*) procedure
Returns the size of the given image object in points.

(set-image-size! *image size*) procedure
Sets the size of *image* to *size*, a size in points.

(bitmap? *obj*) procedure
Returns `#t` if *obj* is a bitmap-based image. Otherwise, it returns `#f`.

(bitmap-size *bmap*) procedure
Returns the size of the bitmap *bmap* in points. If *bmap* is not a bitmap object, `bitmap-size` returns `#f`
.

(bitmap-pixels *bmap*) procedure
Returns the number of horizontal and vertical pixels of the bitmap *bmap* as a size. If *bmap* is not a bitmap object, `bitmap-size` returns `#f`.

(bitmap-exif-data *bmap*) procedure
Returns the EXIF metadata associated with bitmap *bmap*. EXIF metadata is represented as an association list in which symbols are used as keys.

```
> (define photo (load-image (asset-file-path "Regensberg" "jpeg" "Images")))
> (bitmap-exif-data photo)
((ExposureBiasValue . 0)
 (CustomRendered . 6)
 (SensingMethod . 2)
 (SubsecTimeOriginal . "615")
 (SubsecTimeDigitized . "615")
 (Flash . 0)
 (ExposureTime . 0.00040306328093510683)
 (OffsetTime . "+01:00")
 (PixelXDimension . 8066)
 (ExifVersion 2 3 1)
 (OffsetTimeDigitized . "+01:00")
```

```
(ISOSpeedRatings 25)
(OffsetTimeOriginal . "+01:00")
(DateTimeDigitized . "2019:10:27 14:21:39")
(FlashPixVersion 1 0)
(WhiteBalance . 0)
(PixelYDimension . 3552)
(LensSpecification 4.25 4.25 1.7999999523162842 1.7999999523162842)
(ColorSpace . 65535)
(LensModel . "iPhone XS back camera 4.25mm f/1.8")
(SceneCaptureType . 0)
(ApertureValue . 1.6959938128383605)
(SceneType . 1)
(ShutterSpeedValue . 11.276932534193945)
(FocalLength . 4.25)
(FNumber . 1.8)
(LensMake . "Apple")
(FocalLenIn35mmFilm . 26)
(BrightnessValue . 10.652484683458134)
(ComponentsConfiguration 1 2 3 0)
(MeteringMode . 5)
(DateTimeOriginal . "2019:10:27 14:21:39"))
```

(set-bitmap-exif-data! ***bmap exif*****)** procedure

Sets the EXIF metadata for the given bitmap *bmap* to *exif*. *exif* is an association list defining all the EXIF attributes with symbols being used as keys.

```
> (define photo (load-image (asset-file-path "Regensberg" "jpeg" "Images")))

> (set-bitmap-exif-data! photo
    '((ExposureBiasValue . 0)
      (Flash . 0)
      (ExposureTime . 0.0005)
      (PixelXDimension . 8066)
      (PixelYDimension . 3552)
      (ExifVersion 2 3 1)
      (ISOSpeedRatings 25)
      (FlashPixVersion 1 0)
      (WhiteBalance . 0)
      (LensSpecification 4.25 4.25 1.7999999523162842 1.7999999523162842)
      (ColorSpace . 65535)
      (SceneCaptureType . 0)
      (ApertureValue . 1.6959938128383605)
      (SceneType . 1)
      (ShutterSpeedValue . 11.276932534193945)
      (FocalLength . 4.25)
      (FNumber . 1.8)
      (BrightnessValue . 10.652484683458134)
      (ComponentsConfiguration 1 2 3 0)
      (OffsetTime . "+01:00")
      (OffsetTimeOriginal . "+01:00")
      (DateTimeOriginal . "2019:10:27 14:21:39")
      (OffsetTimeDigitized . "+01:00")
      (DateTimeDigitized . "2019:10:27 14:21:39"))
  )
```

(make-bitmap ***drawing size*****)** procedure
(make-bitmap ***drawing size ppi*****)**

Creates a new bitmap-based image by drawing the object *drawing* into an empty bitmap of size *size* in points. *ppi* determines the number of pixels per inch. By default, *ppi* is set to 72. In this case, the number of pixels of the bitmap corresponds to the number of points (since 1 pixel corresponds to 1/72 of an inch). For a *ppi* value of 144, the horizontal and vertial number of pixels is doubled, etc.

(bitmap-crop *bitmap rect*) procedure

Crops a rectangle from the given bitmap and returns the result in a new bitmap. *rect* is a rectangle in pixels. Its intersection with the dimensions of *bitmap* (in pixels) are used for cropping.

(bitmap-blur *bitmap radius*) procedure

Blurs the given bitmap with the given blur radius and returns the result in a new bitmap of the same size.

(save-bitmap *path bitmap format*) procedure

Saves a given bitmap-based image *bitmap* in a file at filepath *path*. *format* is a symbol specifying the image file format. Supported are: `png`, `jpg`, `gif`, `bmp`, and `tiff`.

(bitmap->bytevector *bitmap format*) procedure

Returns a bytevector with an encoding of *bitmap* in the given format. *format* is a symbol specifying the image format. Supported are: `png`, `jpg`, `gif`, `bmp`, and `tiff`.

20.4 Transformations

A transformation is an immutable object defining an affine transformation. Transformations can be used to:

- shift,
- scale, and
- rotate coordinate systems.

Transformations are typically used in drawings to transform drawing instructions. They can also be used to transform shapes.

(transformation? *obj*) procedure

Returns `#t` if *obj* is a transformation. Otherwise, it returns `#f`.

(transformation *tf* ...) procedure

Creates a new transformation by composing the given transformations *tf*.

(invert *tf*) procedure

Inverts transformation *tf* and returns a new transformation object for it.

(translate *dx dy*) procedure
(translate *dx dy tf*)

Returns a transformation for shifting the coordinate system by *dx* and *dy*. If transformation *tf* is provided, the translation transformation extends *tf*.

(scale *dx dy*) procedure
(scale *dx dy tf*)

Returns a transformation for scaling the coordinate system by *dx* and *dy*. If transformation *tf* is provided, the scaling transformation extends *tf*.

(rotate *angle*) procedure
(rotate *angle tf*)

Returns a transformation for rotating the coordinate system by *angle* (in radians). If transformation *tf* is provided, the rotation transformation extends *tf*.

20.5 Colors

Colors are immutable objects defining colors in terms of four components: red, green, blue and alpha. Library `(lispkit draw)` currently only supports RGB color spaces.

`(lispkit draw)` supports the concept of *color lists* on *macOS*. A color list is provided as a `.plist` file and stored in the "ColorLists" asset directory of LispKit. It maps color names expressed as symbols to color values. Color lists need to be loaded explicitly via procedure `load-color-list`.

(color? *obj*) procedure

Returns `#t` if *obj* is a color. Otherwise, it returns `#f`.

(color *spec*) procedure
(color *name clist*)
(color *r g b*)
(color *r g b alpha*)

This procedure returns new color objects. If *spec* is provided, it either is a string containing a color description in hex format, or it is a symbol referring to the name of a color in the default color list `(White Yellow Red Purple Orange Magenta Green Cyan Brown Blue Black)`. If a different color list should be used, its name can be specified via string *clist*. Procedure `(available-color-lists)` returns a list of all available color lists. If the color is specified via a hex string, the following formats can be used: `"ccc"`, `"#ccc"`, `"rrggbb"`, and `"#rrggbb"`.

The color can also be specified using color components *r, g, b,* and *alpha*. *alpha* determines the transparency of the color (0.0 = fully transparent, 1.0 = no transparency). The default value for *alpha* is 1.0.

(color-red *color*) procedure

Returns the red color component of *color*.

(color-green *color*) procedure

Returns the green color component of *color*.

(color-blue *color*) procedure

Returns the blue color component of *color*.

(color-alpha *color*) procedure

Returns the alpha color component of *color*.

```
(color->alpha (color 1.0 0.5 0.1))      ⇒  1.0
(color->alpha (color 1.0 0.1 0.5 0.4))  ⇒  0.4
```

(color->hex *color*) procedure

Returns a representation of the given *color* in hex form as a string.

```
(color->hex (color 1.0 0.5 0.1))  ⇒  "#FF801A"
(color->hex (color "#6AF"))       ⇒  "#66AAFF"
(color->hex red)                  ⇒  "#FF0000"
```

black object
gray
white
red
green
blue
yellow

Predefined color objects.

(available-color-lists) procedure

Returns a list of available color lists. The LispKit installation guarantees that there is at least color list "HTML" containing all named colors from the HTML 5 specification.

```
(available-color-lists)
⇒  ("HTML" "Web Safe Colors" "Crayons" "System" "Apple")
```

(load-color-list *name path*) procedure

Loads a new color list stored as a `.plist` file in the assets directory of LispKit at the given file *path* (which can also refer to color lists outside of the assets directory via absolute file paths). *name* is a string which specifies the name of the color list. It is added to the list of available colors if loading of the color list was successful. `load-color-list` returns `#t` if the color list could be successfully loaded, `#f` otherwise.

(available-colors *clist*) procedure

Returns a list of color identifiers supported by the given color list. *clist* is a string specifying the name of the color list.

```
(available-colors "HTML")
⇒  (YellowGreen Yellow WhiteSmoke White Wheat Violet Turquoise Tomato Thistle Teal Tan
 ↪  SteelBlue Snow SlateGrey SlateGray SlateBlue SkyBlue Silver Sienna SeaShell SeaGreen
 ↪  SandyBrown Salmon SaddleBrown RoyalBlue RosyBrown Red RebeccaPurple Purple PowderBlue Plum
 ↪  Pink Peru PeachPuff PapayaWhip PaleVioletRed PaleTurquoise PaleGreen PaleGoldenRod Orchid
 ↪  OrangeRed Orange OliveDrab Olive OldLace Navy NavajoWhite Moccasin MistyRose MintCream
 ↪  MidnightBlue MediumVioletRed MediumTurquoise MediumSpringGreen MediumSlateBlue
 ↪  MediumSeaGreen MediumPurple MediumOrchid MediumBlue MediumAquaMarine Maroon Magenta Linen
 ↪  LimeGreen Lime LightYellow LightSteelBlue LightSlateGrey LightSlateGray LightSkyBlue
 ↪  LightSeaGreen LightSalmon LightPink LightGrey LightGreen LightGray LightGoldenRodYellow
 ↪  LightCyan LightCoral LightBlue LemonChiffon LawnGreen LavenderBlush Lavender Khaki Ivory
 ↪  Indigo IndianRed HotPink HoneyDew Grey GreenYellow Green Gray GoldenRod Gold GhostWhite
 ↪  Gainsboro Fuchsia ForestGreen FloralWhite FireBrick DodgerBlue DimGrey DimGray DeepSkyBlue
 ↪  DeepPink DarkViolet DarkTurquoise DarkSlateGrey DarkSlateGray DarkSlateBlue DarkSeaGreen
 ↪  DarkSalmon DarkRed DarkOrchid DarkOrange DarkOliveGreen DarkMagenta DarkKhaki DarkGrey
 ↪  DarkGreen DarkGray DarkGoldenRod DarkCyan DarkBlue Cyan Crimson Cornsilk CornflowerBlue
 ↪  Coral Chocolate Chartreuse CadetBlue BurlyWood Brown BlueViolet Blue BlanchedAlmond Black
 ↪  Bisque Beige Azure Aquamarine Aqua AntiqueWhite AliceBlue)
```

20.6 Fonts

Fonts are immutable objects defining fonts in terms of a font name and a font size (in points).

(font? *obj*) procedure

Returns `#t` if *obj* is a font. Otherwise, it returns `#f`.

(font *fontname size*) procedure
(font *familyname size weight trait ...*)

If only two arguments, *fontname* and *size*, are provided, `font` will return a new font object for a font with the given font name and font size (in points). If more than two arguments are provided, `font` will return a new font object for a font with the given font family name, font size (in points), font weight, as well as a number of font traits.

The weight of a font is specified as an integer on a scale from 0 to 15. Library `(lispkit draw)` exports the following weight constants:

- `ultralight` (1)

- `thin` (2)
- `light` (3)
- `book` (4)
- `normal` (5)
- `medium` (6)
- `demi` (7)
- `semi` (8)
- `bold` (9)
- `extra` (10)
- `heavy` (11)
- `super` (12)
- `ultra` (13)
- `extrablack` (14)

Font traits are specified as integer masks. The following trait constants are exported from library `(lispkit draw)`:

- `italic`
- `boldface`
- `unitalic`
- `unboldface`
- `narrow`
- `expanded`
- `condensed`
- `small-caps`
- `poster`
- `compressed`
- `monospace`

(font-name *font*) procedure
Returns the font name of *font*.

(font-family-name *font*) procedure
Returns the font family name of *font*.

(font-size *font*) procedure
Returns the font size of *font* in points.

(font-weight *font*) procedure
Returns the font weight of *font*. See documentation of function `font` for details.

(font-traits *font*) procedure
Returns the font traits of *font* as an integer bitmask. See documentation of function `font` for details.

(font-has-traits *font trait ...*) procedure
Returns `#t` if font *font* has all the given traits.

(available-fonts) procedure
(available-fonts *trait ...*)

Returns all the available fonts that have matching font traits.

(available-font-families) procedure
Returns all the available font families, i.e. all font families for which there is at least one font installed.

20.7 Points

A *point* describes a location on a two-dimensional plane consisting of a *x* and *y* coordinate. Points are represented as pairs of floating-point numbers where the car representes the x-coordinate and the cdr represents the y-coordinate. Even though an expression like `'(3.5 . -2.0)` does represent a point, it is recommended to always construct points via function `point`; e.g. `(point 3.5 -2.0)`.

(point? *obj*) procedure
Returns `#t` if *obj* is a valid point. Otherwise, it returns `#f`.

(point *x y*) procedure
Returns a point for coordinates *x* and *y*.

(move-point *point dx fy*) procedure
Moves *point* by *dx* and *dy* and returns the result as a point.

(point-x *point*) procedure
Returns the x-coordinate for *point*.

(point-y *point*) procedure
Returns the y-coordinate for *point*.

zero-point object
The *zero point*, i.e `(point 0.0 0.0)`.

20.8 Size

A *size* describes the dimensions of a rectangle consisting of *width* and *height* values. Sizes are immutable objects, represented as pairs of floating-point numbers where the car representes the width and the cdr represents the height. Even though an expression like `'(5.0 . 3.0)` does represent a valid size object, it is recommended to always construct sizes via function `size`; e.g. `(size 5.0 3.0)`.

(size? *obj*) procedure
Returns `#t` if *obj* is a valid size. Otherwise, it returns `#f`.

(size *w h*) procedure
Returns a size object for the given width *w* and height *h*.

(size-width *size*) procedure
Returns the width of *size*.

(size-height *size*) procedure
Returns the height of *size*.

(increase-size *size dx dy*) procedure
Returns a new size object based on `size` whose width is increased by *dx* and whose height is increased by *dy*.

(scale-size *size factor*) procedure
Returns a new size object based on `size` whose width and height is multiplied by *factor*. `(scale-size s factor)` is equivalent to:

```
(size (* (size-width s) factor)
      (* (size-height s) factor))
```

20.9 Rects

A *rect* describes a rectangle in terms of an upper left *point* and a *size*. Rects are represented as pairs whose car is a point and whose cdr is a size. Even though an expression like `'((1.0 . 2.0) . (3.0 4.0))` does represent a rect, it is recommended to always construct rects via function `rect`; e.g. `(rect (point 1.0 2.0) (size 3.0 4.0))`.

(rect? *obj*) procedure
Returns `#t` if *obj* is a valid rect. Otherwise, it returns `#f`.

(rect *point size*) procedure
(rect *x y width height*)

Returns a rect either from the given *point* and *size*, or from x-coordinate *x*, y-coordinate *y*, width *w*, and height *h*.

(move-rect *rect dx dy*) procedure
Moves *rect* by *dx* and *dy* and returns the result.

(rect-point *rect*) procedure
Returns the upper left corner point of *rect*.

(rect-size *rect*) procedure
Returns the size of the *rect*.

(rect-x *rect*) procedure
Returns the x-coordinate of the upper left corner point of *rect*.

(rect-y *rect*) procedure
Returns the y-coordinate of the upper left corner point of *rect*.

(rect-size *rect*) procedure
Returns the size of *rect* as a size, i.e. as a pair of floating-point numbers where the car representes the width and the cdr represents the height of *rect*.

(rect-width *rect*) procedure
Returns the width of *rect*.

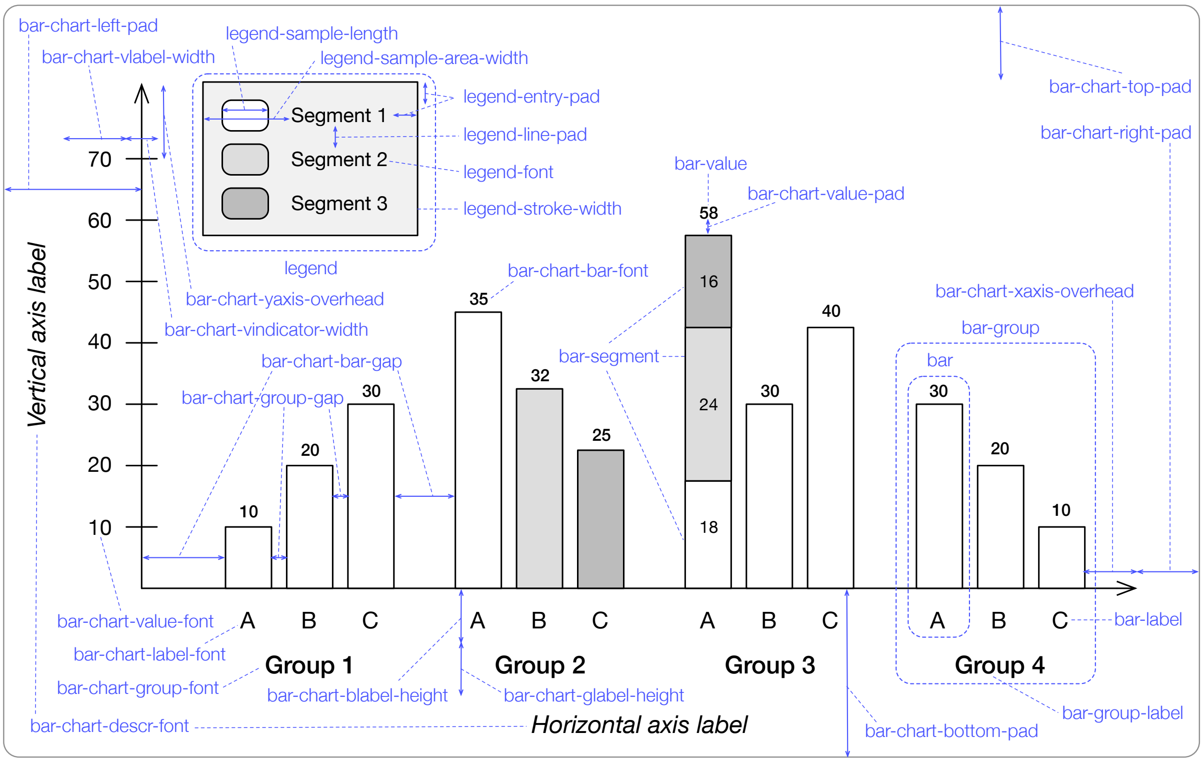
(rect-height *rect*) procedure
Returns the height of *rect*.

21 LispKit Draw Chart Bar

Library `(lispkit draw chart bar)` supports drawing bar charts based on a data-driven API in which *bar charts* are being described declaratively. A drawing function is then able to draw a bar chart into a given drawing as defined by library `(lispkit draw)`.

21.1 Bar Chart Model

The following diagram shows a generic bar chart model including the various elements that make up bar charts and parameters that can be configured:



The bar chart model on which library `(lispkit draw chart bar)` is based on consists of the following components:

- A bar chart is defined by a list of *bar groups*.
- Each *bar group* consists of a list of *bars* and an optional label.
- Each *bar* consists of a list of *values* and optionally a label and a color.
- Each *value* of a bar matches a *bar segment* (via the position in the list).
- A list of *bar segments* is provided when a bar chart is being drawn. Each *bar segment* consists of a label, a bar color, and optionally, a text color (for the value shown in the bar).
- A *legend* shows the provided *bar segments* and the color used to highlight them in the bar chart.

- A *legend configuration* specifies how the legend is layed out (see the various parameters in the diagram above)
- A *bar chart configuration* specifies how the bar chart overall is being drawn. This includes all fonts, offsets, padding values, etc. that are shown in the diagram above.

21.2 Legend Configurations

A *legend configuration* is a record encapsulating all parameters needed for drawing a legend in a bar chart. Legend configurations are mutable objects that are created via procedure `make-legend-config`. For every parameter, there is an accessor and a setter procedure.

(make-legend-config *key val* ...) procedure
Creates a new legend configuration object from the provided keyword/value pairs. The following keyword arguments are supported. The default value is provided in parenthesis.

- `font:` Font used for all text in a legend (Helvetica 10).
- `stroke-width:` Width of a stroke for drawing the bounding box of the legend (1.0).
- `horizontal-offset:` Horizontal offset from chart bounds (negative values are interpreted as offsets from the right bound) (70).
- `vertical-offset:` Vertical offset from chart bounds (negative values are interpreted as offsets from the bottom bound) (10).
- `sample-area-width:` Width of the sample area (17).
- `sample-length:` Heigh and width of color sample boxes for the various segments (10).
- `line-pad:` Padding between segment lines (3).
- `entry-pad:` Top/bottom and left/right padding around segment descriptions including the sample area (6).

```
(make-legend-config
  'font: (font "Helvetica" 7)
  'stroke-width: 0.4
  'entry-pad: 5
  'sample-area-width: 16
  'sample-length: 8
  'horizontal-offset: 50)
```

(legend-config? *obj*) procedure
Returns `#t` if *obj* is a legend configuration object, `#f` otherwise.

(legend-font *lconf*) procedure
Returns the font defined by the given legend configuration *lconf*.

(legend-font-set! *lconf font*) procedure
Sets the font for the given legend configuration *lconf* to *font*.

(legend-stroke-width *lconf*) procedure
Returns the stroke width defined by the given legend configuration *lconf*.

(legend-stroke-width-set! *lconf val*) procedure
Sets the stroke width for the given legend configuration *lconf* to *val*.

(legend-horizontal-offset *lconf*) procedure
Returns the horizontal offset defined by the given legend configuration *lconf*.

(legend-horizontal-offset-set! *lconf val*) procedure
Sets the horizontal offset for the given legend configuration *lconf* to *val*.

(legend-vertical-offset *lconf*) procedure
Returns the vertical offset defined by the given legend configuration *lconf*.

(legend-vertical-offset-set! *lconf val*) procedure
Sets the horizontal offset for the given legend configuration *lconf* to *val*.

(legend-sample-area-width *lconf*) procedure
Returns the sample area width defined by the given legend configuration *lconf*.

(legend-sample-area-width-set! *lconf val*) procedure
Sets the sample area width for the given legend configuration *lconf* to *val*.

(legend-sample-length *lconf*) procedure
Returns the sample length defined by the given legend configuration *lconf*.

(legend-sample-length-set! *lconf val*) procedure
Sets the sample length for the given legend configuration *lconf* to *val*.

(legend-line-pad *lconf*) procedure
Returns the line padding defined by the given legend configuration *lconf*.

(legend-line-pad-set! *lconf val*) procedure
Sets the line padding for the given legend configuration *lconf* to *val*.

(legend-entry-pad *lconf*) procedure
Returns the entry padding defined by the given legend configuration *lconf*.

(legend-entry-pad-set! *lconf val*) procedure
Sets the entry padding for the given legend configuration *lconf* to *val*.

21.3 Bar Chart Configurations

A *bar chart configuration* is a record encapsulating all parameters needed for drawing a bar chart (excluding the bar chart legend). Bar chart configurations are mutable objects that are created via procedure `make-bar-chart-config` . For every parameter of the configuration, there is an accessor and a setter procedure.

(make-bar-chart-config *key val ...*) procedure
Creates a new bar chart configuration object from the provided keyword/value pairs. The following keyword arguments are supported. The default value is provided in parenthesis.

- `size:` The rectangle in which the chart is drawn (495 x 200)
- `color:` Color of text, axis, etc. (black)
- `bg-color:` Color of legend background (white)
- `value-font:` Font for displaying values (Helvetica 10)
- `bar-font:` Font for displaying values on top of bars (Helvetica 11)
- `label-font:` Font for displaying bar labels (Helvetica 12)
- `group-font:` Font for displaying bar group labels (Helvetica-Bold 12)
- `descr-font:` Font for describing the axis (Helvetica-LightOblique 10)
- `stroke-width:` Width of a stroke in pixels (1.0)
- `top-pad:` Top padding in pixels (20)
- `bottom-pad:` Padding below the x axis (5)
- `right-pad:` Right-side padding (15)
- `left-pad:` Padding left to the y axis (10)
- `bar-gap:` Space between two bar groups (20)
- `group-gap:` Space between two bars within a group (5)

- `vlabel-width:` Width of labels on y axis (50)
- `vindicator-width:` Width of y label indicator lines (10)
- `vline-lengths:` List of alternating dash/space lengths; can be set to `#f` to disable the line ((1 2))
- `value-pad:` Padding between bar and displayed value (1)
- `blabel-height:` Height of bar labels (14)
- `glabel-height:` Height of group labels (30)
- `xaxis-overhead:` Overhead on x axis (20)
- `yaxis-overhead:` Overhead on y axis (20)

```
(make-bar-chart-config
  'size: (size 495 200)
  'color: (color 0.9 0.9 0.9)
  'value-font: (font "Helvetica" 8.5)
  'bar-font: (font "Helvetica" 8)
  'label-font: (font "Helvetica" 9)
  'top-pad: 5
  'left-pad: 10
  'right-pad: 5
  'bar-gap: 10
  'vlabel-width: 34
  'glabel-height: 5
  'blabel-height: 20)
```

(bar-chart-config? *obj*) procedure

Returns `#t` if *obj* is a bar chart configuration, otherwise `#f` is returned.

(bar-chart-size *bconf*) procedure

Returns the size defined by the given bar chart configuration *bconf*.

(bar-chart-size-set! *bconf size*) procedure

Sets the size for the given bar chart configuration *bconf* to *size*. *size* is a size object.

(bar-chart-value-font *bconf*) procedure

Returns the value font defined by the given bar chart configuration *bconf*.

(bar-chart-value-font-set! *bconf font*) procedure

Sets the value font for the given bar chart configuration *bconf* to *font*.

(bar-chart-bar-font *bconf*) procedure

Returns the bar font defined by the given bar chart configuration *bconf*.

(bar-chart-bar-font-set! *bconf font*) procedure

Sets the bar font for the given bar chart configuration *bconf* to *font*.

(bar-chart-label-font *bconf*) procedure

Returns the label font defined by the given bar chart configuration *bconf*.

(bar-chart-label-font-set! *bconf font*) procedure

Sets the label font for the given bar chart configuration *bconf* to *font*.

(bar-chart-group-font *bconf*) procedure

Returns the group font defined by the given bar chart configuration *bconf*.

(bar-chart-group-font-set! *bconf font*) procedure

Sets the group font for the given bar chart configuration *bconf* to *font*.

(bar-chart-descr-font *bconf*) procedure

Returns the description font defined by the given bar chart configuration *bconf*.

(bar-chart-descr-font-set! *bconf font*) procedure
Sets the description font for the given bar chart configuration *bconf* to *font*.

(bar-chart-stroke-width *bconf*) procedure
Returns the stroke width defined by the given bar chart configuration *bconf*.

(bar-chart-stroke-width-set! *bconf val*) procedure
Sets the stroke width for the given bar chart configuration *bconf* to *val*.

(bar-chart-top-pad *bconf*) procedure
Returns the top padding defined by the given bar chart configuration *bconf*.

(bar-chart-top-pad-set! *bconf val*) procedure
Sets the top padding for the given bar chart configuration *bconf* to *val*.

(bar-chart-bottom-pad *bconf*) procedure
Returns the bottom padding defined by the given bar chart configuration *bconf*.

(bar-chart-bottom-pad-set! *bconf val*) procedure
Sets the bottom padding for the given bar chart configuration *bconf* to *val*.

(bar-chart-right-pad *bconf*) procedure
Returns the right padding defined by the given bar chart configuration *bconf*.

(bar-chart-right-pad-set! *bconf val*) procedure
Sets the right padding for the given bar chart configuration *bconf* to *val*.

(bar-chart-left-pad *bconf*) procedure
Returns the left padding defined by the given bar chart configuration *bconf*.

(bar-chart-left-pad-set! *bconf val*) procedure
Sets the left padding for the given bar chart configuration *bconf* to *val*.

(bar-chart-bar-gap *bconf*) procedure
Returns the bar gap defined by the given bar chart configuration *bconf*.

(bar-chart-bar-gap-set! *bconf val*) procedure
Sets the bar gap for the given bar chart configuration *bconf* to *val*.

(bar-chart-group-gap *bconf*) procedure
Returns the group gap defined by the given bar chart configuration *bconf*.

(bar-chart-group-gap-set! *bconf val*) procedure
Sets the group gap for the given bar chart configuration *bconf* to *val*.

(bar-chart-vlabel-width *bconf*) procedure
Returns the vertical label width defined by the given bar chart configuration *bconf*.

(bar-chart-vlabel-width-set! *bconf val*) procedure
Sets the vertical label width for the given bar chart configuration *bconf* to *val*.

(bar-chart-vindicator-width *bconf*) procedure
Returns the vertical value indicator width defined by the given bar chart configuration *bconf*.

(bar-chart-vindicator-width-set! *bconf val*) procedure
Sets the vertical value indicator width for the given bar chart configuration *bconf* to *val*.

(bar-chart-vline-lengths *bconf*) procedure
Returns a list of alternating dash/space lengths defined by the given bar chart configuration *bconf*. If `#f` is returned, no horizontal value lines are drawn.

(bar-chart-vline-lengths-set! *bconf val*) procedure
Sets the list of alternating dash/space lengths for the given bar chart configuration *bconf* to *val*. *val* may be set to `#f` to disable drawing horizontal value lines.

(bar-chart-value-pad *bconf*) procedure
Returns the value padding defined by the given bar chart configuration *bconf*.

(bar-chart-value-pad-set! *bconf val*) procedure
Sets the value padding for the given bar chart configuration *bconf* to *val*.

(bar-chart-blabel-height *bconf*) procedure
Returns the value padding, i.e. the space between bar and displayed value, defined by the given bar chart configuration *bconf*.

(bar-chart-blabel-height-set! *bconf val*) procedure
Sets the value padding, i.e. the space between bar and displayed value, for the given bar chart configuration *bconf* to *val*.

(bar-chart-glabel-height *bconf*) procedure
Returns the group label height defined by the given bar chart configuration *bconf*.

(bar-chart-glabel-height-set! *bconf val*) procedure
Sets the group label height for the given bar chart configuration *bconf* to *val*.

(bar-chart-xaxis-overhead *bconf*) procedure
Returns the overhead on the x axis of the coordinate system defined by the given bar chart configuration *bconf*.

(bar-chart-xaxis-overhead-set! *bconf val*) procedure
Sets the overhead on the x axis of the coordinate system for the given bar chart configuration *bconf* to *val*.

(bar-chart-yaxis-overhead *bconf*) procedure
Returns the overhead on the y axis of the coordinate system defined by the given bar chart configuration *bconf*.

(bar-chart-yaxis-overhead-set! *bconf val*) procedure
Sets the overhead on the y axis of the coordinate system for the given bar chart configuration *bconf* to *val*.

21.4 Constructing Bar Charts

(bar-spec? *obj*) procedure
Returns `#t` if *obj* is a bar diagram specification, `#f` otherwise. A bar diagram specification is a list of bars and bar groups.

(bar? *obj*) procedure
Returns `#t` if *obj* is a bar object, otherwise `#f` is returned.

(bar *label value ...*) procedure
(bar *label color value ...*)

Creates a new bar object. A bar without bar segments consists of a single value and an optional label string (`#f` disables the label) and color. A segmented bar has a value for all segments (i.e. all bars of a bar diagram should have the same number of segments). A segment is disabled by setting its value to 0.

(bar-label *bar*) procedure
Returns the label of the given bar object *bar*.

(bar-color *bar*) procedure
Returns the color of the given bar object *bar*.

(bar-values *bar*) procedure

Returns the values of the given bar object *bar*.

(bar-group? *obj*) procedure

Returns `#t` if *obj* is a bar group object, otherwise `#f` is returned.

(bar-group *label bar* ...) procedure

Creates a new bar group from the bars *bar* ... with string *label* as label.

(bar-group-label *group*) procedure

Returns the label of the given bar group *group*.

(bar-group-bars *group*) procedure

Returns the bars of the given bar group *group*.

(bar-segment *label col*) procedure
(bar-segment *label col textcol*)

Creates a bar segment represented by label string *label* and segment color *col*. Text color *textcol* is optional (and might be `#f`).

21.5 Drawing Bar Charts

(draw-bar-chart *bars col ystep ydescr xdescr loc config legend*) procedure
(draw-bar-chart *bars col ystep ydescr xdescr loc config legend drawing*)

Draws the bar diagram *bars* with *col* as the default bar color into the drawing *drawing*. *ystep* defines the increment between values on the y axis. *ydescr* defines the label of the y axis. *xdescr* defines the label of the x axis. *loc* is a point at which the bar diagram is drawn with the bar diagram configuration *config*. If a legend should be drawn, a legend configuration needs to be provided as parameter *legend*.

```
(draw-bar-chart
  (list
    (bar "Jan" 0) (bar "Feb" 2) (bar "Mar" 6)
    (bar "Apr" 9) (bar "May" 14) (bar "Jun" 16)
    (bar "Jul" 19) (bar "Aug" 18) (bar "Sep" 15)
    (bar "Oct" 11) (bar "Nov" 5) (bar "Dec" 2))
  gray 5
  "Temperature [C°]" "Month"
  (point 50 105)
  (make-bar-chart-config
    'size: (size 495 200))
  #f)
```

22 LispKit Draw Turtle

Library `(lispkit draw turtle)` defines a simple "turtle graphics" API. The API provides functionality for creating turtles and for moving turtles on a plane generating *drawings* as a side-effect. A *drawing* is a data structure defined by library `(lispkit draw)`.

A *turtle* is defined in terms of the following components: - A position *(x, y)* defining the coordinates where the turtle is currently located within a coordinate system defined by parameters used to create the turtle via `make-turtle` - A heading *angle* which defines the direction in degrees into which the turtle is moving - A boolean flag *pen down* which, if set to `#t`, will make the turtle draw lines on the graphics plane when moving. - A *line width* defining the width of lines drawn by the turtle - A *color* defining the color of lines drawn by the turtle - A *drawing* which records the moves of the turtle while the pen is down.

Turtles are mutable objects created via `make-turtle`. The functions listed below change the state of a turtle. In particular, they generate a drawing as a side-effect which can be accessed via `turtle-drawing`. For most functions, the turtle is an optional argument. If it is not provided, the function applies to the turtle provided by the `current-turtle` parammeter object.

current-turtle `parameter object`
Defines the *current turtle*, which is used as a default by all functions for which the turtle argument is optional. If there is no current turtle, this parameter is set to `#f`.

(turtle? *obj*) `procedure`
Returns `#t` if *obj* is a turtle. Otherwise, it returns `#f`.

(make-turtle *x y scale*) `procedure`
Returns a new turtle object. *x* and *y* determine the "home point" of the turtle. This is equivalent to the zero point of the coordinate system in which the turtle navigates. *scale* is a scaling factor.

(turtle-drawing *turtle*) `procedure`
Returns the drawing associated with the given *turtle*.

(pen-up) `procedure`
(pen-up *turtle*)

Lifts *turtle* from the plane. If *turtle* is not provided, the turtle defined by `current-turtle` is used. Subsequent `forward` and `backward` operations don't lead to lines being drawn. Only the current coordinates are getting updated.

(pen-down) `procedure`
(pen-down *turtle*)

Drops *turtle* onto the plane. If *turtle* is not provided, the turtle defined by `current-turtle` is used. Subsequent `forward` and `backward` operations will lead to lines being drawn.

(pen-color *color*) `procedure`
(pen-color *color turtle*)

Sets the drawing color of *turtle* to *color*. If *turtle* is not provided, the turtle defined by `current-turtle` is used. *color* is a color object as defined by library `(lispkit draw)`.

(pen-size *size*) procedure
(pen-size *size turtle*)

Sets the pen size of *turtle* to *size*. If *turtle* is not provided, the turtle defined by `current-turtle` is used. The pen size corresponds to the width of lines drawn by `forward` and `backward`.

(home) procedure
(home *turtle*)

Moves *turtle* to its home position. If *turtle* is not provided, the turtle defined by `current-turtle` is used.

(move *x y*) procedure
(move *x y turtle*)

Moves *turtle* to the position described by the coordinates *x* and *y*. If *turtle* is not provided, the turtle defined by `current-turtle` is used.

(heading *angle*) procedure
(heading *angle turtle*)

Sets the heading of *turtle* to *angle*. If *turtle* is not provided, the turtle defined by `current-turtle` is used. *angle* is expressed in terms of degrees.

(turn *angle*) procedure
(turn *angle turtle*)

Adjusts the heading of *turtle* by *angle* degrees. If *turtle* is not provided, the turtle defined by `current-turtle` is used.

(right *angle*) procedure
(right *angle turtle*)

Adjusts the heading of *turtle* by *angle* degrees. If *turtle* is not provided, the turtle defined by `current-turtle` is used.

(left *angle*) procedure
(left *angle turtle*)

Adjusts the heading of *turtle* by -*angle* degrees. If *turtle* is not provided, the turtle defined by `current-turtle` is used.

(forward *distance*) procedure
(forward *distance turtle*)

Moves *turtle* forward by *distance* units drawing a line if the pen is down. If *turtle* is not provided, the turtle defined by `current-turtle` is used.

(backward *distance*) procedure
(backward *distance turtle*)

Moves *turtle* backward by *distance* units drawing a line if the pen is down. If *turtle* is not provided, the turtle defined by `current-turtle` is used.

23 LispKit Dynamic

23.1 Dynamic bindings

(make-parameter *init*) procedure
(make-parameter *init converter*)

Returns a newly allocated parameter object, which is a procedure that accepts zero arguments and returns the value associated with the parameter object. Initially, this value is the value of `(` *converter init* `)`, or of *init* if the conversion procedure *converter* is not specified. The associated value can be temporarily changed using `parameterize`. The default associated value can be changed by invoking the parameter object as a function with the new value as the only argument.

Parameter objects can be used to specify configurable settings for a computation without the need to pass the value to every procedure in the call chain explicitly.

(parameterize ((*param value*) ...) *body*) syntax

A `parameterize` expression is used to change the values returned by specified parameter objects *param* during the evaluation of *body*. The *param* and *value* expressions are evaluated in an unspecified order. The *body* is evaluated in a dynamic environment in which calls to the parameters return the results of passing the corresponding values to the conversion procedure specified when the parameters were created. Then the previous values of the parameters are restored without passing them to the conversion procedure. The results of the last expression in the *body* are returned as the results of the entire `parameterize` expression.

```
(define radix
  (make-parameter 10 (lambda (x)
                       (if (and (exact-integer? x) (<= 2 x 16))
                           x
                           (error "invalid radix")))))
(define (f n) (number->string n (radix)))
(f 12)                              ⇒  "12"
(parameterize ((radix 2)) (f 12))   ⇒  "1100"
(f 12)                              ⇒  "12"
(radix 16)
(parameterize ((radix 0)) (f 12))   ⇒  error: invalid radix
```

(make-dynamic-environment) syntax

Returns a newly allocated copy of the current dynamic environment. Dynamic environments are represented as mutable hashtables.

(dynamic-environment) syntax

Returns the current dynamic environment represented as mutable hashtables.

(set-dynamic-environment! *hashtable*) syntax

Sets the current dynamic environment to the given dynamic environment object. Dynamic environments are modeled as hashtables.

23.2 Continuations

(continuation? *obj*) procedure

Returns #t if *obj* is a continuation procedure, #f otherwise.

(call-with-current-continuation *proc*) procedure
(call/cc *proc*)

The procedure call-with-current-continuation (or its equivalent abbreviation call/cc) packages the current continuation as an "escape procedure" and passes it as an argument to *proc*. It is an error if *proc* does not accept one argument.

The escape procedure is a Scheme procedure that, if it is later called, will abandon whatever continuation is in effect at that later time and will instead use the continuation that was in effect when the escape procedure was created. Calling the escape procedure will cause the invocation of before and after thunks installed using dynamic-wind.

The escape procedure accepts the same number of arguments as the continuation to the original call to call-with-current-continuation. Most continuations take only one value. Continuations created by the call-with-values procedure (including the initialization expressions of define-values, let-values, and let*-values expressions), take the number of values that the consumer expects. The continuations of all non-final expressions within a sequence of expressions, such as in lambda, case-lambda, begin, let, let*, letrec, letrec*, let-values, let*-values, let-syntax, letrec-syntax, parameterize, guard, case, cond, when, and unless expressions, take an arbitrary number of values because they discard the values passed to them in any event. The effect of passing no val- ues or more than one value to continuations that were not created in one of these ways is unspecified.

The escape procedure that is passed to proc has unlimited extent just like any other procedure in Scheme. It can be stored in variables or data structures and can be called as many times as desired. However, like the raise and error procedures, it never returns to its caller.

The following examples show only the simplest ways in which call-with-current-continuation is used. If all real uses were as simple as these examples, there would be no need for a procedure with the power of call-with-current-continuation.

```
(call-with-current-continuation
  (lambda (exit)
    (for-each (lambda (x) (if (negative? x) (exit x)))
              '(54 0 37 -3 245 19)) #t))  ⇒  -3
(define list-length
  (lambda (obj)
    (call-with-current-continuation
      (lambda (return)
        (letrec
          ((r (lambda (obj)
                (cond ((null? obj) 0)
                      ((pair? obj) (+ (r (cdr obj)) 1))
                      (else        (return #f))))))
          (r obj))))))
(list-length '(1 2 3 4))      ⇒  4
(list-length '(a b . c))      ⇒  #f
```

(dynamic-wind *before thunk after*) procedure

Calls *thunk* without arguments, returning the result(s) of this call. *before* and *after* are called, also without arguments, as required by the following rules. Note that, in the absence of calls to continuations captured using call-with-current-continuation, the three arguments are called once each, in order. *before* is called whenever execution enters the dynamic extent of the call to *thunk* and *after* is called whenever it exits that dynamic extent. The dynamic extent of a procedure call is the period between when the call is initiated and when it returns. The *before* and *after* thunks are called in the same dynamic environment as the call to `dynamic-wind`. In Scheme, because of `call-with-current-continuation`, the dynamic extent of a call is not always a single, connected time period. It is defined as follows:

- The dynamic extent is entered when execution of the body of the called procedure begins.
- The dynamic extent is also entered when execution is not within the dynamic extent and a continuation is invoked that was captured (using `call-with-current-continuation`) during the dynamic extent.
- It is exited when the called procedure returns.
- It is also exited when execution is within the dynamic extent and a continuation is invoked that was captured while not within the dynamic extent.

If a second call to `dynamic-wind` occurs within the dynamic extent of the call to *thunk* and then a continuation is invoked in such a way that the *afters* from these two invocations of `dynamic-wind` are both to be called, then the *after* associated with the second (inner) call to `dynamic-wind` is called first.

If a second call to `dynamic-wind` occurs within the dynamic extent of the call to *thunk* and then a continuation is invoked in such a way that the *befores* from these two invocations of `dynamic-wind` are both to be called, then the before associated with the first (outer) call to `dynamic-wind` is called first.

If invoking a continuation requires calling the *before* from one call to `dynamic-wind` and the after from another, then the *after* is called first.

The effect of using a captured continuation to enter or exit the dynamic extent of a call to *before* or *after* is unspecified.

```
(let ((path ’())
      (c #f))
  (let ((add (lambda (s)
         (set! path (cons s path)))))
    (dynamic-wind
      (lambda () (add 'connect))
      (lambda () (add (call-with-current-continuation
                        (lambda (c0) (set! c c0) 'talk1))))
      (lambda () (add 'disconnect)))
    (if (< (length path) 4)
        (c 'talk2)
        (reverse path))))
  ⇒  (connect talk1 disconnect connect talk2 disconnect)
```

(unwind-protect *body cleanup* ...) syntax

Executes expression *body* guaranteeing that statements *cleanup* ... are executed when *body*'s execution is finished or when an exception is thrown during the execution of *body*. `unwind-protect` returns the result of executing *body*.

(return *obj*) procedure

Returns to the top-level of the read-eval-print loop with *obj* as the result (or terminates the program with *obj* as its return value).

23.3 Exceptions

(with-exception-handler *handler thunk*) procedure

The `with-exception-handler` procedure returns the results of invoking *thunk*. *handler* is installed as the current exception handler in the dynamic environment used for the invocation of *thunk*. It is an error if *handler* does not accept one argument. It is also an error if *thunk* does not accept zero arguments.

```
(call-with-current-continuation
  (lambda (k)
    (with-exception-handler
      (lambda (x)
        (display "condition: ")(write x)(newline)(k 'exception))
      (lambda ()
        (+ 1 (raise 'an-error))))))  ⇒ exception; prints "condition: an-error"
(with-exception-handler
  (lambda (x) (display "something went wrong\n"))
  (lambda () (+ 1 (raise 'an-error))))   ⇒ prints "something went wrong"
```

After printing, the second example then raises another exception: “exception handler returned”.

(try *thunk*) procedure
(try *thunk handler*)

`try` executes argument-less procedure *thunk* and returns the result as the result of `try` if *thunk*’s execution completes normally. If an exception is thrown, procedure *handler* is called with the exception object as its argument. The result of executing *handler* is returned by *try*.

(guard (*var cond-clause* ...) *body*) syntax
The *body* is evaluated with an exception handler that binds the raised object to *var* and, within the scope of that binding, evaluates the clauses as if they were the clauses of a `cond` expression. That implicit `cond` expression is evaluated with the continuation and dynamic environment of the `guard` expression. If every *cond-clause*’s *test* evaluates to `#f` and there is no “else” clause, then `raise-continuable` is invoked on the raised object within the dynamic environment of the original call to `raise` or `raise-continuable` , except that the current exception handler is that of the `guard` expression.

Please note that each *cond-clause* is as in the specification of `cond` .

```
(guard (condition
         ((assq 'a condition) => cdr)
         ((assq 'b condition)))
  (raise (list (cons 'a 42))))         ⇒  42
(guard (condition
         ((assq 'a condition) => cdr)
         ((assq 'b condition)))
  (raise (list (cons 'b 23))))         ⇒ (b . 23)
```

(make-error *message irrlist*) procedure
Returns a newly allocated custom error object consisting of *message* as its error message and the list of irritants *irrlist*.

(make-assertion-error *procname expr*) procedure
Returns a newly allocated assertion error object referring to a procedure of name *procname* and an expression *expr* which triggered the assertion. Assertion errors that were raised should never be caught as they indicate a violation of an invariant.

(raise *obj*) procedure
Raises an exception by invoking the current exception handler on *obj*. The handler is called with the same dynamic environment as that of the call to `raise` , except that the current exception handler is the one that was in place when the handler being called was installed. If the handler returns, a secondary exception is raised in the same dynamic environment as the handler. The relationship between *obj* and the object raised by the secondary exception is unspecified.

(raise-continuable *obj*) procedure
Raises an exception by invoking the current exception handler on *obj*. The handler is called with the same dynamic environment as the call to `raise-continuable` , except that: (1) the current exception handler

is the one that was in place when the handler being called was installed, and (2) if the handler being called returns, then it will again become the current exception handler. If the handler returns, the values it returns become the values returned by the call to `raise-continuable`.

```
(with-exception-handler
  (lambda (con)
    (cond ((string? con) (display con))
          (else          (display "a warning has been issued")))
    42)
  (lambda ()
    (+ (raise-continuable "should be a number") 23)))
prints: should be a number
  ⇒  65
```

(error *message obj* ...) procedure

Raises an exception as if by calling `raise` on a newly allocated error object which encapsulates the information provided by *message*, as well as any *obj*, known as the irritants. The procedure `error-object?` must return `#t` on such objects. *message* is required to be a string.

```
(define (null-list? l)
  (cond ((pair? l) #f)
        ((null? l) #t)
        (else (error "null-list?: argument out of domain" l))))
```

(assertion *expr*) procedure

Raises an exception as if by calling `raise` on a newly allocated assertion error object encapsulating *expr* as the expression which triggered the assertion failure and the current procedure's name. Assertion errors that are raised via `assertion` should never be caught as they indicate a violation of a critical invariant.

```
(define (null-list? l)
  (cond ((pair? l) #f)
        ((null? l) #t)
        (else (assertion '(list? l)))))
```

(assert *expr0 expr1* ...) syntax

Executes *expr0*, *expr1*, ... in the given order and raises an assertion error as soon as the first expression is evaluating to `#f`. The raised assertion error encapsulates the expression that evaluated to `#f` and the name of the procedure in which the `assert` statement was placed.

```
(define (drop-elements xs n)
  (assert (list? xs) (fixnum? n) (not (negative? n)))
  (if (or (null? xs) (zero? n)) xs (drop-elements (cdr xs) (fx1- n))))
```

(error-object? *obj*) procedure

Returns `#t` if *obj* is an error object, `#f` otherwise. Error objects are either implicitly created via `error` or they are created explicitly with procedure `make-error`.

(error-object-message *err*) procedure

Returns the message (which is a string) encapsulated by the error object *err*.

(error-object-irritants *err*) procedure

Returns a list of the irritants encapsulated by the error object *err*.

(error-object-stacktrace *err*) procedure

Returns a list of procedures representing the stack trace encapsulated by the error object *err*. The stack trace reflects the currently active procedures at the time the error object was created (either implicitly via `error` or explicitly via `make-error`).

(read-error? *obj*) procedure

This error type predicate returns `#t` if *obj* is an error object raised by the `read` procedure; otherwise, it returns `#f`.

(file-error? *obj*) procedure

This error type predicate returns `#t` if *obj* is an error object raised by the inability to open an input or output port on a file; otherwise, it returns `#f`.

23.4 Exiting

(exit) procedure
(exit *obj*)

Runs all outstanding `dynamic-wind` after procedures, terminates the running program, and communicates an exit value to the operating system. If no argument is supplied, or if *obj* is `#t`, the `exit` procedure should communicate to the operating system that the program exited normally. If *obj* is `#f`, the `exit` procedure will communicate to the operating system that the program exited abnormally. Otherwise, `exit` should translate *obj* into an appropriate exit value for the operating system, if possible. The `exit` procedure must not signal an exception or return to its continuation.

(emergency-exit) procedure
(emergency-exit *obj*)

Terminates the program without running any outstanding `dynamic-wind` "after procedures" and communicates an exit value to the operating system in the same manner as `exit`.

24 LispKit Enum

Library `(lispkit enum)` implements an API for enumeration types, enumeration values, and enumeration sets.

An enumeration type is defined by a list of tagged enumeration names. It encapsulates enumeration values which can be accesses either by name or ordinal value. Sets of these enumeration values are called enumeration sets. Each enumeration set is based on an enumeration type and contains a set of enumeration values.

24.1 Declarative API

(define-enum *type-name (symbol ...) constructor*) syntax
(define-enumeration *type-name (symbol ...) constructor*) syntax
The `define-enum` and `define-enumeration` forms defines an enumeration type and provide two macros for constructing its members and sets of its members.

type-name is an identifier that is bound as a syntactic keyword; *symbol* ... are the symbols that comprise the universe of the enumeration (in order).

`(` *type-name symbol* `)` checks whether the name of *symbol* is in the universe associated with *type-name*. If it is, `(` *type-name symbol* `)` is equivalent to *symbol*. It is a syntax violation if it is not.

constructor is an identifier that is bound to a syntactic form that, given any finite sequence of the symbols in the universe, possibly with duplicates, expands into an expression that evaluates to the enumeration set of those symbols.

`(` *constructor symbol* ... `)` checks whether every ... is in the universe associated with *type-name*. It is a syntax violation if one or more is not. Otherwise `(` *constructor symbol*> ... `)` is equivalent to `((enum-set-constructor (`*constructor-syntax*`)) '(`*symbol* ...`))`.

Here is a complete example:

```
(define-enumeration color
  (black white purple maroon)
  color-set)
(color black)                 ⇒ black
(color purpel)                ⇒ error: symbol not in enumeration universe
(enum-set->list (color-set))  ⇒ ()
(enum-set->list
  (color-set maroon white))   ⇒ (white maroon)
```

24.2 Enum types

(enum-type? *obj*) procedure
Returns `#t` if *obj* is an enum type, and `#f` otherwise.

(make-enum-type *list*) procedure
(make-enum-type *name list*)
(make-enum-type *name list tag*)

Returns a newly allocated enum type containing a fixed set of newly allocated enums. Both enums and enum types are immutable, and it is not possible to create an enum except as part of creating an enum type. *name* is the name of the enumeration as a string or symbol, *tag* is an arbitrary object attached to the enum type (which can be accessed via `enum-type-tag`).

The elements of *list* are either symbols or two-element lists, where each list has a symbol as the first element and any value as the second element (this is the enum's tag). Each list element causes a single enum to be generated, and the enum's name is specified by the symbol. It is an error unless all the symbols are distinct within an enum type. The position of the element in *list* is the ordinal of the corresponding enum, so ordinals within an enum type are also distinct. If a value is given, it becomes the value of the enum; otherwise the enum's value is the same as the ordinal.

Here are a few examples:

```
(define color
  (make-enum-type
    '(red orange yellow green cyan blue violet)))

(define us-traffic-light
  (make-enum-type '(red yellow green)))

(define pizza
  (make-enum-type
    '((margherita "tomato and mozzarella")
      (funghi "mushrooms")
      (chicago "deep-dish")
      (hawaiian "pineapple and ham"))))
```

(enum-type-type-tag *enum-type*) procedure
Returns the type tag, i.e. an uninterned symbol, representing the type of enums as defined by *enum-type*.

(enum-type-size *enum-type*) procedure
Returns an exact integer equal to the number of enums in *enum-type*.

(enum-min *enum-type*) procedure
Returns the enum belonging to *enum-type* whose ordinal is 0.

(enum-max *enum-type*) procedure
Returns the enum belonging to *enum-type* whose ordinal is equal to the number of enums in the enum type minus 1.

(enum-type-tag *enum-type*) procedure
Returns the tag associated with *enum-type*.

(enum-type-enums *enum-type*) procedure
Returns a list of the enums belonging to *enum-type* ordered by increasing ordinal.

(enum-type-names *enum-type*) procedure
Returns a list of the names of the enums belonging to *enum-type* ordered by increasing ordinal.

(enum-type-tags *enum-type*) procedure
Returns a list of the values of the enums belonging to *enum-type* ordered by increasing ordinal.

(enum-type-contains? *enum-type enum*) procedure
Returns `#t` if *enum* belongs to *enum-type*, and `#f` otherwise.

```
(enum-type-contains? color
  (enum-name->enum color 'red)) ⇒ #t
(enum-type-contains? pizza
  (enum-name->enum color 'red)) ⇒ #f
```

(enum-type-test-predicate *enum-type*) procedure
Returns a procedure which given an object, checks that this object is an enum that belongs to *enum-type*.

(enum-set-type-test-predicate *enum-type*) procedure
Returns a procedure which given an object, checks that this object is an enum set whose type matches *enum-type*.

(enum-name->enum *enum-type symbol*) procedure
If there exists an enum belonging to *enum-type* named *symbol*, returns it; otherwise return `#f`.

(enum-name->ordinal *enum-type symbol*) procedure
Returns the ordinal of the enum belonging to *enum-type* whose name is *symbol*. It is an error if there is no such enum.

(enum-name->tag *enum-type symbol*) procedure
Returns the value of the enum belonging to *enum-type* whose name is *symbol*. It is an error if there is no such enum.

(enum-ordinal->enum *enum-type exact-integer*) procedure
If there exists an enum belonging to *enum-type* whose ordinal is *exact-integer*, returns it; otherwise return `#f`.

(enum-ordinal->name *enum-type exact-integer*) procedure
Returns the name of the enum belonging to *enum-type* whose ordinal is *exact-integer*. It is an error if there is no such enum.

(enum-ordinal->tag *enum-type exact-integer*) procedure
Returns the value of the enum belonging to *enum-type* whose ordinal is *exact-integer*. It is an error if there is no such enum.

24.3 Enum values

(enum? *obj*) procedure
Returns `#t` if *obj* is an enum, and `#f` otherwise.

(enum-type *enum*) procedure
Returns the enum type to which *enum* belongs.

(enum-name *enum*) procedure
Returns the name (symbol) associated with *enum*.

(enum-ordinal *enum*) procedure
Returns the ordinal (exact integer) associated with *enum*.

(enum-tag *enum*) procedure
Returns the tag associated with *enum*.

(enum-next *enum*) procedure
Returns the enum that belongs to the same enum type as *enum* and has an ordinal one greater than *enum*. Returns `#f` if there is no such enum.

```
(enum-name (enum-next color-red)) ⇒ orange
(enum-next (enum-max color))      ⇒ #f
```

(enum-prev *enum*) procedure

Returns the enum that belongs to the same enum type as *enum* and has an ordinal one less than *enum*. Returns `#f` if there is no such enum.

(enum=? *enum0 enum1* ...) procedure

Returns `#t` if all the arguments are the same enum in the sense of `eq?` (which is equivalent to having the same name and ordinal) and `#f` otherwise. It is an error to apply `enum=?` to enums belonging to different enum types.

```
(enum=? color-red color-blue)      ⇒ #f
(enum=? pizza-funghi
  (enum-name->enum pizza 'funghi)) ⇒ #t
(enum=? color-red
  (enum-name->enum color 'red)
  color-blue)                      ⇒ #f
```

(enum<? *enum0 enum1* ...) procedure
(enum>? *enum0 enum1* ...) procedure
(enum<=? *enum0 enum1* ...) procedure
(enum>=? *enum0 enum1* ...) procedure

These predicates return `#t` if their arguments are enums whose ordinals are in increasing, decreasing, non-decreasing, and non-increasing order respectively, and `#f` otherwise. It is an error unless all of the arguments belong to the same enum type.

24.4 Enum sets

(enum-set? *obj*) procedure

Returns `#t` if *obj* is an enum set and `#f` otherwise.

(enum-set *enum-type enum* ...) procedure

Returns an enum set that can contain enums of the type *enum-type* and containing the *enums*. It is an error unless all the *enums* belong to *enum-type*.

```
(enum-set-contains?
  (enum-set color color-red color-blue)
  color-red)     ⇒  #t
(enum-set-contains?
  (enum-set color color-red color-blue)
  color-orange)  ⇒  #f
```

(list->enum-set *enum-type list*) procedure

Returns an enum set with the specified *enum-type* that contains the members of *list*. *list* may contain enums, enum names, or enum ordinals. It is an error unless all the members refer to enums belonging to *enum-type*.

(enum-set-copy *enum-set*) procedure

Returns a copy of *enum-set*.

(enum-set-empty? *enum-set*) procedure

Returns `#t` if *enum-set* is empty, and `#f` otherwise.

(enum-set-contains? *enum-set enum*) procedure

Returns `#t` if *enum* is a member of *enum-set*. It is an error if *enum* does not belong to the same enum type as the members of *enum-set*.

(enum-set-disjoint? *enum-set1 enum-set2*) procedure

Returns `#t` if *enum-set1* and *enum-set2* do not have any enum objects in common, and `#f` otherwise.

```
(define reddish
  (list->enum-set
    (map (lambda (name)
           (enum-name->enum color name))
         '(red orange))))
(define ~reddish
  (list->enum-set
    (map (lambda (name)
           (enum-name->enum color name))
         '(yellow green cyan blue violet))))
(enum-set-disjoint? color-set reddish) ⇒ #f
(enum-set-disjoint? reddish ~reddish)  ⇒ #t
```

(enum-set-projection *enum-set1 enum-set2*) procedure

Projects *enum-set1* into the universe of *enum-set2*, dropping any elements of *enum-set1* that do not belong to the universe of *enum-set2*. If *enum-set1* is a subset of the universe of its second, no elements are dropped, and the injection is returned.

```
(let ((e1 (make-enumeration '(red green blue black)))
      (e2 (make-enumeration '(red black white))))
  (enum-set->list (enum-set-projection e1 e2))))
⇒ (red black)
```

(enum-set-member? *symbol enum-set*) procedure
(enum-set-subset? *enum-set1 enum-set2*) procedure

The `enum-set-member?` procedure returns `#t` if its first argument is an element of its second argument, `#f` otherwise.

The `enum-set-subset?` procedure returns `#t` if the universe of *enum-set1* is a subset of the universe of *enum-set2* (considered as sets of symbols) and every element of *enum-set1* is a member of *enum-set2*. It returns `#f` otherwise.

```
(let* ((e (make-enumeration '(red green blue)))
       (c (enum-set-constructor e)))
  (list
    (enum-set-member? 'blue (c '(red blue)))
    (enum-set-member? 'green (c '(red blue)))
    (enum-set-subset? (c '(red blue)) e)
    (enum-set-subset? (c '(red blue)) (c '(blue red)))
    (enum-set-subset? (c '(red blue)) (c '(red)))
    (enum-set=? (c '(red blue)) (c '(blue red)))))
⇒ (#t #f #t #t #f #t)
```

(enum-set=? *enum-set1 enum-set2*) procedure

Returns `#t` if the members of *enum-set-1* are the same as of *enum-set-2*. It is an error if the members of the enum sets do not belong to the same type.

(enum-set<? *enum-set1 enum-set2*) procedure

Returns `#t` if the members of *enum-set-1* are a proper subset of *enum-set-2*. It is an error if the members of the enum sets do not belong to the same type.

(enum-set>? *enum-set1 enum-set2*) procedure

Returns `#t` if the members of *enum-set-1* are a proper superset of *enum-set-2*. It is an error if the members of the enum sets do not belong to the same type.

(enum-set<=? *enum-set1 enum-set2*) procedure

Returns `#t` if the members of *enum-set-1* are a subset of *enum-set-2*. It is an error if the members of the enum sets do not belong to the same type.

(enum-set>=? *enum-set1 enum-set2*) procedure

Returns `#t` if the members of *enum-set-1* are a superset of *enum-set-2*. It is an error if the members of the enum sets do not belong to the same type.

(enum-set->list *enum-set*) procedure

Returns a list of the names of enums that belong to *enum-set*. The list is in increasing order of the enums.

```
(let*
  ((e (make-enumeration '(red green blue)))
   (c (enum-set-constructor e)))
  (enum-set->list (c '(blue red))))
⇒ (red blue)
```

(enum-set->enum-list *enum-set*) procedure

Returns a list containing the enum members of *enum-set*. The list is in increasing order of the enums.

```
(let*
  ((e (make-enumeration '(red green blue)))
   (c (enum-set-constructor e)))
  (enum-set->enum-list (c '(blue red))))
⇒ (#<enum enum-3: 0> #<enum enum-3: 2>)
```

(enum-set-next *enum-set e*) procedure

Returns the ordinally next enum in *enum-set* following enum *e*. *e* is either a name, ordinal, or enum value. `enum-set-next` returns `#f` if there is no next enum.

(enum-set-type *enum-set*) procedure

Returns the enum type associated with *enum-set*.

(enum-set-bitset *enum-set*) procedure

Returns a bit set (as defined by library `(lispkit bitset)`) representing all ordinals of *enum-set*.

(enum-set-size *enum-set*) procedure

Returns the number of elements in *enum-set*.

(enum-set-adjoin! *enum-set e …*) procedure

Adds enums *e …* to *enum-set*. Enums are defined either via a name, an ordinal, or an enum object. It is an error if the enums denoted by *e …* do not all belong to the same enum type.

(enum-set-adjoin-all! *enum-set list*) procedure

list is a list of enums. Enums are defined either via a name, an ordinal, or an enum object. `enum-set-adjoin-all!` adds all enums of *list* to *enum-set*. It is an error if the enums denoted by *list* do not all belong to the same enum type.

(enum-set-delete! *enum-set e …*) procedure

Removes enums *e …* from *enum-set*. Enums are defined either via a name, an ordinal, or an enum object. It is an error if the enums denoted by *e …* do not all belong to the same enum type.

(enum-set-delete-all! *enum-set list*) procedure

list is a list of enums. Enums are defined either via a name, an ordinal, or an enum object. `enum-set-delete-all!` removes all enums of *list* from *enum-set*. It is an error if the enums denoted by *list* do not all belong to the same enum type.

(enum-set-union *enum-set1 enum-set2*) procedure
(enum-set-intersection *enum-set1 enum-set2*)
(enum-set-difference *enum-set1 enum-set2*)
Arguments *enum-set1* and *enum-set2* must be enumeration sets that have the same enumeration type.

The `enum-set-union` procedure returns the union of *enum-set1* and *enum-set2*. The `enum-set-intersection` procedure returns the intersection of *enum-set1* and *enum-set2*. The `enum-set-difference` procedure returns the difference of *enum-set1* and *enum-set2*.

```
(let* ((e (make-enumeration '(red green blue)))
       (c (enum-set-constructor e)))
  (list (enum-set->list (enum-set-union (c '(blue)) (c '(red))))
        (enum-set->list
          (enum-set-intersection (c '(red green)) (c '(red blue))))
        (enum-set->list
         (enum-set-difference (c '(red green)) (c '(red blue))))))
⇒ ((red blue) (red) (green))
```

(enum-set-xor *enum-set1 enum-set2*) procedure
Procedure `enum-set-xor` returns the exclusive disjunction of *enum-set1* and *enum-set2*. Arguments *enum-set1* and *enum-set2* must be enumeration sets that have the same enumeration type.

(enum-set-complement *enum-set*) procedure
Returns *enum-set*'s complement with respect to its universe.

```
(let* ((e (make-enumeration '(red green blue)))
       (c (enum-set-constructor e)))
  (enum-set->list (enum-set-complement (c '(red)))))
⇒ (green blue)
```

(enum-set-union! *enum-set1 enum-set2*) procedure
Creates the union of *enum-set1* and *enum-set2* and stores its result in *enum-set1*. *enum-set1* and *enum-set2* must be enumeration sets that have the same enumeration type.

(enum-set-intersection! *enum-set1 enum-set2*) procedure
Creates the intersection of *enum-set1* and *enum-set2* and stores its result in *enum-set1*. *enum-set1* and *enum-set2* must be enumeration sets that have the same enumeration type.

(enum-set-difference! *enum-set1 enum-set2*) procedure
Creates the set difference between *enum-set1* and *enum-set2* and stores its result in *enum-set1*. *enum-set1* and *enum-set2* must be enumeration sets that have the same enumeration type.

(enum-set-xor! *enum-set1 enum-set2*) procedure
Creates the exclusive disjunction between *enum-set1* and *enum-set2* and stores its result in *enum-set1*. *enum-set1* and *enum-set2* must be enumeration sets that have the same enumeration type.

(enum-set-complement! *enum-set*) procedure
Replaces *enum-set* with its complement with respect to the type of *enum-set*.

(enum-set-indexer *enum-set*) procedure
Returns a unary procedure that, given a symbol that is in the universe of *enum-set*, returns its 0-origin index within the canonical ordering of the symbols in the universe; given a value not in the universe, the unary procedure returns `#f`.

```
(let* ((e (make-enumeration '(red green blue)))
       (i (enum-set-indexer e)))
  (list (i 'red) (i 'green) (i 'blue) (i 'yellow)))
 ⇒ (0 1 2 #f)
```

The `enum-set-indexer` procedure could be defined as follows using the `memq` procedure:

```
(define (enum-set-indexer set)
  (let* ((symbols (enum-set->list (enum-set-universe set)))
         (cardinality (length symbols)))
    (lambda (x)
      (cond ((memq x symbols) =>
              (lambda (probe) (- cardinality (length probe))))
            (else #f)))))
```

(enum-set-any? *pred enum-set*) procedure

Returns `#t` if any application of *pred* to the elements of *enum-set* returns true, and `#f` otherwise.

(enum-set-every? *pred enum-set*) procedure

Returns `#t` if every application of *pred* to the elements of *enum-set* returns true, and `#f` otherwise.

(enum-set-count *pred enum-set*) procedure

Returns an exact integer, the number of elements of *enum-set* that satisfy *pred*.

(enum-set-map->list *proc enum-set*) procedure

Invokes *proc* on each member of *enum-set* in increasing ordinal order. The results are returned as a list.

(enum-set-for-each *proc enum-set*) procedure

Invokes *proc* on each member of *enum-set* in increasing ordinal order and discards the rest.

(enum-set-fold *proc nil enum-set*) procedure

The current state is initialized to *nil*, and *proc* is invoked on each element of *enum-set* in increasing ordinal order and the current state, setting the current state to the result. The algorithm is repeated until all the elements of *enum-set* have been processed. Then the current state is returned.

(enum-set-filter *pred enum-set*) procedure

Returns an enum set containing the enums in *enum-set* that satisfy *pred*.

(enum-set-remove *pred enum-set*) procedure

Returns an enum set containing the enums in *enum-set* that do not satisfy *pred*.

24.5 R6RS Compatibility

(make-enumeration *symbol-list*) procedure

Argument *symbol-list* must be a list of symbols. The `make-enumeration` procedure creates a new enumeration type whose universe consists of those symbols (in canonical order of their first appearance in the list) and returns that universe as an enumeration set whose universe is itself and whose enumeration type is the newly created enumeration type.

(enum-set-universe *enum-set*) procedure

Returns the set of all symbols that comprise the universe of its argument *enum-set*, as an enum set.

(enum-set-constructor *enum-set*) procedure

Returns a unary procedure that, given a list of symbols that belong to the universe of *enum-set*, returns a subset of that universe that contains exactly the symbols in the list. The values in the list must all belong to the universe.

(enum-constructor *enum-set*) procedure

Given an enum set, `enum-constructor` returns a procedure which takes an enum name and returns the corresponding enum object.

25 LispKit Format

Library `(lispkit format)` provides an implementation of Common Lisp's `format` procedure for LispKit. Procedure `format` can be used for creating formatted text using a format string similar to `printf`. The formatting formalism, though, is significantly more expressive, allowing users to display numbers in various formats (e.g. hex, binary, octal, roman numerals, natural language), applying conditional formatting, outputting text in a tabular format, iterating over data structures, and even applying `format` recursively to handle data that includes its own preferred formatting strings.

25.1 Usage overview

In its most simple form, procedure `format` gets invoked with a *control string* followed by an arbitrary number of *arguments*. The *control string* consists of characters that are copied verbatim into the output as well as *formatting directives*. All formatting directives start with a tilde (`~`) and end with a single character identifying the type of the directive. Directives may also take prefix *parameters* written immediately after the tilde character, separated by comma as well as *modifiers* (see below for details).

For example, the call of `format` below injects two integer arguments into the control string via directive `~D` and returns the resulting string:

```
(format "There are ~D warnings and ~D errors." 12 7)
⇒ "There are 12 warnings and 7 errors."
```

25.1.1 Simple Directives

Here is a simple control string which injects a readable description of an argument via directive `~A`: `"I received ~A as a response"`. Directive `~A` refers to a the *next argument* provided to `format` when compiling the formatted output:

```
(format "I received ~A as a response" "nothing")
⇒ "I received nothing as a response"
(format "I received ~A as a response" "a long email")
⇒ "I received a long email as a response"
```

Directive `~A` may be given parameters to influence the formatted output. The first parameter of `~A`-directives defines the minimal length. If the length of the textual representation of the next argument is smaller than the minimal length, padding characters are inserted:

```
(format "|Name: ~10A|Location: ~13A|" "Smith" "New York")
⇒ "|Name: Smith     |Location: New York     |"
(format "|Name: ~10A|Location: ~13A|" "Williams" "San Francisco")
⇒ "|Name: Williams  |Location: San Francisco|"
(format "|Name: ~10,,,'_@A|Location: ~13,,,'-A|" "Garcia" "Los Angeles")
⇒ "|Name: ____Garcia|Location: Los Angeles--|"
```

The third example above utilizes more than one parameter and, in one case, includes a `@` modifier. The directive `~13,,,'-A` defines the first and the fourth parameter. The second and third parameter are omitted and thus defaults are used. The fourth parameter defines the padding character. If character literals are used in the parameter list, they are prefixed with a quote `'`. The directive `~10,,,'_@A` includes an `@` modifier which will result in padding of the output on the left.

It is possible to inject a parameter from the list of arguments. The following examples show how parameter `v` is used to do this for formatting a floating-point number with a configurable number of fractional digits.

```
(format "length = ~,vF" 2 pi)
⇒ "length = 3.14"
(format "length = ~,vF" 4 pi)
⇒ "length = 3.1416"
```

Here `v` is used as the second parameter of the fixed floating-point directive `~F`, indicating the number of fractional digits. It refers to the next provided argument (which is either 2 or 4 in the examples above).

25.1.2 Composite Directives

The next example shows how one can refer to the total number of arguments that are not yet consumed in the formatting process by using `#` as a parameter value.

```
(format "~A left for formatting: ~#[none~;one~;two~:;many~]."
        "Arguments" "eins" 2)
⇒ "Arguments left for formatting: two."
(format "~A left for formatting: ~#[none~;one~;two~:;many~]."
        "Arguments")
⇒ "Arguments left for formatting: none."
(format "~A left for formatting: ~#[none~;one~;two~:;many~]."
        "Arguments", "eins", 2, "drei", "vier")
⇒ "Arguments left for formatting: many."
```

In these examples, the *conditional directive* `~[` is used. It is followed by *clauses* separared by directive `~;` until `~]` is reached. Thus, there are four clauses in the example above: `none`, `one`, `two`, and `many`. The parameter in front of the `~[` directive determines which of the clauses is being output. All other clauses will be discarded. For instance, `~1[zero~;one~;two~:;many~]` will output `one` as clause 1 is chosen (which is the second one, given that numbering starts with zero). The last clause is special because it is prefixed with the `~;` directive using a `:` modifier: this is a *default clause* which is chosen when none of the others are applicable. Thus, `~8[zero~;one~;two~:;many~]` outputs `many`. This also explains how the example above works: here `#` refers to the number of arguments that are still available and this number drives what is being returned in this directive: `~#[...~]`.

Another powerful composite directive is the *iteration directive* `~{`. With the iteration directive it is possible to iterate over all elements of a sequence such as a list or vector. The control string between `~{` and `~}` gets repeated as long as there are still elements left in the sequence which is provided as an argument. For instance, `Numbers:~{ ~A~}` applied to `("one" "two" "three")` results in the output `Numbers: one two three`. The control string between `~{` and `~}` can also consume more than one element of the sequence. Thus, `Numbers:~{ ~A=>~A~}` applied to argument `("one" 1 "two" 2)` outputs `Numbers: one=>1 two=>2`.

Of course, it is also possible to nest arbitrary composite directives. Here is an example for a control string that uses a combination of iteration and conditional directives to output the elements of a sequence separated by a comma: `(~{~#[~;~A~:;~A, ~]~})`. When this control string is used with the argument `("one" "two" "three")`, the following formatted output is generated: `(one, two, three)`.

25.2 Formatting language

Control strings consist of characters that are copied verbatim into the output as well as *formatting directives*. All formatting directives start with a tilde (`~`) and end with a single character identifying the type of the directive. Directives may take prefix *parameters* written immediately after the tilde character, separated by comma. Both integers and characters are allowed as parameters. They may be followed by formatting *modifiers* `:` , `@` , and `+` . This is the general format of a formatting directive:

```
~param1,param2,...mX
```

where

- `m` is a potentially empty modifier, consisting of an arbitrary sequence of modifier characters `:` , `@` , and `+`
- `X` is a character identifying a directive type
- `paramN` is either a nummeric or character parameter according to the specification below.

The following grammar describes the syntax of directives formally:

```
<directive>  = "~" <modifiers> <char>
             | "~" <parameters> <modifiers> <char>
<modifiers>  = <empty>
             | ":" <modifiers>
             | "@" <modifiers>
             | "+" <modifiers>
<parameters> = <parameter>
             | <parameter> "," <parameters>
<parameter>  = <empty>
             | "#"
             | "v"
             | <number>
             | "-" <number>
             | <character>
<number>     = <digit>
             | <digit> <number>
<digit>      = "0" | "1" | "2" | "3" | "4" | "5" | "6" | "7" | "8" | "9"
<character>  = "'" <char>
```

25.3 Formatting directives

The formatting directives supported by library `(lispkit format)` are based on the directives specified in Common Lisp the Language, 2nd Edition by Guy L. Steele Jr. Some directives have been extended to meet today's formatting requirements (e.g. to support localization) and to enable a powerful usage throughout LispKit. Extensions were introduced in a way to not impact backward compatibility.

`~a`
`~A`

ASCII: *~mincol,colinc,minpad,padchar,maxcol,elchar*A

The next argument *arg* is output as if procedure `display` was used, i.e. the output is without escape characters and if *arg* is a string, its characters will be output verbatim without surrounding quotes.

mincol (default: 0) specifies the minimal "width" of the output of the directive in characters, *maxcol* (default: ∞) specifies the maximum width. *padchar* (default: ' ') defines the character that is used to pad the output to make sure it is at least *mincol* characters long.

By default, the output is padded on the right with at least *minpad* (default: 0) copies of *padchar*. Padding characters are then inserted *colinc* (default: 1) characters at a time until the total width is at least *mincol*. Padding is capped such that the output never exceeds *maxcol* characters. If, without padding, the output is already longer than *maxcol*, the output is truncated at width *maxcol - 1* and the ellipsis character *elchar* (default: '…') is inserted at the end.

Modifier `@` enables padding on the left to right-align the output.

`~w`
`~W` **WRITE: *~mincol,colinc,minpad,padchar,maxcol,elchar*W**

The next argument *arg* is output as if procedure `write` was used, i.e. the output is with escape characters and if *arg* is a string, its characters will be output surrounded by quotes.

Parameters *mincol* (default: 0), *colinc* (default: 1), *minpad* (default: 0), *padchar* (default: ' '), *maxcol* (default: ∞), and *elchar* (default: '…') are used just as described for the ASCII directive `~A`. Modifier `@` enables padding on the left to right-align the output.

`~s`
`~S` **SOURCE: *~mincol,colinc,minpad,padchar,maxcol,elchar*S**

The next argument *arg* is output using a type-specific control string. If no control string is registered for the type of *arg*, then `~S` behaves like `~W` for *arg*.

Parameters *mincol* (default: 0), *colinc* (default: 1), *minpad* (default: 0), *padchar* (default: ' '), *maxcol* (default: ∞), and *elchar* (default: '…') are used just as described for the ASCII directive `~A`. Modifier `@` enables padding on the left to right-align the output.

`~c`
`~C` **CHARACTER: ~C**

The next argument *arg* should be a character or a string consisting of one character. Directive `~C` outputs *arg* in a form dependent on the modifiers used. Without any modifiers, *arg* is output as if the character was used in a string without any escaping.

If the `@` modifier is provided alone, the character is output using Scheme's syntax for character literals. The modifier combination `@:` will lead to *arg* being output as Unicode code points. The combination `@:+` will output *arg* as a sequence of Unicode scalar property names, separated by comma.

If the `:` modifier is used (without `@`), a representation of *arg* for the usage in XML documents is chosen. By default, a Unicode-based XML character encoding is used, unless `:` is combined with `+`, in which case the character is represented as a XML named character entity when possible, otherwise, the character is output in raw form.

If the `+` modifiers is used alone, the character is output as if it is a character of a string, escaped if necessary, and surrounded by quotes.

```
(format "~C" #\A ) ⇒ A
(format "~+C" #\A ) ⇒ "A"
(format "~+C" #\newline ) ⇒ "\n "
(format "~@C" "A") ⇒ #\A
(format "~@C" "\t ") ⇒ #\tab
(format "~@:C" "©") ⇒ U+00A9
(format "~@:+C" "©") ⇒ COPYRIGHT SIGN
(format "~:C" "©") ⇒ &#xA9;
(format "~:+C" "©") ⇒ &copy;
```

~d
~D

DECIMAL: ***~mincol,padchar,groupchar,groupcol* D**

The next argument *arg* is output in decimal radix. *arg* should be an integer, in which case no decimal point is printed. For floating-point numbers which do not represent an integer, a decimal point and a fractional part are output.

mincol (default: 0) specifies the minimal ”width” of the output of the directive in characters with *padchar* (default: ' ') defining the character that is used to pad the output on the left to make sure it is at least *mincol* characters long.

```
(format "Number: ~D" 8273) ⇒ Number: 8273
(format "Number: ~6D" 8273) ⇒  Number: 8273
(format "Number: ~6,'0D" 8273) ⇒ Number: 008273
```

By default, the number is output without grouping separators. *groupchar* specifies which character should be used to separate sequences of *groupcol* digits in the output. Grouping of digits gets enabled with the `:` modifier.

```
(format "|~10:D|" 1734865) ⇒ | 1,734,865|
(format "|~10,,'.:D|" 1734865) ⇒ | 1.734.865|
```

A sign is output only if the number is negative. With the modifier `@` it is possible to force output also of positive signs. To facilitate the localization of output, procedure `format` supports a locale parameter, which is also available via format config objects. Locale-specific output can be enabled for the `~D` directive by using the `+` modifier.

```
(format 'de_CH "~+D" 14321) ⇒ 14'321
```

~b
~B

BINARY: ***~mincol,padchar,groupchar,groupcol* B**

Binary directive `~B` is just like decimal directive `~D` but it outputs the next argument in binary radix (radix 2) instead of decimal. It uses the space character as the default for *groupchar* and has a default grouping size of 4 as the default for *groupcol*.

```
(format "bin(~D) = ~B" 178 178) ⇒ bin(178) = 10110010
(format "~:B" 59701) ⇒ 1110 1001 0011 0101
(format "~19,'0,'.:B" 31912) ⇒ 0111.1100.1010.1000
```

~o
~O

OCTAL: ***~mincol,padchar,groupchar,groupcol* O**

Octal directive `~O` is just like decimal directive `~D` but it outputs the next argument in octal radix (radix 8) instead of decimal. It uses the space character as the default for groupchar and has a default grouping size of 4 as the default for groupcol.

```
(format "bin(~D) = ~O" 178 178) ⇒ bin(178) = 262
(format "~:O" 59701) ⇒ 16 4465
(format "~9,'0,',:O" 31912) ⇒ 0007,6250
```

~x
~X

HEXADECIMAL: ***~mincol,padchar,groupchar,groupcol* X**

Hexadecimal directive `~X` is just like decimal directive `~D` but it outputs the next argument in hexadecimal radix (radix 16) instead of decimal. It uses the colon character as the default for *groupchar* and has a default grouping size of 2 as the default for *groupcol*. With modifier `+`, upper case characters are used for representing hexadecimal digits.

```
(format "bin(~D) = ~X" 9968 9968) ⇒ bin(9968) = 26f0
(format "~:X" 999701) ⇒ f:41:15
(format "~+X" 999854) ⇒ F41AE
```

~r
~R

RADIX: *~radix,mincol,padchar,groupchar,groupcol* R

The next argument *arg* is expected to be a fixnum number. It will be output with radix *radix* (default: 10). *mincol* (default: 0) specifies the minimal "width" of the output of the directive in characters with *padchar* (default: ' ') defining the character that is used to pad the output on the left to make it at least *mincol* characters long.

```
(format "Number: ~10R" 1272) ⇒ Number: 1272
(format "Number: ~16,8,'0R" 7121972) ⇒ Number: 006cac34
(format "~+X" 999854) ⇒ Number: 10101101
```

By default, the number is output without grouping separators. *groupchar* specifies which character should be used to separate sequences of *groupcol* digits in the output. Grouping of digits is enabled with the `:` modifier.

```
(format "~16,8,,':,2:R" 7121972) ⇒ 6c:ac:34
(format "~2,14,'0,'.,4:R" 773) ⇒ 0011.0000.0101
```

A sign is output only if the number is negative. With modifier `@` it is possible to force output also of positive signs.

If parameter *radix* is not specified at all, then an entirely different interpretation is given. `~R` outputs *arg* as a cardinal number in natural language. The form `~:R` outputs *arg* as an ordinal number in natural language. `~@R` outputs *arg* as a Roman numeral.

```
(format "~R" 572) ⇒ five hundred seventy-two
(format "~:R" 3) ⇒ 3rd
(format "~@R" 1272) ⇒ MCCLXXII
```

Whenever output is provided in natural language, English is used as the default language. By specifying the + modifier, it is possible to switch the language to the language of the locale provided to procedure format. In fact, modifier + plays two different roles: If the given radix is greater than 10, upper case characters are used for representing alphabetic digits. If the radix is omitted, usage of modifier + enables locale-specific output determined by the locale: parameter of procedure format.

```
(format 'de_DE "~+R" 572) ⇒ fünfhundertzweiundsiebzig
(format 'de_CH "~10+R" 14321) ⇒ 14'321
(format "~16R vs ~16+R" 900939 900939) ⇒ dbf4b vs DBF4B
```

~f
~F

FIXED FLOAT: *~w,d,k,overchar,padchar,groupchar,groupcol* F

The next argument *arg* is output as a floating-point number in a fixed format (ideally without exponent) of exactly *w* characters, if *w* is specified. First, leading *padchar* characters (default: ' ') are output, if necessary, to pad the field on the left. If *arg* is negative, then a minus sign is printed. If *arg* is not negative, then a plus sign is printed if and only if the `@` modifier was specified. Then a sequence of digits, containing a single embedded decimal point, is printed. If parameter *d* is provided, then exactly *d* decimal places are output. This represents the magnitude of the value of *arg* times 10^k, rounded to *d* fractional digits. There are no leading zeros, except that a single zero digit is output before the decimal point, if the printed value is less than 1.0, and this single zero digit is not output after all if $w = d + 1$.

If it is impossible to print the value in the required format in a field of width *w*, then one of two actions is taken: If the parameter *overchar* is specified, then *w* copies of this character are printed. If *overchar* is omitted, then the scaled value of *arg* is printed using more than *w* characters.

If the width parameter *w* is omitted, then the output is of variable width and a value is chosen for *w* in such a way that no leading padding characters are needed and exactly *d* characters will follow the decimal point. For example, the directive `~,2F` will output exactly two digits after the decimal point and as many as necessary before the decimal point.

If *d* is omitted, then there is no constraint on the number of digits to appear after the decimal point. A value is chosen for *d* in such a way that as many digits as possible may be printed subject to the width constraint imposed by *w* and the constraint that no trailing zero digits may appear in the fraction, except that if the fraction is zero, then a single zero digit should appear after the decimal point if permitted by the width constraint.

If *w* is omitted, then if the magnitude of *arg* is so large (or, if *d* is also omitted, so small) that more than 100 digits would have to be printed, then *arg* is output using exponential notation instead.

The `~F` directive also supports grouping of the integer part of *arg*; this can be enabled via the `:` modifier. *groupchar* (default: ',') specifies which character should be used to separate sequences of *groupcol* (default: 3) digits in the integer part of the output. If locale-specific settings should be used, the `+` modifier needs to be set.

```
(format "~F" 123.1415926) ⇒ 123.1415926
(format "~8F" 123.1415926) ⇒ 123.1416
(format "~8,,,'-F" 123.1415926) ⇒ 123.1416
(format "~8,,,'-F" 123456789.12) ⇒ --------
(format "~8,,,,'0F" 123.14) ⇒ 00123.14
(format "~8,3,,,'0F" 123.1415926) ⇒ 0123.142
(format "~,4F" 123.1415926) ⇒ 123.1416
(format "~,2@F" 123.1415926) ⇒ +123.14
(format "~,2,-2@F" 314.15926) ⇒ +3.14
(format "~,2:F" 1234567.891) ⇒ 1,234,567.89
(format "~,2,,,,'',3:F" 1234567.891) ⇒ 1'234'567.89
```

`~e`
`~E`

EXPONENTIAL FLOAT: *~w,d,e,k,overchar,padchar,expchar* F

The next argument *arg* is output as a floating-point number in an exponential format of exactly *w* characters, if *w* is specified. Parameter *d* is the number of digits to print after the decimal point, *e* is the number of digits to use when printing the exponent, and *k* is a scale factor that defaults to 1.

First, leading *padchar* (default: ' ') characters are output, if necessary, to pad the output on the left. If *arg* is negative, then a minus sign is printed. If *arg* is not negative, then a plus sign is printed if and only if the `@` modifier was specified. Then a sequence of digits, containing a single embedded decimal point, is output. The form of this sequence of digits depends on the scale factor *k*. If *k* is zero, then *d* digits are printed after the decimal point, and a single zero digit appears before the decimal point. If *k* is positive, then it must be strictly less than $d + 2$ and *k* significant digits are printed before the decimal point, and $d - k + 1$ digits are printed after the decimal point. If *k* is negative, then it must be strictly greater than $-d$. A single zero digit appears before the decimal point and after the decimal point, first $-k$ zeros are output followed by $d + k$ significant digits.

Following the digit sequence, the exponent is output following character *expchar* (default: 'E') and the sign of the exponent, i.e. either the plus or the minus sign. The exponent consists of *e* digits representing the power of 10 by which the fraction must be multiplied to properly represent the rounded value of *arg*.

If it is impossible to print the value in the required format in a field of width *w*, then one of two actions is taken: If the parameter *overchar* is specified, then *w* copies of this character are printed instead of *arg*. If *overchar* is omitted, then *arg* is printed using more than *w* characters, as many more as may be needed. If *d* is too small for the specified *k* or *e* is too small, then a larger value is used for *d* or *e* as may be needed.

If parameter *w* is omitted, then the output is of variable width and a value is chosen for *w* in such a way that no leading padding characters are needed.

```
(format "~E" 31.415926) ⇒ 3.1415926E+1
(format "~,5E" 0.0003141592) ⇒ 3.14159E-4
(format "~,4,2E" 0.0003141592) ⇒ 3.1416E-04
(format "~9E" 31.415926) ⇒ 3.1416E+1
(format "~10,3,,,,'#E" 31.415926) ⇒ ##3.142E+1
(format "~10,4,,3,,'#E" 31.415926) ⇒ #314.16E-1
(format "~7,3,2,,'-E" 31.415926) ⇒ -------
(format "~10,4,,4,,'#@E" 31.415926) ⇒ +3141.6E-2
```

~g
~G

GENERAL FLOAT: *~w,d,e,k,overchar,padchar,expchar* G

The next argument *arg* is output as a floating-point number in either fixed-format or exponential notation as appropriate. The format in which to print *arg* depends on the magnitude (absolute value) of *arg*. Let *n* be an integer such that $10^{n-1} \leq arg < 10^n$. If *arg* is zero, let *n* be 0. Let *ee* equal *e* + 2, or 4 if *e* is omitted. Let *ww* equal *w* − ee, or nil if *w* is omitted. If *d* is omitted, first let *q* be the number of digits needed to print *arg* with no loss of information and without leading or trailing zeros; then let *d* equal *max(q, min(n, 7))*. Let *dd* equal *d* − *n*.

If $0 \leq dd \leq d$, then *arg* is output as if by the format directives:
`~ww,dd,,overchar,padcharF~ee@T`.
Note that the scale factor *k* is not passed to the `~F` directive. For all other values of *dd*, *arg* is printed as if by the format directive: `~w,d,e,k,overchar,padchar,expcharE`.
In either case, an `@` modifier is specified to the `~F` or `~E` directive if and only if one was specified for the `~G` directive.

```
(format "|~G|" 712.72) ⇒ |712.72 |
(format "|~12G|" 712.72) ⇒ | 712.72 |
(format "|~9,2G|~9,3,2,3G|~9,3,2,0G|" 0.031415 0.031415 0.031415)
 ⇒ | 3.14E-2|314.2E-04|0.314E-01|
(format "|~9,2G|~9,3,2,3G|~9,3,2,0G|" 0.314159 0.314159 0.314159)
 ⇒ | 0.31 |0.314 |0.314 |
(format "|~9,2G|~9,3,2,3G|~9,3,2,0G|" 3.14159 3.14159 3.14159)
 ⇒ | 3.1 | 3.14 | 3.14 |
(format "|~9,2G|~9,3,2,3G|~9,3,2,0G|" 314.159 314.159 314.159)
 ⇒ | 3.14E+2| 314 | 314 |
(format "|~9,2G|~9,3,2,3G|~9,3,2,0G|" 3141.59 3141.59 3141.59)
 ⇒ | 3.14E+3|314.2E+01|0.314E+04|
```

~$

DOLLARS FLOAT: *~d,n,w,padchar,curchar,groupchar,groupcol* $

The next argument *arg* is output as a floating-point number in a fixed-format notation that is particularly well suited for outputting monetary values. Parameter *d* (default: 2) defines the number of digits to print after the decimal point. Parameter *n* (default: 1) defines the minimum number of digits to print before the decimal point. Parameter *w* (default: 0) is the minimum total width of the output.

First, padding and the sign are output. If *arg* is negative, then a minus sign is printed. If *arg* is not negative, then a plus sign is printed if and only if the `@` modifier was specified. If the `:` modifier is used, the sign appears before any padding, and otherwise after the padding. If the number of characters, including the sign and a potential currency symbol is below width *w*, then character *padchar* (default: ' ') is used for padding the number in front of the integer part such that the overall output has *w* characters. After the padding, the currency symbol *curchar* is inserted, if available, followed by *n* digits representing the integer part of *arg*, prefixed by the right amount of '0' characters. If either parameter *groupchar* or *groupcol* is provided, the integer part is output in groups of *groupcol* characters (default: 3) separated by *groupchar* (default: ','). After the integer part, a decimal point is output followed by d digits of fraction, properly rounded.

If the magnitude of *arg* is so large that the integer part of *arg* cannot be output with at most *n* characters, then more characters are generated, as needed, and the total width might overrun as well.

For cases where a simple currency symbol is not sufficient, it is possible to use a numeric currency code as defined by ISO 4217 for parameter *curchar*. For positive codes, the shortest currency symbol is being used. For negative currency codes, the corresponding alphabetic code (ignoring the sign) is being used. Library `(lispkit system)` provides a conventient API to access currency codes.

By specifying the + modifier, it is possible to enable locale-specific output of the monetary value using the locale provided to `format`. In this case, also the currency associated with this locale is being used.

```
(format "~$" 4930.351) ⇒ 4930.35
(format "~3$" 4930.351) ⇒ 4930.351
(format "~,6$" 4930.351) ⇒ 004930.35
(format "~,6,12,'_$" 4930.351) ⇒ ___004930.35
(format "~,6,12,'_@$" 4930.351) ⇒ __+004930.35
(format "~,6,12,'_@:$" 4930.351) ⇒ +__004930.35
(format "~,6,12,'_,'€$" 4930.351) ⇒ __€004930.35
(format "~,6,12,'_,'€@$" 4930.351) ⇒ _+€004930.35
(format "~,,,,,,3$" 4930.351) ⇒ 4,930.35
(format "~,6,,,,,3$" 4930.351) ⇒ 004,930.35
(format "~,,,,208$" 1234.567) ⇒ kr 1234.57
(format "~,,,,-208$" 1234.567) ⇒ DKK 1234.57
(format 'de_CH "~+$" 4930.351) ⇒ CHF 4930.35
(format 'en_US "~,,,,,,3+$" 4930.351) ⇒ $4,930.35
(format 'de_DE "~,6,14,'_,,,3+$" 4930.351) ⇒ __004.930,35 €
```

~% **NEWLINE: *~n* %**

This directive outputs *n* (default: 1) newline characters, thereby terminating the current output line and beginning a new one. No arguments are being consumed. Simply putting *n* newline escape characters `"\n "` into the control string would also work, but ~% is often used because it makes the control string look nicer and more consistent.

~& **FRESHLINE: *~n* &**

Unless it can be determined that the output is already at the beginning of a line, this directive outputs a newline if $n > 0$. This conditional newline is followed by $n - 1$ newlines, it $n > 1$. Nothing is output if $n = 0$.

`~|` **PAGE SEPARATOR: *~n* |**

This directive outputs *n* (default: 1) page separator characters `#\page`.

`~~` **TILDE: *~n* ~**

This directive outputs *n* (default: 1) tilde characters.

`~p` `~P` **PLURAL: ~ P**

Depending on the next argument *arg*, which is expected to be an integer value, a different string is output. If *arg* is not equal to 1, a lowercase `s` is output. If *arg* is equal to 1, nothing is output. If the `:` modifier is provided, the last argument is used instead for *arg*. This is useful after outputting a number using `~D`. With the `@` modifier, `y` is output if *arg* is 1, or `ies` if it is not.

```
(format "~D tr~:@P/~D win~:P" 7 1) ⇒ 7 tries/1 win
(format "~D tr~:@P/~D win~:P" 1 0) ⇒ 1 try/0 wins
```

`~t` `~T` **TABULATE: *~colnum,colinc* T**

This directive will output sufficient spaces to move the cursor to column *colnum* (default: 1). If the cursor is already at or beyond column *colnum*, the directive will output spaces to move the cursor to column $colnum + k \times colinc$ for the smallest positive integer *k* possible, unless *colinc* (default: 1) is zero, in which case no spaces are output if the cursor is already at or beyond column *colnum*.

If modifier `@` is provided, relative tabulation is performed. In this case, the directive outputs *colnum* spaces and then outputs the smallest non-negative number of additional spaces necessary to move the cursor to a column that is a multiple of *colinc*. For example, the directive `~3,8@T` outputs three spaces and then moves the cursor to a "standard multiple-of-eight tab stop" if not at one already. If the current output column cannot be determined, however, then *colinc* is ignored, and exactly *colnum* spaces are output.

`~*` **IGNORE ARGUMENT: *~n* ***

The next *n* (default: 1) arguments are ignored. If the `:` modifier is provided, arguments are "ignored backwards", i.e. `~:*` backs up in the list of arguments so that the argument last processed will be processed again. `~n:*` backs up *n* arguments. When within a `~{` construct, the ignoring (in either direction) is relative to the list of arguments being processed by the iteration.

The form `~n@*` is an "absolute goto" rather than a "relative goto": the directive goes to the *n*-th argument, where 0 means the first one. *n* defaults to 0 for this form, so `~@*` goes back to the first argument. Directives after a `~n@*` will take arguments in sequence beginning with the one gone to. When within a `~{` construct, the "goto" is relative to the list of arguments being processed by the iteration.

`~?` **INDIRECTION: ~ ?**

The next argument *arg* must be a string, and the one after it, *lst*, must be a sequence (e.g. an array). Both arguments are consumed by the directive. *arg* is processed as a format control string, with the elements of the list *lst* as the arguments. Once the recursive processing of the control string has been finished, then processing of the control string containing the `~?` directive is resumed.

```
(format "~? ~D" "[~A ~D]" '("Foo" 5) 7) ⇒ [Foo 5] 7
(format "~? ~D" "[~A ~D]" '("Foo" 5 14) 7) ⇒ [Foo 5] 7
```

Note that in the second example, three arguments are supplied to the control string `"(~A ~D)"`, but only two are processed and the third is therefore ignored. With the `@` modifier, only one argument is directly consumed. The argument must be a string. It is processed as part of the control string as if it had appeared in place of the `~@?` directive, and any directives in the recursively processed control string may consume arguments of the control string containing the `~@?` directive.

```
(format "~@? ~D" "[~A ~D]" "Foo" 5 7) ⇒ [Foo 5] 7
(format "~@? ~D" "[~A ~D]" "Foo" 5 14 7) ⇒ [Foo 5] 14
```

`~(...~)` **CONVERSION: ~(*str* ~)**

The contained control string *str* is processed, and what it produces is subject to a conversion. Without the + modifier, a case conversion is performed. `~(` converts every uppercase character to the corresponding lowercase character, `~:(` capitalizes all words, `~@(` capitalizes just the first word and forces the rest to lowercase, and `~:@(` converts every lowercase character to the corresponding uppercase character.

```
(format "Result: ~:(~R error~:P~)" 0) ⇒ Result: Zero Errors
(format "Result: ~@(~R error~:P~)" 1) ⇒ Result: One error
(format "Result: ~:@(~R error~:P~)" 23) ⇒ Result: TWENTY-THREE ERRORS
```

If the + modifier is provided together with the `:` modifier, all characters corresponding to named XML entities are being converted into names XML entities. If modifier `@` is added, then only those characters are converted which conflict with the XML syntax. The modifier combination `+@` converts the output by stripping off all diacritics. Modifier + only will escape characters such that the result can be used as a Scheme string literal.

```
(format "~+:(~A~)" "© 2021–2023 TÜV") ⇒ &copy; 2021&ndash;2023 T&Uuml;V
(format "~+:@(~A~)" "<a href=\"t.html\">© TÜV</a>")
   ⇒ &lt;a href=&quot;t.html&quot;&gt;© TÜV&lt;/a&gt;
(format "~+@(~A~)" "épistèmê") ⇒ episteme
(format "~+(~A~)" "Hello \"World\"\n ") ⇒ Hello \"World\"\n
```

`~[...~]` **CONDITIONAL: ~[str_0 ~; str_1 ~; ... ~; str_n ~]**

This is a set of control strings str_i, called clauses, one of which is chosen and used. Which clause is chosen depends on the next argument *arg*, which is expected to be a fixnum. The clauses are separated by `~;` and the construct is terminated by `~]`.

Without default: From a conditional directive `~[` str_0`~;` str_1`~;` ... `~;` str_n `~]`, the *arg*-th clause is selected, where the first clause is number 0. If a prefix parameter is given as `~n[`, then the parameter *n* is used instead of argument *arg*. This is useful only if the parameter is specified via `#`, to dispatch on the number of arguments remaining to be processed. If *arg* or *n* is out of range, then no clause is selected and no error is signaled. After the selected alternative has been processed, the control string continues after the `~]`.

With default: Whenever the directive has the form `~[` str_0`~;` str_1`~;` ... `~:;` *default* `~]`, i.e. the last clause is separated via `~:;`, then the conditional directive has a default clause which gets performed whenever no other clause could be selected.

Optional selector: Whenever the directive has the form `~:[` *none* `~;` *some* `~]`, the *none* control string is chosen if *arg* is `#f`, otherwise the *some* control string is chosen.

Nil selector: Whenever the directive has the form `~+[` *empty* `~;` *elems* `~]` the *empty* control string is chosen if *arg* is the empty list, otherwise the *elems* control string is chosen.

Selector test: Whenever the directive has the form `~@[` *true* `~]`, the next argument *arg* is tested for being `#f`. If *arg* is not `#f`, then the argument is not used up by the `~@[` directive but remains as the next one to be processed, and the one clause *true* is processed. If *arg* is `#f`, then the argument is used up, and the clause is not processed. Therefore, the clause should normally use exactly one argument, and may expect it to be different from `#f`.

`~{ ... ~}` **ITERATION: *~n { str ~}***

The iteration directive is used to control how a sequence is output. Thus, the next argument *arg* should be a sequence which is used as a list of arguments as if for a recursive call to `format`. The string *str* is used repeatedly as the control string until all elements from *arg* are consumed. Each iteration can absorb as many elements of *arg* as it needs. For instance, if *str* uses up two arguments by itself, then two elements of sequence *arg* will get used up each time around the loop. If before any iteration step the sequence is empty, then the iteration is terminated. Also, if a prefix parameter *n* is given, then there will be at most *n* repetitions of processing of *str*. Finally, the `~^` directive can be used to terminate the iteration prematurely. If the iteration is terminated before all the remaining arguments are consumed, then any arguments not processed by the iteration remain to be processed by any directives following the iteration construct.

```
(format "Winners:~{ ~A~}." '("Fred" "Harry" "Jill"))
   ⇒ Winners: Fred Harry Jill.
(format "Winners: ~{~#[~;~A~:;~A, ~]~}." '("Fred" "Harry" "Jill"))
   ⇒ Winners: Fred, Harry, Jill.
(format "Pairs:~{ <~A,~S>~}." '("A" 1 "B" 2 "C" 3))
   ⇒ Pairs: <A, 1> <B, 2> <C, 3>.
```

~n,m`:{`*str* `~}` is similar, but the argument should be a list of sublists. At each repetition step (capped by *n*), one sublist is used as the list of arguments for processing *str* with an iteration cap of *m*. On the next repetition, a new sublist is used, whether or not all elements of the last sublist had been processed.

```
(format "Pairs:~:{ <~A,~S>~}." '(("A" 1) ("B" 2) ("C" 3)))
   ⇒ Pairs: <A, 1> <B, 2> <C, 3>.
```

`~@{`*str* `~}` is similar to `~{`*str* `~}`, but instead of using one argument that is a sequence, all the remaining arguments are used as the list of arguments for the iteration.

```
(format "Pairs:~@{ <~A,~S>~}." "A" 1 "B" 2 "C" 3)
   ⇒ Pairs: <A, 1> <B, 2> <C, 3>.
```

`~:@{`*str* `~}` combines the features of `~:{`*str* `~}` and `~@{`*str* `~}`. All the remaining arguments are used, and each one must be a sequence. On each iteration, the next argument is used as a list of arguments to *str*.

```
(format "Pairs:~:@{ <~A,~S>~}." '("A" 1) '("B" 2) '("C" 3))
   ⇒ Pairs: <A, 1> <B, 2> <C, 3>.
```

Terminating the repetition directive with `~:}` instead of `~}` forces *str* to be processed at least once, even if the initial sequence is empty. However, it will not override an explicit prefix parameter of zero. If *str* is empty, then an argument is used as *str*. It must be a string and precede any arguments processed by the iteration.

~<...~> **JUSTIFICATION: *~mincol,colinc,minpad,padchar,maxcol,elchar< str ~>***

This directive justifies the text produced by processing control string *str* within a field which is at least *mincol* columns wide (default: 0). *str* may be divided up into segments via directive `~;`, in which case the spacing is evenly divided between the text segments.

With no modifiers, the leftmost text segment is left-justified in the field and the rightmost text segment is right-justified. If there is only one text element, it is right-justified. The `:` modifier causes spacing to be introduced before the first text segment. The `@` modifier causes spacing to be added after the last text segment. The *minpad* parameter (default: 0) is the minimum number of padding characters to be output between each segment. Whenever padding is needed, the padding character *padchar* (default: ”) is used. If the total width needed to satisfy the constraints is greater than *mincol*, then the width used is $mincol + k \times colinc$ for the smallest possible non-negative integer *k* with *colinc* defaulting to 1.

```
(format "|~10,,,'.<foo~;bar~>|") ⇒ |foo....bar|
(format "|~10,,,'.:<foo~;bar~>|") ⇒ |..foo..bar|
(format "|~10,,,'.:@<foo~;bar~>|") ⇒ |..foo.bar.|
(format "|~10,,,'.<foobar~>|") ⇒ |....foobar|
(format "|~10,,,'.:<foobar~>|") ⇒ |....foobar|
(format "|~10,,,'.@<foobar~>|") ⇒ |foobar....|
(format "|~10,,,'.:@<foobar~>|") ⇒ |..foobar..|
```

Note that *str* may include format directives. All the clauses in *str* are processed in order. It is the resulting pieces of text that are justified. The `~^` directive may be used to terminate processing of the clauses prematurely, in which case only the completely processed clauses are justified.

If the first clause of a `~<` directive is terminated with `~:;` instead of `~;`, then it is used in a special way. All of the clauses are processed, but the first one is not used in performing the spacing and padding. When the padded result has been determined, then, if it fits on the current line of output, it is output, and the text for the first clause is discarded. If, however, the padded text does not fit on the current line, then the text segment for the first clause is output before the padded text. The first clause ought to contain a newline (such as a % directive). The first clause is always processed, and so any arguments it refers to will be used. The decision is whether to use the resulting segment of text, not whether to process the first clause. If `~:;` has a prefix parameter *n*, then the padded text must fit on the current line with *n* character positions to spare to avoid outputting the first clause's text.

For example, the control string in the following example can be used to print a list of items separated by comma without breaking items over line boundaries, beginning each line with `;;`. The prefix parameter 1 in `~1:;` accounts for the width of the comma that will follow the justified item if it is not the last element in the list, or the period if it is. If `~:;` has a second prefix parameter, like below, then it is used as the width of the line, overriding the line width as specified by the current format configuration object.

```
(format "~%;; ~{~<~%;; ~1,30:; ~S~>~^,~}.~%"
        '("first line" "second" "a long third line"
        "fourth" "fifth"))
 ⇒
  ;; "first line", "second",
  ;; "a long third line",
  ;; "fourth", "fifth".
```

~^ **UP AND OUT: ~^**

Continue: The ~^ directive is an escape construct. If there are no more arguments remaining to be processed, then the immediately enclosing ~{ or ~< directive is terminated. If there is no such enclosing directive, then the entire formatting operation is terminated. In the case of ~<, the formatting is performed, but no more segments are processed before doing the justification. The ~^ directive should appear only at the beginning of a ~< clause, because it aborts the entire clause it appears in, as well as all following clauses. ~^ may appear anywhere in a ~{ construct.

```
(format "Done.~^ ~D warning~:P.~^ ~D error~:P.") ⇒ Done.
(format "Done.~^ ~D warning~:P.~^ ~D error~:P." 3) ⇒ Done. 3 warnings.
(format "Done.~^ ~D warning~:P.~^ ~D error~:P." 1 5)
   ⇒ Done. 1 warning. 5 errors.
```

If the directive has the form ~n^, then termination occurs if *n* is zero. If the directive has the form ~n,m^, termination occurs if the value of *n* equals the value of *m*. If the directive has the form ~n,m,o^, termination occurs if $n \le m \le o$. Of course, this is useless if all the prefix parameters are literals and at least one of them should be a # or a v parameter.

Break: If ~^ is used within a ~:{ directive, then it merely terminates the current iteration step because in the standard case, it tests for remaining arguments of the current step only and the next iteration step commences immediately. To terminate the entire iteration process, use ~:^. ~:^ may only be used if the directive it would terminate is ~:{ or ~:@{ . The entire iteration process is terminated if and only if the sublist that is supplying the arguments for the current iteration step is the last sublist (in the case of terminating a ~:{ directive) or the last argument to that call to format (in the case of terminating a ~:@{ directive).

Note that while ~^ is equivalent to ~#^ in all circumstances, ~:^ is not equivalent to ~#:^ because the latter terminates the entire iteration if and only if no arguments remain for the current iteration step, as opposed to no arguments remaining for the entire iteration process.

```
(format "~:{/~A~^ …~}",/ } .~ % "
        '(("hot" "dog") ("hamburger") ("ice" "cream") ("french" "fries")))
   ⇒ /hot …/hamburger/ice …/french …
(format "~:{/~A~:^ …~}",/ } .~ % "
        '(("hot" "dog") ("hamburger") ("ice" "cream") ("french" "fries")))
   ⇒ /hot …/hamburger …/ice …/french
(format "~:{/~A~#:^ …~}",/ } .~ % "
        '(("hot" "dog") ("hamburger") ("ice" "cream") ("french" "fries")))
   ⇒ /hot …/hamburger
```

~`...~‘ **UNPACK: ~` *str* ~‘**

This directive is used to format composite objects, such as rational numbers, complex numbers, colors, date-time objects, error objects, records, etc. Such objects get decomposed into a sequence of individual values which are formatted by control string *str*.

The next argument *arg* can be any Scheme object. If there is a decomposition predefined for this type of objects, it is applied to *arg* and *str* is used to format the resulting sequence of values. If no decomposition is possible, *str* is output assuming there is one argument *arg*.

```
(format "~S~:* = ~`(~S, ~S)~‘" 17/3) ⇒ 17/3 = (17, 3)
(format "Bits =~`~*~{ ~D~}~‘" (bitset 1 2 7)) ⇒ Bits = 1 2 7
(format "Color: ~`R=~F, G=~F, B=~F~‘" (color 0.3 1.0 0.74))
   ⇒ Color: R=0.3, G=1.0, B=0.74
```

25.4 Formatting configurations

A few formatting directives provided by procedure `format` require access to environment variables such as the locale, the width of tab characters, the length of lines, etc. Also the type-specific customization of the formatting of native and user-defined objects, e.g. via the `~S` directive, is based on a formatting control registry defined by an environment variable.

All relevant environment variables are bundled together into *format config* objects. Format configurations are organized hierarchically. Each format configuration optionally refers to a parent configuration. It inherits all environment variables and allows their values to be overridden.

The root of this format configuration hierarchy constitutes `base-format-config`. Typically, changes to this object impact all invocations of `format`, unless format is called with a custom format config object which is not derived from `base-format-config`. Without a custom format config, `format` reads the environment variables from the *current format config* parameter `current-format-config` (which, by default, inherits from `base-format-config`). Like every other parameter object, it is possible to define a new config dynamically via `parameterize`.

Format config objects are also used in combination with type-specific formatting as provided by the `~S` directive, as explained in the next section.

25.5 Type-specific formatting

Procedure `format` provides great means to format numbers, characters, strings, as well as sequences, i.e. lists and vectors. But as soon as values of data types encapsulating their state have to be output, only the default textual representation is supported, which is also used when a value is output via procedure `write`.

For this reason, procedure `format` supports the customization of how composite objects are formatted. The approach for doing this is simple: Internally, a composite object can be mapped ("unpacked") into a vector of "field values". These field values are then interpreted as arguments for an object type-specific control string which defines how the field values of such objects are formatted. If there is no object type-specific control string available, the object is output as if it was written via procedure `write`.

The following example shows how to customize the formatting of objects defined by a record type. The following record is used to model colored 2-dimensional points:

```
(define-record-type <point>
  (make-point x y c)
  point?
  (x point-x)
  (y point-y)
  (c point-color))
```

By default, objects of type `<point>` are output in the following way:

```
(define pt (make-point 7 13 (color 0.5 0.9 0)))
(format "~S" pt)
 ⇒ "#<record <point>: x=7, y=13, c=#<color 0.5 0.9 0.0>>"
```

LispKit defines a type tag for every type. A type tag will later be used to define a custom format for records of type `<point>`. We can retrieve the type tag for `<point>` via procedure `record-type-tag`:

```
(define point-type-tag (record-type-tag <point>))
```

Now we can define a custom format for objects of type `<point>` in which we refer to the unpacked fields in the order as defined in the `<point>` record type definition following a fixnum value denoting the identity of the record. The following control string formats `<point>` records in this way: `point{x=?,y=?,color=?}`. Note that it skips the record identity via the `~*` directive.

```
"point{x=~*~S,y=~S,c=~S}"
```

`format` refers to a number of environment variables via a formatting configuration (see previous section). The default configuration is defined by definition `base-format-config` and it includes custom type-specific formats. With procedure `format-config-control-set!` we can declare that all objects of type `<point>` should be formatted with the control string shown above:

```
(format-config-control-set!
  base-format-config
  point-type-tag
  "point{x=~*~S,y=~S,c=~S}")
```

Formatting records of type `<point>` via the `~S` directive is now based on this new control string.

```
(format "~S" pt)
 ⇒ "point{x=7,y=13,c=#<color 0.5 0.9 0.0>}"
```

If we wanted to also change how colors are formatted, we could do that in a similar way:

```
(format-config-control-set!
  base-format-config
  color-type-tag
  "color{~S, ~S, ~S}")
```

Now colors are formatted differently:

```
(format "~S" pt)  ⇒ "point{x=7,y=13,c=#<color 0.5 0.9 0.0>}"
(format "~S" (color 1.0 0.3 0.7))  ⇒ "color{1.0, 0.3, 0.7}"
```

If we wanted to change the way how colors are formatted only in the context of formatting points, we could do that by creating a formatting configuration for colors and associate it only with the formatting control string for points. The following code first removes the global color format so that colors are formatted again using the default mechanism. Then it redefines the formatting control for points by also specifying a format configuration that is used while applying the point formatting control string.

```
(format-config-control-remove! base-format-config color-type-tag)
(format-config-control-set!
  base-format-config
  point-type-tag
  "point{x=~*~S,y=~S,c=~S}"
  (format-config (list color-type-tag "color{~S, ~S, ~S}")))
(format "~S" (color 1.0 0.3 0.7))  ⇒ "#<color 1.0 0.3 0.7>"
(format "~S" pt)  ⇒ "point{x=7,y=13,c=color{0.5, 0.9, 0.0}}"
```

25.6 API

format-config-type-tag `object`
Symbol representing the `format-config` type. The `type-for` procedure of library `(lispkit type)` returns this symbol for all formatting configurations objects.

base-format-config `object`
Formatting configurations can have parent configurations from which all formatting environment variables are being inherited. `base-format-config` is the root formatting configuration for `repl-format-config` and `current-format-config`.

repl-format-config `object`
The formatting configuration that a read-eval-print loop might use for displaying the result of an evaluation. Initially, `repl-format-config` is set to an empty formatting configuration with parent `base-format-config`.

current-format-config `parameter object`
Parameter object referring to the current formatting configuration that is used as a default whenever no specific formatting configuration is specified, e.g. by procedure `format`. Initially, `current-format-config` is set to an empty formatting configuration with parent `base-format-config`.

(format *[port] [config] [locale] [tabw [linew]] cntrl arg* …) `procedure`
`format` is the universal formatting procedure provided by library `(lispkit format)`. `format` creates formatted output by outputting the characters of the control string *cntrl* while interpreting formatting directives embedded in *cntrl*. Each formatting directive is prefixed with a tilde which might be preceded by formatting parameters and modifiers. The next character identifies the formatting directive and thus determines what output is being generated by the directive. Most directives use one or more arguments *arg* as input.

Formatting configuration *config* defines environment variables influencing the output of some formatting directives. If *config* is not provided, the formatting configuration from parameter object `current-format-config` is used. For convenience, some environment variables, such as *locale*, can be overridden if they are provided when `format` is being invoked. *locale* refers to a locale identifier like `en_US` that is used by locale-specific formatting directives. *tabw* defines the maximum number of space characters that correspond to a single tab character. *linew* specifies the number of characters per line; this is used by the justification directive only.

(format-config? *obj*) `procedure`
Returns `#t` if *obj* is a formatting configuration; otherwise `#f` is returned.

(format-config *[parent] [locale] [tabw [linew]] (tag cntrl [config])* …) `procedure`
Creates a new formatting configuration with *parent* as parent configuration. If *parent* is not provided explicitly, `current-format-config` is used. If *parent* is `#f`, the new formatting configuation will not have a parent configuration. *locale* refers to a locale identifier like `en_US` that is used by locale-specific formatting directives. *tabw* defines the maximum number of space characters that correspond to a single tab character. *linew* specifies the maximum number of characters per line.

(make-format-config *parent*) `procedure`
(make-format-config *parent locale*)
(make-format-config *parent locale tabw*)
(make-format-config *parent locale tabw linew*)

Creates a new formatting configuration with *parent* as parent configuration. If *parent* is `#f`, the new formatting configuation does not have a parent configuration. The remaining arguments define overrides for the environment variables inherited from *parent*.

locale refers to a locale identifier like `en_US` that is used by locale-specific formatting directives. *tabw* defines the maximum number of space characters that correspond to a single tab character. *linew* specifies the maximum number of characters per line.

(copy-format-config *config*) procedure
(copy-format-config *config collapse?*)

Returns a copy of formatting configuration *config*. If either *collapse?* is omitted or set to `#f`, a 1:1 copy of *config* is being made. If *collapse?* is set to true, a new format config without parent configuration is created which contains the same values for the supported formatting environment variables as *config*.

(merge-format-config *child parent*) procedure
Merges the format configurations *child* and *parent* by creating a new collapsed copy of *child* whose parent configuration *parent* is.

(format-config-locale) procedure
(format-config-locale *config*)

Returns the locale defined by format configuration *config*. If *config* defines a locale itself, it is being returned. Otherwise, the locale of the parent configuration of *config* gets returned. If *config* is not provided, the default configuration `current-format-config` is used.

(format-config-locale-set! *locale*) procedure
(format-config-locale-set! *config locale*)

Sets the locale of the format configuration *config* to *locale*. If *locale* is `#f`, the locale setting gets removed from *config* (but might still get inherited from *config*'s parents). If *config* is not provided, the default configuration `current-format-config` gets mutated.

(format-config-tabwidth) procedure
(format-config-tabwidth *config*)

Returns the width of a tab character in terms of space characters defined by format configuration *config*. If *config* defines a tab width itself, it is being returned. Otherwise, the tab width of the parent configuration of *config* gets returned. If *config* is not provided, the default configuration `current-format-config` is used.

(format-config-tabwidth-set! *tabw*) procedure
(format-config-tabwidth-set! *config tabw*)

Sets the tab width of the format configuration *config* to *tabw*. If *tabw* is `#f`, the tab width setting gets removed from *config* (but might still get inherited from *config*'s parents). If *config* is not provided, the default configuration `current-format-config` gets mutated. The "tab width" is the maximum number of space characters representing one tab character.

(format-config-linewidth) procedure
(format-config-linewidth *config*)

Returns the maximum number of characters per line defined by format configuration *config*. If *config* defines a line width itself, it is being returned. Otherwise, the line width of the parent configuration of *config* gets returned. If *config* is not provided, the default configuration `current-format-config` is used.

(format-config-linewidth-set! *linew*) procedure
(format-config-linewidth-set! *config linew*)

Sets the line width of the format configuration *config* to *linew*. If *linew* is `#f`, the line width setting gets removed from *config* (but might still get inherited from *config*'s parents). If *config* is not provided, the default configuration `current-format-config` gets mutated. The "line width" is the maximum number of characters per line.

(format-config-control-set! *tag cntrl*) procedure
(format-config-control-set! *tag cntrl sconf*)
(format-config-control-set! *config tag cntrl*)
(format-config-control-set! *config tag cntrl sconf*)

Declares for formatting configuration *config* that objects whose type has type tag *tag* are being formatted with control string *cntrl* by formatting directive `~S`. If formatting configuration *sconf* is provided, it is used as a type-specific configuration that is merged with the current configuration when `~S` formats objects of type tag *tag*. If *cntrl* is `#f`, type-specific formatting rules for *tag* are being removed from *conf* (but might still be inherited from the parent of *conf*). If *cntrl* is `#t`, native formatting is being forced for *tag*, no matter what is inherited from the parent of *config*. If *config* is not provided, the default configuration `current-format-config` gets mutated.

(format-config-control-remove! *tag*) procedure
(format-config-control-remove! *config tag*)

Removes any type-specific formatting with directive `~S` for objects whose type has tag *tag* from formatting configuration *config*. If *config* is not provided, the default configuration `current-format-config` gets mutated.

(format-config-controls) procedure
(format-config-controls *config*)

Returns a list of type tags, i.e. symbols, for which there is a type-specific formatting control string defined by formatting configuration *config* or its parents. If *config* is not provided, the default configuration `current-format-config` gets mutated.

(format-config-parent) procedure
(format-config-parent *config*)

Returns the parent configuration of format configuration *config*. If *config* is not provided, the default configuration `current-format-config` is used. `format-config-parent` returns `#f` if *config* does not have a parent formatting configuration.

26 LispKit Graph

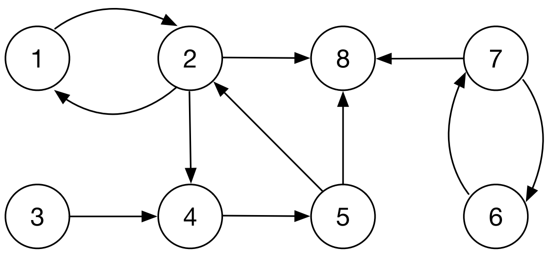
Library `(lispkit graph)` provides a simple API for representing, manipulating, and reasoning about directed graphs.

Graphs are mutable objects encapsulating *nodes* and *edges*. Since graphs organize nodes internally in a hashtable, each graph requires an equivalence and hash function upon creation.

Here is an example for creating and initializing a directed graph with library `(lispkit graph)`. Graph `g` consists of the nodes {1, 2, 3, 4, 5, 6, 7, 8} and the edges {1 → 2, 2 → 1, 2 → 4, 3 → 4, 4 → 5, 5 → 2, 5 → 8, 6 → 7, 7 → 6, 7 → 8, 2 → 8}.

```
(define g (graph
            identity           ; node hash function
            =                  ; node equality function
            '(1 2 3 4 5 6 7 8) ; the nodes
            '((1 2)(2 1)(2 4)  ; the edges
              (3 4)(4 5)(5 2)
              (5 8)(6 7)(7 6)
              (7 8)(2 8))))
```

This is the graph that is implemented by this code:



The following lines illustrate some of the features of library `(lispkit graph)`:

```
(graph-out-degree g 2)       ⇒ 3
(graph-has-edge? g 3 1)      ⇒ #f
(graph-neighbors g 2)        ⇒ (8 4 1)
(graph-reachable? g 3 1)     ⇒ #t
(graph-reachable-nodes g 3) ⇒ (1 2 8 5 4 3)
```

There are also a few advanced algorithms for directed graphs implemented by the library:

```
(graph-weakly-connected-components g)
⇒ ((1 2 6 8 7 5 4 3))
(graph-strongly-connected-components g)
⇒ ((3) (5 2 4 1) (7 6) (8))
(graph-shortest-path g 3 8)
⇒ (3 4 5 8)
  3.0
```

26.1 Constructors

(make-graph *hash equiv*) procedure

Creates and returns a new empty graph object using *hash* as the hash function and *equiv* as the equivalence function for nodes.

(make-eq-graph) procedure

Returns a new empty graph object using `eq-hash` as the hash function and `eq?` as the equivalence function for nodes.

(make-eqv-graph) procedure

Returns a new empty graph object using `eqv-hash` as the hash function and `eqv?` as the equivalence function for nodes.

(make-equal-graph) procedure

Returns a new empty graph object using `equal-hash` as the hash function and `equal?` as the equivalence function for nodes.

(graph *hash equiv nodes edges*) procedure

Creates and returns a new graph object using *hash* as the hash function and *equiv* as the equivalence function for nodes. *nodes* is a list of all the graph's initial nodes, and *edges* is a list of all the initial edges of the graph. Each edge is represented as a list of the form (*from to*) or (*from to* . *label*) where *from* and *to* are nodes, and *label* is an arbitrary Lisp object used as a label annotation.

```
(define g0
  (graph identity =
         '(1 2 3)
         '((1 2 . "one") (1 3 . "two"))))
(graph-neighbors+labels g0 1)
⇒ ((3 . "two") (2 . "one"))
```

(eq-graph *nodes edges*) procedure

Creates and returns a new graph object using `eq-hash` as the hash function and `eq?` as the equivalence function for nodes. *nodes* is a list of all the graph's initial nodes, and *edges* is a list of all the initial edges of the graph. Each edge is represented as a list of the form (*from to*) or (*from to* . *label*) where *from* and *to* are nodes, and *label* is an arbitrary Lisp object used as a label annotation.

```
(define g (eq-graph '(1 2 3) '((1 2) (1 3))))
(graph-neighbors g 1)
⇒ (3 2)
```

(eqv-graph *nodes edges*) procedure

Creates and returns a new graph object using `eqv-hash` as the hash function and `eqv?` as the equivalence function for nodes. *nodes* is a list of all the graph's initial nodes, and *edges* is a list of all the initial edges of the graph. Each edge is represented as a list of the form (*from to*) or (*from to* . *label*) where *from* and *to* are nodes, and *label* is an arbitrary Lisp object used as a label annotation.

```
(define g (eqv-graph '(1 2 3) '((1 2) (2 3) (1 3))))
(graph-edges g)
⇒ ((1 2) (1 3) (2 3))
```

(equal-graph *nodes edges*) procedure

Creates and returns a new graph object using `equal-hash` as the hash function and `equal?` as the equivalence function for nodes. *nodes* is a list of all the graph's initial nodes, and *edges* is a list of all the initial edges of the graph. Each edge is represented as a list of the form (*from to*) or (*from to* . *label*) where *from* and *to* are nodes, and *label* is an arbitrary Lisp object used as a label annotation.

```
(define g (equal-graph '(1 2 3) '((1 2) (1 3 . 9.8))))
(graph-neighbors+labels g 1)
⇒ ((3 . 9.8) (2 . #f))
```

graph-type-tag object

Symbol representing the `graph` type. The `type-for` procedure of library `(lispkit type)` returns this symbol for all graph objects.

26.2 Predicates

(graph? ***obj*****)** procedure

Returns `#t` if *obj* is a graph.

(eq-graph? ***obj*****)** procedure

Returns `#t` if *obj* is a graph using `eq-hash` as hash function and `eq?` as equivalence function for nodes.

(eqv-graph? ***obj*****)** procedure

Returns `#t` if *obj* is a graph using `eqv-hash` as hash function and `eqv?` as equivalence function for nodes.

(equal-graph? ***obj*****)** procedure

Returns `#t` if *obj* is a graph using `eq-hash` as hash function and `eq?` as equivalence function for nodes.

(graph-empty? ***graph*****)** procedure

Returns `#t` if *graph* is an empty graph, i.e. a graph without nodes and edges.

(graph-cyclic? ***graph*****)** procedure

Returns `#t` if *graph* is a cyclic graph, i.e. a graph which has at least one node with a path to itself.

```
(define g
  (eq-graph '(1 2 3 4) '((1 2)(2 3)(3 4))))
(graph-cyclic? g)
⇒ #f
(graph-add-edge! g 4 2)
(graph-cyclic? g)
⇒ #t
```

26.3 Introspection

(node-equivalence-function ***graph*****)** procedure

Returns the node equivalence function used by *graph*.

(node-hash-function ***graph*****)** procedure

Returns the node hash function used by *graph*.

(graph-has-node? ***graph node*****)** procedure

Returns `#t` if *graph* contains *node*; otherwise `#f` is returned.

(graph-nodes ***graph*****)** procedure

Returns a list of all nodes of *graph*.

(graph-has-edge? ***graph edge*****)** procedure
(graph-has-edge? ***graph from to*****)**

Returns `#t` if *edge* is contained in *graph*, `#f` otherwise. *edge* is a list with at least two elements: the first element is the starting node, the second element is the end node. Alternatively, it is possible to provide the start and end node explicitly as parameters *from* and *to*.

(graph-edges *graph*) procedure

Returns a list of all edges of *graph*. Each edge is represented as a list of the form (*from to*) or (*from to* . *label*) where *from* and *to* are nodes, and *label* is an arbitrary Lisp object representing the edge label.

```
(define g
  (eq-graph '(1 2 3) '((1 2 . "a") (1 3))))
(graph-edges g)
⇒ ((1 2 . "a") (1 3))
```

(graph-edge-label *graph from to*) procedure

Returns the label for the edge from node *from* to node *to*. If there is no label associated with the edge, `#f` is returned. It is an error when `graph-edge-label` gets invoked for an edge that does not exist.

(graph-in-degree *graph node*) procedure

Returns the number of edges that lead into *node* in *graph*. It is an error if *node* is not contained in *graph*.

(graph-in-degree *graph node*) procedure

Returns the number of edges that lead into *node* in *graph*. It is an error if *node* is not contained in *graph*.

(graph-out-degree *graph node*) procedure

Returns the number of edges that originate in *node* within *graph*. It is an error if *node* is not contained in *graph*.

(graph-neighbors *graph from*) procedure

Returns the number of edges that originate in *node* within *graph*. It is an error if *node* is not contained in *graph*.

(graph-neighbors *graph from*) procedure

Returns a list of neighbors of node *from* in *graph*. A neighbor `n` is a node for which there is an edge originating in *from* and leading to `n`. `graph-neighbors` returns `#f` if *from* is not a node of *graph*.

(graph-neighbors+labels *graph from*) procedure

Returns a list of pairs, consisting of neighbors with associated labels of node *from* in *graph*. A neighbor `n` is a node for which there is an edge originating in *from* and leading to `n`. The associated label is the label of this edge of `#f` if it does not exist. `graph-neighbors+labels` returns `#f` if *from* is not a node of *graph*.

```
(define g
  (eq-graph '(1 2 3) '((1 2 . "a") (1 3))))
(graph-neighbors+labels g 1)
⇒ ((3 . #f) (2 . "a"))
```

26.4 Mutation

(graph-add-node! *graph node*) procedure

Adds *node* to *graph*. It is an error if the comparison and hash functions associated with *graph* are incompatible with *node*.

(graph-remove-node! *graph node*) procedure

Removes *node* from *graph* if it contains *node*, and does nothing otherwise. It is an error if the comparison and hash functions associated with *graph* are incompatible with *node*.

(graph-add-edge! ***graph edge*****)** procedure
(graph-add-edge! ***graph edge*****)**
(graph-add-edge! ***graph from to label*****)**

Adds *edge* to *graph*. *edge* is represented as a list of the form (*from to*) or (*from to* . *label*) where *from* and *to* are nodes, and *label* is an arbitrary Lisp object used as a label annotation.

(graph-remove-edge! ***graph edge*****)** procedure
(graph-remove-edge! ***graph from to*****)**
(graph-remove-edge! ***graph from to label*****)**

Removes *edge* from *graph*. *edge* is represented as a list of the form (*from to*) or (*from to* . *label*) where *from* and *to* are nodes, and *label* is an arbitrary Lisp object used as a label annotation. The label given in *edge* does not need to match the label of that edge in *graph* for the edge to be removed.

26.5 Transformation

(graph-copy ***graph*****)** procedure
(graph-copy ***graph rnodes*****)**

Returns a copy of *graph* that only includes nodes from list *rnodes*; i.e. *rnodes* acts as a node filter. Edges that either originate or lead into nodes that are not contained in *rnodes* will not be copied over.

(graph-transpose ***graph*****)** procedure
Returns a copy of *graph* with all edges reversed, i.e. start and end nodes swapped.

```
(define g
  (eq-graph '(1 2 3 4) '((1 2 . "a") (1 3) (4 1))))
(graph-edges (graph-transpose g))
⇒ ((3 1) (1 4) (2 1 . "a"))
```

(graph-complement ***graph*****)** procedure
Returns a new graph containing all possible edges that have not been contained in *graph*. The new graph does not have any edge labels.

```
(define g
  (eq-graph '(1 2 3) '((1 2 . "a") (1 3))))
(graph-edges (graph-complement g))
⇒ ((3 2) (3 1) (3 3) (1 1) (2 2) (2 1) (2 3))
```

26.6 Processing graphs

(graph-fold-nodes ***f z graph*****)** procedure
This is the fundamental iterator for graph nodes. Applies the procedure *f* across all nodes of *graph* using initial state value *z*. That is, if *graph* is empty, the procedure returns *z*. Otherwise, some node *n* of *graph* is chosen; let *g'* be the remaining graph without node *n*. `graph-fold-nodes` returns `(graph-fold-nodes f (f n z) g')`.

(graph-fold-edges ***f z graph*****)** procedure
This is the fundamental iterator for graph edges. Applies the procedure *f* across all edges of *graph* using initial state value *z*. That is, if *graph* is empty, the procedure returns *z*. Otherwise, some edge of *graph* is chosen with start node *s*, end node *e* and label *l*; let *g'* be the remaining graph without this edge. `graph-fold-edges` returns `(graph-fold-edges f (f s e l z) g')`.

(graph-reachable? *graph from to*) procedure

Returns `#t` if node `to` is reachable from node *from*, i.e. there is a path/sequence of edges for getting from node *from* to node *to*. Otherwise, `#f` is returned.

(graph-reachable-nodes *graph from*) procedure
(graph-reachable-nodes *graph from limit*)

Returns a list of all nodes that are reachable from node *from* within *graph*. These are nodes for which there is a path/sequence of edges starting from node *from*. *limit* specifies the maximum number of edges in the paths to explore.

(graph-topological-sort *graph*) procedure

Returns a list of nodes of *graph* that are topologically sorted. A topological sort of a directed graph is a linear ordering of its nodes such that for every directed edge from node *u* to node *v*, *u* comes before *v* in the ordering. `graph-topological-sort` returns `#f` if *graph* is cyclic. In this case, it is not possible to sort all nodes topologically.

(graph-weakly-connected-components *graph*) procedure

Returns a list of all weakly connected components of *graph*. Each component is represented as a list of nodes. A weakly connected component is a subgraph of *graph* where all nodes are connected to each other by some path, ignoring the direction of edges.

(graph-strongly-connected-components *graph*) procedure

Returns a list of all strongly connected components of *graph*. Each component is represented as a list of nodes. A strongly connected component is a subgraph of *graph* where all nodes are connected to each other by some path.

(graph-shortest-path *graph from to*) procedure
(graph-shortest-path *graph from to distance*)

Returns a shortest path from node *from* to node *to* in *graph*. *distance* is a distance function taking a starting and ending node as arguments. By default *distance* returns 1.0 for all edges. A path is represented as a list of nodes.

(graph-shortest-paths *graph from*) procedure
(graph-shortest-paths *graph from distance*)

Returns all the shortest paths from node *from* to node *to* in *graph*. *distance* is a distance function taking a starting and ending node as arguments. By default *distance* returns 1.0 for all edges. Paths are represented as lists of nodes.

27 LispKit Gvector

This library defines an API for *growable vectors*. Just like regular vectors, *growable vectors* are heterogeneous sequences of elements which are indexed by a range of integers. Unlike for regular vectors, the length of a *growable vector* is not fixed. Growable vectors may expand or shrink in length. Nevertheless, growable vectors are fully compatible to regular vectors and all operations from library `(lispkit vector)` may also be used in combination with growable vectors. The main significance of library `(lispkit gvector)` is in providing functions to construct growable vectors. Growable vectors are always *mutable* by design.

Just like for vectors with a fixed length, the valid indexes of a growable vector are the exact, non-negative integers less than the length of the vector. The first element in a vector is indexed by zero, and the last element is indexed by one less than the length of the growable vector.

Two growable vectors are `equal?` if they have the same length, and if the values in corresponding slots of the vectors are `equal?`. A growable vector is never `equal?` a regular vector of fixed length.

Growable vectors are written using the notation `#g(obj ...)`. For example, a growable vector of initial length 3 containing the number one as element 0, the list (8 16 32) as element 1, and the string "Scheme" as element 2 can be written as follows: `#g(1 (8 16 32) "Scheme")`. Growable vector constants are self-evaluating, so they do not need to be quoted in programs.

27.1 Predicates

(gvector? *obj*) procedure

Returns `#t` if *obj* is a growable vector; otherwise returns `#f`.

(gvector-empty? *obj*) procedure

Returns `#t` if *obj* is a growable vector of length zero; otherwise returns `#f`.

27.2 Constructors

(make-gvector) procedure
(make-gvector *c*)

Returns a newly allocated growable vector of capacity *c*. The capacity is used to pre-allocate space for up to *c* elements.

(gvector *obj* ...) procedure

Returns a newly allocated growable vector whose elements contain the given arguments.

```
(gvector 'a 'b 'c)  ⇒  #g(a b c)
```

(list->gvector *list*) procedure

The `list->gvector` procedure returns a newly created growable vector initialized to the elements of the list *list* in the order of the list.

```
(list->gvector '(a b c)) ⇒ #g(a b c)
```

(vector->gvector *vector*) procedure

Returns a newly allocated growable vector initialized to the elements of the vector *vector* in the order of *vector*.

(gvector-copy *vector*) procedure
(gvector-copy *vector start*)
(gvector-copy *vector start end*)

Returns a newly allocated copy of the elements of the given growable vector between *start* and *end*, but excluding the element at index *end*. The elements of the new vector are the same (in the sense of `eqv?`) as the elements of the old.

(gvector-append *vector* ...) procedure

Returns a newly allocated growable vector whose elements are the concatenation of the elements of the given vectors.

```
(gvector-append #(a b c) #g(d e f)) ⇒ #g(a b c d e f)
```

(gvector-concatenate *vector xs*) procedure

Returns a newly allocated growable vector whose elements are the concatenation of the elements of the vectors in *xs*. *xs* is a proper list of vectors.

```
(gvector-concatenate '(#g(a b c) #(d) #g(e f))) ⇒ #g(a b c d e f)
```

(gvector-map *f vector1 vector2* ...) procedure

Constructs a new growable vector of the shortest size of the vector arguments *vector1*, *vector2*, etc. Each element at index *i* of the new vector is mapped from the old vectors by `(f (vector-ref vector1 i) (vector-ref vector2 i) ...)` . The dynamic order of the application of f is unspecified.

```
(gvector-map + #(1 2 3 4 5) #g(10 20 30 40)) ⇒ #g(11 22 33 44)
```

(gvector-map/index *f vector1 vector2* ...) procedure

Constructs a new growable vector of the shortest size of the vector arguments *vector1*, *vector2*, etc. Each element at index *i* of the new vector is mapped from the old vectors by `(f i (vector-ref vector1 i) (vector-ref vector2 i) ...)` . The dynamic order of the application of f is unspecified.

```
(gvector-map/index (lambda (i x y) (cons i (+ x y)))
                   #g(1 2 3)
                   #(10 20 30))
⇒ #g((0 . 11) (1 . 22) (2 . 33))
```

27.3 Iterating over vector elements

(gvector-for-each *f vector1 vector2* ...) procedure

`gvector-for-each` implements a simple vector iterator: it applies *f* to the corresponding list of parallel elements from vectors *vector1 vector2* ... in the range *[0, length)*, where *length* is the length of the smallest vector argument passed. In contrast with `gvector-map` , *f* is reliably applied to each subsequent element, starting at index 0, in the vectors.

```
(gvector-for-each (lambda (x) (display x) (newline))
                  #g("foo" "bar" "baz" "quux" "zot"))
⇒
foo
bar
baz
quux
zot
```

(gvector-for-each/index *f vector1 vector2 ...*) procedure

`gvector-for-each/index` implements a simple vector iterator: it applies *f* to the index *i* and the corresponding list of parallel elements from *vector1 vector2* ... in the range *[0, length)*, where *length* is the length of the smallest vector argument passed. The only difference to `gvector-for-each` is that `gvector-for-each/index` always passes the current index as the first argument of *f* in addition to the elements from the vectors *vector1 vector2* ....

```
(gvector-for-each/index
  (lambda (i x) (display i)(display ": ")(display x)(newline))
  #g("foo" "bar" "baz" "quux" "zot"))
⇒
0: foo
1: bar
2: baz
3: quux
4: zot
```

27.4 Managing vector state

(gvector-length *vector*) procedure

Returns the number of elements in growable vector *vector* as an exact integer.

(gvector-ref *vector k*) procedure

The `gvector-ref` procedure returns the contents of element *k* of *vector*. It is an error if *k* is not a valid index of *vector* or if *vector* is not a growable vector.

```
(gvector-ref '#g(1 1 2 3 5 8 13 21) 5) ⇒  8
(gvector-ref '#g(1 1 2 3 5 8 13 21) (exact (round (* 2 (acos -1))))) ⇒  13
```

(gvector-set! *vector k obj*) procedure

The `vector-set!` procedure stores *obj* in element *k* of growable vector *vector*. It is an error if *k* is not a valid index of *vector* or if *vector* is not a growable vector.

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

(gvector-add! *vector obj ...*) procedure

Appends the values *obj*, ... to growable vector *vector*. This increases the length of the growable vector by the number of *obj* arguments.

```
(let ((vec (gvector 0 '(2 2 2 2) "Anna")))
  (gvector-add! vec "Micha")
  vec)
  ⇒ #g(0 (2 2 2 2) "Anna" "Micha")
```

(gvector-insert! *vector k obj*) procedure

Inserts the value *obj* into growable vector *vector* at index *k*. This increases the length of the growable vector by one.

```
(let ((vec (gvector 0 '(2 2 2 2) "Anna")))
  (gvector-insert! vec 1 "Micha")
  vec)
  ⇒ #g(0 "Micha" (2 2 2 2) "Anna")
```

(gvector-remove! *vector k*) procedure

Removes the element at index *k* from growable vector *vector*. This decreases the length of the growable vector by one.

```
(let ((vec (gvector 0 '(2 2 2 2) "Anna")))
  (gvector-remove! vec 1)
  vec)
  ⇒ #g(0 "Anna")
```

(gvector-remove-last! *vector*) procedure

Removes the last element of the growable vector *vector*. This decreases the length of the growable vector by one.

```
(let ((vec (gvector 0 '(2 2 2 2) "Anna")))
  (gvector-remove-last! vec)
  vec)
  ⇒ #g(0 (2 2 2 2))
```

27.5 Destructive growable vector operations

Procedures which operate only on a part of a growable vector specify the applicable range in terms of an index interval [*start*; *end*[; i.e. the *end* index is always exclusive.

(gvector-copy! *to at from*) procedure
(gvector-copy! *to at from start*)
(gvector-copy! *to at from start end*)

Copies the elements of vector *from* between *start* and *end* to growable vector *to*, starting at *at*. The order in which elements are copied is unspecified, except that if the source and destination overlap, copying takes place as if the source is first copied into a temporary vector and then into the destination. *start* defaults to 0 and *end* defaults to the length of *vector*.

It is an error if *at* is less than zero or greater than the length of *to*. It is also an error if `(- (gvector-length to) at)` is less than `(- end start)`.

```
(define a (vector 1 2 3 4 5))
(define b (gvector 10 20 30 40 50))
(gvector-copy! b 1 a 0 2)
b  ⇒ #g(10 1 2 40 50)
```

(gvector-append! ***vector v1 ...*****)** procedure

Appends the elements of the vectors *v1* ... to the growable vector *vector* in the given order.

(gvector-reverse! ***vector*****)** procedure
(gvector-reverse! ***vector start*****)**
(gvector-reverse! ***vector start end*****)**

Procedure `gvector-reverse!` destructively reverses the contents of growable *vector* between *start* and *end*. *start* defaults to 0 and *end* defaults to the length of *vector*.

```
(define a (gvector 1 2 3 4 5))
(vector-reverse! a)
a  ⇒  #g(5 4 3 2 1)
```

(gvector-sort! ***pred vector*****)** procedure

Procedure `gvector-sort!` destructively sorts the elements of growable vector *vector* using the "less than" predicate *pred*.

```
(define a (gvector 7 4 9 1 2 8 5))
(gvector-sort! < a)
a  ⇒  #g(1 2 4 5 7 8 9)
```

(gvector-map! ***f vector1 vector2 ...*****)** procedure

Similar to `gvector-map` which maps the various elements into a new vector via function *f*, procedure `gvector-map!` destructively inserts the mapped elements into growable vector *vector1*. The dynamic order in which *f* gets applied to the elements is unspecified.

```
(define a (gvector 1 2 3 4))
(gvector-map! + a #(10 20 30))
a  ⇒  #g(11 22 33 4)
```

(gvector-map/index! ***f vector1 vector2 ...*****)** procedure

Similar to `gvector-map/index` which maps the various elements together with their index into a new vector via function *f*, procedure `gvector-map/index!` destructively inserts the mapped elements into growable vector *vector1*. The dynamic order in which *f* gets applied to the elements is unspecified.

```
(define a #g(1 2 3 4))
(gvector-map/index! (lambda (i x y) (cons i (+ x y))) a #(10 20 30))
a  ⇒  #g((0 . 11) (1 . 22) (2 . 33) 4)
```

27.6 Converting growable vectors

(gvector->list ***vector*****)** procedure
(gvector->list ***vector start*****)**
(gvector->list ***vector start end*****)**

The `gvector->list` procedure returns a newly allocated list of the objects contained in the elements of growable *vector* between *start* and *end* in the same order as in *vector*.

```
(gvector->list '#g(dah dah didah))  ⇒  (dah dah didah)
(gvector->list '#g(dah dah didah) 1 2)  ⇒  (dah)
```

(gvector->vector *vector*) procedure
(gvector->vector *vector start*)
(gvector->vector *vector start end*)

The `gvector->list` procedure returns a newly allocated list of the objects contained in the elements of growable vector *vector* between *start* and *end* in the same order as in *vector*.

```
(gvector->list '#(dah dah didah))  ⇒  error since the argument is not a gvector
(gvector->list '#g(dah dah didah) 1 2)  ⇒  (dah)
```

28 LispKit Hashtable

Library `(lispkit hashtable)` provides a native implementation of hashtables based on the API defined by R6RS.

A hashtable is a data structure that associates keys with values. Any object can be used as a key, provided a hash function and a suitable equivalence function is available. A hash function is a procedure that maps keys to exact integer objects. It is the programmer's responsibility to ensure that the hash function is compatible with the equivalence function, which is a procedure that accepts two keys and returns true if they are equivalent and `#f` otherwise. Standard hashtables for arbitrary objects based on the `eq?` , `eqv?` , and `equal?` predicates are provided. Also, hash functions for arbitrary objects, strings, and symbols are included.

The specification below uses the *hashtable* parameter name for arguments that must be hashtables, and the *key* parameter name for arguments that must be hashtable keys.

28.1 Constructors

(make-eq-hashtable) procedure
(make-eq-hashtable *k*)

Returns a newly allocated mutable hashtable that accepts arbitrary objects as keys and compares those keys with `eq?` . If an argument is given, the initial capacity of the hashtable is set to approximately *k* elements.

(make-eqv-hashtable) procedure
(make-eqv-hashtable *k*)

Returns a newly allocated mutable hashtable that accepts arbitrary objects as keys and compares those keys with `eqv?` . If an argument is given, the initial capacity of the hashtable is set to approximately *k* elements.

(make-equal-hashtable) procedure
(make-equal-hashtable *k*)

Returns a newly allocated mutable hashtable that accepts arbitrary objects as keys and compares those keys with `equal?` . If an argument is given, the initial capacity of the hashtable is set to approximately *k* elements.

(make-hashtable *hash equiv*) procedure
(make-hashtable *hash equiv k*)

Returns a newly allocated mutable hashtable using *hash* as the hash function and *equiv* as the equivalence function for comparing keys. If a third argument *k* is given, the initial capacity of the hashtable is set to approximately *k* elements.

hash and *equiv* must be procedures. *hash* should accept a key as an argument and should return a non-negative exact integer object. *equiv* should accept two keys as arguments and return a single boolean value. Neither procedure should mutate the hashtable returned by `make-hashtable` . Both *hash* and *equiv* should behave like pure functions on the domain of keys. For example, the `string-hash` and `string=?` procedures are permissible only if all keys are strings and the contents of those strings are never changed so long as any of them continues to serve as a key in the hashtable. Furthermore, any pair of keys for which *equiv* returns true should be hashed to the same exact integer objects by *hash*.

(alist->eq-hashtable *alist*) procedure
(alist->eq-hashtable *alist k*)

Returns a newly allocated mutable hashtable consisting of the mappings contained in the association list *alist*. Keys are compared with `eq?`. If argument *k* is given, the capacity of the returned hashtable is set to at least *k* elements.

(alist->eqv-hashtable *alist*) procedure
(alist->eqv-hashtable *alist k*)

Returns a newly allocated mutable hashtable consisting of the mappings contained in the association list *alist*. Keys are compared with `eqv?`. If argument *k* is given, the capacity of the returned hashtable is set to at least *k* elements.

(alist->equal-hashtable *alist*) procedure
(alist->equal-hashtable *alist k*)

Returns a newly allocated mutable hashtable consisting of the mappings contained in the association list *alist*. Keys are compared with `equal?`. If argument *k* is given, the capacity of the returned hashtable is set to at least *k* elements.

(hashtable-copy *hashtable*) procedure
(hashtable-copy *hashtable mutable*)

Returns a copy of *hashtable*. If the *mutable* argument is provided and is true, the returned hashtable is mutable; otherwise it is immutable.

(hashtable-empty-copy *hashtable*) procedure
Returns a new mutable hashtable that uses the same hash and equivalence functions like *hashtable*.

28.2 Type tests

(hashtable? *obj*) procedure
Returns `#t` if *obj* is a hashtable. Otherwise, it returns `#f`.

(eq-hashtable? *obj*) procedure
Returns `#t` if *obj* is a hashtable which uses `eq?` for comparing keys. Otherwise, it returns `#f`.

(eqv-hashtable? *obj*) procedure
Returns `#t` if *obj* is a hashtable which uses `eqv?` for comparing keys. Otherwise, it returns `#f`.

(equal-hashtable? *obj*) procedure
Returns `#t` if *obj* is a hashtable which uses `equal?` for comparing keys. Otherwise, it returns `#f`.

28.3 Inspection

(hashtable-equivalence-function *hashtable*) procedure
Returns the equivalence function used by *hashtable* to compare keys. For hashtables created with `make-eq-hashtable`, `make-eqv-hashtable`, and `make-equal-hashtable`, returns `eq?`, `eqv?`, and `equal?` respectively.

(hashtable-hash-function *hashtable*) procedure
(hashtable-hash-function *hashtable force?*)

Returns the hash function used by *hashtable*. For hashtables created by `make-eq-hashtable` and `make-eqv-hashtable` , `#f` is returned. This behavior can be disabled if boolean parameter *force?* is being provided and set to `#t` . In this case, `hashtable-hash-function` will also return hash functions for `eq` and `eqv` -based hashtables.

(hashtable-mutable? *hashtable*) procedure
Returns `#t` if hashtable is mutable, otherwise `#f` .

28.4 Hash functions

The `equal-hash` , `string-hash` , and `string-ci-hash` procedures are acceptable as the hash functions of a hashtable only, if the keys on which they are called are not mutated while they remain in use as keys in the hashtable.

(equal-hash *obj*) procedure
Returns an integer hash value for *obj*, based on its structure and current contents. This hash function is suitable for use with `equal?` as an equivalence function. Like `equal?` , the `equal-hash` procedure must always terminate, even if its arguments contain cycles.

(eqv-hash *obj*) procedure
Returns an integer hash value for *obj*, based on *obj*'s identity. This hash function is suitable for use with `eqv?` as an equivalence function.

(eq-hash *obj*) procedure
Returns an integer hash value for *obj*, based on *obj*'s identity. This hash function is suitable for use with `eq?` as an equivalence function.

(boolean-hash *b*) procedure
Returns an integer hash value for boolean *b*.

(char-hash *ch*) procedure
Returns an integer hash value for character *ch*. This hash function is suitable for use with `char=?` as an equivalence function.

(char-ci-hash *ch*) procedure
Returns an integer hash value for character *ch*, ignoring case. This hash function is suitable for use with `char-ci=?` as an equivalence function.

(string-hash *str*) procedure
Returns an integer hash value for string *str*, based on its current characters. This hash function is suitable for use with `string=?` as an equivalence function.

(string-ci-hash *str*) procedure
Returns an integer hash value for string *str* based on its current characters, ignoring case. This hash function is suitable for use with `string-ci=?` as an equivalence function.

(symbol-hash *sym*) procedure
Returns an integer hash value for symbol *sym*.

(number-hash *x*) procedure
Returns an integer hash value for numeric value *x*.

(combine-hash *h* ...) procedure
Combines the integer hash values *h* ... into a single hash value.

28.5 Procedures

(hashtable-size *hashtable*) procedure
Returns the number of keys contained in hashtable as an exact integer object.

(hashtable-load *hashtable*) procedure
Returns the load factor of the hashtable. The load factor is defined as the ratio between the number of keys and the number of hash buckets of *hashtable*.

(hashtable-ref *hashtable key default*) procedure
Returns the value in *hashtable* associated with *key*. If *hashtable* does not contain an association for *key*, *default* is returned.

(hashtable-get *hashtable key*) procedure
Returns a pair consisting of a key matching *key* and associated value from *hashtable*. If *hashtable* does not contain an association for *key*, `hashtable-get` returns `#f`.

For example, for a hashtable `ht` containing the mapping `3` to `"three"`, `(hashtable-get ht 3)` will return `(3 . "three")`.

(hashtable-set! *hashtable key obj*) procedure
Changes *hashtable* to associate *key* with *obj*, adding a new association or replacing any existing association for *key*.

(hashtable-delete! *hashtable key*) procedure
Removes any association for *key* within *hashtable*.

(hashtable-add! *hashtable key obj*) procedure
Changes *hashtable* to associate *key* with *obj*, adding a new association for *key*. The difference to `hashtable-set!` is that existing associations of *key* will remain in *hashtable*, whereas `hashtable-set!` replaces an existing association for *key*.

(hashtable-remove! *hashtable key*) procedure
Removes the association for *key* within *hashtable* which was added last, and returns it as a pair consisting of the key matching *key* and its associated value. If there is no association of *key* in *hashtable*, `hashtable-remove!` will return `#f`.

(alist->hashtable! *hashtable alist*) procedure
Adds all the associations from *alist* to *hashtable* using `hashtable-add!`.

(hashtable-contains? *hashtable key*) procedure
Returns `#t` if *hashtable* contains an association for *key*, `#f` otherwise.

(hashtable-update! *hashtable key proc default*) procedure
`hashtable-update!` applies *proc* to the value in *hashtable* associated with *key*, or to *default* if *hashtable* does not contain an association for *key*. The hashtable is then changed to associate *key* with the value returned by *proc*. *proc* is a procedure which should accept one argument, it should return a single value, and should not mutate *hashtable*. The behavior of `hashtable-update!` is equivalent to the following code:

```
(hashtable-set! hashtable
                key
                (proc (hashtable-ref hashtable key default)))
```

(hashtable-clear! *hashtable*) procedure
(hashtable-clear! *hashtable k*)

Removes all associations from *hashtable*. If a second argument *k* is given, the current capacity of the hashtable is reset to approximately *k* elements.

(hashtable-keys *hashtable*) procedure
Returns an immutable vector of all keys in *hashtable*.

(hashtable-values *hashtable*) procedure
Returns an immutable vector of all values in *hashtable*.

(hashtable-entries *hashtable*) procedure
Returns two values, an immutable vector of the keys in *hashtable*, and an immutable vector of the corresponding values.

(hashtable-key-list *hashtable*) procedure
Returns a list of all keys in *hashtable*.

(hashtable-value-list *hashtable*) procedure
Returns a list of all values in *hashtable*.

(hashtable->alist *hashtable*) procedure
Returns a list of all associations in *hashtable* as an association list. Each association is represented as a pair consisting of the key and the corresponding value.

(hashtable-for-each *proc hashtable*) procedure
Applies *proc* to every association in *hashtable*. *proc* should be a procedure accepting two values, a key and a corresponding value.

(hashtable-map! *proc hashtable*) procedure
Applies *proc* to every association in *hashtable*. *proc* should be a procedure accepting two values, a key and a corresponding value, and returning one value. This value and the key will replace the existing binding.

28.6 Composition

(hashtable-union! *hashtable1 hashtable2*) procedure
Includes all associations from *hashtable2* in *hashtable1* if the key of the association is not already contained in *hashtable1*.

(hashtable-intersection! *hashtable1 hashtable2*) procedure
Removes all associations from *hashtable1* for which the key of the association is not contained in *hashtable2*.

(hashtable-difference! *hashtable1 hashtable2*) procedure
Removes all associations from *hashtable1* for which the key of the association is contained in *hashtable2*.

29 LispKit Heap

Library `(lispkit heap)` provides an implementation of a *priority queue* in form of a *binary max heap*. A *max heap* is a tree-based data structure in which for any given node *C*, if *P* is a parent node of *C*, then the value of *P* is greater than or equal to the value of *C*. Heaps are mutable objects.

heap-type-tag — object

Symbol representing the `heap` type. The `type-for` procedure of library `(lispkit type)` returns this symbol for all heap objects.

(make-heap *pred<?*) — procedure

Returns a new empty binary max heap with *pred<?* being the associated ordering function.

(heap-empty? *hp*) — procedure

Returns `#t` if the heap *hp* is empty, otherwise `#f` is returned.

(heap-max *hp*) — procedure

Returns the largest item in heap *hp*, i.e. the item which is larger than all others according to the comparison function of *hp*. Note, `heap-max` does not remove the largest item as opposed to `heap-delete-max!`. If there are no items on the heap, an error is signaled.

(heap-add! *hp e1 ...*) — procedure

Inserts an item into the heap. The same item can be inserted multiple times.

(heap-delete-max! *hp*) — procedure

Returns the largest item in heap *hp*, i.e. the item which is larger than all others according to the comparison function of *hp*, and removes the item from the heap. If the heap is empty, an error is signaled.

(heap-clear! *hp*) — procedure

Removes all items from *hp*. After this procedure has been executed, the heap is empty.

(heap-copy *hp*) — procedure

Returns a copy of heap *hp*.

(heap->vector *hp*) — procedure

Returns a new vector containing all items of the heap *hp* in descending order.

(heap->list *hp*) — procedure

Returns a list containing all items of the heap *hp* in descending order.

(heap->reversed-list *hp*) — procedure

Returns a list containing all items of the heap *hp* in ascending order.

(list->heap! *hp items*) — procedure

Inserts all the items from list *items* into heap *hp*.

(list->heap *items pred<?*) — procedure

Creates a new heap for the given ordering predicate *pred<?* and inserts all the items from list *items* into it. `list->heap` returns the new heap.

(vector->heap *vec pred<?*) — procedure

Creates and returns a new heap for the given ordering predicate *pred<?* and inserts all the items from vector *vec* into it.

30 LispKit Iterate

Library `(lispkit iterate)` defines syntactical forms supporting frequently used iteration patterns. Some of the special forms were inspired by Common Lisp.

(dotimes (*var count*) *body* ...) syntax
(dotimes (*var count result*) *body* ...)

`dotimes` iterates variable *var* over the integer range *[0, count[*, executing *body* for every iteration.

`dotimes` first evaluates *count*, which has to evaluate to a fixnum. If *count* evaluates to zero or a negative number, *body* ... is not executed. `dotimes` then executes *body* ... once for each integer from 0 up to, but not including, the value of *count*, with *var* bound to each integer. Then, *result* is evaluated and its value is returned as the value of the `dotimes` form. If *result* is not provided, no value is being returned.

```
(let ((res 0))
  (dotimes (i 10 res)
    (set! res (+ res i))))
⇒ 45
```

(dolist (*var lst*) *body* ...) syntax
(dolist (*var lst result*) *body* ...)

`dolist` iterates variable *var* over the elements of list *lst*, executing *body* ... for every iteration.

`dolist` first evaluates *lst*, which has to evaluate to a list. It then executes *body* ... once for each element in the list, with *var* bound to the current element of the list. Then, *result* is evaluated and its value is returned as the value of the `dolist` form. If *result* is not provided, no value is being returned.

```
(let ((res ""))
  (dolist (x '("a" "b" "c") res)
    (set! res (string-append res x))))
⇒ "abc"
```

(loop *break body* ...) syntax
`loop` iterates infinitely, executing *body* ... in each iteration. *break* is a variable bound to an exit function which can be used to leave the `loop` form. *break* receives one argument which is the result of the `loop` form.

```
(let ((i 1))
  (loop break
    (if (> i 100) (break i) (set! i (* i 2)))))
⇒ 128
```

(while *condition body* ...) syntax
(while *condition* unless *break body* ...)

`while` iterates as long as *condition* evaluates to a value other than `#f`, executing *body* ... in each iteration. `unless` can be used to bind an exit funtion to variable *break* so that it is possible to leave the loop by calling thunk *break*. `while` forms never return a result.

```
(let ((i 0)(sum 0))
  (while (< sum 100) unless exit
    (if (> i 10) (exit))
    (set! sum (+ sum i))
    (set! i (fx1+ i)))
  (cons i sum))
⇒  (11 . 55)
```

(for *var* from *lo* to *hi* *body* ...) syntax
(for *var* from *lo* to *hi* step *s* *body* ...)

This form of `for` iterates through all the fixnums from *lo* to *hi* (both inclusive), executing *body* ... in each iteration. If step *s* is provided, *s* is used as the increment of variable *var* which iterates through the elements of the given range.

When this `for` form is being executed, first *lo* and *hi* are evaluated. Both have to evaluate to a fixnum. Then, *body* ... is executed once for each integer in the given range, with *var* bound to the current integer. The form returns no result.

```
(let ((res '()))
  (for x from 1 to 16 step 2
    (set! res (cons x res)))
  res)
⇒  (15 13 11 9 7 5 3 1)
```

(for *var* in *lst* *body* ...) syntax
(for *var* in *lst* where *condition* *body* ...)
(for *var* from *(x ...)* *body* ...)

This form of `for` iterates through all the elements of a list, executing *body* ... in each iteration. The list is either explicitly given via *lst* or its elements are enumerated in the form *(x ...)*. If a *where* predicate is provided, the it acts as a filter on the elements through which variable *var* is iterated.

When this `for` form is being executed, first *lst* or *(x ...)* is evaluated. Then, *body* ... is executed once for each element in the list, with *var* bound to the current element of the list. The form returns no result.

```
(let ((res '()))
  (for x in (iota 16) where (odd? x)
    (set! res (cons x res)))
  res)
⇒  (15 13 11 9 7 5 3 1)
```

(exit-with *break* *body* ...) syntax
(exit-with *break* from *body* ...)

`exit-with` is not an iteration construct by itself. It is often used in combination with iteration constructs to declare an exit function for leaving statements *body* .... *break* is a variable which gets bound to the exit function in the scope of statements *body* .... `exit-with` either returns the result of the last statement of *body* ... or it returns the value passed to *break* in case the exit function gets called.

```
(exit-with break
  (display "hello")
  (break #f)
  (display "world"))
⇒  #f  ; printing "hello"
```

31 LispKit List

Lists are heterogeneous data structures constructed out of *pairs* and an *empty list* object.

A *pair* consists of two fields called *car* and *cdr* (for historical reasons). Pairs are created by the procedure `cons`. The *car* and *cdr* fields are accessed by the procedures `car` and `cdr`. As opposed to most other Scheme implementations, lists are immutable in LispKit. Thus, it is not possible to set the car and cdr fields of an already existing pair.

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

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

The objects in the car fields of successive pairs of a list are the *elements* of the list. For example, a two-element list is a pair whose car is the first element and whose cdr is a pair whose car is the second element and whose cdr is the empty list. The *length* of a list is the number of elements, which is the same as the number of pairs.

The empty list is a special object of its own type. It is not a pair, it has no elements, and its length is zero.

The most general notation (external representation) for Scheme pairs is the "dotted" notation `(c1 . c2)` where `c1` is the value of the car field and `c2` is the value of the cdr field. For example `(4 . 5)` is a pair whose car is `4` and whose cdr is `5`. Note that `(4 . 5)` is the external representation of a pair, not an expression that evaluates to a pair.

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

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

and

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

are equivalent notations for a list of symbols.

A chain of pairs not ending in the empty list is called an *improper list*. Note that an improper list is not a list. The list and dotted notations can be combined to represent improper lists:

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

is equivalent to

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

31.1 Basic constructors and procedures

(cons *x y*) procedure

Returns a pair whose car is *x* and whose cdr is *y*.

(car *xs*) procedure

Returns the contents of the car field of pair *xs*. Note that it is an error to take the car of the empty list.

(cdr *xs*) procedure

Returns the contents of the cdr field of pair *xs*. Note that it is an error to take the cdr of the empty list.

(caar *xs*) procedure
(cadr *xs*)
(cdar *xs*)
(cddr *xs*)

These procedures are compositions of `car` and `cdr` as follows:

```
(define (caar x) (car (car x)))
(define (cadr x) (car (cdr x)))
(define (cdar x) (cdr (car x)))
(define (cddr x) (cdr (cdr x)))
```

(caaar *xs*) procedure
(caadr *xs*)
(cadar *xs*)
(caddr *xs*)
(cdaar *xs*)
(cdadr *xs*)
(cddar *xs*)
(cdddr *xs*)

These eight procedures are further compositions of `car` and `cdr` on the same principles. For example, `caddr` could be defined by `(define caddr (lambda (x) (car (cdr (cdr x)))))`. Arbitrary compositions up to four deep are provided.

(caaaar *xs*) procedure
(caaadr *xs*)
(caadar *xs*)
(caaddr *xs*)
(cadaar *xs*)
(cadadr *xs*)
(caddar *xs*)
(cadddr *xs*)
(cdaaar *xs*)
(cdaadr *xs*)
(cdadar *xs*)
(cdaddr *xs*)
(cddaar *xs*)
(cddadr *xs*)
(cdddar *xs*)
(cddddr *xs*)

These sixteen procedures are further compositions of `car` and `cdr` on the same principles. For example, `cadddr` could be defined by `(define cadddr (lambda (x) (car (cdr (cdr (cdr x))))))`. Arbitrary compositions up to four deep are provided.

(make-list *k*) procedure
(make-list *k* *fill*)

Returns a list of *k* elements. If argument *fill* is given, then each element is set to *fill*. Otherwise the content of each element is the empty list.

(list *x* …) procedure
Returns a list of its arguments, i.e. (*x* …).

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

(cons* *e1* *e2* …) procedure
Like `list`, but the last argument provides the tail of the constructed list, returning `(cons e1 (cons e2 (cons ... en)))`. This function is called `list*` in Common Lisp.

```
(cons* 1 2 3 4)  ⇒  (1 2 3 . 4)
(cons* 1)        ⇒  1
```

(length *xs*) procedure
Returns the length of list *xs*.

```
(length '(a b c))          ⇒  3
(length '(a (b) (c d e)))  ⇒  3
(length '())               ⇒  0
```

31.2 Predicates

(pair? *obj*) procedure
Returns `#t` if *obj* is a pair, `#f` otherwise.

(null? *obj*) procedure
Returns `#t` if *obj* is an empty list, `#f` otherwise.

(list? *obj*) procedure
Returns `#t` if *obj* is a proper list, `#f` otherwise. A chain of pairs ending in the empty list is called a *proper list*.

(every? *pred* *xs* …) procedure
Applies the predicate *pred* across the lists *xs* …, returning `#t` if the predicate returns `#t` on every application. If there are *n* list arguments *xs1* … *xsn*, then *pred* must be a procedure taking *n* arguments and returning a single value, interpreted as a boolean. If an application of *pred* returns *#f*, then `every?` returns `#f` immediately without applying *pred* further anymore.

(any? *pred* *xs* …) procedure
Applies the predicate *pred* across the lists *xs* …, returning `#t` if the predicate returns `#t` for at least one application. If there are *n* list arguments *xs1* … *xsn*, then *pred* must be a procedure taking *n* arguments and returning a single value, interpreted as a boolean. If an application of *pred* returns *#t*, then `any?` returns `#t` immediately without applying *pred* further anymore.

31.3 Composing and transforming lists

(append *xs* …) procedure
Returns a list consisting of the elements of the first list *xs* followed by the elements of the other lists. If

there are no arguments, the empty list is returned. If there is exactly one argument, it is returned. The last argument, if there is one, can be of any type. An improper list results if the last argument is not a proper list.

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

(concatenate *xss*) procedure

This procedure appends the elements of the list of lists *xss*. That is, `concatenate` returns `(apply append` *xss*`)`.

(reverse *xs*) procedure

Procedure `reverse` returns a list consisting of the elements of list *xs* in reverse order.

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

(filter *pred xs*) procedure

Returns all the elements of list *xs* that satisfy predicate *pred*. Elements in the result list occur in the same order as they occur in the argument list *xs*.

```
(filter even? '(0 7 8 8 43 -4))  ⇒  (0 8 8 -4)
```

(remove *pred xs*) procedure

Returns a list without the elements of list *xs* that satisfy predicate *pred*: `(lambda (pred list) (filter (lambda (x) (not (pred x))) list))`. Elements in the result list occur in the same order as they occur in the argument list *xs*.

```
(remove even? '(0 7 8 8 43 -4))  ⇒  (7 43)
```

(partition *pred xs*) procedure

Partitions the elements of list *xs* with predicate *pred* returning two values: the list of in-elements (i.e. elements from *xs* satisfying *pred*) and the list of out-elements. Elements occur in the result lists in the same order as they occur in the argument list *xs*.

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

(map *f xs* ...) procedure

The `map` procedure applies procedure *proc* element-wise to the elements of the lists *xs* ... and returns a list of the results, in order. If more than one list is given and not all lists have the same length, map terminates when the shortest list runs out. The dynamic order in which *proc* is applied to the elements of the lists is unspecified.

It is an error if *proc* does not accept as many arguments as there are lists *xs* ... and return a single value.

```
(map cadr '((a b) (d e) (g h)))            ⇒  (b e h)

(map (lambda (n) (expt n n)) '(1 2 3 4 5))  ⇒  (1 4 27 256 3125)
```

```
(map + '(1 2 3) '(4 5 6 7))              ⇒  (5 7 9)

(let ((count 0))
  (map (lambda (ignored)
         (set! count (+ count 1)) count)
       '(a b)))                          ⇒  (1 2)
```

(append-map *f xs* ...) procedure

Maps *f* over the elements of the lists *xs* ..., just as in function `map`. However, the results of the applications are appended together to determine the final result. `append-map` uses `append` to append the results together. The dynamic order in which the various applications of *f* are made is not specified. At least one of the list arguments *xs* ... must be finite.

This is equivalent to `(apply append (map f xs ...))`.

```
(append-map!
  (lambda (x)
    (list x (- x))) '(1 3 8))
⇒ (1 -1 3 -3 8 -8)
```

(filter-map *f xs* ...) procedure

This function works like `map`, but only values differently from `#f` are being included in the resulting list. The dynamic order in which the various applications of *f* are made is not specified. At least one of the list arguments *xs* ... must be finite.

```
(filter-map
  (lambda (x)
    (and (number? x) (* x x))) '(a 1 b 3 c 7))
⇒ (1 9 49)
```

(for-each *f xs* ...) procedure

The arguments to `for-each` *xs* ... are like the arguments to `map`, but `for-each` calls *proc* for its side effects rather than for its values. Unlike `map`, `for-each` is guaranteed to call *proc* on the elements of the lists in order from the first element to the last. If more than one list is given and not all lists have the same length, `for-each` terminates when the shortest list runs out.

(fold-left *f z xs* ...) procedure

Fundamental list recursion operator applying *f* to the elements *x1* ... *xn* of list *xs* in the following way: `(f ... (f (f z x1) x2) ... xn)`. In other words, this function applies *f* recursively based on the following rules, assuming one list parameter *xs*:

```
(fold-left f z xs)   ⇒  (fold-left f (f z (car xs)) (cdr xs))
(fold-left f z '())  ⇒  z
```

If *n* list arguments are provided, then function *f* must take *n + 1* parameters: one element from each list, and the "seed" or fold state, which is initially *z* as its very first argument. The `fold-left` operation terminates when the shortest list runs out of values.

```
(fold-left (lambda (x y) (cons y x)) '() '(1 2 3 4))  ⇒  (4 3 2 1)
(define (xcons+ rest a b) (cons (+ a b) rest))
(fold-left xcons+ '() '(1 2 3 4) '(10 20 30 40 50))   ⇒  (44 33 22 11)
```

Please note, compared to function `fold` from library `(srfi 1)`, this function applies the "seed"/fold state always as its first argument to *f*.

(fold-right *f z xs* ...) procedure

Fundamental list recursion operator applying *f* to the elements *x1* ... *xn* of list *xs* in the following way: `(f x1 (f x2 ... (f xn z)))`. In other words, this function applies *f* recursively based on the following rules, assuming one list parameter *xs*:

```
(fold-right f z xs)              ⇒  (f (car xs) (fold-right f z (cdr xs)))
(fold-right f z '())             ⇒  z
(define (xcons xs x) (cons x xs))
(fold-left xcons '() '(1 2 3 4))  ⇒  (4 3 2 1)
```

If *n* list arguments *xs* ... are provided, then function *f* must take *n + 1* parameters: one element from each list, and the "seed" or fold state, which is initially *z*. The `fold-right` operation terminates when the shortest list runs out of values.

```
(fold-right (lambda (x l) (if (even? x) (cons x l) l)) '()
            '(1 2 3 4 5 6))  ⇒  (2 4 6)
```

As opposed to `fold-left`, procedure `fold-right` is not tail-recursive.

(sort *less xs*) procedure

Returns a sorted list containing all elements of *xs* such that for every element *xi* at position *i*, `(` less *xj xi* `)` returns `#t` for all elements *xj* at position *j* where $j < i$.

(merge *less xs ys*) procedure

Merges two lists *xs* and *ys* which are both sorted with respect to the total ordering predicate *less* and returns the result as a list.

(tabulate *count proc*) procedure

Returns a list with *count* elements. Element *i* of the list, where $0 \leq i < count$, is produced by *(proc i)*.

```
(tabulate 4 fx1+)  ⇒  (1 2 3 4)
```

(iota *count*) procedure
(iota *count start*)
(iota *count start step*)

Returns a list containing the elements `(start start+step ... start+(count-1)*step)`. The *start* and *step* parameters default to 0 and 1.

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

31.4 Finding and extracting elements

(list-tail *xs k*) procedure

Returns the sublist of list *xs* obtained by omitting the first *k* elements. Procedure `list-tail` could be defined by

```
(define (list-tail xs k)
  (if (zero? k) xs (list-tail (cdr xs) (- k 1))))
```

(list-ref *xs k*) procedure

Returns the *k*-th element of list *xs*. This is the same as the car of `(list-tail xs k)`.

(memq *obj xs*) procedure
(memv *obj xs*)
(member *obj xs*)
(member *obj xs compare*)

These procedures return the first sublist of *xs* whose car is *obj*, where the sublists of *xs* are the non-empty lists returned by `(list-tail xs k)` for *k* less than the length of *xs*. If *obj* does not occur in *xs*, then `#f` is returned. The `memq` procedure uses `eq?` to compare *obj* with the elements of *xs*, while `memv` uses `eqv?` and `member` uses *compare*, if given, and `equal?` otherwise.

(delq *obj xs*) procedure
(delv *obj xs*)
(delete *obj xs*)
(delete *obj xs compare*)

Returns a copy of list *xs* with all entries equal to element *obj* removed. `delq` uses `eq?` to compare `obj` with the elements in list *xs*, `delv` uses `eqv?`, and `delete` uses *compare* if given, and `equal?` otherwise.

(assq *obj alist*) procedure
(assv *obj alist*)
(assoc *obj alist*)
(assoc *obj alist compare*)

alist must be an association list, i.e. a list of key/value pairs. This family of procedures finds the first pair in *alist* whose car field is *obj*, and returns that pair. If no pair in *alist* has *obj* as its car, then `#f` is returned. The `assq` procedure uses `eq?` to compare *obj* with the car fields of the pairs in *alist*, while `assv` uses `eqv?` and `assoc` uses *compare* if given, and `equal?` otherwise.

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

(alist-delq *obj alist*) procedure
(alist-delv *obj alist*)
(alist-delete *obj alist*)
(alist-delete *obj alist compare*)

Returns a copy of association list *alist* with all entries removed whose `car` is equal to element *obj*. `alist-delq` uses `eq?` to compare `obj` with the first elements of all members of list *xs*, `alist-delv` uses `eqv?`, and `alist-delete` uses *compare* if given, and `equal?` otherwise.

(key *xs*) procedure
(key *xs default*)

Returns `(car xs)` if *xs* is a pair, otherwise *default* is being returned. If *default* is not provided as an argument, `#f` is used instead.

(value *xs*) procedure
(value *xs default*)

Returns `(cdr xs)` if *xs* is a pair, otherwise *default* is being returned. If *default* is not provided as an argument, `#f` is used instead.

32 LispKit List Set

Library `(lispkit list set)` provides a simple API for list-based sets, called `lset`. Such sets are simply represented as lists (without duplicate entries) with respect to a given equality relation. Every `lset` procedure is provided as its first argument such an equality predicate. It is up to the client of the API to make sure that equality predicate and the given list-based sets are compatible and are used consistently.

An equality predicate `=` is required to be consistent with `eq?`, i.e. it must satisfy `(eq? x y) ⇒ (= x y)`. This implies, in turn, that two lists that are `eq?` are also set-equal by any compliant comparison procedure. This allows for constant-time determination of set operations on `eq?` lists.

(lset = *x* ...) procedure

Returns a list-based set containing all the elements *x* ... without duplicates when using equality predicate = for comparing elements.

(list->lset = *xs*) procedure

Returns a list-based set containing all the elements of list *xs* without duplicates when using equality predicate = for comparing elements.

(lset<=? = *xs* ...) procedure

Returns `#t` iff every list xs_i is a subset of list xs_{i+1} using equality predicate = for comparing elements, otherwise `#f` is returned. List *A* is a subset of list *B* if every element in *A* is equal to some element of *B*. When performing an element comparison, the = procedure's first argument is an element of *A*, its second argument is an element of *B*.

```
(lset<=? eq? '(a) '(a b a) '(a b c c)) ⇒ #t
(lset<=? eq?) ⇒ #t
(lset<=? eq? '(a b)) ⇒ #t
```

(lset=? = *xs* ...) procedure

Returns `#t` iff every list xs_i is set-equal to xs_{i+1} using equality predicate = for comparing elements, otherwise `#f` is returned. "Set-equal" simply means that xs_i is a subset of xs_{i+1}, and xs_{i+1} is a subset of xs_i. When performing an element comparison, the = procedure's first argument is an element of xs_i, its second argument is an element of xs_{i+1}.

```
(lset=? eq? '(b e a) '(a e b) '(e e b a)) ⇒ #t
(lset=? eq?) ⇒ #t
(lset=? eq? '(a b)) ⇒ #t
```

(lset-contains? = *xs* *x*) procedure

Returns `#t` if element *x* is contained in *xs* using equality predicate = for comparing elements. Otherwise, `#f` is returned.

```
(lset-contains? eq? '(a b c) 'b) ⇒ #t
(lset-contains? eq? '(a b c) 'd) ⇒ #f
(lset-contains? eq? '() 'd) ⇒ #f
```

(lset-adjoin = *xs x* ...) procedure

Adds the elements *x* ... not already in the list *xs* and returns the result as a list. The new elements are added to the front of the list, but no guarantees are made about their order. The = parameter is an equality predicate used to determine if an element *x* is already a member of *xs*. Its first argument is an element of *xs*; its second is one of the *x* ... elements. *xs* is always a suffix of the result returned by `lset-adjoin`, even if *xs* contains repeated elements, these are not reduced.

```
(lset-adjoin eq? '(a b c d c e) 'a 'e 'i 'o) ⇒ (o i a b c d c e)
```

(lset-union = *xs* ...) procedure

Returns the union of the lists *xs* using equality predicate = for comparing elements. The union of lists *A* and *B* is constructed as follows:

- If *A* is the empty list, the answer is *B*
- Otherwise, the result is initialised to be list *A*
- Proceed through the elements of list *B* in a left-to-right order. If *b* is such an element of *B*, compare every element *r* of the current result list to *b*: *(= r b)*. If all comparisons fail, *b* is consed onto the front of the result.

In the n-ary case, the two-argument list-union operation is simply folded across the argument lists *xs* ....

```
(lset-union eq? '(a b c d e) '(a e i o)) ⇒ (o i a b c d e)
(lset-union eq? '(a a c) '(x a x)) ⇒ (x a a c)
(lset-union eq?) ⇒ ()
(lset-union eq? '(a b c)) ⇒ (a b c)
```

(lset-intersection = *xs* ...) procedure

Returns the intersection of the lists *xs* using equality predicate = for comparing elements.

The intersection of lists *A* and *B* is comprised of every element of *A* that is = to some element of *B*: *(= a b)*, for *a* in *A*, and *b* in *B*. This implies that an element which appears in *B* and multiple times in list *A* will also appear multiple times in the result.

The order in which elements appear in the result is the same as they appear in xs_1, i.e. `lset-intersection` essentially filters xs_1, without disarranging element order.

In the n-ary case, the two-argument `lset-intersection` operation is simply folded across the argument lists.

```
(lset-intersection eq? '(a b c d e) '(a e i o u)) ⇒ (a e)
(lset-intersection eq? '(a x y a) '(x a x z)) ⇒ (a x a)
(lset-intersection eq? '(a b c)) ⇒ (a b c)
```

(lset-difference = *xs* ...) procedure

Returns the difference of the lists *xs* ... using equality predicate = for comparing elements. The result is a list of all the elements of xs_1 that are not = to any element from one of the other xs_i lists.

The = procedure's first argument is always an element of xs_1 whereas its second argument is an element of one of the other xs_i. Elements that are repeated multiple times in xs_1 will occur multiple times in the result. The order in which elements appear in the result list is the same as they appear in xs_1, i.e. `lset-difference` essentially filters xs_1, without disarranging element order.

```
(lset-difference eq? '(a b c d e) '(a e i o u)) ⇒ (b c d)
(lset-difference eq? '(a b c)) ⇒ (a b c)
```

(lset-xor = *xs* ...) procedure

Returns the exclusive-or of the list-based sets *xs* ... using equality predicate = for comparing elements. If there are exactly two lists, this is all the elements that appear in exactly one of the two lists. The operation is associative, and thus extends to the n-ary case.

More precisely, for two lists *A* and *B*, *A* "xor" *B* is a list of

- every element *a* of *A* such that there is no element *b* of *B* such that *(= a b)*, and
- every element *b* of *B* such that there is no element *a* of *A* such that *(= b a)*.

However, an implementation is allowed to assume that = is symmetric, i.e., that $(= ab) \Rightarrow (= ba)$. This means, e.g. that if a comparison *(= a b)* returns `#t` for some *a* in *A* and *b* in *B*, both *a* and *b* may be removed from inclusion in the result.

In the n-ary case, the binary-xor operation is simply folded across the lists *xs* ....

```
(lset-xor eq? '(a b c d e) '(a e i o u)) ⇒ (d c b i o u)
(lset-xor eq?) ⇒ ()
(lset-xor eq? '(a b c d e)) ⇒ (a b c d e)
```

(lset-diff+intersection = *xs* ...) procedure

Returns two values: the difference and the intersection of the list-based sets *xs* ... using equality predicate = for comparing elements. Is equivalent to but can be implemented more efficiently than the code below. The = procedure's first argument is an element of xs_1, its second arguments is an element of one of the other xs_i. `lset-diff+intersection` essentially partitions xs_1 into elements that are unique to xs_1 and elements that are shared with other xs_i.

```
(values
  (lset-difference = xs ...)
  (lset-intersection = xs1 (lset-union = xs2 ...)))
```

Some of this documentation is derived from SRFI 1 by Olin Shivers.

33 LispKit Log

Library `(lispkit log)` defines a simple logging API for LispKit. Log entries are sent to a *logger*. A logger processes each log entry, e.g. by adding or filtering information, and eventually persists it if the severity of the log entry is at or above the level of the severity of the logger. Supported are logging to a port and into a file. The macOS IDE *LispPad* implements a special version of `(lispkit log)` which makes log messages available in a session logging user interface supporting filtering, sorting, and exporting of log entries.

A log entry consists of the following four components: a timestamp, a severity, a sequence of tags, and a log message. Timestamps are generated via `current-second`. There are five severities, represented as symbols, supported by this library: `debug`, `info`, `warn`, `err`, and `fatal`. Also tags are represented as symbols. The sequence of tags is represented as a list of symbols. A log message is a string.

Logging functions take the logger as an optional argument. If it is not provided, the *current logger* is chosen. The current logger is represented via the parameter object `current-logger`. The current logger is initially set to `default-logger`.

33.1 Log severities

Log severities are represented using symbols. The following symbols are supported:

- `debug` (0),
- `info` (1),
- `warn` (2),
- `err` (3), and
- `fatal` (4).

Each severity has an associated *severity level* (previously listed in parenthesis for each severity). The higher the level, the more severe a logged issue.

default-severity [object]
The default logging severity that is used if no severity is specified (initially `'debug`) when a new empty logger is created via procedure `logger`.

(severity? *obj*) [procedure]
Returns `#t` if *obj* is an object representing a log severity, `#f` otherwise. The following symbols are representing severities: `debug`, `info`, `warn`, `err`, and `fatal`.

(severity->level *sev*) [procedure]
Returns the severity level of severity *sev* as a fixnum.

(severity->string *sev*) [procedure]
Returns a human readable string (in English) for the default textual representation of the given severity *sev*.

33.2 Log formatters

Log formatters are used by port and file loggers to map a structured logging request consisting of a timestamp, severity, log message, and logging tags into a string.

default-log-formatter object

The default log formatting procedure. It is used by default when a new port or file logger gets created and no formatter procedure is provided.

(long-log-formatter *timestamp sev message tags*) procedure

Formatter procedure using a long format.

(short-log-formatter *timestamp sev message tags*) procedure

Formatter procedure using a short format.

33.3 Logger objects

default-logger object

The default logger that is initially created by the logging library. The native implementation for LispKit logs to standard out, the native implementation for LispPad logs into the session logging system of LispPad.

current-logger parameter object

Parameter object referring to the current logger that is used as a default if no logger object is provided for a logging request. Initially `current-logger` is set to `default-logger`.

logger-type-tag object

Symbol representing the `logger` type. The `type-for` procedure of library `(lispkit type)` returns this symbol for all logger objects.

(logger? *obj*) procedure

Returns `#t` if *obj* is a logger object, `#f` otherwise.

(logger) procedure
(logger *sev*)

Returns a new empty logger with the lowest persisted severity *sev*. The logger does not perform any logging action. If *sev* is not provided, `default-severity` is used as a default.

(make-logger *logproc lg*) procedure
(make-logger *logproc deinit lg*)

Returns a new logger with logging procedure *logproc*, the de-initialization thunk *deinit*, and a logger object *lg* which can be used as a delegate and whose state will be inherited (e.g. the lowest persisted severity).

logproc gets called by logging requests via procedures such as `log`, `log-debug`, etc. *logproc* is a procedure with the following signature: `(logproc timestamp sev message tags)`. *timestamp* is a floating-point number representing the number of seconds since 00:00 UTC on January 1, 1970 (e.g. returned by `current-second`), *sev* is a severity, *message* is the log message string, and *tags* is a list of logging tags. A tag is represented as a symbol.

Procedure *deinit* is called without parameters when the logger gets closed via *close-logger* before the de-initialization procedure of *lg* is called.

(make-port-logger *port*) procedure
(make-port-logger *port formatter*)
(make-port-logger *port formatter sev*)

Returns a new port logger object which forwards log messages formatted by *formatter* to *port* if the severity is above the lowest persisted severity *sev*.

formatter is a procedure with the following signature: `(formatter timestamp sev message tags)` . *timestamp* is a floating-point number representing the number of seconds since 00:00 UTC on January 1, 1970, *sev* is a severity, *message* is the log message string, and *tags* is a list of logging tags. A tag is represented as a symbol.

(make-file-logger *path*) procedure
(make-file-logger *path formatter*)
(make-file-logger *path formatter sev*)

Returns a new file logger object which writes log messages formatted by *formatter* into a new file at the given file path *path* if the severity is above the lowest persisted severity *sev*.

formatter is a procedure with the following signature: `(formatter timestamp sev message tags)` . *timestamp* is a floating-point number representing the number of seconds since 00:00 UTC on January 1, 1970, *sev* is a severity, *message* is the log message string, and *tags* is a list of logging tags. A tag is represented as a symbol.

(make-tag-logger *tag lg*) procedure
Returns a new logger which includes *tag* into the tags to log and forwards the logging request to logger *lg*.

(make-filter-logger *filter lg*) procedure
Returns a new logger which filters logging requests via procedure *filter* and forwards the requests which pass the filter to logger *lg*.

filter is a predicate with the following signature: `(filter timestamp sev message tags)` . *timestamp* is a floating-point number representing the number of seconds since 00:00 UTC on January 1, 1970, *sev* is a severity, *message* is the log message string, and *tags* is a list of logging tags. A tag is represented as a symbol.

(close-logger *lg*) procedure
Closes the logger *lg* by calling the deinitialization procedures of the full logger chain of *lg*.

(logger-addproc *lg*) procedure
Returns the logging request procedure used by logger *lg*.

(logger-severity *lg*) procedure
Returns the default logging severity used by logger *lg*.

(logger-severity-set! *lg sev*) procedure
Sets the default logging severity used by logger *lg* to *sev*.

33.4 Logging procedures

(log *sev message*) procedure
(log *sev message tag*)
(log *sev message lg*)
(log *sev message tag lg*)

Logs message string *message* with severity *sev* into logger *lg* with *tag* if provided. If *lg* is not provided, the current logger (as defined by parameter object `current-logger`) is used.

(log-debug *message*) procedure
(log-debug *message tag*)

(log-debug *message lg*)
(log-debug *message tag lg*)

Logs message string *message* with severity `debug` into logger *lg* with *tag* if provided. If *lg* is not provided, the current logger (as defined by parameter object `current-logger`) is used.

(log-info *message*) procedure
(log-info *message tag*)
(log-info *message lg*)
(log-info *message tag lg*)

Logs message string *message* with severity `info` into logger *lg* with *tag* if provided. If *lg* is not provided, the current logger (as defined by parameter object `current-logger`) is used.

(log-warn *message*) procedure
(log-warn *message tag*)
(log-warn *message lg*)
(log-warn *message tag lg*)

Logs message string *message* with severity `warn` into logger *lg* with *tag* if provided. If *lg* is not provided, the current logger (as defined by parameter object `current-logger`) is used.

(log-error *message*) procedure
(log-error *message tag*)
(log-error *message lg*)
(log-error *message tag lg*)

Logs message string *message* with severity `error` into logger *lg* with *tag* if provided. If *lg* is not provided, the current logger (as defined by parameter object `current-logger`) is used.

(log-fatal *message*) procedure
(log-fatal *message tag*)
(log-fatal *message lg*)
(log-fatal *message tag lg*)

Logs message string *message* with severity `fatal` into logger *lg* with *tag* if provided. If *lg* is not provided, the current logger (as defined by parameter object `current-logger`) is used.

33.5 Logging syntax

(log-time *expr*) syntax
(log-time *expr descr*)
(log-time *expr descr tag*)
(log-time *expr descr tag lg*)

Log the time for executing expression *expr* into logger *lg*. *descr* is a description string and *tag* is a logging tag. If *lg* is not provided, the current logger (as defined by parameter object `current-logger`) is used.

(log-using *lg body* ...) syntax
Assigns *lg* as the current logger and executed expressions *body* ... in the context of this assignment.

(log-into-file *filepath body* ...) syntax
Creates a new file logger at file path *filepath*, assigns the new file logger to parameter object `current-logger` and executes the statements *body* ... in the context of this assignment.

(log-with-tag *tag body* ...) syntax
Creates a new logger which appends *tag* to the tags logged to `current-logger` and assigns the new logger to `current-logger`. *body* ... gets executed in the context of this assignment.

(log-from-severity *sev body* …) syntax

Modifies the current logger setting its lowest persisted severity to *sev* and executing *body* … in the context of this change. Once *body* … has been executed, the lowest persisted severity is set back to its original value.

(log-dropping-below-severity *sev body* …) syntax

Creates a new logger on top of `current-logger` which filters out all logging requests with a severity level below the severity level of *sev* and assigns the new logger to `current-logger` . *body* … gets executed in the context of this assignment.

34 LispKit Markdown

Library `(lispkit markdown)` provides an API for programmatically constructing Markdown documents, for parsing strings in Markdown format, as well as for mapping Markdown documents into corresponding HTML. The Markdown syntax supported by this library is based on the CommonMark Markdown specification.

34.1 Data Model

Markdown documents are represented using an abstract syntax that is implemented by three algebraic datatypes `block`, `list-item`, and `inline`, via `define-datatype` of library `(lispkit datatype)`.

34.1.1 Blocks

At the top-level, a Markdown document consist of a list of *blocks*. The following recursively defined datatype shows all the supported block types as variants of type `block`.

```
(define-datatype block markdown-block?
  (document blocks)
      where (markdown-blocks? blocks)
  (blockquote blocks)
      where (markdown-blocks? blocks)
  (list-items start tight items)
      where (and (opt fixnum? start) (markdown-list? items))
  (paragraph text)
      where (markdown-text? text)
  (heading level text)
      where (and (fixnum? level) (markdown-text? text))
  (indented-code lines)
      where (every? string? lines)
  (fenced-code lang lines)
      where (and (opt string? lang) (every? string? lines))
  (html-block lines)
      where (every? string? lines)
  (reference-def label dest title)
      where (and (string? label) (string? dest) (every? string? title))
  (table header alignments rows)
      where (and (every? markdown-text? header)
                 (every? symbol? alignments)
                 (every? (lambda (x) (every? markdown-text? x)) rows))
  (definition-list defs)
      where (every? (lambda (x)
                      (and (markdown-text? (car x))
                           (markdown-list? (cdr x)))) defs)
  (thematic-break))
```

`(document blocks)` represents a full Markdown document consisting of a list of blocks. `(blockquote blocks)` represents a blockquote block which itself has a list of sub-blocks. `(list-items start tight` `items)` defines either a bullet list or an ordered list. *start* is `#f` for bullet lists and defines the first item number for ordered lists. *tight* is a boolean which is `#f` if this is a loose list (with vertical spacing between the list items). *items* is a list of list items of type `list-item` as defined as follows:

```
(define-datatype list-item markdown-list-item?
  (bullet ch tight? blocks)
      where (and (char? ch) (markdown-blocks? blocks))
  (ordered num ch tight? blocks)
      where (and (fixnum? num) (char? ch) (markdown-blocks? blocks)))
```

The most frequent Markdown block type is a paragraph. `(paragraph text)` represents a single paragraph of text where *text* refers to a list of inline text fragments of type `inline` (see below). `(heading level text)` defines a heading block for a heading of a given level, where *level* is a number starting with 1 (up to 6). `(indented-code lines)` represents a code block consisting of a list of text lines each represented by a string. `(fenced-code lang lines)` is similar: it defines a code block with code expressed in the given language *lang*. `(html lines)` defines a HTML block consisting of the given lines of text. `(reference-def label dest title)` introduces a reference definition consisting of a given *label*, a destination URI *dest*, as well as a *title* string. `(table header alignments rows)` defines a table consisting of *headers*, a list of markdown text describing the header of each column, *alignments*, a list of symbols `l` (= left), `c` (= center), and `r` (= right), and *rows*, a list of lists of markdown text. `(definition-list defs)` represents a definition list where *defs* refers to a list of definitions. A definition has the form `(name def ...)` where *name* is markdown text defining a name, and *def* is a bullet item using `:` as bullet character. Finally, `(thematic-break)` introduces a thematic break block separating the previous and following blocks visually, often via a line.

34.1.2 Inline Text

Markdown text is represented as lists of inline text segments, each represented as an object of type `inline`. `inline` is defined as follows:

```
(define-datatype inline markdown-inline?
  (text str)
      where (string? str)
  (code str)
      where (string? str)
  (emph text)
      where (markdown-text? text)
  (strong text)
      where (markdown-text? text)
  (link text uri title)
      where (and (markdown-text? text) (string? uri) (string? title))
  (auto-link uri)
      where (string? uri)
  (email-auto-link email)
      where (string? uri)
  (image text uri title)
      where (and (markdown-text? text) (string? uri) (string? title))
  (html tag)
      where (string? tag)
  (line-break hard?))
```

`(text str)` refers to a text segment consisting of string *str*. `(code str)` refers to a code string *str* (often displayed as verbatim text). `(emph text)` represents emphasized *text* (often displayed as italics). `(strong text)` represents *text* in boldface. `(link text uri title)` represents a hyperlink with *text* linking to *uri* and *title* representing a title for the link. `(auto-link uri)` is a link where *uri* is both the text and the destination URI. `(email-auto-link email)` is a “mailto:” link to the given email address *email*. `(image text uri title)` inserts an image at *uri* with image description *text* and image link title *title*. `(html tag)` represents a single HTML tag of the form < *tag* >. Finally, `(line-break #f)` introduces a “soft line break”, whereas `(line-break #t)` inserts a “hard line break”.

34.2 Creating Markdown documents

Markdown documents can either be constructed programmatically via the datatypes introduced above, or a string representing a Markdown documents gets parsed into the internal abstract syntax representation via function `markdown`.

For instance, `(markdown "# My title\n\nThis is a paragraph.")` returns a markdown document consisting of two blocks: a *header block* for header “My title” and a *paragraph block* for the text “This is a paragraph”:

```
(markdown "# My title\n\nThis is a paragraph.")
⇒ #block:(document (#block:(heading 1 (#inline:(text "My title"))) #block:(paragraph
 ↪  (#inline:(text "This is a paragraph.")))))
```

The same document can be created programmatically in the following way:

```
(document
  (list
    (heading 1 (list (text "My title")))
    (paragraph (list (text "This is a paragraph.")))))
⇒ #block:(document (#block:(heading 1 (#inline:(text "My title"))) #block:(paragraph
 ↪  (#inline:(text "This is a paragraph.")))))
```

34.3 Processing Markdown documents

Since the abstract syntax of Markdown documents is represented via algebraic datatypes, pattern matching can be used to deconstruct the data. For instance, the following function returns all the top-level headers of a given Markdown document:

```
(import (lispkit datatype))  ; this is needed to import `match`

(define (top-headings doc)
  (match doc
    ((document blocks)
      (filter-map (lambda (block)
                    (match block
                      ((heading 1 text) (text->raw-string text))
                      (else #f)))
                  blocks))))
```

An example for how `top-headings` can be applied to this Markdown document:

```
# *header* 1
Paragraph.
# __header__ 2
## header 3
The end.
```

is shown here:

```
(top-headings
  (markdown "# *header* 1\nParagraph.\n# __header__ 2\n## header 3\nThe end."))
⇒ ("header 1" "header 2")
```

34.4 API

block-type-tag object

Symbol representing the markdown `block` type. The `type-for` procedure of library `(lispkit type)` returns this symbol for all block objects.

list-item-type-tag object

Symbol representing the markdown `list-item` type. The `type-for` procedure of library `(lispkit type)` returns this symbol for all list item objects.

inline-type-tag object

Symbol representing the markdown `inline` type. The `type-for` procedure of library `(lispkit type)` returns this symbol for all inline objects.

(markdown-blocks? *obj*) procedure

Returns `#t` if *obj* is a proper list of objects *o* for which `(markdown-block? o)` returns `#t`; otherwise the procedure returns `#f`.

(markdown-block? *obj*) procedure

Returns `#t` if *obj* is a mMrkdown block object, i.e. a variant of algebraic datatype `block`.

(markdown-block=? *lhs rhs*) procedure

Returns `#t` if Markdown blocks *lhs* and *rhs* are equals; otherwise it returns `#f`.

(markdown-list? *obj*) procedure

Returns `#t` if *obj* is a proper list of list items *i* for which `(markdown-list-item? i )` returns `#t`; otherwise the procedure returns `#f`.

(markdown-list-item? *obj*) procedure

Returns `#t` if *obj* is a Markdown list item, i.e. a variant of algebraic datatype `list-item`.

(markdown-list-item=? *lhs rhs*) procedure

Returns `#t` if Markdown list items *lhs* and *rhs* are equals; otherwise it returns `#f`.

(markdown-text? *obj*) procedure

Returns `#t` if *obj* is a proper list of objects *o* for which `(markdown-inline? o)` returns `#t`; otherwise the procedure returns `#f`.

(markdown-inline? *obj*) procedure

Returns `#t` if *obj* is an inline text object, i.e. a variant of algebraic datatype `inline`.

(markdown-inline=? *lhs rhs*) procedure

Returns `#t` if the two Markdown inline text objects *lhs* and *rhs* are equals; otherwise the procedure returns `#f`.

(markdown? *obj*) procedure

Returns `#t` if *obj* is a valid Markdown document, i.e. an instance of the `document` variant of datatype `block`; otherwise the procedure returns `#f`.

(markdown=? *lhs rhs*) procedure

Returns `#t` if Markdown documents *lhs* and *rhs* are equals; otherwise it returns `#f`.

(markdown *str*) procedure

Parses the text in Markdown format in *str* and returns a representation of the abstract syntax using the algebraic datatypes `block`, `list-item`, and `inline`.

(markdown->html *md*) procedure

Converts a Markdown document *md* into HTML, represented in form of a string. *md* needs to satisfy the *markdown?* predicate.

(blocks->html *bs*) procedure
(blocks->html *bs tight*)

Converts a Markdown block or list of blocks *bs* into HTML, represented in form of a string. *tight?* is a boolean and should be set to true if the conversion should consider tight typesetting (see CommonMark specification for details).

(text->html *txt*) procedure

Converts Markdown inline text or a list of inline text objects *txt* into HTML and returns the generated HTML in form of a string.

(markdown->html-doc *md*) procedure
(markdown->html-doc *md style*)
(markdown->html-doc *md style codestyle*)
(markdown->html-doc *md style codestyle cblockstyle*)
(markdown->html-doc *md style codestyle cblockstyle colors*)

Converts a Markdown document *md* into a styled HTML document, represented in form of a string. *md* needs to satisfy the *markdown?* predicate. *style* is a list with up to three elements: *(size font color)*. It specifies the default text style of the document. *size* is the point size of the font, *font* is a font name, and *color* is a HTML color specification (e.g. `"#FF6789"`). *codestyle* specifies the style of inline code in the same format. *colors* is a list of HTML color specifications for the following document elements in this order: the border color of code blocks, the color of blockquote “bars”, the color of H1, H2, H3 and H4 headers.

(text->string *text*) procedure

Converts given inline text *text* into a string representation which encodes markup in *text* using Markdown syntax. *text* needs to satisfy the *markdown-text?* predicate.

(text->raw-string *text*) procedure

Converts given inline text *text* into a string representation ignoring markup in *text*. *text* needs to satisfy the *markdown-text?* predicate.

35 LispKit Match

`(lispkit match)` ports Alex Shinn's portable hygienic pattern matcher library to LispKit and adapts it to match LispKit's features. For instance, `(lispkit match)` assumes all pairs are immutable. At this point, the library does not support matching against algebraic datatypes. Procedure `match` of library `(lispkit datatype)` needs to be used for this purpose.

35.1 Simple patterns

Patterns are written to look like the printed representation of the objects they match. The basic usage for matching an expression `expr` against a pattern `pat` via procedure `match` looks like this:

```
(match expr (pat body ...) ...)
```

Here, the result of `expr` is matched against each pattern in turn, and the corresponding body is evaluated for the first to succeed. Thus, a list of three elements matches a list of three elements.

```
(let ((ls (list 1 2 3)))
  (match ls ((1 2 3) #t)))  ⇒  #t
```

If no patterns match, an error is signaled. Identifiers will match anything, and make the corresponding binding available in the body.

```
(match (list 1 2 3) ((a b c) b))  ⇒  2
```

If the same identifier occurs multiple times, the first instance will match anything, but subsequent instances must match a value which is `equal?` to the first.

```
(match (list 1 2 1) ((a a b) 1) ((a b a) 2))  ⇒  2
```

The special identifier `_` matches anything, no matter how many times it is used, and does not bind the result in the body.

```
(match (list 1 2 1) ((_ _ b) 1) ((a b a) 2))  ⇒  1
```

To match a literal identifier (or list or any other literal), use `quote`.

```
(match 'a ('b 1) ('a 2))  ⇒  2
```

Analogous to its normal usage in scheme, `quasiquote` can be used to quote a mostly literally matching object with selected parts unquoted.

```
(match (list 1 2 3) (`(1 ,b ,c) (list b c)))  ⇒  (2 3)
```

Often you want to match any number of a repeated pattern. Inside a list pattern you can append `...` after an element to match zero or more of that pattern, similar to a regular expression *Kleene star*.

```
(match (list 1 2) ((1 2 3 ...) #t))        ⇒  #t
(match (list 1 2 3) ((1 2 3 ...) #t))      ⇒  #t
(match (list 1 2 3 3 3) ((1 2 3 ...) #t))  ⇒  #t
```

Pattern variables matched inside the repeated pattern are bound to a list of each matching instance in the body.

```
(match (list 1 2) ((a b c ...) c))        ⇒  ()
(match (list 1 2 3) ((a b c ...) c))      ⇒  (3)
(match (list 1 2 3 4 5) ((a b c ...) c))  ⇒  (3 4 5)
```

More than one `...` may not be used in the same list, since this would require exponential backtracking in the general case. However, `...` need not be the final element in the list, and may be succeeded by a fixed number of patterns.

```
(match (list 1 2 3 4) ((a b c ... d e) c))        ⇒  ()
(match (list 1 2 3 4 5) ((a b c ... d e) c))      ⇒  (3)
(match (list 1 2 3 4 5 6 7) ((a b c ... d e) c))  ⇒  (3 4 5)
```

`___` is provided as an alias for `...` when it is inconvenient to use the ellipsis (as in a syntax-rules template).

The `..1` syntax is exactly like the `...` except that it matches one or more repetitions like a `+` in regular expressions.

```
(match (list 1 2) ((a b c ..1) c))    ⇒  [error] no matching pattern
(match (list 1 2 3) ((a b c ..1) c))  ⇒  (3)
```

35.2 Composite patterns

The boolean operators `and`, `or` and `not` can be used to group and negate patterns analogously to their Scheme counterparts.

The `and` operator ensures that all subpatterns match. This operator is often used with the idiom `(and x pat)` to bind `x` to the entire value that matches `pat`, similar to “as-patterns” in ML and Haskell. Another common use is in conjunction with `not` patterns to match a general case with certain exceptions.

```
(match 1 ((and) #t))      ⇒  #t
(match 1 ((and x) x))     ⇒  1
(match 1 ((and x 1) x))   ⇒  1
```

The `or` operator ensures that at least one subpattern matches. If the same identifier occurs in different subpatterns, it is matched independently. All identifiers from all subpatterns are bound if the `or` operator matches, but the binding is only defined for identifiers from the subpattern which matched.

```
(match 1 ((or) #t) (else #f))  ⇒  #f
(match 1 ((or x) x))           ⇒  1
(match 1 ((or x 2) x))         ⇒  1
(match 1 ((or 1 x) x))         ⇒  [error] variable not yet initialized: x
```

The `not` operator succeeds if the given pattern does not match. None of the identifiers used are available in the body.

```
(match 1 ((not 2) #t))  ⇒  #t
```

The more general operator `?` can be used to provide a predicate. The usage is `(? predicate pat ...)` where `predicate` is a Scheme expression evaluating to a predicate called on the value to match, and any optional patterns after the predicate are then matched as in an `and` pattern.

```
(match 1 ((? odd? x) x))  ⇒  1
```

35.3 Advanced patterns

The field operator `=` is used to extract an arbitrary field and match against it. It is useful for more complex or conditional destructuring that can't be more directly expressed in the pattern syntax. The usage is `(= field pat)`, where `field` can be any expression, and should evaluate to a procedure of one argument which gets applied to the value to match to generate a new value to match against `pat`.

Thus the pattern `(and (= car x) (= cdr y))` is equivalent to `(x . y)`, except it will result in an immediate error if the value isn't a pair.

```
(match '(1 . 2) ((= car x) x))          ⇒  1
(match '(1 . 2)
  ((and (= car x) (= cdr y)) (+ x y)))  ⇒  3
(match 4 ((= square x) x))              ⇒  16
```

The record operator `$` is used as a concise way to match records. The usage is `($ rtd field ...)`, where `rtd` should be the *record type descriptor* specified as the first argument to `define-record-type`, and each `field` is a subpattern matched against the fields of the record in order. Not all fields must be present. For more information on *record type descriptors* see library `(lispkit record)`.

```
(define-record-type employee
  (make-employee name title)
  employee?
  (name get-name)
  (title get-title))
(match (make-employee "Bob" "Doctor")
  (($ employee n t) (list t n)))
⇒  ("Doctor" "Bob")
```

For records with more fields it can be helpful to match them by name rather than position. For this you can use the `@` operator, originally a Gauche extension:

```
(define-record-type employee
  (make-employee name title)
  employee?
  (name get-name)
```

```
  (title get-title))
(match (make-employee "Bob" "Doctor")
  ((@ employee (title t) (name n)) (list t n)))
⇒ ("Doctor" "Bob")
```

The `set!` and `get!` operators are used to bind an identifier to the setter and getter of a field, respectively. The setter is a procedure of one argument, which mutates the field to that argument. The getter is a procedure of no arguments which returns the current value of the field.

```
(let ((x (mcons 1 2)))
  (match x ((1 . (set! s)) (s 3) x)))  ⇒  #<pair 1 3>
(match '(1 . 2) ((1 . (get! g)) (g)))  ⇒  2
```

The new operator `***` can be used to search a tree for subpatterns. A pattern of the form `(x *** y)` represents the subpattern `y` located somewhere in a tree where the path from the current object to `y` can be seen as a list of the form `(x ...)` . `y` can immediately match the current object in which case the path is the empty list. In a sense, it is a two-dimensional version of the `...` pattern. As a common case the pattern `(_ *** y)` can be used to search for `y` anywhere in a tree, regardless of the path used.

```
(match '(a (a (a b))) ((x *** 'b) x))  ⇒  (a a a)
(match '(a (b) (c (d e) (f g)))
  ((x *** 'g) x))                      ⇒  (a c f)
```

35.4 Pattern grammar

```
pat = patvar                          ;; anything, and binds pattern var
    | _                               ;; anything
    | ()                              ;; the empty list
    | #t                              ;; #t
    | #f                              ;; #f
    | string                          ;; a string
    | number                          ;; a number
    | character                       ;; a character
    | 'sexp                           ;; an s-expression
    | 'symbol                         ;; a symbol (special case of s-expr)
    | (pat1 ... patN)                 ;; list of n elements
    | (pat1 ... patN . patN+1)        ;; list of n or more
    | (pat1 ... patN patN+1 ooo)      ;; list of n or more, each element
                                      ;;   of remainder must match patN+1
    | #(pat1 ... patN)                ;; vector of n elements
    | #(pat1 ... patN patN+1 ooo)     ;; vector of n or more, each element
                                      ;;   of remainder must match patN+1
    | ($ record-type pat1 ... patN)   ;; a record (patK matches in slot order)
    | (struct struct-type pat1 ... patN)     ;; ditto
    | (@ record-type (slot1 pat1) ...)       ;; a record (using slot names)
    | (object struct-type (slot1 pat1) ...) ;; ditto
    | (= proc pat)                    ;; apply proc, match the result to pat
    | (and pat ...)                   ;; if all of pats match
    | (or pat ...)                    ;; if any of pats match
    | (not pat ...)                   ;; if no pats match
    | (? predicate pat ...)           ;; if predicate true and all pats match
    | (set! patvar)                   ;; anything, and binds setter
    | (get! patvar)                   ;; anything, and binds getter
    | (pat1 *** pat2)                 ;; tree subpattern (*)
    | `qp                             ;; a quasi-pattern
```

```
patvar = a symbol except _, quote,
         $, struct, @, object, =,
         and, or, not, ?, set!, get!,
         quasiquote, ..., ___, ..1,
         ..=, ..*.

ooo = ...                             ;; zero or more
    | ___                             ;; zero or more
    | ..1                             ;; one or more
    | ..= k                           ;; exactly k where k is an integer. (*)
                                      ;;   Example: ..= 1, ..= 2 ...
    | ..* k j                         ;; between k and j, where k and j are (*)
                                      ;;   integers. Example: ..* 3 4, match 3
                                      ;;   or 4 of a pattern ..* 1 5 match from
                                      ;;   1 to 5 of a pattern

qp  = ()                              ;; the empty list
    | #t                              ;; #t
    | #f                              ;; #f
    | string                          ;; a string
    | number                          ;; a number
    | character                       ;; a character
    | identifier                      ;; a symbol
    | (qp_1 ... qp_n)                 ;; list of n elements
    | (qp_1 ... qp_n . qp_{n+1})      ;; list of n or more
    | (qp_1 ... qp_n qp_n+1 ooo)      ;; list of n or more, each element
                                      ;;   of remainder must match qp_n+1
    | #(qp_1 ... qp_n)                ;; vector of n elements
    | #(qp_1 ... qp_n qp_n+1 ooo)     ;; vector of n or more, each element
                                      ;;   of remainder must match qp_n+1
    | ,pat                            ;; a pattern
    | ,@pat                           ;; a pattern
```

35.5 Matching API

(match *expr* (*pat* . *body*) ...) procedure
(match *expr* (*pat* (=> *failure*) . *body*) ...)

The result of *expr* is matched against each pattern *pat* in turn until the first pattern matches. When a match is found, the corresponding *body* statements are evaluated in order, and the result of the last expression is returned as the result of the entire `match` evaluation. If a failure occurs, then it is bound to a procedure of no arguments which continues processing at the next pattern. If no pattern matches, an error is signaled.

(match-lambda (*pat body* ...) ...) procedure
This is a shortcut for `lambda` in combination with `match`. `match-lambda` returns a procedure of one argument, and matches that argument against each clause.

```
(lambda (expr) (match expr (pat body ...) ...))
```

(match-lambda* (*pat body* ...) ...) procedure
`match-lambda*` is similar to `match-lambda`. It returns a procedure of any number of arguments, and matches the argument list against each clause.

```
(lambda expr (match expr (pat body ...) ...))
```

(match-let ((*var value*) ...) *body* ...) procedure
(match-let *loop* ((*var value*) ...) *body* ...)

`match-let` matches each variable *var* to the corresponding expression, and evaluates the body with all match variables in scope. It raises an error if any of the expressions fail to match. This syntax is analogous to named `let` and can also be used for recursive functions which match on their arguments as in `match-lambda*`.

(match-let* ((*var value*) ...) *body* ...) procedure

Similar to `match-let`, but analogously to `let*`, `match-let*` matches and binds the variables in sequence, with preceding match variables in scope.

(match-letrec ((*var value*) ...) *body* ...) procedure

Similar to `match-let`, but analogously to `letrec`, `match-letrec` matches and binds the variables with all match variables in scope.

This documentation was derived from code and documentation written by Andrew K. Wright, Robert Cartwright, Alex Shinn, Panicz Maciej Godek, Felix Thibault, Shiro Kawai and Ludovic Cortès.

36 LispKit Math

Library `(lispkit math)` defines functions on numbers. Numbers are arranged into a tower of subtypes in which each level is a subset of the level above it:

- number
- complex number
- real number
- rational number
- integer

For example, 3 is an integer. Therefore 3 is also a rational, a real, and a complex number. These types are defined by the predicates `number?` , `complex?` , `real?` , `rational?` , and `integer?` .

There is no simple relationship between a number's type and its representation inside a computer. Scheme's numerical operations treat numbers as abstract data, as independent of their representation as possible.

36.1 Numerical constants

pi [object]
The constant pi.

e [object]
Euler's number, i.e. the base of the natural logarithm.

fx-width [object]
Number of bits used to represent fixnum numbers (typically 64).

fx-greatest [object]
Greatest fixnum value (typically 9223372036854775807).

fx-least [object]
Smallest fixnum value (typically -9223372036854775808).

fl-epsilon [object]
Bound to the appropriate machine epsilon for the hardware representation of flonum numbers, i.e. the positive difference between 1.0 and the next greater representable number.

fl-greatest [object]
This value compares greater than or equal to all finite floating-point numbers, but less than infinity.

fl-least [object]
This value compares less than or equal to all positive floating-point numbers, but greater than zero.

36.2 Predicates

(number? ***obj*****)** [procedure]
(complex? ***obj*****)**

(real? ***obj*****)**
(rational? ***obj*****)**
(integer? ***obj*****)**

These numerical type predicates can be applied to any kind of argument, including non-numbers. They return `#t` if the object is of the named type, and otherwise they return `#f`. In general, if a type predicate is true of a number then all higher type predicates are also true of that number. Consequently, if a type predicate is false of a number, then all lower type predicates are also false of that number.

If *z* is a complex number, then `(real? z)` is true if and only if `(zero? (imag-part z))` is true. If *x* is an inexact real number, then `(integer? x)` is true if and only if `(= x (round x))`.

The numbers `+inf.0`, `-inf.0`, and `+nan.0` are real but not rational.

```
(complex? 3+4i)      ⇒  #t
(complex? 3)         ⇒  #t
(real? 3)            ⇒  #t
(real? -2.5+0i)      ⇒  #t
(real? -2.5+0.0i)    ⇒  #f
(real? #e1e10)       ⇒  #t
(real? +inf.0)       ⇒  #t
(real? +nan.0)       ⇒  #t
(rational? -inf.0)   ⇒  #f
(rational? 3.5)      ⇒  #t
(rational? 6/10)     ⇒  #t
(rational? 6/3)      ⇒  #t
(integer? 3+0i)      ⇒  #t
(integer? 3.0)       ⇒  #t
(integer? 8/4)       ⇒  #t
```

(fixnum? ***obj*****)** procedure
Returns `#t` if object *obj* is a fixnum; otherwise returns `#f`. A fixnum is an exact integer that is small enough to fit in a machine word. LispKit fixnums are 64-bit words. Fixnums are signed and encoded using 2's complement.

(ratnum? ***obj*****)** procedure
Returns `#t` if object `obj` is a fractional number, i.e. a rational number which isn't an integer.

(bignum? ***obj*****)** procedure
Returns `#t` if object *obj* is a large integer number, i.e. an integer which isn't a fixnum.

(flonum? ***obj*****)** procedure
Returns `#t` if object *obj* is a floating-point number.

(cflonum? ***obj*****)** procedure
Returns `#t` if object *obj* is a complex floating-point number.

(exact? ***obj*****)** procedure
(inexact? ***obj*****)**

These numerical predicates provide tests for the exactness of a quantity. For any Scheme number, precisely one of `exact?` and `inexact?` is true.

```
(exact? 3.0)     ⇒  #f
(exact? #e3.0)   ⇒  #t
(inexact? 3.)    ⇒  #t
```

(exact-integer? ***obj*****)** procedure
Returns `#t` if *obj* is both exact and an integer; otherwise returns `#f`.

```
(exact-integer? 32)   ⇒  #t
(exact-integer? 32.0) ⇒  #f
(exact-integer? 32/5) ⇒  #f
```

(finite? *obj*) procedure

The `finite?` procedure returns `#t` on all real numbers except `+inf.0`, `-inf.0`, and `+nan.0`, and on complex numbers if their real and imaginary parts are both finite. Otherwise it returns `#f`.

```
(finite? 3)          ⇒  #t
(finite? +inf.0)     ⇒  #f
(finite? 3.0+inf.0i) ⇒  #f
```

(infinite? *obj*) procedure

The `infinite?` procedure returns `#t` on the real numbers `+inf.0` and `-inf.0`, and on complex numbers if their real or imaginary parts or both are infinite. Otherwise it returns `#f`.

```
(infinite? 3)          ⇒  #f
(infinite? +inf.0)     ⇒  #t
(infinite? +nan.0)     ⇒  #f
(infinite? 3.0+inf.0i) ⇒  #t
```

(nan? *obj*) procedure

The `nan?` procedure returns `#t` on `+nan.0`, and on complex numbers if their real or imaginary parts or both are `+nan.0`. Otherwise it returns `#f`.

```
(nan? +nan.0)      ⇒  #t
(nan? 32)          ⇒  #f
(nan? +nan.0+5.0i) ⇒  #t
(nan? 1+2i)        ⇒  #f
```

(positive? *x*) procedure

Returns `#t` if number *x* is positive, i.e. `x > 0`.

(negative? *x*) procedure

Returns `#t` if number *x* is negative, i.e. `x < 0`.

(zero? *z*) procedure

Returns `#t` if number *z* is zero, i.e. `z = 0`.

(even? *n*) procedure

Returns `#t` if the integer number *n* is even.

(odd? *n*) procedure

Returns `#t` if the integer number *n* is odd.

36.3 Exactness and rounding

Scheme distinguishes between numbers that are represented exactly and those that might not be. This distinction is orthogonal to the dimension of type. A number is exact if it was written as an exact constant or was derived from exact numbers using only exact operations. A number is inexact if it was written as an inexact constant, if it was derived using inexact ingredients, or if it was derived using inexact operations.

Rational operations such as `+` should always produce exact results when given exact arguments. If the operation is unable to produce an exact result, then it either reports the violation of an implementation restriction or it silently coerces its result to an inexact value.

(exact *z*) `procedure`
(inexact *z*)

The procedure `inexact` returns an inexact representation of *z*. The value returned is the inexact number that is numerically closest to the argument. For inexact arguments, the result is the same as the argument. For exact complex numbers, the result is a complex number whose real and imaginary parts are the result of applying inexact to the real and imaginary parts of the argument, respectively. If an exact argument has no reasonably close inexact equiv- alent (in the sense of `=`), then a violation of an implementation restriction may be reported.

The procedure `exact` returns an exact representation of *z*. The value returned is the exact number that is numerically closest to the argument. For exact arguments, the result is the same as the argument. For inexact non-integral real arguments, the function may return a rational approximation. For inexact complex arguments, the result is a complex number whose real and imaginary parts are the result of applying exact to the real and imaginary parts of the argument, respectively. If an inexact argument has no reasonably close exact equivalent, (in the sense of `=`), then a violation of an implementation restriction may be reported.

These procedures implement the natural one-to-one correspondence between `exact` and `inexact` integers throughout an implementation-dependent range.

(approximate *x delta*) `procedure`
Procedure `approximate` approximates floating-point number *x* returning a rational number which differs at most *delta* from *x*.

(rationalize *x y*) `procedure`
The `rationalize` procedure returns the simplest rational number differing from *x* by no more than *y*. A rational number *r1* is simpler than another rational number *r2* if *r1 = p1/q1* and *r2 = p2/q2* (in lowest terms) and |p1| ≤ |p2| and |q1| ≤ |q2|. Thus `3/5` is simpler than `4/7`. Although not all rationals are comparable in this ordering (consider `2/7` and `3/5`), any interval contains a rational number that is simpler than every other rational number in that interval (the simpler `2/5` lies between `2/7` and `3/5`). Note that `0 = 0/1` is the simplest rational of all.

```
(rationalize (exact .3) 1/10)  ⇒  1/3
```

(floor *x*) `procedure`
(ceiling *x*)
(truncate *x*)
(round *x*)

These procedures return integers. `floor` returns the largest integer not larger than *x*. `ceiling` returns the smallest integer not smaller than *x*. `truncate` returns the integer closest to *x* whose absolute value is not larger than the absolute value of *x*. `round` returns the closest integer to *x*, rounding to even when *x* is halfway between two integers.

If the argument to one of these procedures is inexact, then the result will also be inexact. If an exact value is needed, the result can be passed to the `exact` procedure. If the argument is *infinite* or a *NaN*, then it is returned.

```
(floor -4.3)     ⇒ -5.0
(ceiling -4.3)   ⇒ -4.0
(truncate -4.3)  ⇒ -4.0
(round -4.3)     ⇒ -4.0
(floor 3.5)      ⇒ 3.0
(ceiling 3.5)    ⇒ 4.0
(truncate 3.5)   ⇒ 3.0
(round 3.5)      ⇒ 4.0 ; inexact
```

```
(round 7/2)      ⇒ 4   ; exact
(round 7)        ⇒ 7
```

36.4 Operations

(+ *z* ...)
(* *z* ...)

procedure

These procedures return the sum or product of their arguments.

```
(+ 34)    ⇒  7
(+ 3)     ⇒  3
(+)       ⇒  0
(* 4)     ⇒  4
(*)       ⇒  1
```

(- *z*)
(- *z1 z2* ...)
(/ *z*)
(/ *z1 z2* ...)

procedure

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

It is an error if any argument of `/` other than the first is an exact zero. If the first argument is an exact zero, the implementation may return an exact zero unless one of the other arguments is a NaN.

```
(- 3 4)      ⇒  -1
(- 3 4 5)    ⇒  -6
(- 3)        ⇒  -3
(/ 3 4 5)    ⇒  3/20
(/ 3)        ⇒  1/3
```

(= *x* ...)
(< *x* ...)
()> *x* ...)
(<= *x* ...)
(>= *x* ...)

procedure

These procedures return `#t` if their arguments are (respectively): equal, monotonically increasing, monotonically decreasing, monotonically non-decreasing, or monotonically non-increasing, and `#f` otherwise. If any of the arguments are `+nan.0`, all the predicates return `#f`. They do not distinguish between inexact zero and inexact negative zero.

(max *x1 x2* ...)
(min *x1 x2* ...)

procedure

These procedures return the maximum or minimum of their arguments.

If any argument is inexact, then the result will also be inexact (unless the procedure can prove that the inaccuracy is not large enough to affect the result, which is possible only in unusual implementations). If `min` or `max` is used to compare numbers of mixed exactness, and the numerical value of the result cannot be represented as an inexact number without loss of accuracy, then the procedure reports an implementation restriction.

(abs *x*) procedure

The `abs` procedure returns the absolute value of its argument *x*.

(square *z*) procedure

Returns the square of *z*. This is equivalent to *(* z z)*.

```
(square 42)   ⇒ 1764
(square 2.0)  ⇒ 4.0
```

(sqrt *z*) procedure

Returns the principal square root of *z*. The result will have either a positive real part, or a zero real part and a non-negative imaginary part.

```
(sqrt 9)   ⇒ 3
(sqrt -1)  ⇒ +i
```

(exact-integer-sqrt *k*) procedure

Returns two non-negative exact integers *s* and *r* where $k = s^2+r$ and $k < (s+1)^2$.

(expt *z1 z2*) procedure

Returns *z1* raised to the power *z2*. For non-zero *z1*, this is $z1^{z2} = e^{(z2 \log z1)}$. The value of 0^z is `1` if `(zero? z)`, 0 if `(real-part z)` is positive, and an error otherwise. Similarly for 0.0z, with inexact results.

(exp *z*) procedure
(log *z*)
(log *z1 z2*)
(sin *z*)
(cos *z*)
(tan *z*)
(asin *z*)
(acos *z*)
(atan *z*)
(atan *y x*)

These procedures compute the usual transcendental functions. The `log` procedure computes the natural logarithm of *z* (not the base-ten logarithm) if a single argument is given, or the base-*z2* logarithm of *z1* if two arguments are given. The `asin`, `acos`, and `atan` procedures compute `arc-sine`, `arc-cosine`, and `arc-tangent`, respectively. The two-argument variant of `atan` computes `(angle (make-rectangular x y))`.

36.5 Division and remainder

(gcd *n* ...) procedure
(lcm *n* ...)

These procedures return the greatest common divisor (`gcd`) or least common multiple (`lcm`) of their arguments. The result is always non-negative.

```
(gcd 32 -36)    ⇒ 4
(gcd)           ⇒ 0
(lcm 32 -36)    ⇒ 288
(lcm 32.0 -36)  ⇒ 288.0  ; inexact
(lcm)           ⇒ 1
```

(truncate/ *n1 n2.*) procedure
(truncate-quotient *n1 n2*)
(truncate-remainder *n1 n2*)

These procedures implement number-theoretic integer division. It is an error if `n2` is zero. `truncate/` returns two integers; the other two procedures return an integer. All the procedures compute a quotient `nq` and remainder `nr` such that `n1 = n2 * nq + nr`. The three procedures are defined as follows:

```
(truncate/ n1 n2)           ⇒ nq nr
(truncate-quotient n1 n2)  ⇒ nq
(truncate-remainder n1 n2) ⇒ nr
```

The remainder `nr` is determined by the choice of integer `nq`: `nr = n1 - n2 * nq` where `nq = truncate(n1/n2)`.

For any of the operators, and for integers `n1` and `n2` with `n2` not equal to 0:

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

provided all numbers involved in that computation are exact.

```
(truncate/ 5 2)      ⇒ 2 1
(truncate/ -5 2)     ⇒ -2 -1
(truncate/ 5 -2)     ⇒ -2 1
(truncate/ -5 -2)    ⇒ 2 -1
(truncate/ -5.0 -2)  ⇒ 2.0 -1.0
```

(floor/ *n1 n2*) procedure
(floor-quotient *n1 n2*)
(floor-remainder *n1 n2*)

These procedures implement number-theoretic integer division. It is an error if `n2` is zero. `floor/` returns two integers; the other two procedures return an integer. All the procedures compute a quotient `nq` and remainder `nr` such that `n1 = n2 * nq + nr`. The three procedures are defined as follows:

```
(floor/ n1 n2)           ⇒ nq nr
(floor-quotient n1 n2)  ⇒ nq
(floor-remainder n1 n2) ⇒ nr
```

The remainder `nr` is determined by the choice of integer `nq`: `nr = n1 - n2 * nq` where `nq = floor(n1/n2)`.

For any of the operators, and for integers `n1` and `n2` with `n2` not equal to 0:

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

provided all numbers involved in that computation are exact.

```
(floor/ 5 2)    ⇒  2 1
(floor/ -5 2)   ⇒  -3 1
(floor/ 5 -2)   ⇒  -3 -1
(floor/ -5 -2)  ⇒  2 -1
```

(quotient *n1 n2*) procedure
(remainder *n1 n2*)
(modulo *n1 n2*)

The `quotient` and `remainder` procedures are equivalent to `truncate-quotient` and `truncate-remainder`, respectively, and `modulo` is equivalent to `floor-remainder`. These procedures are provided for backward compatibility with earlier versions of the Scheme language specification.

36.6 Fractional numbers

(numerator *q*) procedure
(denominator *q*)

These procedures return the numerator or denominator of their rational number `q`. The result is computed as if the argument was represented as a fraction in lowest terms. The denominator is always positive. The denominator of 0 is defined to be 1.

```
(numerator (/ 6 4))              ⇒  3
(denominator (/ 6 4))            ⇒  2
(denominator (inexact (/ 6 4)))  ⇒  2.0
```

36.7 Complex numbers

(make-rectangular *x1 x2*) procedure
Returns the complex number *x1 + x2 * i*. Since in LispKit, all complex numbers are inexact, `make-rectangular` returns an inexact complex number for all *x1* and *x2*.

(make-polar *x1 x2*) procedure
Returns a complex number *z* such that *z = x1 * e ^ (x2 * i)*, i.e. *x1* is the magnitude of the complex number. The `make-polar` procedure may return an inexact complex number even if its arguments are exact.

(real-part *z*) procedure
Returns the real part of the given complex number *z*.

(imag-part *z*) procedure
Returns the imaginary part of the given complex number *z*.

(magnitude *z*) procedure
Returns the magnitude of the given complex number *z*. Assuming *z = x1 * e ^ (x2 * i)*, `magnitude` returns *x1*. The `magnitude` procedure is the same as `abs` for a real argument.

(angle *z*) procedure
Returns the `angle` of the given complex number *z*. The angle is a floating-point number between `-pi` and `pi`.

36.8 Random numbers

(random) procedure
(random *max*)
(random *min max*)

If called without any arguments, `random` returns a random floating-point number between 0.0 (inclusive) and 1.0 (exclusive). If *max* is provided and is an exact integer, `random` returns a random exact integer between 0 (inclusive) and *max* (exclusive). If *max* is inexact, the random number returned by `random` is a floating-point number between 0.0 (inclusive) and *max* (exclusive). If *min* is provided, it is used instead of zero as the included lower-bound of the random number range. If one of *min* and *max* are inexact, the result is inexact. *max* needs to be greater than *min*.

```
(random)           ⇒ 0.17198431800336633
(random 10)        ⇒ 9
(random 10.0)      ⇒ 7.446150392968266
(random 0.1)       ⇒ 0.06781020202176374
(random 100 110)   ⇒ 106
(random 100 109.9) ⇒ 108.30564866186835
```

36.9 String representation

(number->string *z*) procedure
(number->string *z radix*)
(number->string *z radix len*)
(number->string *z radix len prec*)
(number->string *z radix len prec noexp*)

It is an error if *radix* is not one of 2, 8, 10, or 16. The procedure `number->string` takes a number *z* and a *radix* and returns as a string an external representation of the given number in the given radix such that

```
(let ((number number)
      (radix radix))
  (eqv? number (string->number
                 (number->string number radix)
                 radix)))
```

is true. It is an error if no possible result makes this expression true. If omitted, radix defaults to 10.

If *z* is inexact, the radix is 10, and the above expression can be satisfied by a result that contains a decimal point, then the result contains a decimal point and is expressed using the minimum number of digits (exclusive of exponent and trailing zeroes) needed to make the above expression true. Otherwise, the format of the result is unspecified. The result returned by `number->string` never contains an explicit radix prefix.

The error case can occur only when *z* is not a complex number or is a complex number with a non-rational real or imaginary part. If *z* is an inexact number and the radix is 10, then the above expression is normally satisfied by a result containing a decimal point. The unspecified case allows for infinities, NaNs, and unusual representations.

The string representation can be customized via parameters *len*, *prec*, and *noexp*. The absolute value of fixnum *len* determines the length of the string representation in characters. If *len* is negative, then the number is left-aligned; for positive *len*, it is right-aligned; if *len* is zero, no padding is done. *prec* determines the precision of flonum and complex values (i.e. the number of significant digits; default is 16). *noexp* is a boolean for disabling the exponential notation (if *noexp* is set to `#t`).

```
(number->string pi 10 5 5)   ⇒ "3.1416"
(number->string pi 10 9 5)   ⇒ "   3.1416"
(number->string pi 10 -9 5)  ⇒ "3.1416   "
```

(string->number *str*) procedure
(string->number *str radix*)

Returns a number of the maximally precise representation expressed by the given string *str*. It is an error if *radix* is not 2, 8, 10, or 16. If supplied, *radix* is a default radix that will be overridden if an explicit radix prefix is present in string (e.g. `"#o177"`). If *radix* is not supplied, then the default radix is 10. If string *str* is not a syntactically valid notation for a number, or would result in a number that cannot be represented, then `string->number` returns `#f`. An error is never signaled due to the content of string.

```
(string->number "100")     ⇒  100
(string->number "100" 16)  ⇒  256
(string->number "1e2")     ⇒  100.0
```

36.10 Bitwise operations

The following bitwise functions operate on integers including fixnums and bignums.

(bitwise-not *n*) procedure
Returns the bitwise complement of *n*; i.e. all 1 bits are changed to 0 bits and all 0 bits to 1 bits.

(bitwise-and *n* ...) procedure
Returns the *bitwise and* of the given integer arguments *n* ....

(bitwise-ior *n* ...) procedure
Returns the *bitwise inclusive or* of the given integer arguments *n* ....

(bitwise-xor *n* ...) procedure
Returns the *bitwise exclusive or (xor)* of the given integer arguments *n* ....

(bitwise-if *mask n m*) procedure
Merge the integers *n* and *m*, via integer *mask* determining from which integer to take each bit. That is, if the *k*-th bit of *mask* is 0, then the *k*-th bit of the result is the *k*-th bit of *n*, otherwise the *k*-th bit of *m*. `bitwise-if` is defined in the following way:

```
(define (bitwise-if mask n m)
  (bitwise-ior (bitwise-and mask n) (bitwise-and (bitwise-not mask) m)))
```

(bit-count *n*) procedure
Returns the population count of 1's if $n >= 0$, or 0's, if $n < 0$. The result is always non-negative. The R6RS analogue `bitwise-bit-count` procedure is incompatible as it applies `bitwise-not` to the population count before returning it if *n* is negative.

```
(bit-count 0)    ⇒  0
(bit-count -1)   ⇒  0
(bit-count 7)    ⇒  3
(bit-count  13)  ⇒  3
(bit-count -13)  ⇒  2
(bit-count  30)  ⇒  4
```

```
(bit-count -30)  ⇒  4
(bit-count (expt 2 100))          ⇒  1
(bit-count (- (expt 2 100)))      ⇒  100
(bit-count (- (+ 1 (expt 2 100)))) ⇒  1
```

(integer-length *n*) procedure

Returns the number of bits needed to represent *n*, i.e.

```
(ceiling (/ (log (if (negative? integer)
                     (- integer)
                     (+ 1 integer)))
                (log 2)))
```

The result is always non-negative. For non-negative *n*, this is the number of bits needed to represent *n* in an unsigned binary representation. For all *n*, `(+ 1 (integer-length i))` is the number of bits needed to represent *n* in a signed two's-complement representation.

(first-bit-set *n*) procedure

Returns the index of the least significant 1 bit in the two's complement representation of *n*. If *n* is 0, then −1 is returned.

```
(first-bit-set 0)   ⇒  -1
(first-bit-set 1)   ⇒  0
(first-bit-set -4)  ⇒  2
```

(bit-set? *n k*) procedure

k must be non-negative. The `bit-set?` procedure returns `#t` if the *k*-th bit is 1 in the two's complement representation of *n*, and `#f` otherwise. This is the result of the following computation:

```
(not (zero? (bitwise-and (bitwise-arithmetic-shift-left 1 k) n)))
```

(copy-bit *n k b*) procedure

k must be non-negative, and *b* must be either 0 or 1. The `copy-bit` procedure returns the result of replacing the *k*-th bit of *n* by the *k*-th bit of *b*, which is the result of the following computation:

```
(bitwise-if (bitwise-arithmetic-shift-left 1 k)
            (bitwise-arithmetic-shift-left b k)
            n)
```

(arithmetic-shift *n count*) procedure

If *count* > *0*, shifts integer *n* left by *count* bits; otherwise, shifts fixnum *n* right by *count* bits. In general, this procedure returns the result of the following computation: `(floor (* n (expt 2 count)))`.

```
(arithmetic-shift -6 -1)  ⇒  -3
(arithmetic-shift -5 -1)  ⇒  -3
(arithmetic-shift -4 -1)  ⇒  -2
(arithmetic-shift -3 -1)  ⇒  -2
(arithmetic-shift -2 -1)  ⇒  -1
(arithmetic-shift -1 -1)  ⇒  -1
```

(arithmetic-shift-left *n count*) procedure

Returns the result of arithmetically shifting *n* to the left by *count* bits. *count* must be non-negative. The `arithmetic-shift-left` procedure behaves the same as `arithmetic-shift`.

(arithmetic-shift-right *n count*) procedure

Returns the result of arithmetically shifting *n* to the right by *count* bits. *count* must be non-negative. `(arithmetic-shift-right n m)` behaves the same as `(arithmetic-shift n (fx- m))`.

36.11 Fixnum operations

LispKit supports arbitrarily large exact integers. Internally, it has two different representations, one for smaller integers and one for the rest. These are colloquially known as *fixnums* and *bignums* respectively. In LispKit, a *fixnum* is represented as a 64 bit signed integer which is encoded using two-complement.

Fixnum operations perform integer arithmetic on their fixnum arguments. If any argument is not a fixnum, or if the mathematical result is not representable as a fixnum, it is an error. In particular, this means that fixnum operations may return a mathematically incorrect fixnum in these situations without raising an error.

(integer->fixnum *n*) procedure
`integer->fixnum` coerces a given integer *n* into a fixnum. If *n* is a fixnum already, *n* is returned by `integer->fixnum`. If *n* is a bignum, then the first word of the bignum is returned as the result of `integer->fixnum`.

(fx+ *n m* ...) procedure
(fx- *n* ...)
(fx* *n m* ...)
(fx/ *n m* ...)

These procedures return the sum, the difference, the product and the quotient of their fixnum arguments *n m* .... These procedures may overflow without reporting an error. `(fx- n)` is negating `n`.

(fx= *n m o* ...) procedure
(fx< *n m o* ...)
(fx> *n m o* ...)
(fx<= *n m o* ...)
(fx>= *n m o* ...)

These procedures implement the comparison predicates for fixnums. `fx=` returns `#t` if all provided fixnums are equal. `fx<` returns `#t` if all provided fixnums are strictly monotonically increasing. `fx>` returns `#t` if all provided fixnums are strictly monotonically decreasing. `fx<=` returns `#t` if all provided fixnums are monotonically increasing. `fx>=` returns `#t` if all provided fixnums are monotonically decreasing.

(fx1+ *n*) procedure
Increments the fixnum *n* by one and returns the value. This procedure may overflow without raising an error.

(fx1- *n*) procedure
Decrements the fixnum *n* by one and returns the value. This procedure may overflow without raising an error.

(fxzero? *n*) procedure
Returns `#t` if fixnum *n* equals to 0.

(fxpositive? *n*) procedure
Returns `#t` if fixnum *n* is positive, i.e. $n > 0$.

(fxnegative? *n*) procedure
Returns `#t` if fixnum *n* is negative, i.e. $n < 0$.

(fxabs *n*) procedure
Returns the absolute value of its fixnum argument *n*.

(fxremainder *n m*) procedure
This procedure returns a value *r* such that the following equation holds: $n = m * q + r$ where *q* is the

largest number of multiples of *m* that will fit inside *n*. The sign of *m* gets ignored. This means that `(fxremainder n m)` and `(fxremainder n (- m))` always return the same answer.

```
(fxremainder 13 5)   ⇒  3
(fxremainder 13 -5)  ⇒  3
(fxremainder -13 5)  ⇒  -3
(fxremainder -13 -5) ⇒  -3
```

(fxmodulo *n m*) procedure

This procedure computes a remainder similar to `(fxremainder n m)`, but when `(fxremainder n m)` has a different sign than *m*, `(fxmodulo n m)` returns `(+ (fxremainder n m) m)` instead.

```
(fxmodulo 13 5)   ⇒  3
(fxmodulo 13 -5)  ⇒  -2
(fxmodulo -13 5)  ⇒  2
(fxmodulo -13 -5) ⇒  -3
```

(fxsqrt *n*) procedure

Approximates the square root *s* of fixnum *n* such that *s* is the biggest fixnum for which $s \times s \leq n$.

(fxnot *n*) procedure

Returns the *bitwise-logical inverse* for fixnum *n*.

```
(fxnot 0)    ⇒  -1
(fxnot -1)   ⇒  0
(fxnot 1)    ⇒  -2
(fxnot -34)  ⇒  33
```

(fxand *n m*) procedure

Returns the *bitwise-logical and* for *n* and *m*.

```
(fxand #x43 #x0f)  ⇒  3
(fxand #x43 #xf0)  ⇒  64
```

(fxior *n m*) procedure

Returns the *bitwise-logical inclusive or* for *n* and *m*.

(fxxor *n m*) procedure

Returns the *bitwise-logical exclusive or (xor)* for *n* and *m*.

(fxif *mask n m*) procedure

Merges the bit sequences *n* and *m*, with bit sequence *mask* determining from which sequence to take each bit. That is, if the *k*-th bit of *mask* is 1, then the *k*-th bit of the result is the *k*-th bit of *n*, otherwise it's the *k*-th bit of *m*.

```
(fxif 3 1 8)  ⇒  9
(fxif 3 8 1)  ⇒  0
(fxif 1 1 2)  ⇒  3
(fxif #b00111100 #b11110000 #b00001111)  ⇒  #b00110011 = 51
```

`fxif` can be implemented via `(fxior (fxand mask n) (fxand (fxnot mask) m))`.

(fxarithmetic-shift *n count*) procedure

If *count* > 0, shifts fixnum *n* left by *count* bits; otherwise, shifts fixnum *n* right by *count* bits. The absolute value of *count* must be less than `(fixnum-width)`.

```
(fxarithmetic-shift 8 2)    ⇒  32
(fxarithmetic-shift 4 0)    ⇒  4
(fxarithmetic-shift 8 -1)   ⇒  4
(fxarithmetic-shift -1 62)  ⇒  -4611686018427387904
```

`fxarithmetic-shift` can be implemented via `(floor (fx* n (expt 2 m)))` if this computes to a fixnum.

(fxarithmetic-shift-left *n count*) procedure
(fxlshift *n count*)

Returns the result of arithmetically shifting *n* to the left by *count* bits. *count* must be non-negative, and less than `fx-width`. `fxarithmetic-shift-left` behaves the same as `fxarithmetic-shift`.

(fxarithmetic-shift-right *n count*) procedure
(fxrshift *n count*)

Returns the result of arithmetically shifting *n* to the right by *count* bits. *count* must be non-negative, and less than `fx-width`. `(fxarithmetic-shift-right n m)` behaves the same as `(fxarithmetic-shift n (fx- m))`.

(fxlogical-shift-right *n count*) procedure
(fxlrshift *n count*)

Returns the result of logically shifting *n* to the right by *count* bits. *count* must be non-negative, and less than `fx-width`.

```
(fxlogical-shift 8 2)    ⇒  2
(fxlogical-shift 4 0)    ⇒  4
(fxlogical-shift -1 62)  ⇒  3
```

(fxbit-count *n*) procedure
If *n* is non-negative, this procedure returns the number of 1 bits in the two's complement representation of *n*. Otherwise, it returns the following result: `(fxnot (fxbit-count (fxnot n)))`.

(fxlength *n*) procedure
Returns the number of bits needed to represent *n* if it is positive, and the number of bits needed to represent `(fxnot n)` if it is negative, which is the fixnum result of the following computation:

```
(do ((res 0 (fx1+ res))
     (bits (if (fxnegative? n) (fxnot n) n)
           (fxarithmetic-shift-right bits 1)))
    ((fxzero? bits) res))
```

(fxfirst-bit-set *obj*) procedure
Returns the index of the least significant 1 bit in the two's complement representation of *n*. If *n* is 0, then −1 is returned.

```
(fxfirst-bit-set 0)   ⇒  -1
(fxfirst-bit-set 1)   ⇒  0
(fxfirst-bit-set -4)  ⇒  2
```

(fxbit-set? *n k*) procedure
k must be non-negative and less than `fx-width`. The `fxbit-set?` procedure returns `#t` if the *k*-th bit is 1 in the two's complement representation of *n*, and `#f` otherwise. This is the fixnum result of the following computation:

```
(not (fxzero? (fxand n (fxarithmetic-shift-left 1 k))))
```

(fxcopy-bit *n k b*) procedure

k must be non-negative and less than `fx-width`. *b* must be 0 or 1. The `fxcopy-bit` procedure returns the result of replacing the *k*-th bit of *n* by *b*, which is the result of the following computation:

```
(fxif (fxarithmetic-shift-left 1 k)
      (fxarithmetic-shift-left b k)
      n)
```

(fxmin *n m* ...) procedure

Returns the minimum of the provided fixnums *n, m* ....

(fxmax *n m* ...) procedure

Returns the maximum of the provided fixnums *n, m* ....

(fxrandom) procedure
(fxrandom *max*)
(fxrandom *min max*)

Returns a random number between fixnum *min* (inclusive) and fixnum *max* (exclusive). If *min* is not provided, then 0 is assumed to be the minimum bound. *max* is required to be greater than *min*. If called without any arguments, `fxrandom` returns a random fixnum number from the full fixnum range.

```
(fxrandom)          ⇒ 3845975858750874798
(fxrandom 10)       ⇒ 7
(fxrandom -6 -2)    ⇒ -5
```

36.12 Floating-point operations

(make-flonum *x n*) procedure

Returns $x \times 2$ ^ *n*, where *n* is a fixnum with an implementation-dependent range. The significand *x* is a flonum.

(real->flonum *x*) procedure

Returns the best flonum representation of real number *x*.

(flexponent *x*) procedure

Returns the exponent of flonum *x* (using a base of 2).

(flsignificand *x*) procedure

Returns the significand of flonum *x*.

(flnext *x*) procedure

Returns the least representable flonum value that compares greater than flonum *x*.

(flprev *x*) procedure

Returns the greatest representable flonum value that compares less than flonum *x*.

(fl+ *x y*...) procedure
(fl* *x y*...)

These procedures return the flonum sum or product of their flonum arguments *x y* .... In general, they return the flonum that best approximates the mathematical sum or product.

(fl- *x* ...) procedure
(fl/ *x* ...)

These procedures return the flonum difference or quotient of their flonum arguments *x* .... In general, they return the flonum that best approximates the mathematical difference or quotient. `(fl- x)` negates `x`, `(fl/ x)` is equivalent to `(fl/ 1.0 x)`.

(flzero? *x*) procedure

Returns `#t` if *x* = *0.0*, `#f` otherwise.

(flpositive? *x*) procedure

Returns `#t` if *x* > *0.0*, `#f` otherwise.

(flnegative? *x*) procedure

Returns `#t` if *x* < *0.0*, `#f` otherwise.

(fl= *x y z* ...) procedure
(fl< *x y z* ...)
(fl> *x y z* ...)
(fl<= *x y z* ...)
(fl>= *x y z* ...)

These procedures implement the comparison predicates for flonums. `fl=` returns `#t` if all provided flonums are equal. `fl<` returns `#t` if all provided flonums are strictly monotonically increasing. `fl>` returns `#t` if all provided flonums are strictly monotonically decreasing. `fl<=` returns `#t` if all provided flonums are monotonically increasing. `fl>=` returns `#t` if all provided flonums are monotonically decreasing.

```
(fl= +inf.0 +inf.0)  ⇒  #t
(fl= -inf.0 +inf.0)  ⇒  #f
(fl= -inf.0 -inf.0)  ⇒  #t
(fl= 0.0 -0.0)       ⇒  #t
(fl< 0.0 -0.0)       ⇒  #f
(fl= +nan.0 123.0)   ⇒  #f
(fl< +nan.0 123.0)   ⇒  #f
```

(flabs *x*) procedure

Returns the absolute value of *x* as a flonum.

(flmin *x* ...) procedure

Returns the minimum value of the provided flonum values *x* .... If no arguments are provided, positive infinity is returned.

(flmax *x* ...) procedure

Returns the maximum value of the provided flonum values *x* .... If no arguments are provided, negative infinity is returned.

(flrandom) procedure
(flrandom *max*)
(flrandom *min max*)

Returns a random number between flonum *min* (inclusive) and flonum *max* (exclusive). If *min* is not provided, then 0.0 is assumed to be the minimum bound. *max* is required to be greater than *min*. If called without any arguments, `flrandom` returns a random floating-point number from the interval `[0.0, 1.0[`.

```
(flrandom)          ⇒ 0.2179448178976645
(flrandom 123.4)    ⇒ 30.841401002076296
(flrandom -5.0 5.0) ⇒ -2.6619236065396237
```

37 LispKit Math Matrix

Library `(lispkit math matrix)` provides abstractions for representing vectors, matrices and for performing vector and matrix arithmetics. A matrix is a rectangular array of numbers. The library supports common matrix operations such as matrix addition, subtraction, and multiplication. There are operations to create matrices and to manipulate matrix objects. Furthermore, there is support to compute matrix determinants and to transpose and invert matrices. Matrices can be transformed into reduced row echelon form and matrix ranks can be determined.

matrix-type-tag object

Symbol representing the `matrix` type. The `type-for` procedure of library `(lispkit type)` returns this symbol for all matrix objects.

(make-matrix *m n*) procedure

Returns a new matrix with *m* rows and *n* columns.

(matrix *rows*) procedure
(matrix *row0 row1 ...*)

Returns a new matrix consisting of the given rows. Either *rows* is a list of list or numbers, or it is a vector of vectors of numbers. Instead of specifying one *rows* argument, it is also possible to provide the rows *row0*, *row1*, etc. as individual arguments to procedure `matrix`.

```
(display (matrix->string
           (matrix '(1 2 3) '(4 5 6))))
⇒  |1 2 3|
   |4 5 6|
```

(identity-matrix *n*) procedure

Returns a new identity matrix with `n` rows and columns.

```
(display (matrix->string
           (identity-matrix 3)))
⇒  |1 0 0|
   |0 1 0|
   |0 0 1|
```

(matrix-copy *matrix*) procedure

Returns a copy of *matrix*.

(matrix-eliminate *matrix i j*) procedure

Returns a copy of *matrix* with row *i* and column *j* removed.

(matrix-normalize *obj*) procedure

Returns a normalized version of *obj*, representing 1*1 matrices and vectors of length 1 as a number, and m*1 and 1*n matrices as a vector.

(matrix? *obj*) procedure

Returns `#t` if *obj* is a matrix object, otherwise `#f` is returned.

(matrix-vector? *obj*) procedure
Returns `#t` if *obj* is a vector of numbers.

(matrix-zero? *matrix*) procedure
Returns `#t` if *matrix* is a zero matrix, i.e. all its elements are zero.

(matrix-square? *matrix*) procedure
Returns `#t` if *matrix* is a square matrix, i.e. it has size _n*n_.

(matrix-identity? *matrix*) procedure
Returns `#t` if *matrix* is an identity matrix, i.e. it has `1` at the positions *(i, i)* and all other elements are zero.

(matrix-symmetric? *matrix*) procedure
Returns `#t` if *matrix* is symmetric, i.e. *matrix* is equal to its transposed matrix.

(matrix-dimensions? *matrix m n*) procedure
Returns `#t` if *matrix* is a matrix of the given dimensions. *m* is the number of rows, *n* is the number of columns.

(matrix=? *m0 m1 ...*) procedure
Returns `#t` if all matrices *m0*, *m1*, ... are equal to each other, otherwise `#f` is returned.

(matrix-size *matrix*) procedure
Returns the size of *matrix* as a pair whose car is the number of rows and cdr is the number of columns.

(matrix-rows *matrix*) procedure
Returns the number of rows of *matrix*.

(matrix-columns *matrix*) procedure
Returns the number of columns of *matrix*.

(matrix-row *matrix i*) procedure
Returns row *i* of *matrix* as a vector.

(matrix-column *matrix j*) procedure
Returns column *j* of *matrix* as a vector.

(matrix->vector *matrix*) procedure
Returns *matrix* as a vector of vectors.

(matrix-row->list *matrix i*) procedure
Returns row *i* of *matrix* as a list.

(matrix-column->list *matrix j*) procedure
Returns column *j* of *matrix* as a list.

(matrix->list *matrix*) procedure
Returns *matrix* as a list of lists.

(matrix-column-swap! *matrix j k*) procedure
Swaps columns *j* and *k* of *matrix*.

(matrix-row-swap! *matrix i k*) procedure
Swaps rows `i` and `k` of `matrix`.

(matrix-ref *matrix i j*) procedure
Returns the *j*-th element of the *i*-th row of *matrix*.

(matrix-set! *matrix i j x*) procedure
Sets the *j*-th element of the *i*-th row of *matrix* to *x*.

(matrix-for-each *f matrix*) procedure

Invokes *f* for every element of *matrix*. *f* is a procedure taking three arguments: the row, the column, and the element at this position. The traversal order is by row, from left to right.

(matrix-fold *f z matrix*) procedure

Folds the matrix elements row by row from left to right, invoking *f* with the accumulator, the row, the column and the element at this position.

(matrix-transpose *matrix*) procedure

Returns *matrix* in transposed form.

```
(define m (matrix '(1 2 3) '(4 5 6)))
(display (matrix->string
           (matrix-transpose m)))
⇒  |1 4|
   |2 5|
   |3 6|
```

(matrix-sum! *matrix m0 ...*) procedure

Sums up *matrix* and all matrices *m0*, ... storing the result in *matrix*.

(matrix-difference! *matrix m0 ...*) procedure

Subtracts the matrices *m0, ...* from *matrix*, storing the result in *matrix*.

(matrix-add *m0 m1 ...*) procedure

Returns the sum of matrices or vectors *m0, m1, ...*

(matrix-subtract *m0 m1 ...*) procedure

Returns the difference of matrix or vector *m0* and the matrices or vectors *m1, ...* This procedure also supports vector differences.

(matrix-mult *m0 m1 ...*) procedure

Returns the matrix product *m0 * m1 * ...* or the dot product if *m0*, *m1*, ... are vectors.

(matrix-minor *matrix i j*) procedure

Returns a minor of *matrix* by removing row *i* and column *j* and computing the determinant of the remaining matrix. *matrix* needs to be a square matrix.

(matrix-cofactor *matrix i j*) procedure

Returns a minor of *matrix* by removing row *i* and column *j* and computing the determinant of the remaining matrix. The result is negative if the sum of *i* and *j* is odd. *matrix* needs to be a square matrix.

(matrix-determinant *matrix*) procedure

Returns the determinant of *matrix*. *matrix* needs to be a square matrix.

(matrix-inverse *matrix*) procedure

Returns the inverse of *matrix* if it exists. If it does not exist, `#f` is being returned. *matrix* needs to be a square matrix.

```
(define a (matrix '(1 2 3) '(0 1 0) '(1 1 0)))
(define b (matrix-inverse a))
(matrix-identity? (matrix-mult a b))  ⇒  #t
```

(matrix-row-echelon! *matrix*) procedure

Returns the reduced row echelon matrix of `matrix` by continuously applying *Gaussian Elimination*.

```
(define m (matrix '(5 -6 -7   7)
                  '(6 -4 10 -34)
                  '(2  4 -3  29)))
(matrix-row-echelon! m)
(display (matrix->string m))
⇒  |1 0 0  2|
   |0 1 0  4|
   |0 0 1 -3|
```

(matrix-rank *matrix*) procedure

Returns the rank of *matrix*. The rank of *matrix* is the dimension of the vector space generated by its columns. This corresponds to the maximal number of linearly independent columns of *matrix*.

(matrix->string *matrix*) procedure
(matrix->string *matrix len*)
(matrix->string *matrix len prec*)
(matrix->string *matrix len prec noexp*)

Returns a multi-line string representation of *matrix*. The elements *x* of *matrix* are converted into a string by invoking `(number->string x len prec noexp)`.

38 LispKit Math Stats

Library `(lispkit math stats)` implements statistical utility functions. The functions compute summary values for collections of samples, and functions for managing sequences of samples. Most of the functions accept a list of real numbers corresponding to sample values.

(mode *xs*) procedure

Computes the mode of a set of numbers *xs*. The mode is the value that appears most often in *xs*. *xs* is a proper list of numeric values. `=` is used as the equality operator.

(mean *xs*) procedure

Computes the arithmetic mean of a set of numbers *xs*. *xs* is a proper list of numeric values.

(range *xs*) procedure

Computes the range of a set of numbers *xs*, i.e. the difference between the largest and the smallest value. *xs* is a proper list of numeric values which are ordered using the `<` relation.

(variance *xs*) procedure
(variance *xs bias*)

Computes the variance for a set of numbers *xs*, optionally applying bias correction. *xs* is a proper list of numeric values. Bias correction gets enabled by setting *bias* to `#t`. Alternatively, it is possible to provide a positive integer, which is used instead of the number of elements in *xs*.

(stddev *xs*) procedure
(stddev *xs bias*)

Computes the standard deviation for a set of numbers *xs*, optionally applying bias correction. *xs* is a proper list of numeric values. Bias correction gets enabled by setting *bias* to `#t`. Alternatively, it is possible to provide a positive integer, which is used instead of the number of elements in *xs*.

(skewness *xs*) procedure
(skewness *xs bias*)

Computes the skewness for a set of numbers *xs*, optionally applying bias correction. *xs* is a proper list of numeric values. Bias correction gets enabled by setting *bias* to `#t`. Alternatively, it is possible to provide a positive integer, which is used instead of the number of elements in *xs*.

(kurtosis *xs*) procedure
(kurtosis *xs bias*)

Computes the kurtosis for a set of numbers *xs*, optionally applying bias correction. *xs* is a proper list of numeric values. Bias correction gets enabled by setting *bias* to `#t`. Alternatively, it is possible to provide a positive integer, which is used instead of the number of elements in *xs*.

(absdev *xs*) procedure

Computes the average absolute difference between the numbers in list *xs* and `(median xs)`.

(quantile *xs p*) procedure

Computes the p-quantile for a set of numbers *xs* (also known as the *inverse cumulative distribution*). *p* is a real number between 0 and 1.0. For instance, the 0.5-quantile corresponds to the median.

(percentile *xs pct*) procedure

Computes the percentile for a set of numbers *xs* and a given percentage *pct*. *pct* is a number between 0 and 100. For instance, the 90th percentile corresponds to the 0.9-quantile.

(median *xs*) procedure

Computes the median for a set of numbers *xs*.

(interquartile-range *xs*) procedure

Returns the interquartile range for a given set of numbers *xs*. *xs* is a proper list of numeric values. The interquartile range is the difference between the 0.75-quantile and the 0.25-quantile.

(five-number-summary *xs*) procedure

Returns a list of 5 statistics describing the set of numbers *xs*: the minimum value, the lower quartile, the median, the upper quartile, and the maximum value.

(covariance *xs ys*) procedure
(covariance *xs ys bias*)

Computes the covariance of two sets of numbers *xs* and *ys*. Both *xs* and *ys* are proper lists of numbers. Bias correction can be enabled by setting *bias* to `#t`. Alternatively, it is possible to provide a positive integer, which is used instead of the number of elements in *xs*.

(correlation *xs ys*) procedure
(correlation *xs ys bias*)

Computes the correlation of two sets of numbers *xs* and *ys* in form of the *Pearson product-moment correlation coefficient*. Both *xs* and *ys* are proper lists of numbers. Bias correction can be enabled by setting *bias* to `#t`. Alternatively, it is possible to provide a positive integer, which is used instead of the number of elements in *xs*.

39 LispKit Math Util

Library `(lispkit math util)` implements mathematical utility functions.

(sgn *x*) procedure
Implements the sign/signum function. Returns -1 if *x* is negative, 0 (or a signed zero, when inexact) if *x* is zero, and 1 if *x* is a positive number. `sgn` fails if *x* is not a real number.

(numbers *lo hi*) procedure
(numbers *lo hi f*)
(numbers *lo hi guard f*)

Returns a list of numbers by iterating from integer *lo* to integer *hi* (both inclusive) and applying function *f* to each integer in the range for which *guard* returns true. The default guard always returns true. The default for *f* is `identity`.

(sum *xs*) procedure
Returns the sum of all numbers of list *xs*. This procedure fails if there is an element in *xs* which is not a number.

(product *xs*) procedure
Returns the product of all numbers of list *xs*. This procedure fails if there is an element in *xs* which is not a number.

(minimum *xs*) procedure
Returns the minimum of all numbers of list *xs*. This procedure fails if there is an element in *xs* which is not a number.

(maximum *xs*) procedure
Returns the maximum of all numbers of list *xs*. This procedure fails if there is an element in *xs* which is not a number.

(conjugate *x*) procedure
Conjugates number *x*. For real numbers *x*, `conjugate` returns *x*, otherwise *x* is being returned with the opposite sign for the imaginary part.

(degrees->radians *x*) procedure
Converts degrees into radians.

(radians->degrees *x*) procedure
Converts radians into degrees.

(prime? *n*) procedure
Returns `#t` if integer *n* is a prime number, `#f` otherwise.

(make-nan *neg quiet payload*) procedure
Returns a NaN whose sign bit is equal to *neg* (`#t` for negative, `#f` for positive), whose quiet bit is equal to *quiet* (`#t` for quiet, `#f` for signaling), and whose payload is the positive exact integer *payload*. It is an error if *payload* is larger than a NaN can hold.

(nan-negative? *x*) procedure
Returns `#t` if the sign bit of *x* is 1 and `#f` otherwise.

(nan-quiet? *x*) procedure

Returns `#t` if *x* is a quiet NaN.

(nan-payload *x*) procedure

Returns the payload bits of floating-point number *x* as a positive exact integer.

(nan=? *x* *y*) procedure

Returns `#t` if *x* and *y* have the same sign, quiet bit, and payload; and `#f` otherwise.

40 LispKit Object

Library `(lispkit object)` implements a simple, delegation-based object system for LispKit. It provides procedural and declarative interfaces for objects and classes. The class system is optional. It mostly provides means to define and manage new object types and construct objects using object constructors.

40.1 Introduction

Similar to other Scheme and Lisp-based object systems, methods of objects are defined in terms of object/class-specific specializations of generic procedures. A generic procedure consists of methods for the various objects/classes it supports. A generic procedure performs a dynamic dispatch on the first parameter (the `self` parameter) to determine the applicable method.

40.1.1 Generic procedures

Generic procedures can be defined using the `define-generic` form. Here is an example which defines three generic methods, one with only a `self` parameter, and two with three parameters `self`, `x` and `y`. The last generic procedure definition includes a `default` method which is applicable to all objects for which there is no specific method. When a generic procedure without default is applied to an object that does not define its own method implementation, an error gets signaled.

```
(define-generic (point-coordinates self))
(define-generic (set-point-coordinates! self x y))
(define-generic (point-move! self x y)
  (let ((coord (point-coordinate self)))
    (set-point-coordinate! self (+ (car coord) x) (+ (cdr coord) y))))
```

40.1.2 Objects

An object encapsulates a list of methods each implementing a generic procedure. These methods are regular closures which can share mutable state. Objects do not have an explicit notion of a field or slot as in other Scheme or Lisp-based object systems. Fields/slots need to be implemented via generic procedures and method implementations sharing state. Here is an example explaining this approach:

```
(define (make-point x y)
  (object ()
    ((point-coordinates self) (cons x y))
    ((set-point-coordinates! self nx ny) (set! x nx) (set! y ny))
    ((object->string self) (string-append (object->string x) "/" (object->string y)))))
```

This is a function creating new point objects. The `x` and `y` parameters of the constructor function are used for representing the state of the point object. The created point objects implement three generic procedures: `point-coordinates`, `set-point-coordinates`, and `object->string`. The latter procedure is defined directly by the library and, in general, used for creating a string representation of any object. By implementing the `object->string` method, the behavior gets customized for the object.

The following lines of code illustrate how point objects can be used:

```
(define pt (make-point 25 37))
pt                                                 => #object:#<box (...)>
(object->string pt)                                => "25/37"
(point-coordinates pt)                             => (25 . 37)
(set-point-coordinates! pt 5 6)
(object->string pt)                                => "5/6"
(point-coordinates pt)                             => (5 . 6)
```

40.1.3 Inheritance

The LispKit object system supports inheritance via delegation. The following code shows how colored points can be implemented by delegating all point functionality to the previous implementation and by simply adding only color-related logic.

```
(define-generic (point-color self) #f)
(define (make-colored-point x y color)
  (object ((super (make-point x y)))
    ((point-color self) color)
    ((object->string self)
       (string-append (object->string color) ":" (invoke (super object->string) self)))))
```

The object created in function `make-colored-point` inherits all methods from object `super` which gets set to a new point object. It adds a new method to generic procedure `point-color` and redefines the `object->string` method. The redefinition is implemented in terms of the inherited `object->string` method for points. The form `invoke` can be used to refer to overridden methods in delegatee objects. Thus, `(invoke (super object->string) self)` calls the `object->string` method of the `super` object but with the identity (`self`) of the colored point.

The following interaction illustrates the behavior:

```
(define cpt (make-colored-point 100 50 'red))
(point-color cpt)                                  => red
(point-coordinates cpt)                            => (100 . 50)
(set-point-coordinates! cpt 101 51)
(object->string cpt)                               => "red:101/51"
```

Objects can delegate functionality to multiple delegatees. The order in which they are listed determines the methods which are being inherited in case there are conflicts, i.e. multiple delegatees implement a method for the same generic procedure.

40.1.4 Classes

Classes add syntactic sugar, simplying the creation and management of objects. They play the following role in the object-system of LispKit:

1. A class defines a constructor for objects represented by this class.
2. Each class defines an object type, which can be used to distinguish objects created by the same constructor and supporting the same methods.
3. A class can inherit functionality from several other classes, making it easy to reuse functionality.

4. Classes are first-class objects supporting a number of class-related procedures.

The following code defines a `point` class with similar functionality as above:

```
(define-class (point x y) ()
  (object ()
    ((point-coordinates self) (cons x y))
    ((set-point-coordinates! self nx ny) (set! x nx) (set! y ny))
    ((object->string self) (string-append (object->string x) "/" (object->string y)))))
```

Instances of this class are created by using the generic procedure `make-instance` which is implemented by all class objects:

```
(define pt2 (make-instance point 82 10))
pt2                                              => #point:#<box (...)>
(object->string pt2)                             => "82/10"
```

Each object created by a class implements a generic procedure `object-class` referring to the class of the object. Since classes are objects themselves we can obtain their name with generic procedure `class-name` :

```
(object-class pt2)                               => #class:#<box (...)>
(class-name (object-class pt2))                  => point
(instance-of? point pt2)                         => #t
(instance-of? point pt)                          => #f
```

Generic procedure `instance-of?` can be used to determine whether an object is a direct or indirect instance of a given class. The last two lines above show that `pt2` is an instance of `point`, but `pt` is not, even though it is functionally equivalent.

The following definition re-implements the colored point example from above using a class:

```
(define-class (colored-point x y color) (point)
  (if (or (< x 0) (< y 0))
      (error "coordinates are negative: ($0; $1)" x y))
  (object ((super (make-instance point x y)))
    ((point-color self) color)
    ((object->string self)
       (string-append (object->string color) ":" (invoke (super object->string) self)))))
```

The following lines illustrate the behavior of `colored-point` objects vs `point` objects:

```
(point-color cpt2)                               => blue
(point-coordinates cpt2)                         => (128 . 256)
(set-point-coordinates! cpt2 64 32)
(object->string cpt2)                            => "blue:64/32"
(instance-of? point cpt2)                        => #t
(instance-of? colored-point cpt2)                => #t
(instance-of? colored-point cpt)                 => #f
(class-name (object-class cpt2))                 => colored-point
```

40.2 Procedural object interface

object-type-tag object

Symbol representing the `object` type. The `type-for` procedure of library `(lispkit type)` returns this symbol for all objects created via `object` or `make-object`.

(object? *obj*) procedure

Returns `#t` if *obj* is an object as defined by this library. Objects are either created procedurally via `make-object` or declaratively via `object`.

(make-object) procedure
(make-object *delegate* ...)

(method *obj generic*) procedure

(method *obj generic*) procedure

(object-methods *obj*) procedure

(add-method! *obj generic method*) procedure

(delete-method! *obj generic*) procedure

(make-generic-procedure ...) procedure

40.3 Declarative object interface

(object ...) syntax

(define-generic ...) syntax

(invoke ...) syntax

40.4 Procedural class interface

class-type-tag object

Symbol representing the `class` type. The `type-for` procedure of library `(lispkit type)` returns this symbol for all class objects.

(class? *obj*) procedure

Returns `#t` if *obj* is a class object, `#f` otherwise.

root object

The root class object. All class objects have `root` as its direct or indirect superclass object.

(make-class *name superclasses constructor*) procedure

Returns a new class whose name is *name*. *superclasses* is a list of superclass objects. *constructor* is a procedure that is called whenever instances of this new class are being created. *constructor* are passed the arguments passed to `make-instance` when a new object of this class is being created. It returns two values: a list of delegate objects and an initializer procedure which sets up the internal state.

40.4.1 Instance methods

(object-class *obj*) generic procedure

Returns the class of object *obj*.

(object-equal? *obj other*) generic procedure

Returns `#t` if *obj* and *other* are considered equal objects.

(object->string *obj*) procedure

Returns a string representation of object *obj*.

40.4.2 Class methods

(class-name *class*) generic procedure

Returns the class name of *class*.

(class-direct-superclasses *class*) generic procedure

Returns a list of superclass objects of *class*.

(subclass? *class other*) generic procedure

Returns `#t` if *class* is a subclass of class *other*, `#f` otherwise.

(make-instance *class arg* ...) generic procedure

Creates and returns a new object of *class*. *arg* ... are the constructor arguments passed to the constructor of *class*.

(instance-of? *class obj*) generic procedure

Returns `#t` if *obj* is an instance of *class*.

40.5 Declarative class interface

(define-class ...) syntax

41 LispKit Port

Ports represent abstractions for handling input and output. They are used to access files, devices, and similar things on the host system on which LispKit is running.

An *input port* is a LispKit object that can deliver data upon command, while an *output port* is an object that can accept data. In LispKit, input and output port types are disjoint, i.e. a port is either an input or an output port.

Different port types operate on different data. LispKit provides two differnt types of ports: *textual ports* and *binary ports*. Textual ports and binary ports are disjoint, i.e. a port is either textual or binary.

A *textual port* supports reading or writing of individual characters from or to a backing store containing characters using `read-char` and `write-char` , and it supports operations defined in terms of characters, such as `read` and `write` .

A *binary port* supports reading or writing of individual bytes from or to a backing store containing bytes using `read-u8` and `write-u8` below, as well as operations defined in terms of bytes.

41.1 Default ports

current-output-port `parameter object`
current-input-port
current-error-port

These parameter objects represent the current default input port, output port, or error port (an output port), respectively. These parameter objects can be overridden with `parameterize` .

default-output-port `object`
default-input-port

These two ports are the initial values of `current-output-port` and `current-input-port` when LispKit gets initialized. They are typically referring to the default output and input device of the system on which LispKit is running.

41.2 Predicates

(port? *obj*) `procedure`
Returns `#t` if *obj* is a port object; otherwise `#f` is returned.

(input-port? *obj*) `procedure`
(output-port? *obj*)

These predicates return `#t` if *obj* is an input port or output port; otherwise they return `#f` .

(textual-port? *obj*) `procedure`
(binary-port? *obj*)

These predicates return `#t` if *obj* is a textual or a binary port; otherwise they return `#f` .

(input-port-open? *port*) procedure
(output-port-open? *port*)

Returns `#t` if *port* is still open and capable of performing input or output, respectively, and `#f` otherwise.

(eof-object? *obj*) procedure
Returns `#t` if *obj* is an end-of-file object, otherwise returns `#f` .

41.3 General ports

(close-port *port*) procedure
(close-input-port *port*)
(close-output-port *port*)

Closes the resource associated with *port*, rendering the port incapable of delivering or accepting data. It is an error to apply `close-input-port` and `close-output-port` to a port which is not an input or output port, respectively. All procedures for closing ports have no effect if the provided *port* has already been closed.

(with-input-from-port *port thunk*) procedure
(with-output-to-port *port thunk*)

The given port is made to be the value returned by `current-input-port` or `current-output-port` (as used by `(read)` , `(write obj)` , and so forth). The *thunk* is then called with no arguments. When the *thunk* returns, the port is closed and the previous default is restored. It is an error if *thunk* does not accept zero arguments. Both procedures return the values yielded by *thunk*. If an escape procedure is used to escape from the continuation of these procedures, they behave exactly as if the current input or output port had been bound dynamically with `parameterize` .

(call-with-port *port proc*) procedure
The `call-with-port` procedure calls *proc* with *port* as an argument. It is an error if *proc* does not accept one argument.

If *proc* returns, then the port is closed automatically and the values yielded by *proc* are returned. If *proc* does not return, then the port will not be closed automatically unless it is possible to prove that the port will never again be used for a read or write operation.

This is necessary, because LispKit's escape procedures have unlimited extent and thus it is possible to escape from the current continuation but later to resume it. If LispKit would be permitted to close the port on any escape from the current continuation, then it would be impossible to write portable code using both `call-with-current-continuation` and `call-with-port` .

41.4 File ports

(open-input-file *filepath*) procedure
(open-input-file *filepath fail*)

Takes a *filepath* referring to an existing file and returns a textual input port that is capable of delivering data from the file. If the file does not exist or cannot be opened, an error that satisfies `file-error?` is signaled if argument *fail* is not provided. If *fail* is provided, it is returned in case an error occured.

(open-binary-input-file *filepath*) procedure
(open-binary-input-file *filepath fail*)

Takes a *filepath* referring to an existing file and returns a binary input port that is capable of delivering data from the file. If the file does not exist or cannot be opened, an error that satisfies `file-error?` is signaled if argument *fail* is not provided. If *fail* is provided, it is returned in case an error occured.

(open-output-file *filepath*) procedure
(open-output-file *filepath fail*)

Takes a *filepath* referring to an output file to be created and returns a textual output port that is capable of writing data to the new file. If a file with the given name exists already, the effect is unspecified. If the file cannot be opened, an error that satisfies `file-error?` is signaled if argument *fail* is not provided. If *fail* is provided, it is returned in case an error occured.

(open-binary-output-file *filepath*) procedure
(open-binary-output-file *filepath fail*)

Takes a *filepath* referring to an output file to be created and returns a binary output port that is capable of writing data to the new file. If a file with the given name exists already, the effect is unspecified. If the file cannot be opened, an error that satisfies `file-error?` is signaled if argument *fail* is not provided. If *fail* is provided, it is returned in case an error occured.

(with-input-from-file *filepath thunk*) procedure
(with-output-to-file *filepath thunk*)

The file determined by *filepath* is opened for input or output as if by `open-input-file` or `open-output-file`, and the new port is made to be the value returned by `current-input-port` or `current-output-port` (as used by `(read)`, `(write obj)`, and so forth). The *thunk* is then called with no arguments. When the *thunk* returns, the port is closed and the previous default is restored. It is an error if *thunk* does not accept zero arguments. Both procedures return the values yielded by *thunk*. If an escape procedure is used to escape from the continuation of these procedures, they behave exactly as if the current input or output port had been bound dynamically with `parameterize`.

(call-with-input-file *filepath proc*) procedure
(call-with-output-file *filepath proc*)

These procedures create a textual port obtained by opening the file referred to by *filepath* (a string) for input or output as if by `open-input-file` or `open-output-file`. This port and *proc* are then passed to a procedure equivalent to `call-with-port`. It is an error if *proc* does not accept one argument.

41.5 String ports

(open-input-string *str*) procedure
Takes a string and returns a textual input port that delivers characters from the string. If the string is modified, the effect is unspecified.

(open-output-string) procedure
Returns a textual output port that will accumulate characters for retrieval by `get-output-string`.

```
(parameterize ((current-output-port (open-output-string)))
  (display "piece")
  (display " by piece ")
  (display "by piece.")
  (get-output-string (current-output-port)))
   ⇒ "piece by piece by piece."
```

(get-output-string *port*) procedure
It is an error if *port* was not created with `open-output-string`.

Returns a string consisting of the characters that have been output to *port* so far in the order they were output.

```
(parameterize ((current-output-port (open-output-string)))
  (display "piece")
  (display " by piece ")
  (display "by piece.")
  (newline)
  (get-output-string (current-output-port)))
    ⇒ "piece by piece by piece.\n"
```

(with-input-from-string *str thunk*) procedure

String *str* is opened for input as if by `open-input-string`, and the new textual string port is made to be the value returned by `current-input-port`. The *thunk* is then called with no arguments. When the *thunk* returns, the port is closed and the previous default is restored. It is an error if *thunk* does not accept zero arguments. `with-input-from-string` returns the values yielded by *thunk*. If an escape procedure is used to escape from the continuation of these procedures, they behave exactly as if the current input port had been bound dynamically with `parameterize`.

(with-output-to-string *thunk*) procedure

A new string output port is created as if by calling `open-output-string`, and the new port is made to be the value returned by `current-output-port`. The *thunk* is then called with no arguments. When the *thunk* returns, the port is closed and the previous default is restored. It is an error if *thunk* does not accept zero arguments. Both procedures return the values yielded by *thunk*. If an escape procedure is used to escape from the continuation of these procedures, they behave exactly as if the current input or output port had been bound dynamically with `parameterize`.

(call-with-output-string *proc*) procedure

The procedure *proc* is called with one argument, a textual output port. The values yielded by *proc* are ignored. When *proc* returns, `call-with-output-string` returns the port's accumulated output as a string.

This procedure is defined as follows:

```
(define (call-with-output-string procedure)
  (let ((port (open-output-string)))
    (procedure port)
    (get-output-string port)))
```

41.6 Bytevector ports

(open-input-bytevector *bvector*) procedure

Takes a bytevector *bvector* and returns a binary input port that delivers bytes from the bytevector *bvector*.

(open-output-bytevector) procedure

Returns a binary output port that will accumulate bytes for retrieval by `get-output-bytevector`.

(get-output-bytevector *port*) procedure

It is an error if *port* was not created with `open-output-bytevector`. `get-output-bytevector` returns a bytevector consisting of the bytes that have been output to the port so far in the order they were output.

(call-with-output-bytevector *proc*) procedure

The procedure *proc* gets called with one argument, a binary output port. The values yielded by procedure *proc* are ignored. When it returns, `call-with-output-bytevector` returns the port's accumulated output as a newly allocated bytevector.

This procedure is defined as follows:

```
(define (call-with-output-bytevector procedure)
  (let ((port (open-output-bytevector)))
    (procedure port)
    (get-output-bytevector port)))
```

41.7 URL ports

(open-input-url *url*) procedure
(open-input-url *url timeout*)
(open-input-url *url timeout fail*)

Takes a *url* referring to an existing resource and returns a textual input port that is capable of reading data from the resource (e.g. via HTTP). *timeout* specifies a timeout in seconds as a flonum for the operation to wait. If no data is available, the procedure will fail either by throwing an exception or by returning value *fail* if provided.

(open-binary-input-url *url*) procedure
(open-binary-input-url *url timeout*)
(open-binary-input-url *url timeout fail*)

Takes a *url* referring to an existing resource and returns a binary input port that is capable of reading data from the resource (e.g. via HTTP). *timeout* specifies a timeout in seconds as a flonum for the operation to wait. If no data is available, the procedure will fail either by throwing an exception or by returning value *fail* if provided.

(with-input-from-url *url thunk*) procedure
The given *url* is opened for input as if by `open-input-url`, and the new input port is made to be the value returned by `current-input-port`. The *thunk* is then called with no arguments. When the *thunk* returns, the port is closed and the previous default is restored. It is an error if *thunk* does not accept zero arguments. The procedure returns the values yielded by *thunk*. If an escape procedure is used to escape from the continuation of this procedure, they behave exactly as if `current-input-port` had been bound dynamically with `parameterize`.

(call-with-input-url *url proc*) procedure
`call-with-input-url` creates a textual input port by opening the resource at *url* for input as if by `open-input-url`. This port and *proc* are then passed to a procedure equivalent to `call-with-port`. It is an error if *proc* does not accept one argument. Here is an implementation of `call-with-input-url`:

```
(define (call-with-input-url url proc)
  (let* ((port (open-input-url url))
         (res (proc port)))
    (close-input-port port)
    res))
```

(try-call-with-input-url *url proc thunk*) procedure
`try-call-with-input-url` creates a textual input port by opening the resource at *url* for input as if by `open-input-url`. This port and *proc* are then passed to a procedure equivalent to `call-with-port` in case it was possible to open the port. If the port couldn't be opened, *thunk* gets invoked. It is an error if *proc* does not accept one argument and if *thunk* requires at least one argument. Here is an implementation of `try-call-with-input-url`:

```
(define (try-call-with-input-url url proc thunk)
  (let ((port (open-input-url url 60.0 #f)))
    (if port
        (car (cons (proc port) (close-input-port port)))
        (thunk))))
```

41.8 Asset ports

(open-input-asset *name type*) procedure
(open-input-asset *name type dir*)

This function can be used to open a textual LispKit asset file located in one of LispKit's asset paths. An asset is identified via a file *name*, a file *type*, and an optional directory path *dir*. *name*, *type*, and *dir* are all strings. `open-input-asset` constructs a relative file path in the following way (assuming *name* does not have a suffix already):

dir/name.type

It then searches the asset paths in their given order for a file matching this relative file path. Once the first matching file is found, the file is opened as a text file and a corresponding textual input port that is capable of reading data from the file is returned. It is an error if no matching asset is found.

(open-binary-input-asset *name type*) procedure
(open-binary-input-asset *name type dir*)

This function can be used to open a binary LispKit asset file located in one of LispKit's asset paths. An asset is identified via a file *name*, a file *type*, and an optional directory path *dir*. *name*, *type*, and *dir* are all strings. `open-input-asset` constructs a relative file path in the following way (assuming *name* does not have a suffix already):

dir/name.type

It then searches the asset paths in their given order for a file matching this relative file path. Once the first matching file is found, the file is opened as a binary file and a corresponding binary input port that is capable of reading data from the file is returned. It is an error if no matching asset is found.

41.9 Reading from ports

If port is omitted from any input procedure, it defaults to the value returned by `(current-input-port)` . It is an error to attempt an input operation on a closed port.

(read) procedure
(read *port*)

The `read` procedure converts external representations of Scheme objects into the objects themselves by parsing the input. `read` returns the next object parsable from the given textual input *port*, updating *port* to point to the first character past the end of the external representation of the object.

If an end of file is encountered in the input before any characters are found that can begin an object, then an end-of-file object is returned. The port remains open, and further attempts to read will also return an end-of-file object. If an end of file is encountered after the beginning of an object's external representation, but the external representation is incomplete and therefore not parsable, an error that satisfies `read-error?` is signaled.

(read-char) procedure
(read-char *port*)

Returns the next character available from the textual input *port*, updating *port* to point to the following character. If no more characters are available, an end-of-file object is returned.

(peek-char) procedure
(peek-char *port*)

Returns the next character available from the textual input *port*, but without updating *port* to point to the following character. If no more characters are available, an end-of-file object is returned.

Note: The value returned by a call to `peek-char` is the same as the value that would have been returned by a call to `read-char` with the same port. The only difference is that the very next call to `read-char` or `peek-char` on that port will return the value returned by the preceding call to `peek-char`. In particular, a call to `peek-char` on an interactive port will hang waiting for input whenever a call to `read-char` would have hung.

(char-ready?) procedure
(char-ready? *port*)

Returns `#t` if a character is ready on the textual input *port* and returns `#f` otherwise. If `char-ready?` returns `#t` then the next `read-char` operation on the given *port* is guaranteed not to hang. If the *port* is at end of file, then `char-ready?` returns #t.

Rationale: The `char-ready?` procedure exists to make it possible for a program to accept characters from interactive ports without getting stuck waiting for input. Any input editors associated with such ports must ensure that characters whose existence has been asserted by `char-ready?` cannot be removed from the input. If `char-ready?` were to return `#f` at end of file, a port at end of file would be indistinguishable from an interactive port that has no ready characters.

(read-token) procedure
(read-token *port*)
(read-token *port charset*)

Returns the next token of text available from the textual input *port*, updating *port* to point to the following character. A token is a non-empty sequence of characters delimited by characters from character set *charset*. Tokens never contain characters from *charset*. *charset* defaults to the set of all whitespace and newline characters.

(read-line *obj*) procedure
(read-line *port*)

Returns the next line of text available from the textual input *port*, updating *port* to point to the following character. If an end of line is read, a string containing all of the text up to (but not including) the end of line is returned, and *port* is updated to point just past the end of line. If an end of file is encountered before any end of line is read, but some characters have been read, a string containing those characters is returned. If an end of file is encountered before any characters are read, an end-of-file object is returned. For the purpose of this procedure, an end of line consists of either a linefeed character, a carriage return character, or a sequence of a carriage return character followed by a linefeed character.

(read-string *k*) procedure
(read-string *k port*)

Reads the next *k* characters, or as many as are available before the end of file, from the textual input *port* into a newly allocated string in left-to-right order and returns the string. If no characters are available before the end of file, an end-of-file object is returned.

(read-u8) procedure
(read-u8 *port*)

Returns the next byte available from the binary input *port*, updating *port* to point to the following byte. If no more bytes are available, an end-of-file object is returned.

(peek-u8 *obj*) procedure

Returns the next byte available from the binary input *port*, but without updating *port* to point to the following byte. If no more bytes are available, an end-of-file object is returned.

(u8-ready?) procedure
(u8-ready? *port*)

Returns `#t` if a byte is ready on the binary input *port* and returns `#f` otherwise. If `u8-ready?` returns `#t` then the next `read-u8` operation on the given *port* is guaranteed not to hang. If the *port* is at end of file then `u8-ready?` returns `#t`.

(read-bytevector *k*) procedure
(read-bytevector *k port*)

Reads the next *k* bytes, or as many as are available before the end of file, from the binary input *port* into a newly allocated bytevector in left-to-right order and returns the bytevector. If no bytes are available before the end of file, an end-of-file object is returned.

(read-bytevector! *bvector*) procedure
(read-bytevector! *bvector port*)
(read-bytevector! *bvector port start*)
(read-bytevector! *bvector port start end*)

Reads the next *end* − *start* bytes, or as many as are available before the end of file, from the binary input *port* into bytevector *bvector* in left-to-right order beginning at the *start* position. If *end* is not supplied, reads until the end of bytevector *bvector* has been reached. If *start* is not supplied, reads beginning at position 0. Returns the number of bytes read. If no bytes are available, an end-of-file object is returned.

41.10 Writing to ports

If *port* is omitted from any output procedure, it defaults to the value returned by `(current-output-port)`. It is an error to attempt an output operation on a closed port.

(write *obj*) procedure
(write *obj port*)

Writes a representation of *obj* to the given textual output *port*. Strings that appear in the written representation are enclosed in quotation marks, and within those strings backslash and quotation mark characters are escaped by backslashes. Symbols that contain non-ASCII characters are escaped with vertical lines. Character objects are writ- ten using the `#\` notation.

If *obj* contains cycles which would cause an infinite loop using the normal written representation, then at least the objects that form part of the cycle will be represented using datum labels. Datum labels will not be used if there are no cycles.

(write-shared *obj*) procedure
(write-shared *obj port*)

The `write-shared` procedure is the same as `write`, except that shared structures will be represented using datum labels for all pairs and vectors that appear more than once in the output.

(write-simple *obj*) procedure
(write-simple *obj port*)

The `write-simple` procedure is the same as `write`, except that shared structures will never be represented using datum labels. This can cause `write-simple` not to terminate if *obj* contains circular structures.

(write-formatted *obj*) procedure
(write-formatted *obj config*)
(write-formatted *obj config port*)

Writes a representation of *obj* to the given textual output *port* using formatting configuration *config* as defined by library `(lispkit format)`. *config* defines how objects of different types are being written. Using this procedure makes it possible to use custom formatting logic instead of the hardcoded logic as provided by procedure `write`.

(display *obj*) procedure
(display *obj port*)

Writes a representation of *obj* to the given textual output *port*. Strings that appear in the written representation are output as if by `write-string` instead of by `write`. Symbols are not escaped. Character objects appear in the representation as if written by `write-char` instead of by `write`. `display` will not loop forever on self-referencing pairs, vectors, or records.

The `write` procedure is intended for producing machine-readable output and `display` for producing human-readable output.

(display* *obj ...*) procedure
Writes a representation of *obj* ... to the current default textual output port. Strings that appear in the written representation are output as if by `write-string` instead of by `write`. Symbols are not escaped. Character objects appear in the representation as if written by `write-char` instead of by `write`. `display*` will not loop forever on self-referencing pairs, vectors, or records.

(display-format *[port] [config] [locale] [tabw [linew]] cntrl arg ...*) procedure
`display-format` writes formatted output to a textual output *port*, outputting the characters of the control string *cntrl* while interpreting formatting directives embedded in *cntrl*. The control string syntax and the semantics of the provided arguments matches procedure `format` of library `(lispkit format)`. *config* refers to a `format-config` object which defines environment variables influencing the output of some formatting directives. *locale* refers to a locale identifier like symbol `en_US` that is used by locale-specific formatting directives. *tabw* defines the maximum number of space characters that correspond to a single tab character. *linew* specifies the number of characters per line; this is used by the justification directive only.

(newline) procedure
(newline *port*)

Writes an end of line to textual output *port*.

(write-char *char*) procedure
(write-char *char port*)

Writes the character *char* (not an external representation of the character) to the given textual output *port*.

(write-string *str*) procedure
(write-string *str port*)
(write-string *str port start*)
(write-string *str port start end*)

Writes the characters of string *str* from index *start* to *end* (exclusive) in left-to-right order to the textual output *port*. The default of *start* is 0, the default of *end* is the length of *str*.

(write-u8 *byte*) procedure
(write-u8 *byte*)

Writes the *byte* to the given binary output *port*.

(write-bytevector *bvector*) procedure
(write-bytevector *bvector port*)
(write-bytevector *bvector port start*)
(write-bytevector *bvector port start end*)

Writes the bytes of bytevector *bvector* from *start* to *end* (exclusive) in left-to-right order to the binary output *port*. The default of *start* is 0, the default of *end* is the length of *bvector*.

(flush-output-port) procedure
(flush-output-port *port*)

Flushes any buffered output from the buffer of the given output *port* to the underlying file or device.

(eof-object) procedure
Returns an end-of-file object.

42 LispKit Prolog

Library `(lispkit prolog)` implements *Schelog*, an *embedding* of Prolog-style logic programming in Scheme by Dorai Sitaram. This approach allows Prolog-style logic programming and Scheme-style functional programming to be combined. Schelog contains the full repertoire of Prolog features, including meta-logical and second-order ("set") predicates, leaving out only those features that could more easily and more efficiently be done with Scheme subexpressions.

42.1 Simple Goals and Queries

Schelog objects are the same as Scheme objects. However, there are two subsets of these objects that are of special interest to Schelog: *goals* and *predicates*. We will first look at some simple goals. The next section will introduce predicates and ways of making complex goals using predicates.

A *goal* is an object whose truth or falsity we can check. A goal that turns out to be true is said to succeed. A goal that turns out to be false is said to fail. Two simple goals that are provided in Schelog are:

```
%true
%fail
```

The goal `%true` always succeeds, the goal `%fail` always fails.

The names of all Schelog primitive objects start with `%`. This is to avoid clashes with the names of conventional Scheme objects of related meaning. User-created objects in Schelog are not required to follow this convention.

A Schelog user can *query* a goal by wrapping it in a `%which`-form.

```
(%which () %true)
```

evaluates to `()`, indicating success, whereas:

```
(%which () %fail)
```

evaluates to `#f`, indicating failure.

The second subexpression of the `%which`-form is the empty list `()`. Later we will see `%which` used with other lists as the second subform. The distinction between successful and failing goals relies on Scheme distinguishing between `#f` and `()`. We will use the annotation `()` true to signal that `()` is being used as a true value. Henceforth, we will use the notation:

`E => F`

to say that E *evaluates to* F. Thus,

`(%which () %true) => ()` true.

42.2 Predicates

More interesting goals are created by applying a special kind of Schelog object called a *predicate* (or *relation*) to other Schelog objects. Schelog comes with some primitive predicates, such as the arithmetic operators `%=:=` and `%<`, standing for arithmetic "equal" and "less than" respectively. For example, the following are some goals involving these predicates:

```
(%which () (%=:= 1 1))  => ()true
(%which () (%< 1 2))    => ()true
(%which () (%=:= 1 2))  => #f
(%which () (%< 1 1))    => #f
```

Other arithmetic predicates are `%>` ("greater than"), `%<=` ("less than or equal"), `%>=` ("greater than or equal"), and `%=/=` ("not equal").

Schelog predicates are not to be confused with conventional Scheme predicates (such as `<` and `=`). Schelog predicates, when applied to arguments, produce goals that may either succeed or fail. Scheme predicates, when applied to arguments, yield a boolean value. Henceforth, we will use the term "predicate" to mean Schelog predicates. Conventional predicates will be explicitly called "Scheme predicates".

42.2.1 Predicates introducing facts

Users can create their own predicates using the Schelog form `%rel`. For example, the following code defines a predicate `%knows`:

```
(define %knows
  (%rel ()
    (('Odysseus 'TeX))
    (('Odysseus 'Scheme))
    (('Odysseus 'Prolog))
    (('Odysseus 'Penelope))
    (('Penelope 'TeX))
    (('Penelope 'Prolog))
    (('Penelope 'Odysseus))
    (('Telemachus 'TeX))
    (('Telemachus 'calculus))))
```

The expression has the expected meaning. Each *clause* in the `%rel` establishes a *fact*: Odysseus knows TeX, Telemachus knows calculus, etc. In general, if we apply the predicate to the arguments in any one of its clauses, we will get a successful goal. Thus, since `%knows` has a clause that reads `(('Odysseus 'TeX))`, the goal `(%knows 'Odysseus 'TeX)` will be true.

We can now get answers for the following types of queries:

```
(%which () (%knows 'Odysseus 'TeX))       => ()true
(%which () (%knows 'Telemachus 'Scheme))  => #f
```

42.2.2 Predicates with rules

Predicates can be more complicated than the above recitation of facts. The predicate clauses can be *rules*, e.g.

```
(define %computer-literate
  (%rel (person)
    ((person) (%knows person 'TeX)
              (%knows person 'Scheme))
    ((person) (%knows person 'TeX)
              (%knows person 'Prolog))))
```

This defines the predicate `%computer-literate` in terms of the predicate `%knows`. In effect, a person is defined as computer-literate if they know TeX and Scheme, *or* TeX and Prolog.

Note that this use of `%rel` employs a local *logic variable* called person. In general, a `%rel`-expression can have a list of symbols as its second subform. These name new logic variables that can be used within the body of the `%rel`. The following query can now be answered:

`(%which () (%computer-literate 'Penelope))` => `()` true

Since Penelope knows TeX and Prolog, she is computer-literate.

42.2.3 Solving goals

The above queries are yes/no questions. Logic programming allows more: We can formulate a goal with *uninstantiated* logic variables and then ask the querying process to provide, if possible, values for these variables that cause the goal to succeed. For instance, the query:

```
(%which (what)
  (%knows 'Odysseus what))
```

asks for an instantiation of the logic variable `what` that satisfies the goal `(%knows 'Odysseus what)`. In other words, we are asking, "What does Odysseus know?".

Note that this use of `%which`, like `%rel` in the definition of `%computer-literate`, uses a local logic variable `what`. In general, the second subform of `%which` can be a list of local logic variables. The `%which`-query returns an answer that is a list of bindings, one for each logic variable mentioned in its second subform. Thus,

```
(%which (what)
  (%knows 'Odysseus what)) => ((what TeX))
```

But that is not all that Odysseus knows. Schelog provides a zero-argument procedure called `%more` that *retries* the goal in the last `%which`-query for a different solution.

```
(%more) => ((what Scheme))
```

We can keep asking for more solutions:

```
(%more) => ((what Prolog))
(%more) => ((what Penelope))
(%more) => #f
```

The final `#f` shows that there are no more solutions. This is because there are no more clauses in the `%knows` predicate that list Odysseus as knowing anything else.

It is now clear why `()` true was the right choice for truth in the previous `%which`-queries that had no logic variables. `%which` returns a list of bindings for true goals: the list is empty when there are no variables.

42.2.4 Asserting extra clauses

We can add more clauses to a predicate after it has already been defined via `%rel`. Schelog provides the `%assert` form for this purpose.

```
(%assert %knows ()
  (('Odysseus 'archery)))
```

tacks on a new clause at the end of the existing clauses of the `%knows` predicate. Now, the query:

```
(%which (what)
  (%knows 'Odysseus what))
```

gives TeX, Scheme, Prolog, and Penelope, as before, but a subsequent `(%more)` yields a new result: archery. The Schelog form `%assert-a` is similar to `%assert` but adds clauses *before* any of the current clauses.

Both `%assert` and `%assert-a` assume that the variable they are adding to already names a predicate, presumably defined using `%rel`. In order to allow defining a predicate entirely through `%assert`, Schelog provides an empty predicate value `%empty-rel`. `%empty-rel` takes any number of arguments and always fails. Here is a typical use of the `%empty-rel` and `%assert` combination:

```
(define %parent %empty-rel)
(%assert %parent ()
  (('Laertes 'Odysseus)))
(%assert %parent ()
  (('Odysseus 'Telemachus))
  (('Penelope 'Telemachus)))
```

Schelog does not provide a predicate for *retracting* assertions since we can keep track of older versions of predicates using conventional Scheme features such as `let` and `set!`.

42.2.5 Local variables

The local logic variables of `%rel` and `%which`-expressions are in reality introduced by the Schelog syntactic form called `%let`. `%let` introduces new lexically scoped logic variables. Supposing, instead of

```
(%which (what)
  (%knows 'Odysseus what))
```

we had asked

```
(%let (what)
  (%which ()
    (%knows 'Odysseus what)))
```

This query, too, succeeds five times, since Odysseus knows five things. However, `%which` emits bindings only for the local variables that it introduces. Thus, this query emits `()` true five times before `(%more)` finally returns `#f`.

42.3 Using conventional Scheme expressions

The arguments of Schelog predicates can be any Scheme objects. In particular, composite structures such as lists, vectors and strings can be used, as also Scheme expressions using the full array of Scheme's construction and decomposition operators. For instance, consider the following goal:

```
(%member x '(1 2 3))
```

Here, `%member` is a predicate, `x` is a logic variable, and `'(1 2 3)` is a structure. Given a suitably intuitive definition for `%member`, the above goal succeeds for `x` = 1, 2, and 3. Here is a possible definition of `%member`:

```
(define %member
  (%rel (x y xs)
    ((x (cons x xs)))
    ((x (cons y xs))
      (%member x xs))))
```

`%member` is defined with three local variables: `x`, `y`, `xs`. It has two clauses, identifying the two ways of determining membership. The first clause of `%member` states a fact: For any `x`, `x` is a member of a list whose head is also `x`. The second clause of `%member` is a rule: `x` is a member of a list if we can show that it is a member of the *tail* of that list. In other words, the original `%member` goal is translated into a *sub goal*, which is also a `%member` goal.

Note that the variable `y` in the definition of `%member` occurs only once in the second clause. As such, it doesn't need you to make the effort of naming it. Names help only in matching a second occurrence to a first. Schelog lets you use the expression `(_)` to denote an anonymous variable; i.e. `_` is a thunk that generates a fresh anonymous variable at each call. The predicate `%member` can be rewritten in the following way:

```
(define %member
  (%rel (x xs)
    ((x (cons x (_))))
    ((x (cons (_) xs))
      (%member x xs))))
```

42.3.1 Constructors

We can use constructors, i.e. Scheme procedures for creating structures, to simulate data types in Schelog. For instance, let's define a natural-number data-type where `0` denotes zero, and `(succ x)` denotes the natural number whose immediate predecessor is `x`. The constructor `succ` can be defined in Scheme as:

```
(define succ
  (lambda (x)
    (vector 'succ x)))
```

Addition and multiplication can be defined as:

```
(define %add
  (%rel (x y z)
    ((0 y y))
```

```
    (((succ x) y (succ z))
      (%add x y z))))
(define %times
  (%rel (x y z z1)
    ((0 y 0))
    (((succ x) y z)
      (%times x y z1)
      (%add y z1 z))))
```

We can do a lot of arithmetic with this in place. For instance, the factorial predicate looks like:

```
(define %factorial
  (%rel (x y y1)
    ((0 (succ 0)))
    (((succ x) y)
      (%factorial x y1)
      (%times (succ x) y1 y))))
```

42.3.2 `%is`

The above is a very inefficient way to do arithmetic, especially when the underlying language Scheme offers excellent arithmetic facilities, including a comprehensive numeric tower and exact rational arithmetic. One problem with using Scheme calculations directly in Schelog clauses is that the expressions used may contain logic variables that need to be dereferenced. Schelog provides the predicate `%is` that takes care of this. The goal

```
(%is X E)
```

unifies `X` with the value of `E` considered as a Scheme expression. `E` can have logic variables, but usually they should at least be bound, as unbound variables may not be palatable values to the Scheme operators used in `E`. We can now directly use the numbers of Scheme to write a more efficient %factorial predicate:

```
(define %factorial
   (%rel (x y x1 y1)
     ((0 1))
     ((x y) (%is x1 (- x 1))
            (%factorial x1 y1)
            (%is y (* y1 x)))))
```

A price that this efficiency comes with is that we can use `%factorial` only with its first argument already instantiated. In many cases, this is not an unreasonable constraint. In fact, given this limitation, there is nothing to prevent us from using Scheme's factorial directly:

```
(define %factorial
  (%rel (x y)
    ((x y)
      (%is y (scheme-factorial x)))))
```

or better yet, inline any calls to `%factorial` with `%is` -expressions calling scheme-factorial, where the latter is defined in the usual manner:

```
(define scheme-factorial
  (lambda (n)
    (if (= n 0)
        1
        (\* n (factorial (- n 1))))))
```

42.3.3 Lexical scoping

One can use Scheme's lexical scoping to enhance predicate definitions. Here is a list-reversal predicate defined using a hidden auxiliary predicate:

```
(define %reverse
  (letrec ((revaux
             (%rel (x y z w)
               (('() y y))
               (((cons x y) z w)
                 (revaux y (cons x z) w)))))
    (%rel (x y)
      ((x y) (revaux x '() y)))))
```

`(revaux X Y Z)` uses `Y` as an accumulator for reversing `X` into `Z`. `Y` starts out as `()`. Each head of `X` is consed on to `Y`. Finally, when `X` has wound down to `()`, `Y` contains the reversed list and can be returned as `Z`. Here, `revaux` is used purely as a helper predicate for `%reverse`, and so it can be concealed within a lexical contour. We use `letrec` instead of `let` because `revaux` is a recursive procedure.

42.3.4 Type predicates

Schelog provides a couple of predicates that let the user probe the type of objects. The goal

```
(%constant X)
```

succeeds if `X` is an *atomic* object, i.e. not a list or vector. The predicate `%compound`, the negation of `%constant`, checks if its argument is indeed a list or a vector.

The above are merely the logic-programming equivalents of corresponding Scheme predicates. Users can use the predicate `%is` and Scheme predicates to write more type checks in Schelog. Thus, to test if `X` is a string, the following goal could be used:

```
(%is #t (string? X))
```

User-defined Scheme predicates, in addition to primitive Scheme predicates, can thus be imported.

42.4 Backtracking

It is helpful to go into the following evaluation in a little more detail:

```
(%which ()
  (%computer-literate 'Penelope))
  => () true
```

The starting goal is:

```
G0 = (%computer-literate Penelope)
```

Schelog tries to match this with the head of the first clause of `%computer-literate` . It succeeds, generating a binding `(person Penelope)` . But this means it now has two new goals — *subgoals* — to solve. These are the goals in the body of the matching clause, with the logic variables substituted by their instantiations:

```
G1 = (%knows Penelope TeX)
G2 = (%knows Penelope Scheme)
```

For `G1` , Schelog attempts matches with the clauses of `%knows` , and succeeds at the fifth try. There are no subgoals in this case, because the bodies of these "fact" clauses are empty, in contrast to the "rule" clauses of `%computer` -literate. Schelog then tries to solve `G2` against the clauses of `%knows` , and since there is no clause stating that Penelope knows Scheme, it fails.

All is not lost though. Schelog now *backtracks* to the goal that was solved just before: `G1` . It *retries* G1, i.e. tries to solve it in a different way. This entails searching down the previously unconsidered `%knows` clauses for `G1` , i.e. the sixth onwards. Obviously, Schelog fails again, because the fact that Penelope knows TeX occurs only once.

Schelog now backtracks to the goal before `G1` , i.e. `G0` . We abandon the current successful match with the first clause-head of `%computer-literate` , and try the next clause-head. Schelog succeeds, again producing a binding `(person Penelope)` , and two new subgoals:

```
G3 = (%knows Penelope TeX)
G4 = (%knows Penelope Prolog)
```

It is now easy to trace that Schelog finds both `G3` and `G4` to be true. Since both of `G0` 's subgoals are true, `G0` is itself considered true. And this is what Schelog reports. The interested reader can now trace why the following query has a different denouement:

```
(%which ()
  (%computer-literate 'Telemachus))
=> #f
```

42.5 Unification

When we say that a goal matches with a clause-head, we mean that the predicate and argument positions line up. Before making this comparison, Schelog dereferences all already bound logic variables. The resulting structures are then compared to see if they are recursively identical. Thus, `1` unifies with `1` , and `(list 1 2)` with `'(1 2)` ; but `1` and `2` do not unify, and neither do `'(1 2)` and `'(1 3)` .

In general, there could be quite a few uninstantiated logic variables in the compared objects. Unification will then endeavor to find the most natural way of binding these variables so that we arrive at structurally identical objects. Thus, `(list x 1)` , where `x` is an unbound logic variable, unifies with `'(0 1)` , producing the binding `(x 0)` .

Unification is thus a goal, and Schelog makes the unification predicate available to the user as `%=` , e.g.

```
(%which (x)
  (%= (list x 1) '(0 1))
=> ((x 0))
```

Schelog also provides the predicate `%/=`, the *negation* of `%=`. `(%/= X Y)` succeeds if and only if `X` does *not* unify with `Y`.

Unification goals constitute the basic subgoals that all Schelog goals devolve to. A goal succeeds because all the eventual unification subgoals that it decomposes to in at least one of its subgoal-branching succeeded. It fails because every possible subgoal-branching was thwarted by the failure of a crucial unification subgoal.

Going back to the example in the section on backtracking, the goal `(%computer-literate 'Penelope)` succeeds because (a) it unified with `(%computer-literate person)`; and then (b) with the binding `(person Penelope)` in place, `(%knows person 'TeX)` unified with `(%knows 'Penelope 'TeX)` and `(%knows person 'Prolog)` unified with `(%knows 'Penelope 'Prolog)`.

In contrast, the goal `(%computer-literate 'Telemachus)` fails because, with `(person Telemachus)`, the subgoals `(%knows person 'Scheme)` and `(%knows person 'Prolog)` have no facts they can unify with.

The "occurs check"

A robust unification algorithm uses the *occurs check*, which ensures that a logic variable isn't bound to a structure that contains itself. Not performing the check can cause the unification to go into an infinite loop in some cases. On the other hand, performing the occurs check greatly increases the time taken by unification, even in cases that wouldn't require the check.

Schelog uses the global variable `*schelog-use-occurs-check?*` to decide whether to use the occurs check. By default, this variable is `#f`, i.e. Schelog disables the occurs check. To enable the check,

```
(set! *schelog-use-occurs-check?* #t)
```

42.6 Conjuctions and disjunctions

Goals may be combined using the forms `%and` and `%or` to form compound goals. For `%not`, see the section on "Negation as failure". Here is an example:

```
(%which (x)
  (%and (%member x '(1 2 3))
        (%< x 3)))
```

gives solutions for `x` that satisfy both the argument goals of the `%and`, i.e. `x` should both be a member of `'(1 2 3)` *and* be less than `3`. The first solution is

```
((x 1))
```

Typing `(%more)` gives another solution:

```
((x 2))
```

There are no more solutions, because `(x 3)` satisfies the first, but not the second goal. Similarly, the query

```
(%which (x)
  (%or (%member x '(1 2 3))
       (%member x '(3 4 5))))
```

lists all `x` that are members of either list.

```
           ((x 1))
(%more) => ((x 2))
(%more) => ((x 3))
(%more) => ((x 3))
(%more) => ((x 4))
(%more) => ((x 5))
```

Here, `((x 3))` is listed twice. We can rewrite the predicate `%computer-literate` from section "Predicates with rules" using `%and` and `%or` :

```
(define %computer-literate
  (%rel (person)
    ((person)
      (%or (%and (%knows person 'TeX)
                 (%knows person 'Scheme))
           (%and (%knows person 'TeX)
                 (%knows person 'Prolog))))))
```

Or, more succinctly:

```
(define %computer-literate
  (%rel (person)
    ((person)
      (%and (%knows person 'TeX)
            (%or (%knows person 'Scheme)
                 (%knows person 'Prolog))))))
```

We can even dispense `%rel` altogether, turning `%computer-literate` into a conventional Scheme predicate definition:

```
(define %computer-literate
  (lambda (person)
    (%and (%knows person 'TeX)
          (%or (%knows person 'Scheme)
               (%knows person 'Prolog)))))
```

42.7 Manipulating logic variables

Schelog provides special predicates for probing logic variables, without risking them getting bound.

Checking for variables

The goal

```
(%== X Y)
```

succeeds if `X` and `Y` are *identical* objects. This is not quite the unification predicate `%=` , for `%==` doesn't touch unbound objects the way `%=` does. For instance, `%==` will not equate an unbound logic variable with a bound one, nor will it equate two unbound logic variables unless they are the *same* variable.

The predicate `%/==` is the negation of `%==` .

The goal

```
(%var X)
```

succeeds if `X` isn't completely bound, i.e. it has at least one unbound logic variable in its innards.

The predicate `%nonvar` is the negation of `%var`.

Preserving variables

Schelog lets the user protect a term with variables from unification by allowing that term to be treated as a completely bound object. The predicates provided for this purpose are `%freeze`, `%melt`, `%melt-new`, and `%copy`.

The goal

```
(%freeze S F)
```

unifies `F` to the frozen version of `S`. Any lack of bindings in `S` are preserved no matter how much you toss `F` about.

The goal

```
(%melt F S)
```

retrieves the object frozen in `F` into `S`.

The goal

```
(%melt-new F S)
```

is similar to `%melt`, except that when `S` is made, the unbound variables in `F` are replaced by brand-new unbound variables.

The goal

```
(%copy S C)
```

is an abbreviation for `(%freeze S F)` followed by `(%melt-new F C)`.

42.8 The cut (!)

The cut `(called !)`` is a special goal that is used to prune backtracking options. Like the %true`‘ goal, the cut goal too succeeds, when accosted by the Schelog subgoaling engine. However, when a further subgoal down the line fails, and time comes to retry the cut goal, Schelog will refuse to try alternate clauses for the predicate in whose definition the cut occurs. In other words, the cut causes Schelog to commit to all the decisions made from the time that the predicate was selected to match a subgoal till the time the cut was satisfied.

For example, consider again the `%factorial` predicate, as defined in the section on `%is`:

```
(define %factorial
  (%rel (x y x1 y1)
    ((0 1))
    ((x y) (%is x1 (- x 1))
           (%factorial x1 y1)
           (%is y (\* y1 x)))))
```

Clearly,

```
(%which ()
  (%factorial 0 1))
  => () true
(%which (n)
  (%factorial 0 n))
  => ((n 1))
```

But what if we asked for `(%more)` for either query? Backtracking will try the second clause of `%factorial`, and sure enough the clause-head unifies, producing binding `(x 0)`. We now get three subgoals. Solving the first, we get `(x1 -1)`, and then we have to solve `(%factorial -1 y1)`. It is easy to see there is no end to this, as we fruitlessly try to get the factorials of numbers that get more and more negative.

If we placed a cut at the first clause:

```
...
((0 1) !)
...
```

the attempt to find more solutions for `(%factorial 0 1)` is stopped immeditately.

Calling `%factorial` with a *negative* number would still cause an infinite loop. To take care of that problem as well, we use another cut:

```
(define %factorial
  (%rel (x y x1 y1)
    ((0 1) !)
    ((x y) (< x 0) ! %fail)
    ((x y) (%is x1 (- x 1))
           (%factorial x1 y1)
           (%is y (\* y1 x)))))
```

Using *raw* cuts as above can get very confusing. For this reason, it is advisable to use it hidden away in well-understood abstractions. Two such common abstractions are the conditional and negation.

Conditional goals

An “if … then … else …” predicate can be defined as follows

```
(define %if-then-else
  (%rel (p q r)
    ((p q r) p ! q)
    ((p q r) r)))
```

Note that for the first time we have predicate arguments that are themselves goals.

Consider the goal

```
G0 = (%if-then-else Gbool Gthen Gelse)
```

We first unify `G0` with the first clause-head, giving `(p Gbool)`, `(q Gthen)`, `(r Gelse)`. `Gbool` can now either succeed or fail.

Case 1: If `Gbool` fails, backtracking will cause the `G0` to unify with the second clause-head. `r` is bound to `Gelse`, and so `Gelse` is tried, as expected.

Case 2: If `Gbool` succeeds, the cut commits to this clause of the `%if-then-else`. We now try `Gthen`. If `Gthen` should now fail — or even if we simply retry for more solutions — we are guaranteed that the second clause-head will not be tried. If it were not for the cut, `G0` would attempt to unify with the second clause-head, which will of course succeed, and `Gelse` *will* be tried.

Negation as failure

Another common abstraction using the cut is *negation*. The negation of goal `G` is defined as `(%not G)`, where the predicate `%not` is defined as follows:

```
(define %not
  (%rel (g)
    ((g) g ! %fail)
    ((g) %true)))
```

Thus, `g`'s negation is deemed a failure if `g` succeeds, and a success if `g` fails. This is of course confusing goal failure with falsity. In some cases, this view of negation is actually helpful.

42.9 Set predicates

The goal

```
(%bag-of X G Bag)
```

unifies with `Bag` the list of all instantiations of `X` for which `G` succeeds. Thus, the following query asks for all the things known, i.e. the collection of things such that someone knows them:

```
(%which (things-known)
  (%let (someone x)
    (%bag-of x (%knows someone x) things-known)))
=> ((things-known
      (TeX Scheme Prolog
       Penelope TeX Prolog
       Odysseus TeX calculus)))
```

This is the only solution for this goal:

```
(%more) =>#f
```

Note that some things, e.g. TeX, are enumerated more than once. This is because more than one person knows TeX. To remove duplicates, use the predicate `%set-of` instead of `%bag-of`:

```
(%which (things-known)
  (%let (someone x)
    (%set-of x (%knows someone x) things-known)))
=> ((things-known
      (TeX Scheme Prolog
       Penelope Odysseus calculus)))
```

In the above, the free variable someone in the `%knows-goal` is used as if it were existentially quantified. In contrast, Prolog's versions of `%bag-of` and `%set-of` fix it for each solution of the set-predicate goal. We can do it too with some additional syntax that identifies the free variable, for instance:

```
(%which (someone things-known)
  (%let (x)
    (%bag-of x (%free-vars (someone)
                 (%knows someone x)) things-known)))
=> ((someone Odysseus)
    (things-known
      (TeX Scheme Prolog Penelope)))
```

The bag of things known by *one* someone is returned. That someone is Odysseus. The query can be retried for more solutions, each listing the things known by a different someone:

```
(%more) => ((someone Penelope)
            (things-known
              (TeX Prolog Odysseus)))
(%more) => ((someone Telemachus)
            (things-known
              (TeX calculus)))
(%more) => #f
```

Schelog also provides two variants of these set predicates: `%bag-of-1` and `%set-of-1`. These act like `%bag-of` and `%set-of` but fail if the resulting bag or set is empty.

42.10 API

(%/= *E1 E2*) predicate

`%/=` is the negation of predicate `%=`. The goal `(%/= E1 E2)` succeeds if `E1` can not be unified with `E2`.

(%/== *E1 E2*) predicate

`%/==` is the negation of predicate `%==`. The goal `(%/== E1 E2)` succeeds if `E1` and `E2` are not identical.

(%< *E1 E2*) predicate

The goal `(%< E1 E2)` succeeds if `E1` and `E2` are bound to numbers and `E1` is less than `E2`.

(%<= *E1 E2*) predicate

The goal `(%<= E1 E2)` succeeds if `E1` and `E2` are bound to numbers and `E1` is less than or equal to `E2`.

(%= *E1 E2*) predicate

The goal `(%= E1 E2)` succeeds if `E1` can be unified with `E2`. Any resulting bindings for logic variables are kept.

(%=/= *E1 E2*) predicate

The goal `(%=/= E1 E2)` succeeds if `E1` and `E2` are bound to numbers and `E1` is not equal to `E2`.

(%=:= *E1 E2*) predicate

The goal `(%=:= E1 E2)` succeeds if `E1` and `E2` are bound to numbers and `E1` is equal to `E2`.

(%== *E1 E2*) predicate

The goal `(%== E1 E2)` succeeds if `E1` is *identical* to `E2`. They should be structurally equal. If containing logic variables, they should have the same variables in the same position. Unlike a `%=`-call, this goal will not bind any logic variables.

(%> *E1 E2*) predicate

The goal `(%> E1 E2)` succeeds if `E1` and `E2` are bound to numbers and `E1` is greater than `E2`.

(%>= *E1 E2*) predicate

The goal `(%>= E1 E2)` succeeds if `E1` and `E2` are bound to numbers and `E1` is greater than or equal to `E2`.

(%and *G* ...) syntax

The goal `(%and G ...)` succeeds if all the goals `G ...` succeed.

(%append *E1 E2 E3*) predicate

The goal `(%append E1 E2 E3)` succeeds if `E3` is unifiable with the list obtained by appending `E1` and `E2`.

(%assert *Pname (V ...) C ...*) syntax

The form `(%assert Pname (V ...) C ...)` adds the clauses *C* ... to the end of the predicate that is the value of the Scheme variable *Pname*. The variables *V* ... are local logic variables for *C* ....

(%assert-a *Pname (V ...) C ...*) syntax

The form `(%assert-a Pname (V ...) C ...)` adds the clauses *C* ... to the front of the predicate that is the value of the Scheme variable *Pname*. The variables *V* ... are local logic variables for *C* ....

(%bag-of *E1 G E2*) predicate

The goal `(%bag-of E1 G E2)` unifies with `E2` the *bag* (multiset) of all the instantiations of `E1` for which goal `G` succeeds.

(%bag-of-1 *E1 G E2*) predicate

The goal `(%bag-of E1 G E2)` unifies with `E2` the *bag* (multiset) of all the instantiations of `E1` for which goal `G` succeeds. `%bag-of-1` fails if the bag is empty.

(%compound *E*) predicate

The goal `(%compound E)` succeeds if `E` is a non-atomic structure, i.e. a vector or a list.

(%constant *E*) predicate

The goal `(%constant E)` succeeds if `E` is an atomic object, i.e. not a vector and a list.

(%copy *F S*) predicate

The goal `(%copy F S)` unifies with `S` a copy of the frozen structure in `F`.

(%empty-rel *E* ...) predicate

The goal `(%empty-rel E ...)` always fails. The value `%empty-rel` is used as a starting value for predicates that can later be enhanced with `%assert` and `%assert-a`.

%fail goal

The goal `%fail` always fails.

(%free-vars *(V ...) G*) syntax

The form `(%free-vars (V ...) G)` identifies the occurrences of the variables `V ...` in goal `G` as free. It is used to avoid existential quantification in calls to set predicates such as `%bag-of`, `%set-of`, etc.

(%freeze *S F*) predicate

The goal `(%freeze S F)` unifies with `F` a new frozen version of the structure in `S`. Freezing implies that all the unbound variables are preserved. `F` can henceforth be used as *bound* object with no fear of its variables getting bound by unification.

(%if-then-else *G1 G2 G3*) predicate

The goal `(%if-then-else G1 G2 G3)` tries `G1` first: if it succeeds, tries `G2`; if not, tries `G3`.

(%is *E1 E2*) predicate

The goal `(%is E1 E2)` unifies with `E1` the result of evaluating `E2` as a Scheme expression. `E2` may contain logic variables, which are dereferenced automatically. Fails if `E2` contains unbound logic variables. Unlike other predicates, `%is` is implemented as syntax and not a procedure.

(%let *(V ...) E ...*) syntax

The form `(%let (V ...) E ...)` introduces `V ...` as lexically scoped logic variables to be used in `E ...`.

(%melt *F S*) predicate

The goal `(%melt F S)` unifies `S` with the thawed (original) form of the frozen structure in `F`.

(%melt-new *F S*) predicate

The goal `(%melt-new F S)` unifies `S` with a thawed *copy* of the frozen structure in `F`. This means new logic variables are used for unbound logic variables in `F`.

(%member *E1 E2*) predicate

The goal `(%member E1 E2)` succeeds if `E1` is a member of the list in `E2`.

(%nonvar *E*) predicate

`%nonvar` is the negation of `%var`. The goal `(%nonvar E)` succeeds if `E` is completely instantiated, i.e. it has no unbound variables in it.

(%not *G*) predicate

The goal `(%not G)` succeeds if `G` fails.

(%more) procedure

The thunk `%more` produces more instantiations of the variables in the most recent `%which`-form that satisfy the goals in that `%which`-form. If no more solutions can be found, `%more` returns `#f`.

(%or *G ...*) syntax

The goal `(%or G ...)` succeeds if one of `G ...`, tried in that order, succeeds.

(%rel (V ...) C ...) syntax

The form `(%rel (V ...) C ...)` creates a predicate object. Each clause `C` is of the form `((E ...) G ...)`, signifying that the goal created by applying the predicate object to anything that matches `(E ...)` is deemed to succeed if all the goals `G ...` can, in their turn, be shown to succeed.

(%repeat) predicate

The goal `(%repeat)` always succeeds (even on retries). Used for failure-driven loops.

schelog-use-occurs-check? object

If the global flag `*schelog-use-occurs-check?*` is false (the default), unification will not use the occurs check. If it is true, the occurs check is enabled.

(%set-of *E1 G E2*) predicate

The goal `(%set-of E1 G E2)` unifies with `E2` the *set* of all the instantiations of `E1` for which goal `G` succeeds.

(%set-of-1 *E1 G E2*) predicate

The goal `(%set-of-1 E1 G E2)` unifies with `E2` the *set* of all the instantiations of `E1` for which goal `G` succeeds. The predicate fails if the set is empty.

%true goal

The goal `%true` succeeds. Fails on retry.

(%var *E*) predicate

The goal `(%var E)` succeeds if `E` is not completely instantiated, i.e. it has at least one unbound variable in it.

(%which *(V ...) G ...*) syntax

The form `(%which (V ...) G ...)` returns an instantiation of the variables `V ...` that satisfies all of `G ...`. If `G ...` cannot be satisfied, `%which` returns `#f`. Calling the thunk `%more` produces more instantiations, if available.

(_) procedure

A thunk that produces a new logic variable. Can be used in situations where we want a logic variable but don't want to name it. `%let` , in contrast, introduces new lexical names for the logic variables it creates.

---

43 LispKit Queue

Library `(lispkit queue)` provides an implementation for mutable queues, i.e. mutable FIFO buffers.

queue-type-tag [object]
Symbol representing the `queue` type. The `type-for` procedure of library `(lispkit type)` returns this symbol for all queue objects.

(make-queue) [procedure]
Returns a new empty queue.

(queue *x* ...) [procedure]
Returns a new queue with *x* on its first position followed by the remaining parameters.

```
(dequeue! (queue 1 2 3))  ⇒  1
```

(queue? *obj*) [procedure]
Returns `#t` if *obj* is a queue; otherwise `#f` is returned.

(queue-empty? *q*) [procedure]
Returns `#t` if queue *q* is empty.

(queue-size *q*) [procedure]
Returns the size of queue *q*, i.e. the number of elements buffered in *q*.

(queue=? *q1 q2*) [procedure]
Returns `#t` if queue *q1* has the exact same elements in the same order like queue *q2*; otherwise, `#f` is returned.

(enqueue! *q x*) [procedure]
Inserts element *x* at the end of queue *q*.

(queue-front *q*) [procedure]
Returns the first element in queue *q*. If the queue is empty, an error is raised.

(dequeue! *q*) [procedure]
Removes the first element from queue *q* and returns its value.

```
(define q (make-queue))
(enqueue! q 1)
(enqueue! q 2)
(dequeue! q)     ⇒  1
(queue-front q) ⇒  2
(queue-size q)  ⇒  1
```

(queue-clear! *q*) [procedure]
Removes all elements from queue *q*.

(queue-copy *q*) [procedure]
Returns a copy of queue *q*.

(queue->list *q*) procedure

Returns a list consisting of all elements in queue *q* in the order they were inserted, i.e. starting with the first element.

```
(define q (make-queue))
(enqueue! q 1)
(enqueue! q 2)
(enqueue! q 3)
(queue->list q)  ⇒  (1 2 3)
```

(list->queue *l*) procedure

Returns a new queue consisting of the elements of list *l*. The first element in *l* will become the front element of the new queue that is returned.

```
(dequeue! (list->queue '(1 2 3)))  ⇒  1
```

(list->queue! *s l*) procedure

Inserts the elements of list *l* into queue *q* in the order they appear in the list.

```
(define q (list->queue '(1 2 3)))
(list->queue! q '(4 5 6))
(queue->list q)  ⇒  (1 2 3 4 5 6)
```

44 LispKit Record

Library `(lispkit record)` implements extensible record types for LispKit which are compatible with R7RS and SRFI 131. A record provides a simple and flexible mechanism for building structures with named components wrapped in distinct types.

44.1 Declarative API

`record-type` syntax is used to introduce new *record types* in a declarative fashion. Like other definitions, `record-type` can either appear at the outermost level or locally within a body. The values of a *record type* are called *records* and are aggregations of zero or more fields, each of which holds a single location. A predicate, a constructor, and field accessors and mutators are defined for each record type.

44.1.1 Syntax

(define-record-type *<type>* *<constr>* *<pred>* *<field>* …) syntax

<type> defines the name of the new record type and potentially the supertype if the new type is an extension of an existing record type. Thus, it has one of the two possible forms, where *<type name>* is an identifier and *<supertype>* an arbitrary expression evaluating to a record type descriptor:

<type name>, or
(*<type name>* *<supertype>*).

The *<constructor>* is of one of the two possible forms:

<constructor name>, or
(<constructor name> <field name> …_)

When a constructor is of the form (*<constructor name>* *<field name>* …), the following holds:

- Each of the field names can be either a field name declared in the same `define-record-type` form, or any of its ancestors' field names.
- If the record definition contains the same field name as one of its ancestors, it shadows the ancestor's field name for the purposes of the constructor. The constructor's argument initializes the child's slot, and the ancestor's slot of the same name is left uninitialized.
- It is an error if the same identifier appears more than once in the field names of the constructor spec.

<pred> is the identifier denoting the type predicate that recognizes instances of the new record type and its subtypes. Each *<field>* is either of the form:

(*<field name>* *<accessor name>*), or
(*<field name>* *<accessor name>* *<modifier name>*).

It is an error for the same identifier to occur more than once as a field name. It is also an error for the same identifier to occur more than once as an accessor or mutator name.

The `define-record-type` construct is generative: each use creates a new record type that is distinct from all existing types, including the predefined types and other record types - even record types of the same name or structure.

An instance of `define-record-type` is equivalent to the following definitions:

- <*type name*> is bound to a representation of the record type itself. If a <supertype> expression is specified, then it must evaluate to a record type descriptor that serves as the parent record type for the record type being defined.
- <*constructor name*> is bound to a procedure that takes as many arguments as there are <*field name*> elements in the (<*constructor name*> ...) subexpression and returns a new record of type <*name*>. Fields whose names are listed with <*constructor name*> have the corresponding argument as their initial value. The initial values of all other fields are unspecified. It is an error for a field name to appear in <*constructor*> but not as a <*field name*>.
- <*pred*> is bound to a predicate that returns `#t` when given a value returned by the procedure bound to <*constructor name*> or any constructor of a subtype. It returns `#f` for everything else.
- Each <*accessor name*> is bound to a procedure that takes a record of type <*name*> and returns the current value of the corresponding field. It is an error to pass an accessor a value which is not a record of the appropriate type.
- Each <*modifier name*> is bound to a procedure that takes a record of type <*name*> and a value which becomes the new value of the corresponding field. It is an error to pass a modifier a first argument which is not a record of the appropriate type.

44.1.2 Examples

The following record type definition:

```
(define-record-type <pare>
  (kons x y)
  pare?
  (x kar set-kar!)
  (y kdr))
```

defines `kons` to be a constructor, `kar` and `kdr` to be accessors, `set-kar!` to be a modifier, and `pare?` to be a type predicate for instances of `<pare>`.

```
(pare? (kons 1 2))         ⇒  #t
(pare? (cons 1 2))         ⇒  #f
(kar (kons 1 2))           ⇒  1
(kdr (kons 1 2))           ⇒  2
(let ((k (kons 1 2)))
  (set-kar! k 3) (kar k)) ⇒  3
```

Below is another example showcasing how record types can be extended. First a new record type `<point>` is defined:

```
(define-record-type <point>
  (make-point x y)
  point?
  (x point-x set-point-x!)
  (y point-y set-point-y!))
```

Next, a new record type `<color-point>` is defined to extend record type `<point>`:

```
(define-record-type (<color-point> <point>)
  (make-color-point x y color)
  color-point?
  (color point-color))
```

The following transcript shows that `<color-point>` objects are also considered `<point>` objects which inherit all fields from `<point>` :

```
(define cp (make-color-point 12 3 red))
(color-point? cp)     ⇒  #t
(point? cp)           ⇒  #t
(color-point?
  (make-point 1 2))   ⇒  #f
(point-x cp)          ⇒  12
(set-point-x! cp 1)
(point-x cp)          ⇒  1
(point-color cp)      ⇒  #<color 1.0 0.0 0.0>
```

44.2 Procedural API

Besides the syntactical `define-record-type` abstraction for defining record types in a declarative fashion, LispKit provides a low-level, procedural API for creating and instantiating records and record types. Record types are represented in form of *record type descriptor* objects which itself are records.

(record? *obj*) procedure

Returns `#t` if *obj* is a record of any type; returns `#f` otherwise.

(record-type? *obj*) procedure

Returns `#t` if *obj* is a record type descriptor; returns `#f` otherwise.

(record-type *obj*) procedure

Returns the record type descriptor for objects *obj* which are records; returns `#f` for all non-record values.

(make-record-type *name fields*) procedure
(make-record-type *name fields parent*)

Returns a record type descriptor which represents a new data type that is disjoint from all other types. *name* is a string which is only used for debugging purposes, such as the printed representation of a record of the new type. *fields* is a list of symbols naming the fields of a record of the new type. It is an error if the list contains duplicate symbols. *parent* refers to a record type descriptor for the supertype of the newly created record type. By default, *parent* is `#f`, i.e. the new record type does not have a parent type.

(record-type-name *rtd*) procedure

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

(record-type-field-names *rtd*) procedure
(record-type-field-names *rtd all?*)

Returns a list of the symbols naming the fields in members of the type represented by the record type descriptor *rtd*. The returned value is `equal?` to the *fields* argument given in the call to `make-record-type` that created the type represented by *rtd*. If boolean argument *all?* is true (default is false), then a list of all symbols, also defined by parent types of *rtd*, is returned.

(record-type-parent *rtd*) procedure

Returns a record type descriptor for the parent type of the record type represented by the record type descriptor *rtd*. `record-type-parent` returns `#f` if *rtd* does not have a parent type.

(record-type-tag *rtd*) procedure

Returns the type tag, i.e. an uninterned symbol, representing the type of records defined by *rtd*. The result of `record-type-tag` can be used together with procedure `type-of` of library `(lispkit type)`.

(make-record *rtd*) procedure

Returns an uninitialized instance of the record type for which *rtd* is the record type descriptor.

(record-constructor *rtd fields*) procedure

Returns a procedure for constructing new members of the type represented by the record type descriptor *rtd*. The returned procedure accepts exactly as many arguments as there are symbols in the given *fields* list; 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 *fields* are unspecified. It is an error if *fields* contain any duplicates or any symbols not in the *fields* list of the record type descriptor *rtd*.

(record-predicate *rtd*) procedure

Returns a procedure for testing membership in the type represented by the record type descriptor *rtd*. The returned procedure accepts exactly one argument and returns `#t` if the argument is a member of the indicated record type; it returns `#f` otherwise.

(record-field-accessor *rtd field*) procedure

Returns a procedure for reading the value of a particular field of a member of the type represented by the record type descriptor *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* in that record. The symbol *field* 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-field-mutator *rtd field*) procedure

Returns a procedure for writing the value of a particular field of a member of the type represented by the record type descriptor *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* in that record to contain the given value. The returned value of the modifier procedure is unspecified. The symbol *field* must be a member of the list of field names in the call to `make-record-type` that created the type represented by *rtd*.

45 LispKit Regexp

Library `(lispkit regexp)` provides an API for defining regular expressions and applying them to strings. Supported are both matching as well as search/replace.

45.1 Regular expressions

The regular expression syntax supported by this library corresponds to the one of `NSRegularExpression` of Apple's *Foundation* framework. This is also the origin of the documentation of this section.

45.1.1 Meta-characters

`\a` Match the *bell character* `#\alarm`

`\A` Match at the beginning of the input. Differs from `^` in that `\A` will not match after a new line within the input.

`\b` Outside of a [Set], match if the current position is a word boundary. Boundaries occur at the transitions between word (`\w`) and non-word (`\W`) characters, with combining marks ignored. Inside of a [Set], match a *backspace character* `#\backspace`.

`\B` Match if the current position is not a word boundary.

`\c`*X* Match a control-*X* character.

`\d` Match any character with the unicode general category of *Nd*, i.e. numbers and decimal digits.

`\D` Match any character that is not a decimal digit.

`\e` Match an *escape character* `#x1b`.

`\E` Terminates a `\Q ... \E` quoted sequence.

`\f` Match a *form feed character* `#\page`.

`\G` Match if the current position is at the end of the previous match.

`\n` Match a *line feed character* `#\newline`.

`\N{`*name*`}` Match the named unicode character.

`\p{`*prop*`}` Match any character with the specified unicode property. These are the most frequently used properties: `C` (Other), `Cc` (Control), `Cf` (Format), `Cn` (Unassigned), `Co` (Private use), `Cs` (Surrogate), `L` (Letter), `Ll` (Lower case letter), `Lm` (Modifier letter), `Lo` (Other letter), `Lt` (Title case letter), `Lu` (Upper case letter), `L&` (Ll, Lu, or Lt), `M` (Mark), `Mc` (Spacing mark), `Me` (Enclosing mark), `Mn` (Non-spacing mark), `N` (Number), `Nd` (Decimal number), `Nl` (Letter number), `No` (Other number), `P` (Punctuation), `Pc` (Connector punctuation), `Pd` (Dash punctuation), `Pe` (Close punctuation), `Pf` (Final punctuation), `Pi` (Initial punctuation), `Po` (Other punctuation), `Ps` (Open punctuation), `S` (Symbol), `Sc` (Currency symbol), `Sk` (Modifier symbol), `Sm` (Mathematical symbol), `So` (Other symbol), `Z` (Separator), `Zl` (Line separator), `Zp` (Paragraph separator), and `Zs` (Space separator).

\P{*prop*}	Match any character not having the specified unicode property.
\Q	Quotes all following characters until `\E`.
\r	Match a *carriage return character* `#\return`.
\s	Match a whitespace character. Whitespace is defined as `\t`, `\n`, `\f`, `\r`, and `\p{`*Z*`}`.
\S	Match a non-whitespace character.
\t	Match a *horizontal tabulation character* `#\tab`.
\u*hhhh*	Match the character with the hex value *hhhh*, i.e. `#x`*hhhh*.
\U*hhhhhhhh*	Match the character with the hex value *hhhhhhhh*. Exactly eight hex digits must be provided, even though the largest Unicode code point is `#x0010ffff`.
\w	Match a word character. Word characters are `\p{Ll}`, `\p{Lu}`, `\p{Lt}`, `\p{Lo}`, and `\p{Nd}`.
\W	Match a non-word character.
\x{*hhhh*}	Match the character with hex value *hhhh*. From one to six hex digits may be supplied.
\x*hh*	Match the character with two digit hex value *hh*.
\X	Match a grapheme cluster.
\Z	Match if the current position is at the end of input, but before the final line terminator, if one exists.
\z	Match if the current position is at the end of input.
n	Back Reference. Match whatever the *n*-th capturing group matched. *n* must be a number ≥ 1 and ≤ total number of capture groups in the pattern.
\0*ooo*	Match an octal character. *ooo* is from one to three octal digits. 0377 is the largest allowed octal character. The leading zero is required and distinguishes octal constants from back references.
[*chars*]	Match a *character class*, i.e. any one character from *chars*. Possible *chars* patterns are: `[...]`: positive character class `[^...]`: negative character class `[x-y]`: character range (more than one range can be listed)
.	Match any character.
^	Match at the beginning of a line.
$	Match at the end of a line.
****	Quotes the following character. Characters that must be quoted to be treated as literals are `*`, `?`, `+`, `[`, `(`, `)`, `{`, `}`, `^`, `$`, `|`, `\`, `.`, and `/`.

45.1.2 Regular expression operators

|	Alternation. `A|B` matches either `A` or `B`.
*****	Match 0 or more times, as many times as possible.
+	Match 1 or more times, as many times as possible.
?	Match zero or one times, preferring one time if possible.
{n}	Match exactly `n` times.
{n,}	Match at least `n` times, as many times as possible.
{n,m}	Match as many times as possible, but at least `n` and up to `m` times.
***?**	Match zero or more times, as few times as possible.

+?	Match one or more times, as few times as possible.
??	Match zero or one times, preferring zero.
{n}?	Match exactly n times.
{n,}?	Match at least n times, but no more than required for an overall pattern match.
{n,m}?	Match between n and m times, as few times as possible, but not less than n.
***+**	Match zero or more times, as many times as possible when first encountered, do not retry with fewer even if overall match fails (possessive match).
++	Match one or more times (possessive match).
?+	Match zero or one times (possessive match).
{n}+	Match exactly n times.
{n,}+	Match at least n times (possessive match).
{n,m}+	Match between n and m times (possessive match).
(...)	Capturing parentheses; the range of input that matched the parenthesized subexpression is available after the match.
(?:...)	Non-capturing parentheses; groups the included pattern, but does not provide capturing of matching text (more efficient than capturing parentheses).
(?>...)	Atomic-match parentheses; first match of the parenthesized subexpression is the only one tried. If it does not lead to an overall pattern match, back up the search for a match to a position before the "(?>".
(?# ...)	Free-format comment (?# comment).
(?= ...)	Look-ahead assertion. True, if the parenthesized pattern matches at the current input position, but does not advance the input position.
(?! ...)	Negative look-ahead assertion. True, if the parenthesized pattern does not match at the current input position. Does not advance the input position.
(?<= ...)	Look-behind assertion. True, if the parenthesized pattern matches text preceding the current input position, with the last character of the match being the input character just before the current position. Does not alter the input position. The length of possible strings matched by the pattern must not be unbounded (no * or + operators).
(?<! ...)	Negative *look-behind assertion*. True, if the parenthesized pattern does not match text preceding the current input position, with the last character of the match being the input character just before the current position. Does not alter the input position. The length of strings matched by the look-behind pattern must not be unbounded (no * or + operators).
(?ismwx-ismwx:...)	Flag settings. Evaluate the parenthesized expression with the specified flags enabled or disabled.
(?ismwx-ismwx)	Flag settings. Change the flag settings. Changes apply to the portion of the pattern following the setting. For example, (?i) changes to a case insensitive match.

45.1.3 Template Matching

$n The text of capture group n will be substituted for $n. n must be ≥ 0 and not greater than the number of capture groups. A $ not followed by a digit has no special meaning, and will appear in the substitution text as itself, i.e. $.

**** Treat the following character as a literal, suppressing any special meaning. Backslash escaping in substitution text is only required for $ and \`‘, but may be used on any other character.

45.1.4 Flag options

The following flags control various aspects of regular expression matching. These flags get specified within the pattern using the `(?ismx-ismx)` pattern options.

- **`i`** If set, matching will take place in a case-insensitive manner.
- **`x`** If set, allow use of white space and #-comments within patterns.
- **`s`** If set, a ".” in a pattern will match a line terminator in the input text. By default, it will not. Note that a carriage-return/line-feed pair in text behave as a single line terminator, and will match a single ".” in a regular expression pattern.
- **`m`** Control the behavior of `^` and `$` in a pattern. By default these will only match at the start and end, respectively, of the input text. If this flag is set, `^` and `$` will also match at the start and end of each line within the input text.
- **`w`** Controls the behavior of `\b` in a pattern. If set, word boundaries are found according to the definitions of word found in *Unicode UAX 29, Text Boundaries*. By default, word boundaries are identified by means of a simple classification of characters as either *word* or *non-word*, which approximates traditional regular expression behavior.

45.2 API

(regexp? *obj*) procedure
Returns `#t` if *obj* is a regular expression object; otherwise `#f` is returned.

(regexp *str*) procedure
(regexp *str opt …*)

Returns a new regular expression object from the given regular expression pattern *str* and matching options *opt*, … . *str* is a string, matching options *opt* are symbols. The following matching options are supported:

- `case-insensitive` : Match letters in the regular expression independent of their case.
- `allow-comments` : Ignore whitespace and `#` -prefixed comments in the regular expression pattern.
- `ignore-meta` : Treat the entire regular expression pattern as a literal string.
- `dot-matches-line-separator` : Allow `.` to match any character, including line separators.
- `anchors-match-lines` : Allow `^` and `$` to match the start and end of lines.
- `unix-only-line-separators` : Treat only `\n` as a line separator; otherwise, all standard line separators are used.
- `unicode-words` : Use Unicode TR#29 to specify word boundaries; otherwise, all traditional regular expression word boundaries are used.

(regexp-pattern *regexp*) procedure
Returns the regular expression pattern for the given regular expression object *regexp*. A regular expression pattern is a string matching the regular expression syntax supported by library `(lispkit regexp)` .

(regexp-capture-groups *regexp*) procedure
Returns the number of capture groups of the given regular expression object *regexp*.

(escape-regexp-pattern *str*) procedure
Returns a regular expression pattern string by adding backslash escapes to pattern *str* as necessary to protect any characters that would match as pattern meta-characters.

```
(escape-regexp-pattern "(home/objecthub)")
⇒ "\\(home\\/objecthub\\)"
```

(escape-regexp-template *str*) procedure

Returns a regular expression pattern template string by adding backslash escapes to pattern template *str* as necessary to protect any characters that would match as pattern meta-characters.

(regexp-matches *regexp str*) procedure
(regexp-matches *regexp str start*)
(regexp-matches *regexp str start end*)

Returns a *matching spec* if the regular expression object *regexp* successfully matches the entire string *str* from position *start* (inclusive) to *end* (exclusive); otherwise, `#f` is returned. The default for *start* is 0; the default for *end* is the length of the string.

A *matching spec* returned by `regexp-matches` consists of pairs of fixnum positions *(startpos . endpos)* in *str*. The first pair is always representing the full match (i.e. *startpos* is 0 and *endpos* is the length of *str*), all other pairs represent the positions of the matching capture groups of *regexp*.

```
(define email
  (regexp "[A-Z0-9a-z._%+-]+@[A-Za-z0-9.-]+\\.[A-Za-z]{2,4}"))
(regexp-matches email "matthias@objecthub.net")
⇒ ((0 . 22))
(define series
  (regexp "Season\\s+(\\d+)\\s+Episode\\s+(\\d+)"))
(regexp-matches series "Season 3  Episode 12")
⇒ ((0 . 20) (7 . 8) (18 . 20))
```

(regexp-matches? *regexp str*) procedure
(regexp-matches? *regexp str start*)
(regexp-matches? *regexp str start end*)

Returns `#t` if the regular expression object *regexp* successfully matches the entire string *str* from position *start* (inclusive) to *end* (exclusive); otherwise, `#f` is returned. The default for *start* is 0; the default for *end* is the length of the string.

(regexp-search *regexp str*) procedure
(regexp-search *regexp str start*)
(regexp-search *regexp str start end*)

Returns a *matching spec* for the first match of the regular expression *regexp* with a part of string *str* between position *start* (inclusive) and *end* (exclusive). If *regexp* does not match any part of *str* between *start* and *end*, `#f` is returned. The default for *start* is 0; the default for *end* is the length of the string.

A *matching spec* returned by `regexp-search` consists of pairs of fixnum positions *(startpos . endpos)* in *str*. The first pair is always representing the full match of the pattern, all other pairs represent the positions of the matching capture groups of *regexp*.

```
(define email
  (regexp "[A-Z0-9a-z._%+-]+@[A-Za-z0-9.-]+\\.[A-Za-z]{2,4}"))
(regexp-search email "Contact matthias@objecthub.net or foo@bar.org")
⇒ ((8 . 30))
(define series
  (regexp "Season\\s+(\\d+)\\s+Episode\\s+(\\d+)"))
(regexp-search series "New Season 3 Episode 12: Pilot")
⇒ ((4 . 23) (11 . 12) (21 . 23))
```

(regexp-search-all ***regexp str*****)** procedure
(regexp-search-all ***regexp str start*****)**
(regexp-search-all ***regexp str start end*****)**

Returns a list of all *matching specs* for matches of the regular expression *regexp* with parts of string *str* between position *start* (inclusive) and *end* (exclusive). If *regexp* does not match any part of *str* between *start* and *end*, the empty list is returned. The default for *start* is 0; the default for *end* is the length of the string.

A *matching spec* returned by `regexp-search` consists of pairs of fixnum positions *(startpos . endpos)* in *str*. The first pair is always representing the full match of the pattern, all other pairs represent the positions of the matching capture groups of *regexp*.

```
(define email
  (regexp "[A-Z0-9a-z._%+-]+@[A-Za-z0-9.-]+\\.[A-Za-z]{2,4}"))
(regexp-search-all email "Contact matthias@objecthub.net or foo@bar.org")
⇒ (((8 . 30)) ((34 . 45)))
(define series
  (regexp "Season\\s+(\\d+)\\s+Episode\\s+(\\d+)"))
(regexp-search-all series "New Season 3 Episode 12: Pilot")
⇒ (((4 . 23) (11 . 12) (21 . 23)))
```

(regexp-extract ***regexp str*****)** procedure
(regexp-extract ***regexp str start*****)**
(regexp-extract ***regexp str start end*****)**

Returns a list of substrings from *str* which all represent full matches of the regular expression *regexp* with parts of string *str* between position *start* (inclusive) and *end* (exclusive). If *regexp* does not match any part of *str* between *start* and *end*, the empty list is returned. The default for *start* is 0; the default for *end* is the length of the string.

```
(define email
  (regexp "[A-Z0-9a-z._%+-]+@[A-Za-z0-9.-]+\\.[A-Za-z]{2,4}"))
(regexp-extract email "Contact matthias@objecthub.net or foo@bar.org" 10)
⇒ ("tthias@objecthub.net" "foo@bar.org")
(define series
  (regexp "Season\\s+(\\d+)\\s+Episode\\s+(\\d+)"))
(regexp-extract series "New Season 3 Episode 12: Pilot")
⇒ ("Season 3 Episode 12")
```

(regexp-split ***regexp str*****)** procedure
(regexp-split ***regexp str start*****)**
(regexp-split ***regexp str start end*****)**

Splits string *str* into a list of possibly empty substrings separated by non-empty matches of regular expression *regexp* within position *start* (inclusive) and *end* (exclusive). If *regexp* does not match any part of *str* between *start* and *end*, a list with *str* as its only element is returned. The default for *start* is 0; the default for *end* is the length of the string.

```
(define email
  (regexp "[A-Z0-9a-z._%+-]+@[A-Za-z0-9.-]+\\.[A-Za-z]{2,4}"))
(regexp-split email "Contact matthias@objecthub.net or foo@bar.org" 10)
⇒ ("Contact ma" " or " "")
(define series (regexp "Season\\s+(\\d+)\\s+Episode\\s+(\\d+)"))
(regexp-split series "New Season 3 Episode 12: Pilot")
⇒ ("New " ": Pilot")
```

(regexp-partition ***regexp str*****)** procedure
(regexp-partition ***regexp str start*****)**
(regexp-partition ***regexp str start end*****)**

Partitions string *str* into a list of non-empty strings matching regular expression *regexp* within position *start* (inclusive) and *end* (exclusive), interspersed with the unmatched portions of the whole string. The first and every odd element is an unmatched substring, which will be the empty string if *regexp* matches at the beginning of the string or end of the previous match. The second and every even element will be a substring fully matching *regexp*. If *str* is the empty string or if there is no match at all, the result is a list with *str* as its only element.

```
(define email
  (regexp "[A-Z0-9a-z._%+-]+@[A-Za-z0-9.-]+\\.[A-Za-z]{2,4}"))
(regexp-partition email "Contact matthias@objecthub.net or foo@bar.org" 10)
⇒ ("Contact ma" "tthias@objecthub.net" " or " "foo@bar.org" "")
(define series (regexp "Season\\s+(\\d+)\\s+Episode\\s+(\\d+)"))
(regexp-partition series "New Season 3 Episode 12: Pilot")
⇒ ("New " "Season 3 Episode 12" ": Pilot")
```

(regexp-replace ***regexp str subst*****)** procedure
(regexp-replace ***regexp str subst start*****)**
(regexp-replace ***regexp str subst start end*****)**

Returns a new string replacing all matches of regular expression *regexp* in string *str* within position *start* (inclusive) and *end* (exclusive) with string *subst*. `regexp-replace` will always return a new string, even if there are no matches and replacements.

The optional parameters *start* and *end* restrict both the matching and the substitution, to the given positions, such that the result is equivalent to omitting these parameters and replacing on `(substring` *str start end* `)`.

```
(define email
  (regexp "[A-Z0-9a-z._%+-]+@[A-Za-z0-9.-]+\\.[A-Za-z]{2,4}"))
(regexp-replace email "Contact matthias@objecthub.net or foo@bar.org" "<omitted>" 10)
⇒ "Contact ma<omitted> or <omitted>"
(define series
  (regexp "Season\\s+(\\d+)\\s+Episode\\s+(\\d+)"))
(regexp-replace series "New Season 3 Episode 12: Pilot" "Series")
⇒ "New Series: Pilot"
```

(regexp-replace! ***regexp str subst*****)** procedure
(regexp-replace! ***regexp str subst start*****)**
(regexp-replace! ***regexp str subst start end*****)**

Mutates string *str* by replacing all matches of regular expression *regexp* within position *start* (inclusive) and *end* (exclusive) with string *subst*. The optional parameters *start* and *end* restrict both the matching and the substitution. `regexp-replace!` returns the number of replacements that were applied.

The optional parameters *start* and *end* restrict both the matching and the substitution, to the given positions, such that the result is equivalent to omitting these parameters and replacing on `(substring` *str start end* `)`.

```
(define email
  (regexp "[A-Z0-9a-z._%+-]+@[A-Za-z0-9.-]+\\.[A-Za-z]{2,4}"))
(define str "Contact matthias@objecthub.net or foo@bar.org")
(regexp-replace! email str "<omitted>" 10) ⇒ 2
str ⇒ "Contact ma<omitted> or <omitted>"
```

(regexp-fold ***regexp kons knil str*****)** procedure
(regexp-fold ***regexp kons knil str finish*****)**
(regexp-fold ***regexp kons knil str finish start*****)**
(regexp-fold ***regexp kons knil str finish start end*****)**

`regexp-fold` is the most fundamental and generic regular expression matching iterator. It repeatedly searches string *str* for the regular expression *regexp* so long as a match can be found. On each successful match, it applies `(kons` *i regexp-match str acc* `)` where *i* is the index since the last match (beginning with *start*), *regexp-match* is the resulting *matching spec*, and *acc* is the result of the previous *kons* application, beginning with *knil*. When no more matches can be found, `regexp-fold` calls *finish* with the same arguments, except that *regexp-match* is `#f`. By default, *finish* just returns *acc*.

```
(regexp-fold (regexp "(\\w+)")
             (lambda (i m str acc)
               (let ((s (substring str (caar m) (cdar m))))
                 (if (zero? i) s (string-append acc "-" s))))
             ""
             "to  be  or  not  to  be")
⇒ "to-be-or-not-to-be"
```

46 LispKit Set

Library `(lispkit set)` provides a generic implementation for sets of objects. Its API design is compatible to the R6RS-style API of library `(lispkit hashtable)`.

A set is a data structure for representing collections of objects. Any object can be used as element, provided a hash function and a suitable equivalence function is available. A hash function is a procedure that maps elements to exact integer objects. It is the programmer's responsibility to ensure that the hash function is compatible with the equivalence function, which is a procedure that accepts two objects and returns true if they are equivalent and `#f` otherwise. Standard sets for arbitrary objects based on the `eq?`, `eqv?`, and `equal?` predicates are provided.

46.1 Constructors

set-type-tag object
Symbol representing the `set` type. The `type-for` procedure of library `(lispkit type)` returns this symbol for all set objects.

(make-eq-set) procedure
Create a new empty set using `eq?` as equivalence function.

(make-eqv-set) procedure
Create a new empty set using `eqv?` as equivalence function.

(make-equal-set) procedure
Create a new empty set using `equal?` as equivalence function.

(make-set *hash equiv*) procedure
(make-set *hash equiv k*)

Create a new empty set using the given hash function *hash* and equivalence function *equiv*. An initial capacity *k* can be provided optionally.

(eq-set *element* ...) procedure
Create a new set using `eq?` as equivalence function. Initialize it with the values *element* ... .

(eqv-set *element* ...) procedure
Create a new set using `eqv?` as equivalence function. Initialize it with the values *element* ... .

(equal-set *element* ...) procedure
Create a new set using `equal?` as equivalence function. Initialize it with the values *element* ... .

46.2 Inspection

(set-equivalence-function *s*) procedure
Returns the equivalence function used by set *s*.

(set-hash-function *s*) procedure
Returns the hash function used by set *s*.

(set-mutable? *s*) procedure
Returns `#t` if set *s* is mutable.

46.3 Predicates

(set? *obj*) procedure
Returns `#t` if *obj* is a set.

(set-empty? *obj*) procedure
Returns `#t` if *obj* is an empty set.

(set=? *s1 s2*) procedure
Returns `#t` if set *s1* and set *s2* are using the same equivalence function and contain the same elements.

(disjoint? *s1 s2*) procedure
Returns `#t` if set *s1* and set *s2* are disjoint sets.

(subset? *s1 s2*) procedure
Returns `#t` if set *s1* is a subset of set *s2*.

(proper-subset? *s1 s2*) procedure
Returns `#t` if set *s1* is a proper subset of set *s2*, i.e. *s1* is a subset of *s2* and *s1* is not equivalent to *s2*.

(set-contains? *s element*) procedure
Returns `#` if set *s* contains *element*.

(set-any? *s proc*) procedure
Returns true if there is at least one element in set *s* for which procedure *proc* returns true (i.e. not `#f`).

(set-every? *s proc*) procedure
Returns true if procedure *proc* returns true (i.e. not `#f`) for all elements of set *s*.

46.4 Procedures

(set-size *s*) procedure
Returns the number of elements in set *s*.

(set-elements *s*) procedure
Returns the elements of set *s* as a vector.

(set-copy s) procedure
(set-copy *s mutable*)

Copies set *s* creating an immutable copy if `mutable` is set to `#f` or if `mutable` is not provided.

(set-for-each *s proc*) procedure
Applies procedure *proc* to all elements of set *s* in an undefined order.

(set-filter *s pred*) procedure
Creates a new set containing the elements of set *s* for which the procedure *pred* returns true.

(set-union *s s1 ...*) procedure
Creates a new set containing the union of *s* with *s1* ....

(set-intersection *s s1 ...*) procedure
Creates a new set containing the intersection of *s* with *s1* ....

(set-difference *s s1 …*) procedure
Creates a new set containing the difference of `s` and the sets in *s1* … .

(set->list *s*) procedure
Returns the elements of set *s* as a list.

(list->eq-set *elements*) procedure
Creates a new set using the equivalence function `eq?` from the values in list *elements*.

(list->eqv-set *elements*) procedure
Creates a new set using the equivalence function `eqv?` from the values in list *elements*.

(list->equal-set *elements*) procedure
Creates a new set using the equivalence function `equal?` from the values in list *elements*.

46.5 Mutators

(set-adjoin! *s element …*) procedure
Adds *element* … to the set *s*.

(set-delete! *s element …*) procedure
Deletes *element* … from the set *s*.

(set-clear! *s*) procedure
(set-clear! *s k*)

Clears set *s* and reserves a capacity of *k* elements if *k* is provided.

(list->set! *s elements*) procedure
Adds the values of list *elements* to set *s*.

(set-filter! *s pred*) procedure
Removes all elements from set *s* for which procedure *pred* returns `#f`.

(set-union! *s s1 …*) procedure
Stores the union of set *s* and sets *s1* … in *s*.

(set-intersection! *s s1 …*) procedure
Stores the intersection of set *s* and the sets *s1* … in *s*.

(set-difference! *s s1 …*) procedure
Stores the difference of set *s* and the sets *s1* … in *s*.

47 LispKit SQLite

SQLite is a lightweight, embedded, relational open-source database management system. It is simple to use, requires zero configuration, is not based on a server, and manages databases directly in files.

Library `(lispkit sqlite)` provides functionality for creating, managing, and querying SQLite databases in LispKit. `(lispkit sqlite)` is a low-level library that wraps the classial C API for SQLite3. Just like in the C API, the actual SQL statements are represented as strings and compiled into statement objects that are used for executing the statements.

47.1 Introduction

Library `(lispkit sqlite)` exports procedure `open-database` for creating new databases and connecting to existing ones. The following code will create a new database from scratch in file `~/Desktop/TestDatabase.sqlite` if that file does not exist. If the file exists, `open-database` will return a database object for accessing the database:

```
(import (lispkit sqlite))
(define db (open-database "~/Desktop/TestDatabase.sqlite"))
```

A new table can be created in database *db* with the help of an SQL `CREATE TABLE` statement. SQL statements are defined as strings and compiled into statement objects via procedure `prepare-statement`. Procedure `process-statement` is used to execute statement objects.

```
(define stmt0
  (prepare-statement db
    (string-append
      "CREATE TABLE Contacts (id INTEGER PRIMARY KEY,"
      "                       name TEXT NOT NULL,"
      "                       email TEXT NOT NULL UNIQUE,"
      "                       phone TEXT);")))
(process-statement stmt0)
```

Entries can be inserted into the new table `Contacts` with a corresponding SQL statement as shown in the following listing. First, a new SQL statement is being compiled. This SQL statement contains *parameters*. These are placeholders that are defined via `?`. They can be bound to concrete values before the statement is executed using procedures `bind-parameter` and `bind-parameters`.

The SQL statement below has 4 parameters, indexed starting 1. The code below binds these parameters one by one via `bind-parameter` to concrete values before the statement is executed via `process-statement`.

```
(define stmt1 (prepare-statement db "INSERT INTO Contacts VALUES (?, ?, ?, ?);"))
(bind-parameter stmt1 1 1000)
(bind-parameter stmt1 2 "Mickey Mouse")
(bind-parameter stmt1 3 "mickey@disney.net")
(bind-parameter stmt1 4 "+1 101-123-456")
(process-statement stmt1)
```

SQL statements can be reused many times. Typically, this is done by utilizing procedure `reset-statement`. If the previous execution was successful, though, this is not strictly necessary and a reset is done automatically. The code below re-applies the same statement a second time, this time using procedure `bind-parameters` to bind all parameters in one go.

```
(reset-statement stmt1) ; not strictly needed here
(bind-parameters stmt1 '(1001 "Donald Duck" "donald@disney.net" "+1 101-123-456"))
(process-statement stmt1)
```

The following code shows how to query for the total number of distinct phone numbers in table `Contacts`. The first invokation of procedure `process-statement` returns `#f`, indicating that there is a result. `column-count` returns 1, which is the column containing the distinct count. The count is extracted from the statement via `column-value`. The second invokation of `process-statement` now returns `#t` as there are no further query results.

```
; Count the number of distinct phone numbers.
(define stmt2 (prepare-statement db "SELECT COUNT(DISTINCT phone) FROM Contacts;"))
(process-statement stmt2) ; returns `#f`, i.e. there is a result
(display (column-count stmt2))
(newline)
(display (column-value stmt2 0))
(newline)
(process-statement stmt2) ; returns `#t`, i.e. there is no further result
```

The final example code below shows how to iterate effectively over a result table that has more than one result row.

```
; Show all names and email addresses from the `Contacts` table.
(define stmt3 (prepare-statement db "SELECT name, email FROM Contacts;"))
(do ((res '() (cons (row-values stmt3) res)))
    ((process-statement stmt3) res))
```

Executing this code returns the following list:

```
(("Donald Duck" "donald@disney.net") ("Mickey Mouse" "mickey@disney.net"))
```

47.2 API

47.2.1 SQLite version retrieval

(sqlite-version) procedure

The `sqlite-version` procedure returns a string that specifies the version of the SQLite framework in use in the format "*X.Y.Z*", where *X* is the major version number (e.g. 3 for SQLite3), Y is the minor version number, and Z is a release number.

(sqlite-version-number) procedure

The `sqlite-version-number` procedure returns a fixnum with the value *X1000000* + Y1000 + Z where *X* is the major version number (e.g. 3 for SQLite3), Y is the minor version number, and Z is a release number.

47.2.2 Database options

The following fixnum constants are used to specify how databases are opened or created via `make-database` and `open-database`. They can be combined by using an *inclusive or* function such as `fxior`. For instance, `(fxior sqlite-readwrite sqlite-create)` combines the two options `sqlite-create` and `sqlite-readwrite`.

sqlite-readonly object

This is a fixnum value for specifying an option how databases are opened or created via `make-database` and `open-database`. With this option, the database is opened in read-only mode. If the database does not exist already, an exception is thrown.

sqlite-readwrite object

This is a fixnum value for specifying an option how databases are opened or created via `make-database` and `open-database`. With this option, the database is opened for reading and writing if possible, or reading only if the file cannot be written at the operating system-level. If the database does not exist already, an exception is thrown.

sqlite-create object

This is a fixnum value for specifying an option how databases are opened or created via `make-database` and `open-database`. This option needs to be combined with either `sqlite-readwrite` or `sqlite-readonly`. It will lead to the creation of a new database in case there is no database at the specified path.

sqlite-default object

This is a fixnum value for specifying an option how databases are opened or created via `make-database` and `open-database`. With this option, the database is opened for reading and writing if possible, or reading only if the file cannot be written at the operating system-level. If the database does not exist already, a new database is being created.

sqlite-fullmutex object

This is a fixnum value for specifying an option how databases are opened or created via `make-database` and `open-database`. With this option, the database will use the "serialized" threading mode. In this mode, multiple threads can safely attempt to use the same database connection at the same time without the need for synchronization.

sqlite-sharedcache object

This is a fixnum value for specifying an option how databases are opened or created via `make-database` and `open-database`. With this option, the database is opened with shared cache enabled.

sqlite-privatecache object

This is a fixnum value for specifying an option how databases are opened or created via `make-database` and `open-database`. With this option, the database is opened with shared cache disabled.

47.2.3 Database objects

SQLite database objects are either created in memory with procedure `make-database` or they are created on disk by calling procedure `open-database`. `open-database` can also be used for opening an existing database. SQLite stores databases in regular files on disk.

(make-database) procedure
(make-database *options*)

Creates a new temporary in-memory database whose characteristics are described by *options*. *options* is a fixnum value. If no options are specified, `sqlite-default` (= create a new read/write database in memory) is used as the default. Options are represented as fixnum values. Combinations of options are created by performing a *bitwise inclusive or* of several option values, e.g. via `(fxior opt1 opt2)`. The following option values are predefined and can be used with `make-database`:

- `sqlite-default`: A new in-memory database is created and opened for reading and writing.
- `sqlite-fullmutex`: The database will use the "serialized" threading mode. In this mode, multiple threads can safely attempt to use the same database connection at the same time without the need for synchronization.
- `sqlite-sharedcache`: The database is opened with shared cache enabled.
- `sqlite-privatecache`: The database is opened with shared cache disabled.

(open-database *path*) procedure
(open-database *path options*)

Opens a database at file path *path* whose characteristics are described by *options*. *options* is a fixnum value. If no options are specified, `sqlite-default` (= create a new read/write database if there is not database at *path*) is used as the default. Options are represented as fixnum values. Combinations of options are created by performing a *bitwise inclusive or* of several option values, e.g. via `(fxior opt1 opt2)`. The following option values are predefined and can be used with `open-database`:

- `sqlite-readonly`: The database is opened in read-only mode. If the database does not exist already, an exception is thrown.
- `sqlite-readwrite`: The database is opened for reading and writing if possible, or reading only if the file cannot be written at the operating system-level. If the database does not exist already, an exception is thrown.
- `sqlite-create`: This option needs to be combined with either `sqlite-readwrite` or `sqlite-readonly`. It will lead to the creation of a new database in case there is no database at the specified path.
- `sqlite-default`: The database is opened for reading and writing if possible, or reading only if the file cannot be written at the operating system-level. If the database does not exist already, a new database is being created.
- `sqlite-fullmutex`: The database will use the "serialized" threading mode. In this mode, multiple threads can safely attempt to use the same database connection at the same time without the need for synchronization.
- `sqlite-sharedcache`: The database is opened with shared cache enabled.
- `sqlite-privatecache`: The database is opened with shared cache disabled.

(close-database *db*) procedure

Closes database *db* and deallocates all memory related to the database. If a transaction is open at this point, the transaction is automatically rolled back.

(sqlite-database? *obj*) procedure

Returns `#t` if *obj* is a database object. Otherwise, predicate `sqlite-database?` returns `#f`.

(database-path *db*) procedure

Returns the file path as a string at which the database *db* is being persisted. For in-memory databases, this procedure returns `#f`.

(database-last-row-id *db*) procedure

Each entry in a database table (except for *WITHOUT ROWID* tables) has a unique fixnum key called the *row id*. Procedure `database-last-row-id` returns the row id of the most recent successful insert into a table of database *db*. Inserts into *WITHOUT ROWID* tables are not recorded. If no successful inserts into row id tables have ever occurred for an open database, then `database-last-row-id` returns zero.

(database-last-changes *db*) procedure

`database-last-changes` returns the number of rows modified, inserted or deleted by the most recently completed `INSERT`, `UPDATE` or `DELETE` statement on the database *db*. Executing any other type of SQL statement does not modify the value returned by `database-last-changes`.

(database-total-changes *db*) procedure

Procedure `database-total-changes` returns the total number of rows inserted, modified or deleted by all `INSERT`, `UPDATE`, or `DELETE` statements completed since the database *db* was opened. Executing any other type of SQL statement does not affect the value returned by `database-total-changes`.

47.2.4 SQL statements

SQL statements are created with procedure `prepare-statement`. This procedure returns a statement object which encapsulates a compiled SQL query. The compiled SQL query can be executed by repeatedly calling procedure `process-statement`. As long as `process-statement` returns `#f`, a new result row can be extracted from the statement object with procedures such as `column-count`, `column-name`, `column-type`, `column-value`, `row-names`, `row-types`, `row-values`, and `row-alist`. As soon as `process-statement` returns `#t`, processing is complete. With procedure `reset-statement`, a statement object can be reset such that it can be executed again.

(sqlite-statement? *obj*) procedure

Returns `#t` if *obj* is a statement object. Otherwise, predicate `sqlite-statement?` returns `#f`.

(prepare-statement *db str*) procedure

To execute an SQL statement, it must first be compiled into bytecode which then gets executed, potentially multiple times, in a second step. `prepare-statement` compiles an SQL statement contained in string *str* for execution in database *db*. It returns a *statement* object which encapsulates the compiled query. If compilation fails, an execption is thrown.

(parameter-count *stmt*) procedure

Returns the number of parameters contained in statement object *stmt*. If *stmt* contains *N* parameters, they can be referenced by the indices 1 to *N*.

(parameter-index *stmt name*) procedure

Returns the index of named parameter *name* in statement object *stmt*. *name* is a string. The result is a positive fixnum if the named parameter exists, or `#f` if there is no parameter with name *name*.

(parameter-name *stmt idx*) procedure

Returns the name of the named parameter at index *idx* in statement object *stmt* as a string. If such a parameter does not exist, `parameter-name` returns `#f`. *idx* is a positive fixnum.

(bind-parameter *stmt idx val*) procedure

Binds parameter at index *idx* to value *val* in statement object *stmt*.

(bind-parameters *stmt vals*) procedure
(bind-parameters *stmt vals idx*)

Binds the parameters starting at index *idx* to values in list *vals*. If *idx* is not given, 1 is used as a default. `bind-parameters` returns the tail of the list that could not be bound to parameters. *idx* is a positive fixnum.

(process-statement *stmt*) procedure

Procedure `process-statement` starts or proceeds executing statement *stmt*. The result of the execution step is accessible via the statement object *stmt* and can be inspected by procedures such as `column-count`, `column-name`, `column-type`, `column-value`, `row-names`, `row-types`, `row-values`, and `row-alist`. `process-statement` returns `#f` as long as the execution is ongoing and a new resulting table row is available for inspection. When `#t` is returned, execution is complete.

(reset-statement *stmt*) procedure

Resets the statement object *stmt* so that it can be processed another time.

(column-count *stmt*) procedure

`column-count` returns the number of columns of the result of processing statement *stmt*. If *stmt* does not yield data as a result, `column-count` returns 0.

(column-name *stmt idx*) procedure

`column-name` returns the name of column *idx* of the result of executing statement *stmt*. *idx* is a fixnum identifying the column by its 0-based index. `column-name` returns `#f` if column *idx* does not exist.

(column-type *stmt idx*) procedure

`column-type` returns the type of the value at column *idx* of the result of executing statement *stmt*. *idx* is a fixnum identifying the column by its 0-based index. `column-type` returns `#f` if column *idx* does not exist. Types are represented by symbols. The following types are supported:

- `sqlite-integer` : Values are fixnums
- `sqlite-float` : Values are flonums
- `sqlite-text` : Values are strings
- `sqlite-blob` : Values are bytevectors
- `sqlite-null` : There is no value (void is the only supported value)

(column-value *stmt idx*) procedure

`column-value` returns the value at column *idx* of the result of executing statement *stmt*. *idx* is a fixnum identifying the column by its 0-based index. `column-value` returns `#f` if column *idx* does not exist.

(row-names *stmt*) procedure

Returns a list of all column names of the result of executing statement *stmt*.

(row-types *stmt*) procedure

Returns a list of all column types of the result of executing statement *stmt*. Types are represented by symbols. The following types are supported:

- `sqlite-integer` : Values are fixnums
- `sqlite-float` : Values are flonums
- `sqlite-text` : Values are strings
- `sqlite-blob` : Values are bytevectors
- `sqlite-null` : There is no value (void is the only supported value)

(row-values *stmt*) procedure

Returns a list of all column values of the result of executing statement *stmt*.

(row-alist *stmt*) procedure

Returns an association list associating column names with column values of the result of executing statement *stmt*.

48 LispKit Stack

Library `(lispkit stack)` provides an implementation for mutable stacks, i.e. mutable LIFO buffers.

stack-type-tag object

Symbol representing the `stack` type. The `type-for` procedure of library `(lispkit type)` returns this symbol for all stack objects.

(make-stack) procedure

Returns a new empty stack.

(stack *x* ...) procedure

Returns a new stack with *x* on its top position followed by the remaining parameters.

```
(stack-top (stack 1 2 3))  ⇒  1
```

(stack? *obj*) procedure

Returns `#t` if *obj* is a stack; otherwise `#f` is returned.

(stack-empty? *s*) procedure

Returns `#t` if stack *s* is empty.

(stack-size *s*) procedure

Returns the size of stack *s*, i.e. the number of elements buffered in *s*.

(stack=? *s1 s2*) procedure

Returns `#t` if stack *s1* has the exact same elements in the same order like stack *s2*; otherwise, `#f` is returned.

(stack-push! *s x*) procedure

Pushes element *x* onto stack *s*.

(stack-top *s*) procedure

Returns the top element of stack *s*. If the stack is empty, an error is raised.

(stack-pop! *s*) procedure

Removes the top element from stack *s* and returns its value.

```
(define s (make-stack))
(stack-push! s 1)
(stack-push! s 2)
(stack-pop! s)  ⇒  2
(stack-size s)  ⇒  1
```

(stack-clear! *s*) procedure

Removes all elements from stack *s*.

(stack-copy *s*) procedure

Returns a copy of stack *s*.

(stack->list *s*) procedure

Returns a list consisting of all elements on stack *s* in the order they appear, i.e. starting with the top element.

```
(stack->list (stack 1 2 3))
```

(list->stack *l*) procedure

Returns a new stack consisting of the elements of list *l*. The first element in *l* will become the top element of the stack that is returned.

(list->stack! *s* *l*) procedure

Pushes the elements of list *l* onto stack *s* in reverse order.

```
(define s (list->stack '(3 2 1)))
(list->stack! s '(6 5 4))
(stack->list s)  ⇒  (6 5 4 3 2 1)
```

49 LispKit Stream

Streams are a sequential data structure containing elements computed only on demand. They are sometimes also called *lazy lists*.

Streams get constructed with list-like constructors. A stream is either *null* or is a *pair* with a stream in its cdr. Since elements of a stream are computed only when accessed, streams can be infinite. Once computed, the value of a stream element is cached in case it is needed again.

49.1 Benefits of using streams

When used effectively, the primary benefit of streams is improved modularity. Consider a process that takes a sequence of items, operating on each in turn. If the operation is complex, it may be useful to split it into two or more procedures in which the partially-processed sequence is an intermediate result. If that sequence is stored as a list, the entire intermediate result must reside in memory all at once; however, if the intermediate result is stored as a stream, it can be generated piecemeal, using only as much memory as required by a single item. This leads to a programming style that uses many small operators, each operating on the sequence of items as a whole, similar to a pipeline of unix commands.

In addition to improved modularity, streams permit a clear exposition of backtracking algorithms using the "stream of successes" technique, and they can be used to model generators and co-routines. The implicit memoization of streams makes them useful for building persistent data structures, and the laziness of streams permits some multi-pass algorithms to be executed in a single pass. Savvy programmers use streams to enhance their programs in countless ways.

There is an obvious space/time trade-off between lists and streams; lists take more space, but streams take more time (to see why, look at all the type conversions in the implementation of the stream primitives). Streams are appropriate when the sequence is truly infinite, when the space savings are needed, or when they offer a clearer exposition of the algorithms that operate on the sequence.

49.2 Stream abstractions

The `(lispkit stream)` library provides two mutually-recursive abstract data types: An object of type `stream` is a promise that, when forced, is either `stream-null` or is an object of type `stream-pair`. An object of the `stream-pair` type contains a `stream-car` and a `stream-cdr`, which must be a stream. The essential feature of streams is the systematic suspensions of the recursive promises between the two data types.

The object stored in the `stream-car` of a `stream-pair` is a promise that is forced the first time the `stream-car` is accessed; its value is cached in case it is needed again. The object may have any type, and different stream elements may have different types. If the `stream-car` is never accessed, the object stored there is never evaluated. Likewise, the `stream-cdr` is a promise to return a stream, and is only forced on demand.

49.3 Stream API

The design of the API of library `(lispkit stream)` is based on Philip Bewig's SRFI 41. The implementation of the library is LispKit-specific.

stream-type-tag object

Symbol representing the `stream` type. The `type-for` procedure of library `(lispkit type)` returns this symbol for all stream objects.

stream-null object

`stream-null` is a stream that, when forced, is a single object, distinguishable from all other objects, that represents the null stream. `stream-null` is immutable and unique.

(stream? *obj*) procedure

Returns `#t` if *obj* is a stream; otherwise `#f` is returned.

(stream-null? *obj*) procedure

`stream-null?` is a procedure that takes an object *obj* and returns `#t` if the object is the distinguished null stream and `#f` otherwise. If object *obj* is a stream, `stream-null?` must force its promise in order to distinguish `stream-null` from `stream-pair`.

(stream-pair? *obj*) procedure

`stream-pair?` is a procedure that takes an object and returns `#t` if the object is a `stream-pair` constructed by `stream-cons` and `#f` otherwise. If object is a stream, `stream-pair?` must force its promise in order to distinguish `stream-null` from `stream-pair`.

(stream-cons *obj strm*) syntax

`stream-cons` is a special form that accepts an object *obj* and a stream *strm* and creates a newly-allocated stream containing a stream that, when forced, is a `stream-pair` with the object in its `stream-car` and the stream in its `stream-cdr`. `stream-cons` must be syntactic, not procedural, because neither object *obj* nor stream is evaluated when `stream-cons` is called. Since `strm` is not evaluated, when the `stream-pair` is created, it is not an error to call `stream-cons` with a stream that is not of type stream; however, doing so will cause an error later when the `stream-cdr` of the `stream-pair` is accessed. Once created, a `stream-pair` is immutable.

```
(define s (stream-cons 1 (stream-cons 2 (stream-cons 3 stream-null))))
(stream-car s)               ⇒  1
(stream-car (stream-cdr s))  ⇒  2
```

(stream-car *strm*) procedure

`stream-car` is a procedure that takes a stream *strm* and returns the object stored in the `stream-car` of the stream. `stream-car` signals an error if the object passed to it is not a `stream-pair`. Calling `stream-car` causes the object stored there to be evaluated if it has not yet been; the object's value is cached in case it is needed again.

(stream-cdr *strm*) procedure

`stream-cdr` is a procedure that takes a stream *strm* and returns the stream stored in the `stream-cdr` of the stream. `stream-cdr` signals an error if the object passed to it is not a `stream-pair`. Calling `stream-cdr` does not force the promise containing the stream stored in the `stream-cdr` of the stream.

(stream *obj* …) syntax

`stream` is syntax that takes zero or more objects *obj* and creates a newly-allocated stream containing in its elements the objects, in order. Since `stream` is syntactic, the objects are evaluated when they are accessed, not when the stream is created. If no objects are given, as in `(stream)`, the null stream is returned.

(stream-lambda *formals expr0 expr1 …*) syntax

`stream-lambda` creates a procedure that returns a stream to evaluate the body of the procedure. The last body expression to be evaluated must yield a stream. As with the regular `lambda`, *formals* may be a single variable name, in which case all the formal arguments are collected into a single list, or it is a list of variable names, which may be `null` if there are no arguments, proper if there are an exact number of arguments, or dotted, if a fixed number of arguments is to be followed by zero or more arguments collected into a list. The body *expr0 expr1 …* must contain at least one expression, and may contain internal definitions preceding any expressions to be evaluated.

```
(define iter (stream-lambda (f x) (stream-cons x (iter f (f x)))))
(define nats (iter (lambda (x) (+ x 1)) 0))
(stream-car (stream-cdr nats))              ⇒  1

(define stream-add
  (stream-lambda (s1 s2)
    (stream-cons (+ (stream-car s1) (stream-car s2))
                 (stream-add (stream-cdr s1) (stream-cdr s2)))))
(define evens (stream-add nats nats))

(stream-car evens)                          ⇒  0
(stream-car (stream-cdr evens))             ⇒  2
(stream-car (stream-cdr (stream-cdr evens))) ⇒  4
```

(define-stream *(name arg …) expr0 expr1 …*) syntax

`define-stream` creates a procedure *name* that returns a stream, and may appear anywhere a normal define may appear, including as an internal definition, and may have internal definitions of its own, including other `define-streams`. The defined procedure takes arguments *arg …* in the same way as `stream-lambda`. `define-stream` is syntactic sugar on `stream-lambda`.

(stream-let *tag ((var val) …) expr1 expr2 …*) syntax

`stream-let` creates a local scope that binds each variable *var* to the value of its corresponding expression *val*. It additionally binds *tag* to a procedure which takes the bound variables as arguments and body as its defining expressions, binding the tag with `stream-lambda`. *tag* is in scope within body, and may be called recursively. When the expanded expression defined by the `stream-let` is evaluated, `stream-let` evaluates the expressions *expr1 expr2 …* in its body in an environment containing the newly-bound variables, returning the value of the last expression evaluated, which must yield a stream.

`stream-let` provides syntactic sugar on `stream-lambda`, in the same manner as normal `let` provides syntactic sugar on normal `lambda`. However, unlike normal `let`, the `tag` is required, not optional, because unnamed `stream-let` is meaningless.

(display-stream *strm*) procedure
(display-stream *strm n*)
(display-stream *strm n sep*)
(display-stream *strm n sep port*)

`display-stream` displays the first *n* elements of stream *strm* on port *port* using string *sep* as a separator string. If *n* is not provided, all elements are getting displayed. If *sep* is not provided, `", "` is used as a default. If *port* is not provided, the current output port is used.

(list->stream *lst*) procedure

`list->stream` takes a list of objects *lst* and returns a newly-allocated stream containing in its elements the objects in the list. Since the objects are given in a list, they are evaluated when `list->stream` is called, before the stream is created. If the list of objects is null, as in `(list->stream '())`, the null stream is returned.

(port->stream) procedure
(port->stream *port*)

`port->stream` takes a port *port* and returns a newly-allocated stream containing in its elements the characters on the port. If the port is not given, it defaults to the current input port. The returned stream has finite length and is terminated by `stream-null`.

(stream->list *strm*) procedure
(stream->list *strm n*)

`stream->list` takes a natural number *n* and a stream *strm* and returns a newly-allocated list containing in its elements the first *n* items in the stream. If the stream has less than *n* items, all the items in the stream will be included in the returned list. If *n* is not given, it defaults to infinity, which means that unless the stream is finite, `stream->list` will never return.

(stream-append *strm* …) procedure
`stream-append` returns a newly-allocated stream containing in its elements those elements contained in its argument streams *strm* …, in order of input. If any of the input streams is infinite, no elements of any of the succeeding input streams will appear in the output stream; thus, if `x` is infinite, `(stream-append x y)` ≡ `x`.

(stream-concat *strms*) procedure
`stream-concat` takes a stream *strms* consisting of one or more streams and returns a newly-allocated stream containing all the elements of the input streams. If any of the streams in the input stream is infinite, any remaining streams in the input stream will never appear in the output stream.

(stream-constant *obj* …) procedure
`stream-constant` takes one or more objects *obj* … and returns a newly-allocated stream containing in its elements the objects, repeating the objects in succession forever.

(stream-drop *strm n*) procedure
`stream-drop` returns the suffix of the input stream *strm* that starts at the next element after the first *n* elements. The output stream shares structure with the input stream; thus, promises forced in one instance of the stream are also forced in the other instance of the stream. If the input stream has less than *n* elements, `stream-drop` returns the null stream.

(stream-drop-while *pred? strm*) procedure
`stream-drop-while` returns the suffix of the input stream that starts at the first element *x* for which `(pred? x)` is `#f`. The output stream shares structure with the input stream.

(stream-filter *pred? strm*) procedure
`stream-filter` returns a newly-allocated stream that contains only those elements *x* of the input stream for which `(pred? x)` is non-`#f`.

(stream-fold *proc base strm*) procedure
`stream-fold` applies a binary procedure *proc* to *base* and the first element of stream *strm* to compute a new base, then applies the procedure *proc* to the new base (1st argument of *proc*) and the next element of stream (2nd argument of *proc*) to compute a succeeding base, and so on, accumulating a value that is finally returned as the value of `stream-fold` when the end of the stream is reached. *strm* must be finite, or `stream-fold` will enter an infinite loop.

See also `stream-scan`, which is similar to `stream-fold`, but useful for infinite streams. `stream-fold` is a left-fold; there is no corresponding `right-fold`, since `right-fold` relies on finite streams that are fully-evaluated, at which time they may as well be converted to a list.

(stream-for-each *proc strm* …) procedure
`stream-for-each` applies a procedure *proc* elementwise to corresponding elements of the input streams *strm* … for its side-effects. `stream-for-each` stops as soon as any of its input streams is exhausted.

(stream-from *first*) procedure
(stream-from *first delta*)

`stream-from` creates a newly-allocated stream that contains *first* as its first element and increments each succeeding element by *delta*. If *delta* is not given it defaults to 1. *first* and *delta* may be of any numeric type. `stream-from` is frequently useful as a generator in `stream-of` expressions. See also `stream-range` for a similar procedure that creates finite streams.

(stream-iterate *proc base*) procedure

`stream-iterate` creates a newly-allocated stream containing *base* in its first element and applies *proc* to each element in turn to determine the succeeding element.

(stream-length *strm*) procedure

`stream-length` takes an input stream *strm* and returns the number of elements in the stream. It does not evaluate its elements. `stream-length` may only be used on finite streams as it enters an infinite loop with infinite streams.

(stream-map *proc strm …*) procedure

`stream-map` applies a procedure *proc* elementwise to corresponding elements of the input streams *strm* …, returning a newly-allocated stream containing elements that are the results of those procedure applications. The output stream has as many elements as the minimum-length input stream, and may be infinite.

(stream-match *strm-expr (pattern [fender] expr) …*) syntax

`stream-match` provides the syntax of pattern-matching for streams. The input stream *strm-expr* is an expression that evaluates to a stream and is matched against a number of clauses. Each clause `(pattern [fender] expr)` consists of a pattern that matches a stream of a particular shape, an optional *fender* that must succeed if the pattern is to match, and an expression that is evaluated if the pattern matches.

There are four types of patterns:

- `()` : Matches the null stream
- `(pat0 pat1 ...)` : Matches a finite stream with length exactly equal to the number of pattern elements
- `(pat0 pat1 ... . patrest)` : Matches an infinite stream, or a finite stream with length at least as great as the number of pattern elements before the literal dot
- `pat` : Matches an entire stream. Should always appear last in the list of clauses; it's not an error to appear elsewhere, but subsequent clauses could never match

Each pattern element pati may be either:

- *An identifier*: Matches any stream element. Additionally, the value of the stream element is bound to the variable named by the identifier, which is in scope in the *fender* and expression of the corresponding clause. Each identifier in a single pattern must be unique.
- *A literal underscore*: Matches any stream element, but creates no bindings.

The patterns are tested in order, left-to-right, until a matching pattern is found. If *fender* is present, it must evaluate as non- `#f` for the match to be successful. Pattern variables are bound in the corresponding *fender* and expression. Once the matching pattern is found, the corresponding expression is evaluated and returned as the result of the match. An error is signaled if no pattern matches the input stream.

(stream-of *expr rest …*) syntax

`stream-of` provides the syntax of stream comprehensions, which generate streams by means of looping expressions. The result is a stream of objects of the type returned by *expr*. There are four types of clauses:

- `(var in stream-expr)` : Loop over the elements of `stream-expr` , in order from the start of the stream, binding each element of the stream in turn to `var` . `stream-from` and `stream-range` are frequently useful as generators.
- `(var is expr)` : Bind `var` to the value obtained by evaluating `expr` .

- `(pred? expr)` : Include in the output stream only those elements `x` for which `(pred? x)` is non-`#f`.

The scope of variables bound in the stream comprehension is the clauses to the right of the binding clause (but not the binding clause itself) plus the result expression. When two or more generators are present, the loops are processed as if they are nested from left to right; i.e. the rightmost generator varies fastest. A consequence of this is that only the first generator may be infinite and all subsequent generators must be finite. If no generators are present, the result of a stream comprehension is a stream containing the result expression; thus, `(stream-of 1)` produces a finite stream containing only the element 1.

(stream-range *first past*) procedure
(stream-range *first past delta*)

`stream-range` creates a newly-allocated stream that contains *first* as its first element and increments each succeeding element by *step*. The stream is finite and ends before *past*, which is not an element of the stream. If *step* is not given it defaults to 1 if *first* is less than past and `-1` otherwise. First, *past* and *step* may be of any numeric type. `stream-range` is frequently useful as a generator in `stream-of` expressions.

(stream-ref *strm n*) procedure
`stream-ref` returns the *n*-th element of stream, counting from zero. An error is signaled if *n* is greater than or equal to the length of stream.

(stream-reverse *strm*) procedure
`stream-reverse` returns a newly-allocated stream containing the elements of the input stream *strm* but in reverse order. `stream-reverse` may only be used with finite streams; it enters an infinite loop with infinite streams. `stream-reverse` does not force evaluation of the elements of the stream.

(stream-scan *proc base strm*) procedure
`stream-scan` accumulates the partial folds of an input stream *strm* into a newly-allocated output stream. The output stream is the *base* followed by `(stream-fold proc base (stream-take i stream))` for each of the first `i` elements of stream.

(stream-take *strm n*) procedure
`stream-take` takes a non-negative integer *n* and a stream and returns a newly-allocated stream containing the first *n* elements of the input stream. If the input stream has less than *n* elements, so does the output stream.

(stream-take-while *pred? strm*) procedure
`stream-take-while` takes a predicate *pred?* and a stream *strm* and returns a newly-allocated stream containing those elements `x` that form the maximal prefix of the input stream for which `(pred? x)` is non- `#f`.

(stream-unfold *mapper pred? generator base*) procedure
`stream-unfold` is the fundamental recursive stream constructor. It constructs a stream by repeatedly applying *generator* to successive values of *base*, in the manner of `stream-iterate`, then applying *mapper* to each of the values so generated, appending each of the mapped values to the output stream as long as `(pred? base)` is non- `#f`.

(stream-unfolds *proc seed*) procedure
`stream-unfolds` returns *n* newly-allocated streams containing those elements produced by successive calls to the generator *proc*, which takes the current `seed` as its argument and returns *n+1* values:

(*proc seed*) ⇒ *seed result0 … resultn-1*

where the returned seed is the input seed to the next call to the generator and *resulti* indicates how to produce the next element of the *i*-th result stream:

- `(value)` : value is the next car of the result stream

- `#f` : no value produced by this iteration of the generator `proc` for the result stream
- `()` : the end of the result stream

It may require multiple calls of `proc` to produce the next element of any particular result stream.

(stream-zip *strm* ...) procedure

`stream-zip` takes one or more input streams *strm* ... and returns a newly-allocated stream in which each element is a list (not a stream) of the corresponding elements of the input streams. The output stream is as long as the shortest input stream, if any of the input streams is finite, or is infinite if all the input streams are infinite.

50 LispKit String

Strings are sequences of characters. In LispKit, characters are UTF-16 code units. Strings are written as sequences of characters enclosed within quotation marks (`"`). Within a string literal, various escape sequences represent characters other than themselves. Escape sequences always start with a backslash `\` :

- `\a` : alarm (U+0007)
- `\b` : backspace (U+0008)
- `\t` : character tabulation (U+0009)
- `\n` : linefeed (U+000A)
- `\r` : return (U+000D)
- `\"` : double quote (U+0022)
- `\\` : backslash (U+005C)
- `\|` : vertical line (U+007C)
- `\` *line-end*: used for encoding multi-line string literals
- `\x` *hex-scalar-value* `;` : specified character

The result is unspecified if any other character in a string occurs after a backslash.

Except for a line ending, any character outside of an escape sequence stands for itself in the string literal. A line ending which is preceded by a backslash expands to nothing and can be used to encode multi-line string literals.

```
(display "The word \"recursion\" has many meanings.")  ⇒
The word "recursion" has many meanings.
(display "Another example:\ntwo lines of text.")  ⇒
Another example:
two lines of text.
(display "\x03B1; is named GREEK SMALL LETTER ALPHA.")  ⇒
α is named GREEK SMALL LETTER ALPHA.
```

The length of a string is the number of characters, i.e. UTF-16 code units, that it contains. This number is an exact, non-negative integer that is fixed when the string is created. The valid indexes of a string are the exact non-negative integers less than the length of the string. The first character of a string has index 0, the second has index 1, and so on.

Some of the procedures that operate on strings ignore the difference between upper and lower case. The names of the versions that ignore case end with `-ci` (for "case insensitive").

LispKit only supports mutable strings.

50.1 Basic constructors and procedures

(make-string *k*) procedure
(make-string *k* *char*)

The `make-string` procedure returns a newly allocated string of length *k*. If *char* is given, then all the characters of the string are initialized to *char*, otherwise the contents of the string are unspecified.

(string *char* …) procedure
Returns a newly allocated string composed of the arguments. It is analogous to procedure `list`.

(list->string *list*) procedure
Returns a newly allocated string composed of the characters contained in *list*.

(string-ref *str k*) procedure
The `string-ref` procedure returns character *k* of string *str* using zero-origin indexing. It is an error if *k* is not a valid index of string *str*.

(string-set! *str k char*) procedure
The `string-set!` procedure stores *char* in element *k* of string *str*. It is an error if *k* is not a valid index of string *str*.

(string-length *str*) procedure
Returns the number of characters in the given string *str*.

50.2 Predicates

(string? *obj*) procedure
Returns `#t` if *obj* is a string; otherwise returns `#f`.

(string-empty? *str*) procedure
Returns `#t` if *str* is an empty string, i.e. a string of length 0. Otherwise, `string-empty?` returns `#f`.

(string=? *str* …) procedure
Returns `#t` if all the strings have the same length and contain exactly the same characters in the same positions; otherwise `string=?` returns `#f`.

(string<? *str* …) procedure
(string>? *str* …)
(string<=? *str* …)
(string>=? *str* …)

These procedures return `#t` if their arguments are (respectively): monotonically increasing, monotonically decreasing, monotonically non-decreasing, or monotonically non-increasing. These predicates are transitive.

These procedures compare strings in a lexicographic fashion; i.e. `string<?` implements a the lexicographic ordering on strings induced by the ordering `char<?` on characters. If two strings differ in length but are the same up to the length of the shorter string, the shorter string would be considered to be lexicographically less than the longer string.

A pair of strings satisfies exactly one of `string<?`, `string=?`, and `string>?`. A pair of strings satisfies `string<=?` if and only if they do not satisfy `string>?`. A pair of strings satisfies `string>=?` if and only if they do not satisfy `string<?`.

(string-ci=?) procedure
Returns `#t` if, after case-folding, all the strings have the same length and contain the same characters in the same positions; otherwise `string-ci=?` returns `#f`.

(string-ci<? *str* …) procedure
(string-ci<=? *str* …) (string-ci>? *str* …) (string-ci>=? *str* …)

These procedures compare strings in a case-insensitive fashion. The "-ci" procedures behave as if they applied `string-foldcase` to their arguments before invoking the corresponding procedures without "-ci".

(string-contains? *str sub*) procedure

Returns `#t` if string *str* contains string *sub*; returns `#f` otherwise.

(string-prefix? *str sub*) procedure

Returns `#t` if string *str* has string *sub* as a prefix; returns `#f` otherwise.

(string-suffix? *str sub*) procedure

Returns `#t` if string *str* has string *sub* as a suffix; returns `#f` otherwise.

50.3 Composing and extracting strings

Many of the following procedures accept an optional *start* and *end* argument as their last two arguments. If both or one of these optional arguments are not provided, *start* defaults to `0` and *end* defaults to the length of the corresponding string.

(string-contains *str sub*) procedure
(string-contains *str sub start*)
(string-contains *str sub start end*)

This procedure checks whether string *sub* is contained in string *str* within the index range *start* to *end*. It returns the first index into *str* at which *sub* is fully contained within *start* and *end*. If *sub* is not contained in the substring of *str*, then `#f` is returned.

(substring *str start end*) procedure

The `substring` procedure returns a newly allocated string formed from the characters of string *str* beginning with index *start* and ending with index *end*. This is equivalent to calling `string-copy` with the same arguments, but is provided for backward compatibility and stylistic flexibility.

(string-append *str* ...) procedure

Returns a newly allocated string whose characters are the concatenation of the characters in the given strings *str* ....

(string-concatenate *list*) procedure
(string-concatenate *list sep*)

Returns a newly allocated string whose characters are the concatenation of the characters in the strings contained in *list*. *sep* is either a character or string, which, if provided, is used as a separator between two strings that get concatenated. It is an error if *list* is not a proper list containing only strings as elements.

(string-upcase *str*) procedure
(string-downcase *str*)
(string-titlecase *str*)
(string-foldcase *str*)

These procedures apply the Unicode full string uppercasing, lowercasing, titlecasing, and case-folding algorithms to their argument string *str* and return the result as a newly allocated string. It is not guaranteed that the resulting string has the same lenght like *str*. Language-sensitive string mappings and foldings are not used.

(string-normalize-diacritics *str*) procedure

Procedure `string-normalize-diacritics` transforms the given string *str* by normalizing diacritics and returning the result as a newly allocated string.

```
(string-normalize-diacritics "Meet Chloë at São Paulo Café")
⇒ "Meet Chloe at Sao Paulo Cafe"
```

(string-normalize-separators *str*) procedure
(string-normalize-separators *str sep*)
(string-normalize-separators *str sep cset*)

Procedure `string-normalize-separators` normalizes string *str* by replacing sequences of separation characters from character set *cset* with string or character *sep*. If *sep* is not provided, `" "` is used as a default. If *cset* is not provided, all unicode newline and whitespace characters are used as a default for *cset*. *cset* is either a string of separation characters or a character set as defined by library `(lispkit char-set)`.

(string-encode-named-chars *str*) procedure
(string-encode-named-chars *str required-only?*)

Procedure `string-encode-named-chars` returns a new string, replacing characters with their corresponding named XML entity in string *str*. If parameter `required-only?` is set to `#f`, all characters with corresponding named XML entities are being replaced, otherwise only the required characters are replaced.

```
(string-encode-named-chars "<one> & two = 3")
⇒ "&LT;one&gt; &AMP; two &equals; 3"
(string-encode-named-chars "<one> & two = 3" #t)
⇒ "&lt;one&gt; &amp; two = 3"
```

(string-decode-named-chars *str*) procedure
Procedure `string-decode-named-chars` returns a new string, replacing named XML entities with their corresponding character.

```
(string-decode-named-chars "2&Hat;&lcub;3&rcub; &equals; 8")
⇒ "2^{3} = 8"
```

(string-copy *str*) procedure
(string-copy *str start*)
(string-copy *str start end*)

Returns a newly allocated copy of the part of the given string *str* between *start* and *end*. The default for *start* is 0, for *end* it is the length of *str*. Calling `string-copy` is equivalent to calling `substring` with the same arguments. `substring` is provided primarily for backward compatibility.

(string-split *str sep allow-empty?*) procedure
Procedure `string-split` splits string *str* using the separator *sep* and returns a list of the component strings, in order. *sep* is either a string or a character. Boolean argument *allow-empty?* determines whether empty component strings are dropped. *allow-empty?* is `#t` by default.

```
(string-split "name-|-street-|-zip-|-city-|-" "-|-")  ⇒  ("name" "street" "zip" "city" "")
(string-split "name-|-street-|-zip-|-city-|-" "-|-" #f)  ⇒  ("name" "street" "zip" "city")
```

(string-trim *str*) procedure
(string-trim *str chars*)

Returns a newly allocated string by removing all characters from the beginning and end of string *str* that are contained in *chars*. *chars* is either a string or it is a character set. If *chars* is not provided, whitespaces and newlines are being removed.

```
(string-trim "  lispkit is fun ")                          ⇒  "lispkit is fun"
(string-trim "________" "_")                               ⇒  ""
(string-trim "712+72=784" (char-set->string char-set:digit))  ⇒  "+72="
(string-trim "712+72=784" char-set:digit)                  ⇒  "+72="
```

(string-pad-right ***str char k*****)**
(string-pad-right ***str char k force-length?*****)**

procedure

Procedure `string-pad-right` returns a newly allocated string created by padding string *str* at the beginning of the string with character *char* until it is of length *k*. If *k* is less than the length of string *str*, the resulting string gets truncated at length *k* if boolean argument *force-length?* is `#t`; otherwise, the string *str* gets returned as is.

```
(string-pad-right "scheme" #\space 8)    ⇒ "scheme  "
(string-pad-right "scheme" #\x 4)        ⇒ "scheme"
(string-pad-right "scheme" #\x 4 #t)     ⇒ "sche"
(string-pad-right "scheme" "_" 10)       ⇒ "scheme____"
```

(string-pad-left ***str char k*****)**
(string-pad-left ***str char k force-length?*****)**

procedure

Procedure `string-pad-left` returns a newly allocated string created by padding string *str* at the beginning of the string with character *char* until it is of length *k*. If *k* is less than the length of string *str*, the resulting string gets truncated at length *k* if boolean argument *force-length?* is `#t`; otherwise, the string *str* gets returned as is.

```
(string-pad-left "scheme" #\space 8)    ⇒ "  scheme"
(string-pad-left "scheme" #\x 4)        ⇒ "scheme"
(string-pad-left "scheme" #\x 4 #t)     ⇒ "heme"
(string-pad-left "scheme" "_" 10)       ⇒ "____scheme"
```

(string-pad-center ***str char k*****)**
(string-pad-center ***str char k force-length?*****)**

procedure

Procedure `string-pad-center` returns a newly allocated string created by padding string *str* at the beginning and end with character *char* until it is of length *k*, such that *str* is centered in the middle. If *k* is less than the length of string *str*, the resulting string gets truncated at length *k* if boolean argument *force-length?* is `#t`; otherwise, the string *str* gets returned as is.

```
(string-pad-center "scheme" #\space 8)  ⇒ " scheme "
(string-pad-center "scheme" #\x 4)      ⇒ "scheme"
(string-pad-center "scheme" #\x 4 #t)   ⇒ "heme"
(string-pad-center "scheme" "_" 10)     ⇒ "__scheme__"
```

50.4 Manipulating strings

(string-replace! ***str sub repl*****)**
(string-replace! ***str sub repl start*****)**
(string-replace! ***str sub repl start end*****)**

procedure

Replaces all occurences of string *sub* in string *str* between indices *start* and *end* with string *repl* and returns the number of occurences of *sub* that were replaced.

(string-replace-first! ***str sub repl*****)**
(string-replace-first! ***str sub repl start*****)**
(string-replace-first! ***str sub repl start end*****)**

procedure

Replaces the first occurence of string *sub* in string *str* between indices *start* and *end* with string *repl* and returns the index at which the first occurence of *sub* was replaced.

(string-insert! ***str repl*****)** procedure
(string-insert! ***str repl start*****)**
(string-insert! ***str repl start end*****)**

Replaces the part of string *str* between index *start* and *end* with string *repl*. The default for *start* is 0, for *end* it is *start* (i.e. if not provided, *end* is equals to *start*). If both *start* and *end* are not provided, `string-insert!` inserts *repl* at the beginning of *str*. If *start* is provided alone (without *end*), `string-insert!` inserts *repl* at position *start*.

```
(define s "Zenger is my name")
(string-insert! s "Matthias ")
s ⇒ "Matthias Zenger is my name"
(string-insert! s "has always been" 16 18)
s ⇒ "Matthias Zenger has always been my name"
```

(string-append! ***str other …*****)** procedure
Appends the strings *other*, … to mutable string *str* in the given order.

(string-copy! ***to at from*****)** procedure
(string-copy! ***to at from start*****)**
(string-copy! ***to at from start end*****)**

Copies the characters of string *from* between index *start* and *end* to string *to*, starting at index *at*. If the source and destination overlap, copying takes place as if the source is first copied into a temporary string and then into the destination. It is an error if *at* is less than zero or greater than the length of string *to*. It is also an error if `(- (string-length to) at)` is less than `(- end start)`.

(string-fill! ***str fill*****)** procedure
(string-fill! ***str fill start*****)**
(string-fill! ***str fill start end*****)**

The `string-fill!` procedure stores *fill* in the elements of string *str* between index *start* and *end*. It is an error if *fill* is not a character.

50.5 Iterating over strings

(string-map ***proc str …*****)** procedure
The `string-map` procedure applies procedure *proc* element-wise to the characters of the strings *str* … and returns a string of the results, in order. If more than one string *str* is given and not all strings have the same length, `string-map` terminates when the shortest string runs out. It is an error if *proc* does not accept as many arguments as there are strings and returns a single character.

```
(string-map char-foldcase "AbdEgH") ⇒ "abdegh"
(string-map (lambda (c) (integer->char (+ 1 (char->integer c)))) "HAL") ⇒ "IBM"
```

(string-for-each ***proc str …*****)** procedure
The arguments to `string-for-each` are like the arguments to `string-map`, but `string-for-each` calls *proc* for its side effects rather than for its values. Unlike `string-map`, `string-for-each` is guaranteed to call *proc* on the characters of the strings in order from the first character to the last. If more than one string *str* is given and not all strings have the same length, `string-for-each` terminates when the shortest string runs out. It is an error for *proc* to mutate any of the strings. It is an error if *proc* does not accept as many arguments as there are strings.

50.6 Converting strings

(string->list *str*) procedure
(string->list *str start*)
(string->list *str start end*)

The `string->list` procedure returns a list of the characters of string *str* between *start* and *end* preserving the order of the characters.

50.7 Input/Output

(read-file *path*) procedure
Reads the text file at *path* and stores its content in a newly allocated string which gets returned by `read-file`.

(write-file *path str*) procedure
Writes the characters of string *str* into a new text file at *path*. `write-file` returns `#t` if the file could be written successfully; otherwise `#f` is returned.

51 LispKit Styled-Text

Library `(lispkit styled-text)` provides an API to define and manipulate styled text. A styled text object is a string with individual character ranges being layed out using a range of stylistic attributes. Library `(lispkit styled-text)` defines the layout of text in terms of objects encapsulating these stylistic attributes. There are three different style parameter collections: text styles, text block styles, and paragraph styles. Besides textual content, styled text objects may also contain tables (on macOS) and images.

Library `(lispkit styled-text)` also supports loading styled text from RTF, RTFD, and various Word formats (doc, docx). It is also possible to save styled text objects in these formats.

51.1 Styled text

A styled text object is mutable and consists of a string and a set of character ranges associated with stylistic attributes determining how the range of characters is layed out. Both the string and the attributed character ranges can be mutated. The following stylistic attributes are supported:

- `background-color` : Color object defining the background color of the text range.
- `foreground-color` : Color object defining the text color of the text range.
- `strikethrough-color` : Color object defining the strike-through color of the text range.
- `stroke-color` : Color object defining the outline color of the text range, i.e. the stroke color used for text displayed in outlined style.
- `underline-color` : Color object defining the underline color of the text range for text using underlined style.
- `baseline-offset` : The vertical offset for the position of the text in points.
- `expansion` : The expansion factor of the text, i.e. a flonum corresponding to the log of the expansion factor to be applied to the glyphs. 0 is the default, indicating no expansion.
- `kern` : The kerning of the text, i.e. the number of points by which to adjust kern-pair characters. This can be used to reduce/create space between characters. Kerning gets disabled by setting this attribute to 0.
- `obliqueness` : The obliqueness of the text expressed as a flonum indicating skew to be applied to the glyphs. 0 is the default, indicating no skew.
- `stroke-width` : The width of the stroke, i.e. the amount to change the stroke width for outlined text and is specified as a percentage of the font point size. 0 represents the default outline stroke width, negative values make the stroke extend inward, positive values extend it outward.
- `ligature` : A boolean value indicating whether ligatures should be used in the text range.
- `font` : Font object defining the font used in the text range.
- `link` : A string representing the link used for the text range. This is, in most cases, a URL, but could also be a file path.
- `paragraph-style` : Paragraph style object defining all parameters for laying out paragraphs in the text range.
- `shadow` : The shadow of the text range. This is defined by a pair whose car is a size object and cdr is a positive flonum representing the shadow blur radius. The size object represents vertical and horizontal offsets of the shadow. Example: `((2.0 . -3.0) . 6.5)`.

- `superscript` : The superscript of the text expressed as an offset in points.

Library `(lispkit styled-text)` provides functionality to create styled text objects, to compose them, to style them, and to introspect existing stylistic attributes.

(styled-text? *obj*) procedure

Returns `#t` if *obj* is a styled text object, otherwise `#f` is returned.

(styled-text *str*) procedure
(styled-text *str style*)
(styled-text *str font*)
(styled-text *str font color*)
(styled-text *str font color pstyle*)

Creates and returns a styled text object representing string *str* using the stylistic attributes provided by text style object *style*, if provided. Alternatively, the given *font*, *color*, and paragraph style *pstyle* objects are used to define the style of *str*.

(make-styled-text *str key val* ...) procedure
(make-styled-text *image*)

Creates and returns a styled text object representing string *str* layed out by the given stylistic attributes provided as key/value pairs. The attributes are applicable to the whole string. The following attribute *key* symbols are supported:

- `background-color:` Background color
- `foreground-color:` Text color
- `strikethrough-color:` Strike-through color
- `stroke-color:` Outline color
- `underline-color:` Underline color
- `baseline-offset:` Vertical offset for position of the text in points
- `expansion:` Text expansion factor
- `kern:` The kerning of the text, i.e. the number of points by which to adjust kern-pair characters
- `obliqueness:` The skew to be applied to the glyphs
- `stroke-width:` The relative width of the outline stroke
- `ligature:` Boolean indicating whether ligatures are used
- `font:` Font of the text
- `link:` Link target string, e.g. a URL
- `paragraph-style:` Paragraph style for laying out paragraphs
- `shadow:` : The shadow defined by a pair whose car is a size object defining horizontal and vertical offset and cdr is a positive flonum representing the shadow blur radius
- `superscript:` : The superscript offset in points

The second use case for `make-styled-text` is creating a styled text object for a given image.

This example shows how to use `make-styled-text` :

```
(make-styled-text "Mauris scelerisque massa erat."
  'font: (font "Helvetica" 12.0)
  'foreground-color: (color 0.3 0.5 0.7)
  'paragraph-style:
    (make-paragraph-style
      'alignment: 'left
      'head-indent: 40.0
      'paragraph-spacing-before: 4.0))
```

(make-styled-text-table *cols rows*) procedure
(make-styled-text-table *cols rows style*)
(make-styled-text-table *cols rows style pstyle*)
(make-styled-text-table *cols rows style pstyle collapse*)
(make-styled-text-table *cols rows style pstyle collapse hide-empty*)

This procedure is only available on macOS. It creates a styled text table with *cols* number of columns. *rows* is a list of table rows. Each row is a list of table columns. A table column is either a string, styled text, or a text cell descriptor, which is a list containing at least a prefix of the following four components: (*text col-span tbstyle pstyle*). *text* is either a string or styled text, *col-span* is a number indicating how many columns the cell spans, *tbstyle* is a text block style object and *pstyle* is a paragraph style object, both of which are used to lay out the cell content.

style is a default text block style object, and *pstyle* is a default paragraph style object. They are both used to lay out cells which do not come with their own stylistic attributes. *collapse* is a boolean argument which collapses table borders if set to `#t` (which is the default). *hide-empty* is a boolean argument which hides empty cells if set to `#t` (`#f` is the default).

Styled text tables are represented as a styled text object, so `make-styled-text-table` returns styled text.

```
(make-styled-text-table 3
  (list ; list of rows
    (list ; list of columns
      "Cell 1,1"
      "Cell 1,2"
      (styled-text "Cell 1,3" (font "Helvetica" 11.0)))
    (list ; list of columns
      "Cell 2,1"
      (list ; column spanning 2 cells
        (styled-text "Cell 2,3" (font "Times" 10.0) red)
        2
        tbstyle
        pstyle)))
    def-tbstyle ; default text block style
    def-pstyle  ; default paragraph style
    #t)
```

(load-styled-text *path format*) procedure
Loads the document at file *path* and returns its content as styled text. *format* specifies the file format to load. *format* is one of the following symbols:

- `plain` : Plain text file
- `doc` : Microsoft Word file
- `docx` : ECMA Office Open XML text document
- `rtf` : RTF file
- `rtfd` : RTFD file

(save-styled-text *path txt format*) procedure
Saves the styled text *txt* in a new file at file *path* in file *format*. *format* is one of the following symbols:

- `plain` : Plain text file
- `doc` : Microsoft Word file
- `docx` : ECMA Office Open XML text document
- `rtf` : RTF file
- `rtfd` : RTFD file

(copy-styled-text *txt*) procedure
(copy-styled-text *txt start*)
(copy-styled-text *txt start end*)

Returns a copy of styled text *txt* between positions *start* (inclusive) and *end* (exclusive). *start* is an index between 0 and the length of *txt* (default is 0). *end* is an index between *start* and the length of *txt* (default is the length of *txt*).

(html->styled-text *html*) procedure
Returns styled text for the given *html* string.

(styled-text->html *txt*) procedure
(styled-text->html *txt start*)
(styled-text->html *txt start end*)

Returns HTML as a string representing the styled text *txt* between position *start* and *end*. If *end* is not provided, it is assumed to be the length of *txt*. If *start* is not provided, it is assumed to be 0.

(bytevector->styled-text *bvec format*) procedure
(bytevector->styled-text *bvec format start*)
(bytevector->styled-text *bvec format start end*)

`bytevector->styled-text` interprets bytevector *bvec* between *start* and *end* as a file of text *format* and returns its content as a new styled text object. If *end* is not provided, it is assumed to be the length of *bytevector*. If *start* is not provided, it is assumed to be 0.

(styled-text->bytevector *txt format*) procedure
(styled-text->bytevector *txt format start*)
(styled-text->bytevector *txt format start end*)

Stores the styled text *txt* between position *start* and *end* in the given *format* in a new bytevector and returns that bytevector. If *end* is not provided, it is assumed to be the length of *bytevector*. If *start* is not provided, it is assumed to be 0. *format* is one of the following symbols:

- `plain` : Plain text file
- `doc` : Microsoft Word file
- `docx` : ECMA Office Open XML text document
- `rtf` : RTF file

(styled-text=? *txt0 txt1 …*) procedure
Returns `#t` if *txt0*, *txt1*, … are all equals, otherwise `#f` is returned.

(styled-text-string *txt*) procedure
Returns a string for the given styled text *txt*.

(styled-text-insert! *txt obj*) procedure
(styled-text-insert! *txt obj start*)
(styled-text-insert! *txt obj start end*)

Inserts *obj* into the styled text *txt* replacing the characters between the position *start* and *end* with *obj*. If *obj* is `#f`, then the characters between *start* and *end* are being deleted. If *obj* is a string, styled text or an image, *obj* gets converted into styled text and inserted accordingly. If *start* is not provided, it is assumed to be 0. If *end* is not provided, it is assumed to be the same like *start*.

(styled-text-append! *txt obj …*) procedure
Appends the objects *obj*, … to the styled text *txt* in the given order. *obj* may be either a string, styled text, or an image.

(styled-text-ref *txt index*) procedure
Returns a text style object encapsulating all stylistic attributes that are applicable to the character at *index* of styled text *txt*.

(styled-text-set! ***txt start end style*****)** procedure
(styled-text-set! ***txt start end key val*****)**

If *style* is provided, sets the stylistic attributes of styled text *txt* in the character range from *start* (inclusive) to *end* (exclusive) to the attributes encapsulated by *style*. If *key* and *val* are provided instead, procedure `styled-text-set!` sets a single attribute *key* to the value *val* for the given character range of *txt*. Supported keys are: `background-color`, `foreground-color`, `strikethrough-color`, `stroke-color`, `underline-color`, `baseline-offset`, `expansion`, `kern`, `obliqueness`, `stroke-width`, `ligature`, `font`, `link`, `paragraph-style`, `shadow`, and `superscript`.

(styled-text-add! ***txt start end style*****)** procedure
(styled-text-add! ***txt start end key val*****)**

If *style* is provided, adds the stylistic attributes encapsulated by *style* to the existing stylistic attributes of styled text *txt* for the character range from *start* (inclusive) to *end* (exclusive). If *key* and *val* are provided instead, procedure `styled-text-add!` adds a single attribute *key* to the value *val* for the given character range of *txt*. Supported keys are: `background-color`, `foreground-color`, `strikethrough-color`, `stroke-color`, `underline-color`, `baseline-offset`, `expansion`, `kern`, `obliqueness`, `stroke-width`, `ligature`, `font`, `link`, `paragraph-style`, `shadow`, and `superscript`.

(styled-text-remove! ***txt start end key*****)** procedure
Removes the stylistic attribute *key* from the styled text *txt* in the character range from *start* (inclusive) to *end* (exclusive).

(styled-text-attribute ***txt key index*****)** procedure
(styled-text-attribute ***txt key index start*****)**
(styled-text-attribute ***txt key index start end*****)**

This procedure returns two values. The first is the stylistic attribute value for attribute *key* at character *index* in the styled text *txt* within the character range from *start* (inclusive) to *end* (exclusive). The second return value is the longest effective range of this attribute. If the attribute is not set at the given *index*, `styled-text-attribute` returns two `#f` values.

(styled-text-attributes ***txt index*****)** procedure
(styled-text-attributes ***txt index start*****)**
(styled-text-attributes ***txt index start end*****)**

This procedure returns two values. The first is a text style object capturing all stylistic attributes at character *index* in the styled text *txt* within the character range from *start* (inclusive) to *end* (exclusive). The second return value is the longest effective range of this text style. Two `#f` values are returned if no attributes are found.

(styled-text-first-attribute ***text key*****)** procedure
(styled-text-first-attribute ***text key start*****)**
(styled-text-first-attribute ***text key start end*****)**

This procedure returns two values. The first is the first stylistic attribute value for attribute *key* in the styled text *txt* within the character range from *start* (inclusive) to *end* (exclusive). The second return value is the longest effective range of this attribute. If the attribute is not set in range *start* to *end*, `styled-text-first-attribute` returns two `#f` values.

(styled-text-first-attributes ***txt*****)** procedure
(styled-text-first-attributes ***txt start*****)**
(styled-text-first-attributes ***txt start end*****)**

This procedure returns two values. The first is a text style object capturing all stylistic attributes that were found first in the styled text *txt* within the character range from *start* (inclusive) to *end* (exclusive). The second return value is the longest effective range of this text style. Two `#f` values are returned if no attributes are found.

51.2 Text styles

Text style objects encapsulate a set of stylistic attributes, such as `font`, `background-color`, `baseline-offset`, `kern`, `obli queness`, etc. Text style objects are mutable. Attributes can be inspected, added, and removed.

(text-style? *obj*) procedure

Returns `#t` if *obj* is a text style object; otherwise `#f` is returned.

(make-text-style *key val* ...) procedure

Returns a text style object encapsulating the given stylistic attributes provided as key/value pairs. The following attribute *key* symbols are supported:

- `background-color:` Background color
- `foreground-color:` Text color
- `strikethrough-color:` Strike-through color
- `stroke-color:` Outline color
- `underline-color:` Underline color
- `baseline-offset:` Vertical offset for position of the text in points
- `expansion:` Text expansion factor
- `kern:` The kerning of the text, i.e. the number of points by which to adjust kern-pair characters
- `obliqueness:` The skew to be applied to the glyphs
- `stroke-width:` The relative width of the outline stroke
- `ligature:` Boolean indicating whether ligatures are used
- `font:` Font of the text
- `link:` Link target string, e.g. a URL
- `paragraph-style:` Paragraph style for laying out paragraphs
- `shadow:` : The shadow defined by a pair whose car is a size object defining horizontal and vertical offset and cdr is a positive flonum representing the shadow blur radius
- `superscript:` : The superscript offset in points

This example shows how to use `make-text-style`:

```
(make-text-style
  'font: (font "Times" 13.5)
  'foreground-color: (color 0.8 0.8 0.8)
  'paragraph-style:
    (make-paragraph-style
      'alignment: 'left
      'paragraph-spacing-before: 5.0))
```

(copy-text-style *style*) procedure

Returns a copy of *style*.

(text-style-empty? *style*) procedure

Returns `#t` if the *style* object does not include any stylistic attributes; otherwise `#f` is returned.

(text-style-ref *style key*) procedure

Returns the attribute value for stylistic attribute *key* defined by *style*. `#f` is returned if no value is set.

(text-style-set! *style key value*) procedure

Sets the stylistic attribute *key* to *value* in *style*.

(text-style-merge! *style style1* ...) procedure

Merges all the text style objects *style1* ... into *style*. The text style objects are merged in the order provided, i.e. later values override values in earlier style objects.

(text-style-remove! *style key*) procedure

Removes the stylistic attribute *key* from *style*.

(text-style-attributes *expr*) procedure

Returns the stylistic attributes of *style* as an association list. The result is a list of key/value pairs.

```
(text-style-attributes
  (make-text-style
    'font: (font "Times" 13.5)
    'foreground-color: (color 0.8 0.8 0.8)
    'obliqueness: 1.2))
⇒ ((font . #<font Times-Roman 13.5>)
   (obliqueness . 1.2)
   (foreground-color . #<color 0.8 0.8 0.8>))
```

51.3 Text block styles

Text block style objects encapsulate attributes describing how text is layed out in a box. Text block styles are currently only used for defining the layout of cells in a table. The following text block style attributes are supported:

- `width` : The width of the text block in points or as a percentage.
- `height` : The height of the text block in points or as a percentage.
- `margin` : The margin around the text block. This is either a value representing the same margin in points or as a percentage for all four sides, or it is a list of four values `(left right top bottom)` .
- `border` : The “thickness”/size of the border in points or as a percentage. This is either one value representing the same border size for all four sides, or it is a list of four values `(left right top bottom)` . 0 means “no border”.
- `padding` : The padding within the text block in points or as a percentage. This is either one value representing the same padding for all four sides, or it is a list of four values `(left right top bottom)` .
- `background-color` : The background color of the text block.
- `border-color` : The color of the border of the text block.
- `vertical-alignment` : The vertical alignment of the text within the block. Supported are the following four alignment values: `top` , `middle` , `bottom` , and `baseline` .

Text block style objects are mutable. They define values for all attributes, i.e. there are defaults for all attributes set for newly created text block style objects.

(percent *num*) procedure

Some text block style attributes allow for relative values. Procedure `percent` encodes a fixnum *num* as a percentage (i.e. `(num . %)`).

(percent? *obj*) procedure

Returns `#t` if *obj* is an object representing a percentage; otherwise `#f` is returned.

(text-block-style? *obj*) procedure

Returns `#t` if *obj* is a text block style object; otherwise `#f` is returned.

(make-text-block-style *key value* ...) procedure

Returns a text block style object encapsulating the given attributes provided as key/value pairs. The following attribute *key* symbols are supported:

- `width:` The width of the text block in points or as a percentage.
- `height:` The height of the text block in points or as a percentage.

- `margin:` The margin around the text block. This is either a value representing the same margin in points or as a percentage for all four sides, or it is a list of four values `(left right top bottom)`.
- `border:` The "thickness"/size of the border in points or as a percentage. This is either one value representing the same border size for all four sides, or it is a list of four values `(left right top bottom)`. 0 means "no border".
- `padding:` The padding within the text block in points or as a percentage. This is either one value representing the same padding for all four sides, or it is a list of four values `(left right top bottom)`.
- `background-color:` The background color of the text block.
- `border-color:` The color of the border of the text block.
- `vertical-alignment:` The vertical alignment of the text within the block. Supported are the following four alignment values: `top`, `middle`, `bottom`, and `baseline`.

(copy-text-block-style *bstyle*) procedure
Returns a copy of *bstyle*.

(text-block-style=? *bstyle bstyle0 …*) procedure
Returns `#t` if all *bstyle0* … are equal to *bstyle*; otherwise `#f` is returned.

(text-block-style-ref *bstyle key*) procedure
Returns the value associated with text block style attribute *key*.

(text-block-style-set! *bstyle key value*) procedure
Sets the text block style attribute *key* to *value* for the text block style object *bstyle*.

51.4 Paragraph styles

Paragraph style objects define how a paragraph of text is layed out in terms of a number of attributes. The following attributes are supported:

- `alignment`: Horizontal text alignment mode. Supported are: `left`, `right`, `center`, `justified`, and `natural`.
- `first-head-indent`: Distance in points from the leading margin of a text container to the beginning of the paragraph's first line.
- `head-indent`: Distance in points from the leading margin of a text container to the beginning of lines other than the first.
- `tail-indent`: If positive, this is the distance from the leading margin (e.g. the left margin in left-to-right text). If 0 or negative, it's the distance from the trailing margin. For example, a paragraph style designed to fit exactly in a container has a head indent of 0.0 and a tail indent of 0.0. One designed to fit with a quarter-inch margin has a head indent of 0.25 and a tail indent of –0.25.
- `line-height-multiple`: Multiplier for the natural line height of the text (if positive), and constrains the resulting value by the minimum and maximum line height. The default value is 0.0.
- `max-line-height`: This attribute defines the maximum height in points that any line in the paragraph occupies, regardless of the font size or size of any attached image. Default is 0 (= no constaint).
- `min-line-height`: This attribute defines the minimum height in points that any line in the paragraph occupies, regardless of the font size or size of any attached image. Default is 0.
- `line-spacing`: The distance in points between the bottom of one line fragment and the top of the next.
- `paragraph-spacing-after`: Distance between the bottom of a paragraph and the top of the next. The layout algorithm determines the space between paragraphs by adding `paragraph-spacing-after` of the previous paragraph to the next paragraph's `paragraph-spacing-before`.

- `paragraph-spacing-before` : The distance between the paragraph's top and the beginning of its text content.
- `tab-interval` : The default tab interval in points. Tabs after the last specified tab stops are placed at multiples of this distance (if positive). Default is 0.0.
- `text-fit-mode` : Attribute that specifies what happens when a line is too long for a container. Supported are: `word-wrap` , `char-wrap` , `clip` , `truncate` , `truncate-head` , and `truncate-tail` .
- `push-out-line-break` : Boolean value that, if set to `#t` , makes the line layout algorithm push out individual lines to avoid an orphan word on the last line of a paragraph.
- `hypenation-factor` : A paragraph's threshold for hyphenation. The line layout algorithm attempts hyphenation when the ratio of the text width (as broken without hyphenation) to the width of the line fragment is less than the hyphenation factor. Default is 0 (= system-defined hyphenation threshold).
- `writing-direction` : Writing direction of a paragraph. Supported are: `natural` (automatic), `left-to-right` , and `right-to-left` .

Paragraph style objects are mutable. They encapsulate one value for each supported paragraph style attribute.

(paragraph-style? *obj*) procedure
Returns `#t` if *obj* is a paragraph style object *obj*; otherwise `#f` is returned.

(make-paragraph-style *key value* ...) procedure
Returns a paragraph style object encapsulating the given attributes provided as key/value pairs. The following attribute *key* symbols are supported:

- `alignment` : Horizontal text alignment mode; i.e. either `left` , `right` , `center` , `justified` , and `natural` .
- `first-head-indent` : Distance in points from the leading margin of a text container to the beginning of the paragraph's first line.
- `head-indent` : Distance in points from the leading margin of a text container to the beginning of lines other than the first.
- `tail-indent` : Distance from the leading margin if positive. If 0 or negative, it's the distance from the trailing margin.
- `line-height-multiple` : Multiplier for the natural line height of the text.
- `max-line-height` : Defines the maximum height in points that any line in the paragraph occupies.
- `min-line-height` : Defines the minimum height in points that any line in the paragraph occupies.
- `line-spacing` : The distance in points between the bottom of one line fragment and the top of the next.
- `paragraph-spacing-after` : Distance between the bottom of a paragraph and the top of the next.
- `paragraph-spacing-before` : The distance between the paragraph's top and the beginning of its text content.
- `tab-interval` : The default tab interval in points.
- `text-fit-mode` : Specifies what happens when a line is too long; supported are: `word-wrap` , `char-wrap` , `clip` , `truncate` , `truncate-head` , and `truncate-tail` .
- `push-out-line-break` : Boolean that, if set to `#t` , makes the line layout algorithm push out individual lines to avoid an orphan word on the last line of a paragraph.
- `hypenation-factor` : A paragraph's threshold for hyphenation.
- `writing-direction` : Writing direction of a paragraph; supported are: `natural` , `left-to-right` , and `right-to-left` .

(copy-paragraph-style *pstyle*) procedure
Returns a copy of *pstyle*.

(paragraph-style=? *pstyle pstyle0* …) procedure

Returns `#t` if all *pstyle0* … are equal to *pstyle*; otherwise `#f` is returned.

(paragraph-style-ref *pstyle key*) procedure

Returns the value associated with paragraph style attribute *key*.

(paragraph-style-set! *pstyle key value*) procedure

Sets the paragraph style attribute *key* to *value* for the paragraph style object *pstyle*.

(paragraph-style-tabstops *pstyle*) procedure

Returns a list of tab stops for the given paragraph style. Each tab stop is represented as a pair consisting of a location in points and a text alignment, which is either `left`, `right`, `center`, `justified`, or `natural`.

(paragraph-style-tabstop-add! *pstyle loc*) procedure
(paragraph-style-tabstop-add! *pstyle loc align*)
(paragraph-style-tabstop-add! *pstyle loc align cs*)

Adds a new tab stop to paragraph style *pstyle*. *loc* is the location of the tab stop in points, *align* is the alignment of the text at the location (e.g. one of `left`, `right`, `center`, `justified`, and `natural`), and *cs* is a char-set object that is used to determine the terminating character for a tab column. The tab and newline characters are implied even if they don't exist in the character set. The default for *align* is `natural`.

(paragraph-style-tabstop-remove! *pstyle loc*) procedure
(paragraph-style-tabstop-remove! *pstyle loc align*)

Removes the tab stop in *pstyle* at the given location *loc* and using the provided text alignment *align* (default is `natural`).

(paragraph-style-tabstops-clear! *pstyle*) procedure

Removes all tab stops from paragraph style *pstyle*.

52 LispKit System

52.1 File paths

Files and directories are referenced by *paths*. Paths are strings consisting of directory names separated by character `'/'` optionally followed by a file name (if the path refers to a file) and a path extension (sometimes also called *file name suffix*, if the path refers to a file). Paths are either *absolute*, if they start with character `'/'`, or they are *relative* to some unspecified directory.

If a relative path is used to refer to a concrete directory or file, e.g. in the API provided by library `(lispkit port)`, typically the path is interpreted as relative to the path as defined by the parameter object `current-directory`, unless specified otherwise.

current-directory [parameter object]
Defines the path referring to the *current directory*. Each LispKit virtual machine has its own current directory.

source-directory [syntax]
Returns the directory in which the source file is located which is currently being compiled and executed. Typically, such source files are executed via procedure `load`.

(home-directory) [procedure]
(home-directory *username*)
(home-directory *username non-sandboxed?*)

Returns the path of the home directory of the user identified via string *username*. If *username* is not given or is set to `#f`, the name of the current user is used as a default. The name of the current user can be retrieved via procedure `current-user-name`. If boolean argument *non-sandboxed?* is set to `#t`, `home-directory` will return the Unix home directory path, even if LispKit is used in a sandboxed application such as LispPad.

```
(home-directory "objecthub")  ⇒  "/Users/objecthub")
```

(system-directory *type*) [procedure]
Returns a list of paths to system directories specified via symbol *type* for the current user. In most cases, a single value is returned. The following `type` values are supported:

- `desktop` : The “Desktop” folder.
- `downloads` : The “Downloads” folder.
- `movies` : The “Movies” folder.
- `music` : Ths “Music folder.
- `pictures` : The “Pictures” folder.
- `documents` : The “Documents” folder.
- `icloud` : The “Documents” folder on iCloud.
- `shared-public` : The “Public” folder.
- `application-support` : The application support directory on iOS (only accessible to the application hosting LispKit)

- `application-scripts` : The folder where AppleScript source code is stored (only available on macOS).
- `cache` : The cache directory on iOS.
- `temporary` : A shared temporary folder.

```
(system-directory 'documents)  ⇒  ("/Users/objecthub/Documents")
(system-directory 'desktop)  ⇒  ("/Users/objecthub/Desktop")
```

(path *path comp ...*) procedure

Constructs a new relative file or directory path consisting of a relative (or absolute) base path *base* and a number of path components *comp* .... If it is not possible to coonstruct a valid path, this procedure returns `#f` .

```
(path "one" "two" "three.png")  ⇒  "one/two/three.png"
```

(parent-path *path*) procedure

Returns the parent path of *path*. The result is either a relative path if *path* is relative, or the result is an absolute path. `parent-path` returns `#f` if *path* is not a valid path.

```
(parent-path "one/two/three.png")  ⇒  "one/two"
(parent-path "three.png")          ⇒  "."
```

(path-components *path*) procedure

Returns the individual components of a (relative or absolute) *path* as a list of strings. Returns `#f` if *path* is not a valid path.

```
(path-components "one/two/three.png")  ⇒  ("one" "two" "three.png")
```

(file-path *path*) procedure
(file-path *path base*)
(file-path *path base resolve?*)

Constructs a new absolute file or directory path consisting of a base path *base* and a relative file path *path*. Procedure `file-path` will also resolve symbolic links if boolean argument *resolve?* is `#t` . The default for *resolve?* is `#f` . Tilde is always resolved, if provided either in *path* or *base*.

```
(file-path "Photos/img.jpg" "/Users/mz/Desktop")
⇒  "/Users/mz/Desktop/Photos/img.jpg"
(file-path "~/Images/test.jpg")
⇒  "/Users/objecthub/Images/test.jpg"
(file-path "~/Images/test.jpg" "/random")
⇒  "/Users/objecthub/Images/test.jpg"
```

(asset-file-path *name type*) procedure
(asset-file-path *name type dir*)

Returns a new absolute file or directory path to a LispKit *asset*. An asset is identified via a file *name*, a file *type*, and an optional directory path *dir*. *name*, *type*, and *dir* are all strings. An asset is a file which is located directly or indirectly in one of the asset directories part of the LispKit installation. An asset has a *type*, which is the default path extension of the file (e.g. `"png"` for PNG images). If *dir* is provided, it is a relative path to a sub-directory within a matching asset directory.

`asset-file-path` constructs a relative file path in the following way (assuming there is no existing file path extension already):

dir/name.type

It then searches the asset paths in their given order for a file matching this relative file path. Once the first matching file is found, an absolute file path for this file is returned by `asset-file-path`. If no valid (and existing) file is found, `asset-file-path` returns `#f`.

(parent-file-path *path*) procedure

If *path* refers to a file, then `parent-file-path` returns the directory in which this file is contained. If *path* refers to a directory, then `parent-file-path` returns the directory in which this directory is contained. The result of *parent-file-path* is always an absolute path.

(path-extension *path*) procedure

Returns the path extension of *path* or `#f` if there is no path extension.

```
(path-extension "/foo/bar.txt")  ⇒  "txt"
(path-extension "/foo/bar")      ⇒  #f
```

(append-path-extension *path ext opt*) procedure

Appends path extension string *ext* to the file path *path*. The extension is added no matter whether *path* has an extension already or not, unless *opt* is set to `#t`, in which case extension *ext* is only added if there is no extension already.

```
(append-path-extension "/foo/bar" "txt")          ⇒  "/foo/bar.txt"
(append-path-extension "/foo/bar.txt" "mp3")      ⇒  "/foo/bar.txt.mp3"
(append-path-extension "/foo/bar.txt" "mp3" #t)   ⇒  "/foo/bar.txt"
(append-path-extension "" "txt")                  ⇒  #f
```

(remove-path-extension *path*) procedure

Removes the path extension of *path* if one exists and returns the resulting path. If no path extension exists, *path* is returned.

```
(remove-path-extension "/foo/bar")          ⇒  "/foo/bar"
(remove-path-extension "/foo/bar.txt")      ⇒  "/foo/bar"
(remove-path-extension "/foo/bar.txt.mp3")  ⇒  "/foo/bar.txt"
(remove-path-extension "")                  ⇒  ""
```

(file-path-root? *path*) procedure

Returns `#t` if *path* exists and corresponds to the root of the directory hierarchy. The root is typically equivalent to "/". It is an error if *path* is not a string.

52.2 File operations

LispKit supports ways to explore the file system, test if files or directories exist, read and write files, list directory contents, get metadata about files (e.g. file sizes), etc. Most of this functionality is provided by the libraries `(lispkit system)` and `(lispkit port)`.

(file-exists? *filepath*) procedure

The `file-exists?` procedure returns `#t` if the named file exists at the time the procedure is called, and `#f` otherwise. It is an error if *filename* is not a string.

(directory-exists? *dirpath*) procedure

The `directory-exists?` procedure returns `#t` if the named directory exists at the time the procedure is called, and `#f` otherwise. It is an error if *filename* is not a string.

(file-or-directory-exists? *path*) procedure
The `file-or-directory-exists?` procedure returns `#t` if the named file or directory exists at the time the procedure is called, and `#f` otherwise. It is an error if *filename* is not a string.

(file-readable? *path*) procedure
Returns `#t` if the file at *path* exists and is readable; returns `#f` otherwise.

(directory-readable? *path*) procedure
Returns `#t` if the directory at *path* exists and is readable; returns `#f` otherwise.

(file-writable? *path*) procedure
Returns `#t` if the file at *path* exists and is writable; returns `#f` otherwise.

(directory-writable? *path*) procedure
Returns `#t` if the directory at *path* exists and is writable; returns `#f` otherwise.

(file-deletable? *path*) procedure
Returns `#t` if the file at *path* exists and is deletable; returns `#f` otherwise.

(directory-deletable? *path*) procedure
Returns `#t` if the file at *path* exists and is deletab; returns `#f` otherwise.

(delete-file *filepath*) procedure
The `delete-file` procedure deletes the file specified by *filepath* if it exists and can be deleted. If the file does not exist or cannot be deleted, an error that satisfies `file-error?` is signaled. It is an error if *filepath* is not a string.

(delete-directory *dirpath*) procedure
The `delete-directory` procedure deletes the directory specified by *dirpath* if it exists and can be deleted. If the directory does not exist or cannot be deleted, an error that satisfies `file-error?` is signaled. It is an error if *dirpath* is not a string.

(delete-file-or-directory *path*) procedure
The `delete-file-or-directory` procedure deletes the directory or file specified by *path* if it exists and can be deleted. If *path* neither leads to a file nor a directory or the file or directory cannot be deleted, an error that satisfies `file-error?` is signaled. It is an error if *path* is not a string.

(copy-file *filepath targetpath*) procedure
The `copy-file` procedure copies the file specified by *filepath* to the file specified by *targetpath*. An error satisfying `file-error?` is signaled if *filepath* does not lead to an existing file or if a file at *targetpath* cannot be written. It is an error if either *filepath* or *targetpath* are not strings.

(copy-directory *dirpath targetpath*) procedure
The `copy-directory` procedure copies the directory specified by *dirpath* to the directory specified by *targetpath*. An error satisfying `file-error?` is signaled if *dirpath* does not lead to an existing directory or if a directory at *targetpath* cannot be written. It is an error if either *dirpath* or *targetpath* are not strings.

(copy-file-or-directory *sourcepath targetpath*) procedure
The `copy-file-or-directory` procedure copies the file or directory specified by *sourcepath* to the file or directory specified by *targetpath*. An error satisfying `file-error?` is signaled if *sourcepath* does not lead to an existing file or directory, or if a file or directory at *targetpath* cannot be written. It is an error if either *sourcepath* or *targetpath* are not strings.

(move-file *filepath targetpath*) procedure
Moves the file at *filepath* to *targetpath*. This procedure fails if *filepath* does not reference an existing file, or if the file cannot be moved to *targetpath*. It is an error if either *filepath* or *targetpath* are not strings.

(move-directory *dirpath targetpath*) procedure
Moves the directory at *dirpath* to *targetpath*. This procedure fails if *dirpath* does not reference an existing

directory, or if the directory cannot be moved to *targetpath*. It is an error if either *dirpath* or *targetpath* are not strings.

(move-file-or-directory *sourcepath targetpath*) procedure
Moves the file or directory at *sourcepath* to *targetpath*. This procedure fails if *sourcepath* does not reference an existing file or directory, or if the file or directory cannot be moved to *targetpath*. It is an error if either *sourcepath* or *targetpath* are not strings.

(file-size *filepath*) procedure
Returns the size of the file specificed by *filepath* in bytes. It is an error if *filepath* is not a string or if *filepath* does not reference an existing file.

(directory-list *dirpath*) procedure
Returns a list of names of files and directories contained in the directory specified by *dirpath*. It is an error if *dirpath* is not a string or if *dirpath* does not reference an existing directory.

(make-directory *dirpath*) procedure
Creates a directory with path *dirpath*. If the directory exists already or if it is not possible to create a directory with path *dirpath*, `make-directory` fails with an error. It is an error if *dirpath* is not a string.

(open-file *filepath*) procedure
(open-file *filepath app*)
(open-file *filepath app activate*)

Opens the file specified by *filepath* with the application *app*. *app* is either an application name or a file path. *activate* is a boolean argument. If it is `#t`, it will make *app* the frontmost application after invoking it. If *app* is not specified, the default application for the type of the file specified by *filepath* is used. If *activate* is not specified, it is assumed it is `#t`. `open-file` returns `#t` if it was possible to open the file, `#f` otherwise. Example: `(open-file "/Users/objecthub/main.swift" "TextEdit")`.

52.3 Network operations

(open-url *url*) procedure
Opens the given url in the default browser and makes the browser the frontmost application.

(http-get *url*) procedure
(http-get *url timeout*)

`http-get` performs an `http get` request for the given URL. `timeout` is a floating point number defining the time in seconds it should take at most for receiving a response. `http-get` returns two values: the HTTP header in form of an association list, and the content in form of a bytevector. It is an error if the `http get` request fails. Example:

```
(http-get "http://github.com/objecthub")
⇒
(("Date" . "Sat, 17 Nov 2018 22:47:19 GMT")
 ("Referrer-Policy" . "origin-when-cross-origin, strict-origin-when-cross-origin")
 ("X-XSS-Protection" . "1; mode=block")
 ("Status" . "200 OK")
 ("Transfer-Encoding" . "Identity")
 ...
 ("Content-Type" . "text/html; charset=utf-8")
 ("Server" . "GitHub.com"))
#u8(10 10 60 33 68 79 67 84 89 80 69 32 104 116 109 108 62 10 60 104 116 109 108 32 108 97 110
 ↪  103 61 34 101 110 34 62 10 32 32 60 104 101 97 100 62 10 32 32 32 32 60 109 101 116 97 32 99
 ↪  104 97 114 115 101 116 61 34 117 116 102 ...)
```

52.4 Time operations

(current-second) procedure

Returns a floating-point number representing the current time on the International Atomic Time (TAI) scale. The value `0.0` represents midnight on January 1, 1970 TAI (equivalent to ten seconds before midnight UTC) and the value `1.0` represents one TAI second later. **Note:** The current implementation returns the same number like `current-seconds`. This is not conforming to the R7RS spec requiring TAI scale.

(current-jiffy) procedure

Returns the number of jiffies as a fixnum that have elapsed since an arbitrary epoch. A jiffy is a fraction of a second which is defined by the return value of the `jiffies-per-second` procedure. The starting epoch is guaranteed to be constant during a run of the program, but may vary between runs.

(jiffies-per-second) procedure

Returns a fixnum representing the number of jiffies per SI second. Here is an example for how to use `jiffies-per-second`:

```
(define (time-length)
  (let ((list (make-list 100000))
        (start (current-jiffy)))
    (length list)
    (/ (- (current-jiffy) start) (jiffies-per-second))))
```

52.5 Locales

For handling locale-specific behavior, e.g. for formatting numbers and dates, library `(lispkit system)` defines a framework in which

- regions/countries are identified via ISO 3166-1 Alpha 2-code strings,
- languages are identified via ISO 639-1 2-letter strings, and
- locales (i.e. combinations of regions and languages) are identified as symbols.

Library `(lispkit system)` provides functions for returning all available regions, languages, and locales. It also defines functions to map identifiers to human-readable names and to construct identifiers out of other identifiers.

(available-regions) procedure

Returns a list of 2-letter region code identifiers (strings) for all available regions.

(available-region? *obj*) procedure

Returns `#t` if *obj* is a string matching one entry in the list of supported 2-letter region code identifiers. Otherwise, `#f` is returned.

(region-name *ident*) procedure
(region-name *ident locale*)

Returns the name of the region identified by the 2-letter region code string *ident* for the given locale *locale*. If *locale* is not provided, the current (system-provided) locale is used.

(region-flag *ident*) procedure

Returns the flag of the region identified by the 2-letter region code string *ident* as a string containing a single flag emoji.

(available-languages) procedure

Returns a list of 2-letter language code identifiers (strings) for all available languages.

(available-language? *obj*) procedure

Returns `#t` if *obj* is a 2-letter language code identifier string contained in the list of supported/available languages.

(language-name *ident*) procedure
(language-name *ident locale*)

Returns the name of the language identified by the 2-letter language code string *ident* for the given locale *locale*. If *locale* is not provided, the current (system-configured) locale is used.

(available-currencies) procedure

Returns a list of available alpha currency codes based on ISO 4217. A currency code is a 3-letter symbol.

(available-currency? *obj*) procedure

Returns `#t` if *obj* is a valid alpha currency code (symbol) that is contained in the list of supported/available currencies.

(currency-name *ident*) procedure
(currency-name *ident locale*)

Returns the name of the currency identified by the currency identifier *ident* for the given locale *locale*. *ident* can either be a numeric (fixnum) or alpha (symbol or string) currency code as defined by ISO 4217. If *locale* is not provided, the current (system-configured) locale is used.

(currency-code *ident*) procedure

Returns the alpha currency code of the currency identified by the currency identifier *ident*. *ident* can either be a numeric (fixnum) or alpha (symbol or string) currency code as defined by ISO 4217. Returns `#f` if *ident* is not a valid, available currency code symbol.

(currency-numeric-code *ident*) procedure

Returns the numeric currency code of the currency identified by the currency identifier *ident*. *ident* can either be a numeric (fixnum) or alpha (symbol or string) currency code as defined by ISO 4217. Returns `#f` if *ident* is not a valid, available currency code symbol.

(currency-symbol *ident*) procedure
(currency-symbol *ident locale*)

Returns a currency symbol (e.g. "€", "$") for the currency identified by the currency identifier *ident* for the given locale *locale*. *ident* can either be a numeric (fixnum) or alpha (symbol or string) currency code as defined by ISO 4217. If *locale* is not provided, the current (system-configured) locale is used. If there is no currency symbol for a given currency, then `#f` is returned.

(available-locales) procedure

Returns a list of all available locale identifiers (symbols).

(available-locale? *locale*) procedure

Returns `#t` if the symbol *locale* is identifying a locale supported by the operating system; returns `#f` otherwise.

(locale) procedure
(locale *lang*)
(locale *lang country*)

If no argument is provided `locale` returns the current locale (symbol) as configured by the user for the operating system. If the string argument *lang* is provided, a locale representing *lang* (and all countries for which *lang* is supported) is returned. If both *lang* and string *country* are provided, `locale` will return a symbol identifying the corresponding locale.

This function never fails if both *lang* and *country* are strings. It can be used for constructing canonical locale identifiers that are not supported by the underlying operating system. This can be checked with function `available-locale?`.

```
(locale)             ⇒  en_US
(locale "de")        ⇒  de
(locale "en" "GB")   ⇒  en_GB
(locale "en" "de")   ⇒  en_DE
```

(locale-region *locale*) procedure

Returns the 2-letter region code string for the region targeted by the locale identifier *locale*. If *locale* does not target a region, `locale-region` returns `#f`.

(locale-language *locale*) procedure

Returns the 2-letter language code string for the language targeted by the locale identifier *locale*. If *locale* does not target a language, `locale-language` returns `#f`.

(locale-currency *locale*) procedure

Returns the alpha currency code as a symbol for the currency associated with the country targeted by *locale*. If *locale* does not target a country, `locale-currency` returns `#f`.

52.6 Execution environment

(get-environment-variable *name*) procedure

Many operating systems provide each running process with an environment consisting of *environment variables*. Both the name and value of an environment variable are represented as strings. The procedure `get-environment-variable` returns the value of the environment variable *name*, or `#f` if the named environment variable is not found.

```
(get-environment-variable "PATH") ⇒ "/usr/local/bin:/usr/bin:/bin"
```

(get-environment-variables) procedure

Returns the names and values of all the environment variables as an association list, where the car of each entry is the name of an environment variable and the cdr is its value, both as strings. Example: `(("USER" . "root") ("HOME" . "/"))`.

(command-line) procedure

Returns the command line passed to the process as a list of strings. The first string corresponds to the command name.

(features) procedure

Returns a list of the feature identifiers which `cond-expand` treats as true. Here is an example of what `features` might return: `(modules x86-64 lispkit macosx syntax-rules complex 64bit macos little-endian dynamic-loading ratios r7rs)`. LispKit supports at least the following feature identifiers:

- `lispkit`
- `r7rs`
- `ratios`
- `complex`
- `syntax-rules`
- `little-endian`
- `big-endian`
- `dynamic-loading`
- `modules`
- `32bit`

- `64bit`
- `macos`
- `macosx`
- `ios`
- `linux`
- `i386`
- `x86-64`
- `arm64`
- `arm`

(implementation-name) procedure
Returns the name of the Scheme implementation. For LispKit, this function returns the string "LispKit".

(implementation-version) procedure
Returns the version of the Scheme implementation as a string.

(cpu-architecture) procedure
Returns the CPU architecture on which this Scheme implementation is executing as a string.

(machine-name) procedure
Returns a name for the particular machine on which the Scheme implementation is currently running.

(machine-model) procedure
Returns an identifier for the machine on which the Scheme implementation is currently running.

(physical-memory) procedure
Returns the amount of physical memory of the device executing the LispKit code in bytes.

(memory-footprint) procedure
Returns the amount of memory allocated by the application executing the LispKit code in bytes.

(system-uptime) procedure
Returns the uptime of the system in seconds.

(os-type) procedure
Returns the type of the operating system on which the Scheme implementation is running as a string. For macOS, this procedure returns "Darwin".

(os-name) procedure
Returns the name of the operating system on which the Scheme implementation is running as a string. For macOS, this procedure returns "macOS".

(os-version) procedure
Returns the build number of the operating system on which the Scheme implementation is running as a string. For macOS 10.14.1, this procedure returns "18B75".

(os-release) procedure
Returns the (major) release version of the operating system on which the Scheme implementation is running as a string. For macOS 10.14.1, this procedure returns "10.14".

(current-user-name) procedure
Returns the username of the user running the Scheme implementation as a string.

(user-data *username*) procedure
Returns information about the user specified via *username* in form of a list. The list provides the following information in the given order:

1. User id (fixnum)
2. Group id (fixnum)
3. Username (string)

4. Full name (string)
5. Home directory (string)
6. Default shell (string)

Here is an example showing the result for invocation `(user-data "objecthub")`: `(501 20 "objecthub" "Max Mustermann" "/Users/objecthub/" "/bin/bash")`.

(terminal-size) procedure

If a program gets executed in a terminal window, it might be possible to determine the number of columns and rows of that window. In this case, procedure `terminal-size` returns a pair consisting of the number of columns and the number of rows (both in terms of number of characters). If this is not possible, `terminal-size` returns `#f`.

52.7 UUIDs

(make-uuid-string) procedure
(make-uuid-string *bytevector*)
(make-uuid-string *bytevector start end*)
(make-uuid-string *bytevector start end*)

Returns a UUID string. A UUID (Universally Unique Identifier) is a 128-bit label. `make-uuid-string` returns a string with a hexadecimal representation using the 8-4-4-4-12 format. If *bytevector* is not provided, a random UUID is returned. Otherwise, a string representation of the 16-byte bytevector between *start* (inclusive) and *end* (exclusive) is returned.

```
(make-uuid-string) ⇒ "9AF65983-8D44-43CB-AE9B-2FF49ED898BE"
```

(make-uuid-bytevector) procedure
(make-uuid-bytevector str)

Returns a UUID as a bytevector. A UUID (Universally Unique Identifier) is a 128-bit label. If *str* is not provided, a random UUID bytevector is returned. If *str* is provided, it is assumed it is a UUID string (e.g. generated by `make-uuid-string`) and `make-uuid-bytevector` returns a 16-byte representation of this UUID as a bytevector.

53 LispKit System Call

Library `(lispkit system call)` currently defines a single procedure `system-call` for invoking external binaries as a sub-process of the LispKit interpreter. This library is macOS-specific and requires careful usage in portable code.

(system-call ***path args*****)** procedure
(system-call ***path args env*****)**
(system-call ***path args env port*****)**
(system-call ***path args env port input*****)**

Executes the binary at *path* passing the string representation of the elements of list *args* as command-line arguments. *env* is an association list defining environment variables. Both keys and values are strings. The output generated by executing the binary is directed towards *port*, which is a textual output port. The default for *port* corresponds to `current-output-port`, a parameter object defined by library `(lispkit port)`. Providing `#f` as *port* will send the output to `/dev/null`. *input* is an optional string which can be used to pipe data into the binary as input. The current implementation is not able to handle interactive binaries. `system-call` returns the result code for executing the binary (`0` refers to a regular exit).

```
> (system-call "/bin/ls" '(-a -l))
total 863816
drwx------@  47 objecthub    1504 Jun  8 10:56 Desktop
drwx------@  96 objecthub    3072 Jun  7 16:39 Documents
drwx------@ 589 objecthub   18848 May 31 16:59 Downloads
drwx------@  41 objecthub    1312 Dec 19 22:51 Google Drive
drwx------@  84 objecthub    2688 Feb 15 18:32 Library
drwx------+  16 objecthub     512 Oct 20  2019 Movies
drwx------+  10 objecthub     320 Oct 20  2019 Music
drwx------+  10 objecthub     320 May 17 18:37 Pictures
drwxr-xr-x+   5 objecthub     160 Nov 23  2016 Public
0
> (system-call "/usr/bin/bc" '(-q) '() (current-output-port) "10*(11+9)/2\n")
100
0
```

54 LispKit Test

Library `(lispkit test)` provides an API for writing unit tests. The API is largely compatible to similar APIs that are bundled with popular Scheme interpreters and compilers.

54.1 Test groups

Tests are bundled in *test groups*. A test group contains actual *tests* comparing acual with expected values and *nested test groups*. Test groups may be given a *name* which is used for reporting on the testing progress and displaying aggregate test results for each test group.

The following code snippet illustrates how test groups are typically structured:

```
(test-begin "Test group example")
(test "Sum of first 10 integers" 45 (apply + (iota 10)))
(test 64 (gcd 1024 192))
(test-approx 1.414 (sqrt 2.0))
(test-end)
```

This code creates a test group with name `Test group example`. The test group defines three tests, one verifying the result of `(apply + (iota 10))`, one testing `gcd` and one testing `sqrt`. When executed, the following output is shown:

```
╔═══════════════════════════════════════════════════════════════
║ Basic unit tests
╙───────────────────────────────────────────────────────────────
[PASS] Sum of first 10 integers
[PASS] (gcd 1024 192)
[FAIL] (sqrt 2.0): expected 1.414 but received 1.4142135623730951
┌───────────────────────────────────────────────────────────────
│ Basic unit tests
│ 3 tests completed in 0.001 seconds
│ 2 (66.66%) tests passed
│ 1 (33.33%) tests failed
╘═══════════════════════════════════════════════════════════════
```

Procedure `test-begin` opens a new test group. It is optionally given a test group name. Anonymous test groups (without name) are supported, but not encouraged as they make it more difficult to understand the testing output.

Special forms such as `test` and `test-approx` are used to compare expected values with actual result values. Expected values always preceed the actual values. Tests might also be given a name, which is used instead of the expression to test in the test report. `test`, `test-approx`, etc. need to be called in the context of a test group, otherwise the syntactical forms will fail. This is different from other similar libraries which often have an anonymous top-level test group implicitly.

Here is the structure of a more complicated testing setup which has a top-level test group `Library tests` and two nested test groups `Functionality A` and `Functionality B`.

```
(test-begin "Library tests")
  (test-begin "Functionality A")
  (test ...)
  ...
  (test-end)
  (test-begin "Functionality B")
  ...
  (test-end)
(test-end)
```

The syntactic form `test-group` can be used to write small test groups more concisely. This code defines the same test group as above using `test-group`:

```
(test-group "Library tests"
  (test-group "Functionality A"
    (test ...)
    ...)
  (test-group "Functionality B"
    (test ...)
    ...))
```

54.2 Defining test groups

(test-begin) procedure
(test-begin *name*)

A new test group is opened via procedure `test-begin`. *name* defines a name for the test group. The name is primarily used in the test report to refer to the test group.

(test-end) procedure
(test-end *name*)

The currently open test group gets closed by calling procedure `test-end`. Optionally, for documentation and validation purposes, it is possible to provide *name*. If explicitly given, it has to match the name of the corresponding `test-begin` call in terms of `equal?`. When `test-end` is called, a summary gets printed listing stats such as passed/failed tests, the time it took to execute the tests in the group, etc.

(test-exit) procedure
(text-exit *obj*)

This procedure should be placed at the top-level of a test script. It raises an error if it is placed in the context of an open test group. If *obj* is provided and failures were encountered in the previously closed top-level test group, `test-exit` will exit the evaluation of the code by invoking `(exit obj)`.

(test-group *name body* ...) syntax
`test-group` is a syntactical shortcut for opening and closing a new named test group. It is equivalent to:

```
(begin
  (test-begin name)
  body ...
  (test-end))
```

(test-group-failed-tests) procedure
Returns the number of failed tests in the innermost active test group.

(test-group-passed-tests) procedure
Returns the number of passed tests in the innermost active test group.

(failed-tests) procedure
Returns the number of failed tests in all currently active test group.

(passed-tests) procedure
Returns the number of passed tests in all currently active test group.

54.3 Comparing actual with expected values

(test *exp tst*) syntax
(test *name exp tst*)

Main syntax for comparing the result of evaluating expression *tst* with the expected value *exp*. The procedure stored in parameter object *current-test-comparator* is used to compare the actual value with the expected value. *name* is supposed to be a string and used to report success and failure of the test. If not provided, the output of `(display tst)` is used as a name instead. `test` catches errors and prints informative failure messages, including the name, what was expected and what was computed. `test` is a convenience wrapper around `test-equal` that catches common mistakes.

(test-equal *exp tst*) syntax
(test-equal *name exp tst*)
(test-equal *name exp tst eq*)

Compares the result of evaluating expression *tst* with the expected value *exp*. The procedure *eq* is used to compare the actual value with the expected value *exp*. If *eq* is not provided, the procedure stored in parameter object *current-test-comparator* is used as a default. *name* is supposed to be a string and it is used to report success and failure of the test. If not provided, the output of `(display tst)` is used as a name instead. `test-equal` catches errors and prints informative failure messages, including the name, what was expected and what was computed.

(test-assert *tst*) syntax
(test-assert *name tst*)

`test-assert` asserts that the test expression *tst* is not false. It is a convenience wrapper around `test-equal`. *name* is supposed to be a string. It is used to report success and failure of the test. If not provided, the output of `(display tst)` is used as a name instead.

(test-error *tst*) syntax
(test-error *name tst*)

`test-error` asserts that the test expression *tst* fails by raising an error. *name* is supposed to be a string. It is used to report success and failure of the test. If not provided, the output of `(display tst)` is used as a name instead.

(test-approx *exp tst*) syntax
(test-approx *name exp tst*)

Compares the result of evaluating expression *tst* with the expected floating-point value *exp*. The procedure `approx-equal?` is used to compare the actual value with the expected flonum value *exp*. `approx-equal?` uses the parameter object `current-test-epsilon` to determine the precision of the comparison (the default is `0.0000001`). *name* is supposed to be a string. It is used to report success and failure of the test. If not provided, the output of `(display tst)` is used as a name instead. `test-approx` catches errors and prints informative failure messages, including the name, what was expected and what was computed.

(test-not *tst*) syntax
(test-not *name tst*)

`test-not` asserts that the test expression *tst* is false. It is a convenience wrapper around `test-equal`. *name* is supposed to be a string. It is used to report success and failure of the test. If not provided, the output of `(display tst)` is used as a name instead.

(test-values *exp tst*) syntax
(test-values *name exp tst*)

Compares the result of evaluating expression *tst* with the expected values *exp*. *exp* should be of the form `(values x ...)`. As opposed to `test` and `test-equal`, `test-values` works for multiple return values in a portable fashion. The procedure stored in parameter object *current-test-comparator* is used as a comparison procedure. *name* is expected to be a string.

54.4 Test utilities

current-test-comparator parameter object
Parameter object referring to the default comparison procedure for `test` and the `test-*` syntactical forms. By default, `current-test-comparator` refers to `equal?`.

current-test-epsilon parameter object
Maximum difference allowed for inexact comparisons via procedure `approx-equal?`. By default, this parameter object is set to `0.0000001`.

(approx-equal? *x y*) procedure
(approx-equal? *x y epsilon*)

Compares numerical value *x* with numerical value *y* and returns `#t` if *x* and *y* are approximately true. They are approximately true if *x* and *y* differ at most by *epsilon*. If *epsilon* is not provided, the value of parameter object `current-test-epsilon` is used as a default.

(write-to-string *obj*) procedure
Writes value `obj` into a new string using procedure `write`, unless *obj* is a pair, in which case `write-to-string` interprets it as a Scheme expression and uses shortcut syntax for special forms such as `quote`, `quasiquote`, etc. This procedure is used to convert expressions into names of tests.

55 LispKit Text-Table

Library `(lispkit text-table)` provides an API for creating tables of textual content. The library supports column and cell-based text alignment, allows for multi-line rows, and supports different types of row separators.

55.1 Overview

A text table consists of one header row followed by text and separator rows. As part of the header row, it is possible to specify the respective column titles, the text alignment of the header cell, the default text alignment of the corresponding column and a minimum and maximum size of the column (in terms of characters).

Text table rows specify string values for each column. Optionally, it is possible to define a text alignment for each cell that overrides the default column alignment.

The following example shows how text tables are created:

```
(define tt (make-text-table
             '(("ID" center right)
               ("Name" center left)
               ("Address" center left 10 20)
               ("Approved" center center))
             double-line-sep))
(add-text-table-row! tt
  '("1"
    "Mark Smith"
    "2600 Windsor Road\nRedwood City, CA"
    "Yes"))
(add-text-table-separator! tt line-sep)
(add-text-table-row! tt
  '("2"
    "Emily Armstrong"
    "160 Randy Rock Way\nMountain View, CA"
    "No"))
(add-text-table-separator! tt line-sep)
(add-text-table-row! tt
  '("3"
    "Alexander Montgomery"
    "1500 Valencia Street\nSuite 100\nLos Altos, CA"
    "Yes"))
(add-text-table-separator! tt line-sep)
(add-text-table-row! tt
  '("4"
    "Myra Jones"
    "1320 Topaz Street\nPalo Alto, CA"
    "Yes"))
```

A displayable string representation can be generated via procedure `text-table->string`. This is what the result looks like:

ID	Name	Address	Approved
1	Mark Smith	2600 Windsor Road Redwood City, CA	Yes
2	Emily Armstrong	160 Randy Rock Way Mountain View, CA	No
3	Alexander Montgomery	1500 Valencia Street Suite 100 Los Altos, CA	Yes
4	Myra Jones	1320 Topaz Street Palo Alto, CA	Yes

55.2 API

text-table-type-tag object
Symbol representing the `text-table` type. The `type-for` procedure of library `(lispkit type)` returns this symbol for all text table objects.

(text-table? *obj*) procedure
Returns `#t` if *obj* is a text table object; returns `#f` otherwise.

(text-table-header? *obj*) procedure
Returns `#t` if *obj* is a valid text table header. A text table header is a proper list of header cells, one for each column of the text table. A header cell has one of the following forms:

- *"title"*, just specifying the column title.
- *("title" halign)* where *halign* is an alignment specifier (i.e. either `left`, `right`, `center`) that declares how the title is aligned.
- *("title" halign calign)* where *halign* and *calign* are alignment specifiers. *halign* declares how the column title is aligned, *calign* declares how the content in the rest of the column is aligned by default.
- *("title" halign calign min)* where *halign* and *calign* are alignment specifiers and *min* is the minimum size of the column.
- *("title" halign calign min max)* where *halign* and *calign* are alignment specifiers and *min* is the minimum and *max* the maximum size of the column.

(text-table-row? *obj*) procedure
Returns `#t` if *obj* is a valid text table row. A text table row is a proper list of row cells, one for each column of the text table. A row cell has one of the following forms:

- *"content"*, just specifying the content of the cell.
- *("content" align)* where *align* is an alignment specifier (i.e. either `left`, `right`, `center`) that declares how the content in the row cell is aligned.

(make-text-table *headers*) procedure
(make-text-table *headers sep*)
(make-text-table *headers sep edges*)

Returns a new text table with the given header row. *headers* is a valid text table header, *sep* is a separator between header and table rows (i.e. an object for which `text-table-separator?` returns `#t`) and *edges* specifies whether the table edges are round (`round-edges`) or sharp (`sharp-edges`).

```
(make-text-table
  '(("x" center right 3 5) ("f(x)" center right))
  double-line-sep)
```

(add-text-table-row! *table row*) procedure

Adds a new row to the given text table. *row* is a valid text table row, i.e. it is a proper list of row cells, one for each column of the text table. A row cell is either a string or a list with two elements, a string and an alignment specifier (i.e. either `left`, `right`, `center`) which declares how the content in the row cell is aligned.

(add-text-table-separator! *table*) procedure
(add-text-table-separator! *table sep*)

Adds a new row separator to the given table. *sep* is a separator, i.e. it is either `space-sep`, `line-sep`, `double-line-sep`, `bold-line-sep`, `dashed-line-sep`, or `bold-dashed-line-sep`. The default for *sep* is `line-sep`.

(alignment-specifier? *obj*) procedure

Returns `#t` if *obj* is a valid alignment specifier. Supported alignment specifiers are `left`, `right`, and `center`.

left object
right
center

Corresponds to one of the three supported alignment specifiers for text tables.

(text-table-edges? *obj*) procedure

Returns `#t` if *obj* is a valid text table edges specifier. Supported edges specifiers are `no-edges`, `round-edges`, and `sharp-edges`.

no-edges object
round-edges
sharp-edges

Corresponds to one of the three supported edges specifiers for text tables.

(text-table-separator? *obj*) procedure

Returns `#t` if *obj* is a valid text table separator. Supported separators are `no-sep`, `space-sep`, `line-sep`, `double-line-sep`, `bold-line-sep`, `dashed-line-sep`, `bold-dashed-line-sep`.

no-sep object
space-sep
line-sep
double-line-sep
bold-line-sep
dashed-line-sep
bold-dashed-line-sep

Corresponds to one of the seven supported text table separators.

(text-table->string *table*) procedure
(text-table->string *table border*)

Returns the given text table as a string that can be displayed. *border* is a boolean argument specifying whether a border is printed around the table.

56 LispKit Thread

Library `(lispkit thread)` provides programming abstractions facilitating multi-threaded programming. LispKit's thread system offers mechanisms for creating new threads of execution and for synchronizing them. The abstractions provided by this library only offer low-level support for multi-threading and access control. Other libraries such as `(lispkit thread channel)` provide higher-level abstractions built on top of `(lispkit thread)`.

Library `(lispkit thread)` defines the following data types:

- Threads (a virtual processor which shares object space with all other threads)
- Mutexes (a mutual exclusion device, also known as a lock and binary semaphore)
- Condition variables (a set of blocked threads)

Some exception datatypes related to multi-threading are also specified, and a general mechanism for handling such exceptions is provided.

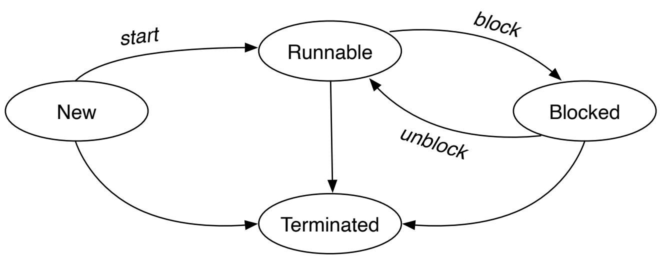
The design of this library as well as most of this documentation originates from SRFI 18 by Marc Feeley.

56.1 Threads

A thread in LispKit encapsulates a *thunk* which it eventually executes, a *name* identifying the thread, a *tag* for storing associated (thread-local) data, a list of mutexes it owns, as well as an end-result and end-exception field for eventually capturing the result of the executed thread. A thread is in exactly one of the following states: new, runnable, blocked, and terminated.

56.1.1 Thread states

A "running" thread is a thread that is currently executing. There can be more than one running thread on a multiprocessor machine. A "runnable" thread is a thread that is ready to execute or running. A thread is "blocked" if it is waiting for a mutex to become unlocked, the end of a "sleep" period, etc. A "new" thread is a thread that has not yet become runnable. A new thread becomes runnable when it is started explicitly. A "terminated" thread is a thread that can no longer become runnable. Deadlocked threads are not considered terminated. The only valid transitions between the thread states are from new to runnable, between runnable and blocked, and from any state to terminated:



The API of library `(lispkit thread)` provides procedures for triggering thread state transitions and for determining the current state of threads.

56.1.2 Primordial thread

The execution of a program is initially under the control of a single thread known as the "primordial thread". The primordial thread has name `main` and a tag referring to a mutable box for storing thread-local data. All threads are terminated when the primordial thread terminates.

Expressions entered in the read-eval-print loop of LispKit are executed on the primordial thread. Whenever execution of an expression is finished, all threads (except for the primordial thread) are terminated automatically.

56.1.3 Memory coherency

Read and write operations on the store, such as reading and writing a variable, an element of a vector or a string, are not necessarily atomic. It is an error for a thread to write a location in the store while some other thread reads or writes that same location. It is the responsibility of the application to avoid write/read and write/write races through appropriate uses of the synchronization primitives. Concurrent reads and writes to ports are allowed, including input and output to the console.

56.1.4 Dynamic environment

The "dynamic environment" is a structure which allows the system to find the value returned by `current-input-port`, `current-output-port`, etc. The procedures `with-input-from-file`, `with-output-to-file`, etc. extend the dynamic environment to produce a new dynamic environment which is in effect for the duration of the call to the thunk passed as the last argument. LispKit provides procedures and special forms to define new "dynamic variables" and bind them in the dynamic environment via `make-parameter` and `parameterize`.

Each thread has its own dynamic environment. When a thread's dynamic environment is extended this does not affect the dynamic environment of other threads. When a thread creates a continuation, the thread's dynamic environment and the dynamic-wind stack are saved within the continuation. When this continuation is invoked, the required dynamic-wind before and after thunks are called and the saved dynamic environment is reinstated as the dynamic environment of the current thread. During the call to each required dynamic-wind before and after thunk, the dynamic environment and the dynamic-wind stack in effect when the corresponding dynamic-wind was executed are reinstated. Note that this specification clearly defines the semantics of calling `call-with-current-continuation` or invoking a continuation within a before or after thunk. The semantics are well defined even when a continuation created by another thread is invoked.

56.1.5 Thread-management API

(current-thread) procedure

Returns the current thread, i.e. the thread executing the current expression.

```
(thread-terminated? (current-thread))  ⇒  #f
```

(thread? *obj*) procedure

Returns `#t` if *obj* is a thread object, otherwise `#f` is returned.

```
(thread? (current-thread))  ⇒  #t
(thread? 12)                ⇒  #f
```

(make-thread *thunk*) procedure
(make-thread *thunk name*)
(make-thread *thunk name tag*)

Creates a new thread for executing *thunk*. Each thread has a thunk to execute as well as a *name* identifying the thread and a *tag* which can be used to associate arbitrary objects with a thread. Both *name* and *tag* can be arbitrary values. The default for *name* and *tag* is `#f`.

New threads are not automatically made runnable; the procedure `thread-start!` must be used for that. Besides *name* and *tag*, a thread encapsulates an end-result, an end-exception, as well as a list of locked/owned mutexes. The thread's execution consists of a call to *thunk* with the "initial continuation". This continuation causes the (then) current thread to store the result in its end-result field, abandon all mutexes it owns, and finally terminate.

The dynamic-wind stack of the initial continuation is empty. The thread inherits the dynamic environment from the current thread. Moreover, in this dynamic environment the exception handler is bound to the "initial exception handler" which is a unary procedure which causes the (then) current thread to store in its end-exception field an "uncaught exception" object whose "reason" is the argument of the handler, abandon all mutexes it owns, and finally terminate.

(thread *stmt* ...) syntax
Creates a new thread for executing the statements *stmt* ... . This statement is equivalent to:

```
(make-thread (thunk stmt ...))
```

(spawn *thunk*) procedure
(spawn *thunk name*)
(spawn *thunk name tag*)

Creates a new thread for executing *thunk* and starts it. Each thread has a thunk to execute as well as a *name* identifying the thread and a *tag* which can be used to associate arbitrary objects with a thread. Both *name* and *tag* can be arbitrary values. This statement is equivalent to:

```
(thread-start! (make-thread thunk name tag))
```

(go *stmt* ...) syntax
Creates a new thread for executing the statements *stmt* ... and starts it. This statement is equivalent to:

```
(thread-start! (make-thread (thunk stmt ...)))
```

(parallel *thunk0 thunk1* ...) procedure
Executes *thunk0* on the current thread, and spawns new threads for executing *thunk1* ... in parallel. `parallel` only terminates when all parallel computations have terminated. It returns *n* results for *n* thunks provided as arguments.

(parallel/timeout *timeout default thunk* ...) procedure
Executes each *thunk* in parallel on a separate thread and terminates only if all parallel threads have terminated or the *timeout* has triggered. *timeout* is a number specifying the maximum time in seconds the computations are allowed to take. `parallel/timeout` returns *n* results for *n* thunks provided as arguments or *default* in case the timeout triggers.

(thread-name *thread*) procedure
Returns the name of the *thread*.

(thread-tag *thread*) procedure
Returns the tag of the *thread*.

(thread-runnable? *thread*) procedure

Returns `#t` if *thread* is in runnable state; otherwise `#f` is returned.

(thread-blocked? *thread*) procedure

Returns `#t` if *thread* is in runnable state; otherwise `#f` is returned.

(thread-terminated? *thread*) procedure

Returns `#t` if *thread* is in terminated state; otherwise `#f` is returned.

(thread-max-stack) procedure
(thread-max-stack *limit*)
(thread-max-stack *thread*)
(thread-max-stack *thread limit*)

Returns the maximum stack size or sets it to a new limit. If no arguments are provided, the maximum stack size of the current thread is returned. If just fixnum *limit* is provided as an argument, the current thread's maximum stack size is set to *limit*. If just *thread* is provided as an argument, the maximum stack size of *thread* is returned. If both *thread* and *limit* are provided, then procedure `thread-max-stack` sets the maximum stack size of *thread* to *limit*.

Changing the stack size while a thread is running is allowed, but it's not always possible to update the limit. The boolean returned by procedure `thread-max-stack` for forms where the maximum stack size is supposed to be updated, indicates whether the update worked. The return value is `#t` in this case.

(thread-start! *thread*) procedure

Makes *thread* runnable. The *thread* must be a new thread. `thread-start!` returns the *thread*. Executing the following code either prints `ba` or `ab`.

```
(let ((t (thread-start! (thread (write 'a)))))
  (write 'b)
  (thread-join! t))
```

(thread-yield!) procedure

The current thread exits the running state as if its quantum had expired. Here is an example how one could use `thread-yield`:

```
; a busy loop that avoids being too wasteful of the CPU
(let loop ()
  ; try to lock m but don't block
  (if (mutex-try-lock! m)
      (begin
        (display "locked mutex m")
        (mutex-unlock! m))
      (begin
        (do-something-else)
        (thread-yield!) ; relinquish rest of quantum
        (loop))))
```

(thread-sleep! *timeout*) procedure

The current thread waits for *timeout* seconds. This blocks the thread only if *timeout* is a positive number.

```
; a clock with a gradual drift:
(let loop ((x 1))
  (thread-sleep! 1)
  (write x)
  (loop (+ x 1)))
; a clock with no drift:
(let ((start (current-second)))
```

```
  (let loop ((x 1))
    (thread-sleep!
      (- (+ start x) (current-second)))
    (write x)
    (loop (+ x 1))))
```

(thread-terminate! ***thread*****)** procedure
(thread-terminate! ***thread wait*****)**

Causes an abnormal termination of the *thread*. If the *thread* is not already terminated, all mutexes owned by the *thread* become unlocked/abandoned and a "terminated thread exception" object is stored in the *thread*'s end-exception field. By default, the termination of the *thread* will occur before `thread-terminate!` returns, unless parameter *wait* is provided and set to `#f`. If *thread* is the current thread, `thread-terminate!` does not return.

This operation must be used carefully because it terminates a thread abruptly and it is impossible for that thread to perform any kind of cleanup. This may be a problem if the thread is in the middle of a critical section where some structure has been put in an inconsistent state. However, another thread attempting to enter this critical section will raise an "abandoned mutex exception" because the mutex is unlocked/abandoned. This helps avoid observing an inconsistent state.

(thread-join! ***thread*****)** procedure
(thread-join! ***thread timeout*****)**
(thread-join! ***thread timeout default*****)**

The current thread waits until the *thread* terminates (normally or not) or until the timeout is reached, if *timeout* is provided. *timeout* is a number in seconds relative to the time `thread-join!` is called. If the timeout is reached, *thread-join!* returns *default* if it is provided, otherwise a "join timeout exception" is raised. If the *thread* terminated normally, the content of the end-result field of *thread* is returned, otherwise the content of the end-exception field is raised. Example:

```
(let ((th (go (+ 1 2 3))))
  (* 10 (thread-join! th)))
⇒ 60
(let ((th (go (error "broken thread"))))
  (* 10 (thread-join! th)))
⇒ raises: [uncaught] [error] broken thread
```

56.2 Mutexes

A mutex is a synchronization abstraction, enforcing mutual exclusive access to a resource when there are many threads of execution. Upon creation, a mutex can be associated with a tag, which is an arbitrary object used in an application-specific way to associate data with the mutex.

56.2.1 Mutex states

A mutex can be in one of four states: *locked* (either *owned* or not *owned*) and *unlocked* (either *abandoned* or *not abandoned*). An attempt to lock a mutex only succeeds if the mutex is in an unlocked state, otherwise the current thread must wait.

A mutex in the *locked/owned* state has an associated "owner" thread, which by convention is the thread that is responsible for unlocking the mutex. This case is typical of critical sections implemented as "lock mutex, perform operation, unlock mutex". A mutex in the *locked/not-owned* state is not linked to a particular thread. A mutex becomes locked when a thread locks it using the `mutex-lock!` primitive. A mutex becomes *unlocked/abandoned* when the owner of a *locked/owned* mutex terminates. A mutex becomes *unlocked/not-abandoned* when a thread unlocks it using the `mutex-unlock!` procedure.

The mutexes provided by library `(lispkit thread)` do not implement "recursive" mutex semantics. An attempt to lock a mutex that is locked already implies that the current thread must wait, even if the mutex is owned by the current thread. This can lead to a deadlock if no other thread unlocks the mutex.

56.2.2 Mutex-management API

(mutex? *obj*) procedure
Returns `#t` if *obj* is a mutex, otherwise returns `#f`.

(make-mutex) procedure
(make-mutex *name*)
(make-mutex *name tag*)

Returns a new mutex in the *unlocked/not-abandoned* state. The optional *name* is an arbitrary object which identifies the mutex (for debugging purposes), defaulting to `#f`. It is also possible to provide a tag, which is an arbitrary object used in an application-specific way to associate data with the mutex. `#f` is used as a default if the tag is not provided.

(mutex-name *mutex*) procedure
Returns the name of the *mutex*.

```
(mutex-name (make-mutex 'foo))  ⇒  foo
```

(mutex-tag *mutex*) procedure
Returns the tag of the *mutex*.

```
(mutex-tag (make-mutex 'id '(1 2 3)))  ⇒  (1 2 3)
```

(mutex-state *mutex*) procedure
Returns the state of the *mutex*. The possible results are:

- *T*: the *mutex* is in the *locked/owned* state and thread *T* is the owner of the *mutex*
- `not-owned`: the *mutex* is in the locked/not-owned state
- `abandoned`: the *mutex* is in the unlocked/abandoned state
- `not-abandoned`: the *mutex* is in the unlocked/not-abandoned state

```
(mutex-state (make-mutex))
⇒  not-abandoned
(let ((mutex (make-mutex)))
  (mutex-lock! mutex #f (current-thread))
  (let ((state (mutex-state mutex)))
    (mutex-unlock! mutex)
    (list state (mutex-state mutex))))
⇒  (#<thread main: runnable> not-abandoned)
```

(mutex-lock! *mutex*) procedure
(mutex-lock! *mutex timeout*)
(mutex-lock! *mutex timeout thread*)

Locks *mutex*. If *mutex* is already locked, the current thread waits until the *mutex* is unlocked, or until the timeout is reached if *timeout* is supplied. If the timeout is reached, `mutex-lock!` returns `#f`. Otherwise, the state of the *mutex* is changed as follows:

- if *thread* is `#f`, the *mutex* becomes *locked/not-owned*,
- otherwise, let *T* be *thread* (or the current thread if *thread* is not supplied): if *T* is terminated the *mutex* becomes *unlocked/abandoned*, otherwise *mutex* becomes *locked/owned* with *T* as the owner.

After changing the state of the *mutex*, an "abandoned mutex exception" is raised if the *mutex* was *unlocked/abandoned* before the state change, otherwise `mutex-lock!` returns `#t`. It is not an error if the *mutex* is owned by the current thread, but the current thread will have to wait.

(mutex-try-lock! *mutex*) procedure
(mutex-try-lock! *mutex thread*)

Locks *mutex* with owner *thread* and returns `#t` if *mutex* is not already locked. Otherwise, `#f` is returned. This is equivalent to: `(mutex-lock! mutex 0 thread)`

(mutex-unlock! *mutex*) procedure
(mutex-unlock! *mutex cvar*)
(mutex-unlock! *mutex cvar timeout*)

Unlocks the *mutex* by making it *unlocked/not-abandoned*. It is not an error to unlock an unlocked mutex and a mutex that is owned by any thread. If *cvar* is supplied, the current thread is blocked and added to the condition variable *cvar* before unlocking *mutex*. The thread can unblock at any time but no later than when an appropriate call to `condition-variable-signal!` or `condition-variable-broadcast!` is performed, and no later than the timeout (if *timeout* is supplied). If there are threads waiting to lock this *mutex*, the scheduler selects a thread, the mutex becomes *locked/owned* or *locked/not-owned*, and the thread is unblocked. `mutex-unlock!` returns `#f` when the timeout is reached, otherwise it returns `#t` .

`mutex-unlock!` is related to the "wait" operation on condition variables available in other thread systems. The main difference is that "wait" automatically locks *mutex* just after the thread is unblocked. This operation is not performed by `mutex-unlock!` and so must be done by an explicit call to `mutex-lock!` . This has the advantages that a different timeout and exception handler can be specified on the mutex-lock! and mutex-unlock! and the location of all the mutex operations is clearly apparent. A typical use with a condition variable is this:

```
(let loop ()
  (mutex-lock! m)
  (if (condition-is-true?)
      (begin
        (do-something-when-condition-is-true)
        (mutex-unlock! m))
      (begin
        (mutex-unlock! m cv)
        (loop))))
```

(with-mutex *mutex stmt0 stmt1* ...) syntax

`with-mutex` locks *mutex* and then executes statements *stmt0*, *stmt1*, ... After all statements are executed, *mutex* is being unlocked. `with-mutex` returns the result of evaluating the last statement. For locking and unlocking t *mutex*, `dynamic-wind` is used so that *mutex* is automatically unlocked if an error or new continuation exits the statements, and it is re-locked, if the statements are re-entered by a captured continuation.

`(with-mutex m stmt0 stmt1 ...)` is equivalent to:

```
(dynamic-wind
  (lambda () (mutex-lock! m))
  (lambda () (begin stmt0 stmt1 ...))
  (lambda () (mutex-unlock! m)))
```

56.3 Condition variables

56.3.1 Semantics

A condition variable represents a set of blocked threads. These blocked threads are waiting for a certain condition to become true. When a thread modifies some program state that might make the condition true, the thread unblocks some number of threads 9one or all depending on the primitive used) so they can check the value of that condition. This allows complex forms of inter-thread synchronization to be expressed more conveniently than with mutexes alone.

Each condition variable has a tag which can be used in an application specific way to associate data with the condition variable.

56.3.2 Condition variable management

(condition-variable? *obj*) procedure
Returns `#t` if *obj* is a condition variable, otherwise returns `#f`.

(make-condition-variable) procedure
(make-condition-variable *name*)
(make-condition-variable *name tag*)

Returns a new empty condition variable. The optional *name* is an arbitrary object which identifies the condition variable for debugging purposes. It defaults to `#f`. It is also possible to provide a tag, which is an arbitrary object used in an application-specific way to associate data with the condition variable. `#f` is used as a default if the tag is not provided.

(condition-variable-name *cvar*) procedure
Returns the name of the condition variable *cvar*.

(condition-variable-tag *cvar*) procedure
Returns the tag of the condition variable *cvar*.

(condition-variable-wait! *cvar mutex*) procedure
(condition-variable-wait! *cvar mutex timeout*)

`condition-variable-wait!` can be used to make the current thread wait on condition variable *cvar*. It is assumed the current thread has locked *mutex* when `condition-variable-wait!` is called. `condition-variable-wait!` will unlock *mutex* and wait on *cvar*, either until *cvar* unblocks the thread again or *timeout* (in seconds) triggers. When the current thread is woken up again, it reclaims the lock on *mutex*.

`condition-variable-wait!` returns `#f` if the timeout triggers, otherwise `#t` is being returned.

(condition-variable-signal! *cvar*) procedure
If there are threads blocked on the condition variable *cvar*, the scheduler selects a thread and unblocks it.

(condition-variable-broadcast! *cvar*) procedure
Unblocks all the threads blocked on the condition variable *cvar*.

56.4 Exception handling

(join-timeout-exception? *obj*) procedure
Returns `#t` if *obj* is a “join timeout exception” object, otherwise returns `#f`. A join timeout exception is

raised when `thread-join!` is called, the timeout is reached and no *default* is supplied.

(abandoned-mutex-exception? *obj*) procedure
Returns `#t` if *obj* is an "abandoned mutex exception" object, otherwise returns `#f`. An abandoned mutex exception is raised when the current thread locks a mutex that was owned by a thread which terminated.

(terminated-thread-exception? *obj*) procedure
Returns `#t` if *obj* is a "terminated thread exception" object, otherwise returns `#f`. A terminated thread exception is raised when `thread-join!` is called and the target thread has terminated as a result of a call to `thread-terminate!`.

(uncaught-exception? *obj*) procedure
Returns `#t` if *obj* is an "uncaught exception" object, otherwise returns `#f`. An uncaught exception is raised when `thread-join!` is called and the target thread has terminated because it raised an exception that called the initial exception handler of that thread.

(uncaught-exception-reason *exc*) procedure
exc must be an "uncaught exception" object. `uncaught-exception-reason` returns the object which was passed to the initial exception handler of that thread.

56.5 Hardware platform and debugging

(processor-count) procedure
(processor-count *active?*)

Returns the number of processors provided by the underlying hardware. If *active?* is set to `#f` (the default), the number of physical processors is returned. If *active?* is set to `#t`, the number of active processors (i.e. available for executing code) is returned.

(runnable-thread-count) procedure
Returns the number of threads that are currently in runnable state.

(allocated-thread-count) procedure
Returns the number of allocated threads, i.e. threads in any state that are not garbage collected yet.

---

57 LispKit Thread Channel

Library `(lispkit thread channel)` implements *channels* for communicating, coordinating and synchronizing threads of execution. LispKit channels are based on the channel abstraction provided by the Go programming language.

LispKit channels are thread-safe FIFO buffers for synchronizing communication between multiple threads. The current implementation supports multiple simultaneous receives and sends. It allows channels to be either synchronous or asynchronous by providing buffering capabilities. Furthermore, the library supports timeouts via channel timers and channel tickers.

The main differences compared to channels in the Go programming language are:

- Channels do not have any type information.
- Sending to a channel that gets closed does not panic, it unblocks all senders immediately with the `fail` flag set to non- `#f` .
- Closing an already closed channel does not result in an error.
- There is support for choosing what channels to select on at runtime via `channel-select*` .

57.1 Channels

(channel? ***obj*****)** procedure

Returns `#t` if *obj* is a channel, otherwise `#f` is returned.

(make-channel) procedure
(make-channel ***capacity*****)**

Returns a new channel with a buffer size of *capacity*. If *capacity* is 0, the channel is synchronous and all its operations will block until a remote client sends/receives messages. Channels with a buffer capacity > 0 are asynchronous, but block if the buffer is exhausted.

(channel-send! ***channel msg*****)** procedure

Sends message *msg* to *channel*. `channel-send!` blocks if the capacity of *channel* is exhausted. `channel-send!` returns the fail flag of the send operation, i.e. `#f` is returned if the send operation succeeded.

(channel-receive! ***channel*****)** procedure
(channel-receive! ***channel none*****)**

Receives a message from *channel* and returns the message. If there is no message available, `channel-receive!` blocks. If the receive operation fails, *none* is returned, if provided. The default for *none* is `#f` .

(channel-try-receive! ***channel*****)** procedure
(channel-try-receive! ***channel none*****)**

Receives a message from *channel* and returns the message. If there is no message available, `channel-try-receive!` returns *none*, if provided. The default for *none* is `#f` .

(channel-select* *channel clauses*) procedure

Procedure `channel-select*` allows selecting channels that are chosen programmatically. It takes input that looks like this:

```
(channel-select*
  `((,chan1 meta1)          ; receive
    (,chan2 meta2 message) ; send
    (,chan3 meta3) ...))
```

`channel-select*` returns three values *msg*, *fail*, and *meta*, where *msg* is the message that was sent over the channel, *fail* is `#t` if the channel was closed and `#f` otherwise, and *meta* is the datum supplied in the arguments.

For example, if a message arrived on *chan3* above, *meta* would be `meta3` in that case. This allows one to see which channel a message came from, i.e. if you supply metadata that is the channel itself.

(channel-select
((chan -> msg) body ...)
((chan -> msg fail) body ...)
((chan <- msg) body ...)
((chan <- msg fail) body ...)
(else body ...)) syntax

This is a channel switch that will send or receive on at most one channel, picking whichever clause is able to complete soonest. If no clause is ready, `channel-select` will block until one does, unless `else` is specified which will execute its body instead of blocking. Multiple send and receive clauses can be specified interchangeably, but only one clause will trigger and get executed. Example:

```
(channel-select
  ((chan1 -> msg fail)
     (if fail
         (print "chan1 closed!")
         (print "chan1 says " msg)))
  ((chan2 -> msg fail)
     (if fail
         (print "chan2 closed!")
         (print "chan2 says " msg))))
```

Receive clauses have the form `((chan -> msg [fail]) body ...)`. They execute *body* with *msg* bound to the message object and *fail* bound to a boolean flag indicating failure. Receiving from a closed channel immediately completes with this *fail* flag set to non- `#f`.

Send clauses have the form `((chan <- msg [fail]) body ...)`. They execute *body* after *msg* has been sent to a receiver, successfully buffered onto the channel, or if channel was closed. Sending to a closed channel immediately completes with the *fail* flag set to `#f`.

A send or receive clause on a closed channel with no *fail*-flag binding specified will immediately return void without executing *body*. This can be combined with recursion like this:

```
;; loop forever until either chan1 or chan2 closes
(let loop ()
   (channel-select
     ((chan1 -> msg)
        (display* "chan1 says " msg) (loop))
     ((chan2 <- 123)
        (display* "chan2 got  " 123) (loop))))
```

Or like this:

```
;; loop forever until chan1 closes. replacing chan2 is
;; important to avoid busy-wait!
(let loop ((chan2 chan2))
  (channel-select
    ((chan1 -> msg)
       (display* "chan1 says " msg)
       (loop chan2))
    ((chan2 -> msg fail)
       (if fail
           (begin
             (display* "chan2 closed, keep going")
             ;; create new forever-blocking channel
             (loop (make-channel 0)))
           (begin
             (display* "chan2 says " msg)
             (loop chan2))))))
```

`channel-select` returns the return value of the executed clause's body. To do a non-blocking receive, you can do the following:

```
(channel-select
  ((chan1 -> msg fail) (if fail #!eof msg))
  (else 'eagain))
```

(channel-range *channel* -> *msg body* …) syntax

`channel-range` continuously waits for messages to arrive on *channel*. Once a message *msg* is available, *body* … gets executed and `channel-range` waits again for the next message to arrive. `channel-range` does not terminate unless *channel* is closed. The following statement is equivalent:

```
(let ((chan channel))
  (let loop ()
    (channel-select
      ((chan -> msg fail)
        (unless fail (begin body ...)(loop))))))
```

(channel-close *channel*) procedure
(channel-close *channel fail*)

Closes *channel*. This will unblock existing receivers and senders waiting for an operation on *channel* with thir fail flag set to a non- `\#f` value. All future receivers and senders will also immdiately unblock in this way, so there is a risk to run into busy-loops.

The optional *fail* flag of `channel-close` can be used to specify an alternative to the default `#t`. As this value is given to all receivers and senders of *channel*, the *fail* flag can be used as a "broadcast" mechanism. *fail* flag must not be set to `#f` though, as that would indicate a successful message transaction.

Closing an already closed channel will results in its fail flag being updated.

57.2 Timers

(timer? *obj*) procedure

Returns `#t` if *obj* is a channel timer as provided by this library. Otherwise `timer?` returns `#f`.

(make-timer *next*) procedure

next is a thunk returning three values: *when-next*, *data*, and *fail*. *when-next* is when to trigger the next time, expressed in seconds since January 1, 1970 TAI (e.g. computed via `(current-second)`), *data* is the payload returned when the triggers (it's usually the time in seconds when it triggers), and *fail* refers to a fail flag, which is usually `#f` for timers.

`next` will be called exaclty once on every timeout and once at "startup" and can thus mutate its own private state. `next` is called within a timer mutex lock and thus does not need to be synchronized.

(timer *duration*) procedure

Returns a timer channel that will "send" a single message after *duration* seconds after its creation. The message is the `current-second` value at the time of the timeout, i.e. not when the message was received. Receiving more than once on an timer channel will block indefinitely or deadlock the second time.

```
(channel-select
  ((chan1 -> msg)
     (display* "chan1 says " msg))
  (((timer 1) -> when)
     (display* "chan1 took too long")))
```

You cannot send to or close a timer channel. Creating timers is a relatively cheap operation. Timers may be garbage-collected before the timer triggers. Creating a timer does not spawn a new thread.

(ticker *duration*) procedure

Returns a ticker channel that will "send" a message every *duration* seconds. The message is the `current-second` value at the time of the tick, i.e. not when it was received.

(ticker-stop! *ticker*) procedure

Stops a ticker channel, i.e. the channel will stop sending "tick" messages.

---

58 LispKit Type

Library `(lispkit type)` provides a simple, lightweight type abstraction mechanism. It allows for creating new types at runtime that are disjoint from all existing types. The library provides two different types of APIs: a purely procedural API for type creation and management, as well as a declarative API which allows for introducing extensible types in a declarative fashion.

A core feature of the type abstraction mechanism is a means that allows for determining the type of any LispKit value. Procedure `type-of` implements this type introspection feature. It returns a *type tag*, i.e. a symbol representing the type, for any given object.

58.1 Usage of the procedural API

New types are created with procedure `make-type`. `make-type` accepts one argument, which is a type label. The type label is either a string or a symbol that is used for debugging purposes but also for defining the string representation of a *type tag* that represents the new type.

The following line introduces a new type for *intervals*:

```
(define-values (interval-type-tag
                new-interval
                interval?
                interval-ref
                make-interval-subtype)
  (make-type 'interval))
```

`(make-type 'interval)` returns five values:

- `interval-type-tag` is an uninterned symbol with the string representation `"interval"` which represents the new type.
- `new-interval` is a procedure which takes one argument, the internal representation of the interval, and returns a new object of the new interval type.
- `interval?` is a type test predicate which accepts one argument and returns `#t` if the argument is of the new interval type, and `#f` otherwise.
- `interval-ref` takes one object of the new interval type and returns its internal representation. `interval-ref` is the inverse operation of `new-interval`.
- `make-interval-subtype` is a type generator (similar to `make-type`), a function that takes a type label and returns five values representing a new subtype of the interval type.

Now it is possible to implement a constructor `make-interval` for intervals:

```
(define (make-interval lo hi)
  (if (and (real? lo) (real? hi) (<= lo hi))
      (new-interval (cons (inexact lo) (inexact hi)))
      (error "make-interval: illegal arguments" lo hi)))
```

`make-interval` first checks that the constructor arguments are valid and then calls `new-interval` to create a new interval object. Interval objects are represented via pairs whose *car* is the lower bound, and *cdr* is the upper bound. Nevertheless, pairs and interval objects are distinct values as the following code shows:

```
(define interval-obj (make-interval 1.0 9.5))
(define pair-obj (cons 1.0 9.5))

(interval? interval-obj)       ⇒ #t
(interval? pair-obj)           ⇒ #f
(equal? interval-obj pair-obj)  ⇒ #f
```

The type is displayed along with the representation in the textual representation of interval objects: `#interval:(1.0 . 9.5)`.

Below are a few functions for interval objects. They all use `interval-ref` to extract the internal representation from an interval object and then operate on the internal representation.

```
(define (interval-length interval)
  (let ((bounds (interval-ref interval)))
    (- (cdr bounds) (car bounds))))

(define (interval-empty? interval)
  (zero? (interval-length interval)))
```

The following function calls show that `interval-ref` fails with a type error if its argument is not an interval object.

```
(interval-length interval-obj)
⇒ 8.5
(interval-empty? '(1.0 . 1.0))
⇒ [error] not an instance of type interval: (1.0 . 1.0)
```

58.2 Usage of the declarative API

The procedural API provides the most flexible way to define a new type in LispKit. On the other hand, this approach comes with two problems:

1. a lot of boilerplate needs to be written, and
2. programmers need to be experienced to correctly encapsulate new data types and to provide means to extend them.

These problems are addressed by the declarative API of `(lispkit type)`. At the core, this API defines a syntax `define-type` for declaring new types of data. `define-type` supports defining simple, encapsulated types as well as provides a means to make types extensible.

The syntax for defining a simple, non-extensible type has the following form:

(`define-type` *name name?*
  ((*make-name x* …) *expr* …)
  name-ref
  functions)

name is a symbol. It defines the name of the new type and the identifier *name* is bound to a type tag representing the new type. *name?* is a predicate for testing whether a given object is of type *name*. *make-name* defines a constructor which returns a value representing the data of the new type. *name-ref* is a function to unwrap values of type *name*. It is optional and normally not needed since *functions* can be declared such that the unwrapping happens implicitly. All functions defined via `define-type` take an object (usually called `self`) of the defined type as their first argument.

There are two forms to declare a function as part of `define-type`: one providing access to `self` directly, and one only providing access to the unwrapped data value:

`((` *name-func self y* ... `)` *expr* ... `)`

provides access directly to *self* (which is a value of type *name*), and

`((` *name-func* `(` *repr* `)` *y* ... `)` *expr* ... `)`

which provides access only to the unwrapped data `repr`.

With this new syntax, type `interval` from the section describing the procedural API, can now be re-written like this:

```
(define-type interval
  interval?
  ((make-interval lo hi)
    (if (and (real? lo) (real? hi) (<= lo hi))
        (cons (inexact lo) (inexact hi))
        (error "make-interval: illegal arguments" lo hi)))
  ((interval-length (bounds))
    (- (cdr bounds) (car bounds)))
  ((interval-empty? self)
    (zero? (interval-length self))))
```

`interval` is a standalone type which cannot be extended. `define-type` provides a simple means to make types extensible such that subtypes can be created reusing the base type definition. This is done with a small variation of the `define-type` syntax:

`(define-type` (*name super*) *name?*
  `((` *make-name x* ... `)` *expr* ... `)`
  name-ref
  functions `)`

In this syntax, *super* refers to the type extended by *name*. All extensible types extend another extensible type and there is one supertype called `obj` provided by library `(lispkit type)` as a primitive.

With this syntactic facility, `interval` can be easily re-defined to be extensible:

```
(define-type (interval obj)
  interval?
  ((make-interval lo hi)
    (if (and (real? lo) (real? hi) (<= lo hi))
        (cons (inexact lo) (inexact hi))
        (error "make-interval: illegal arguments" lo hi)))
  ((interval-length (bounds))
    (- (cdr bounds) (car bounds)))
  ((interval-empty? self)
    (zero? (interval-length self))))
```

It is now possible to define a `tagged-interval` data structure which inherits all functions from `interval` and encapsulates a tag with the interval:

```
(define-type (tagged-interval interval)
  tagged-interval?
  ((make-tagged-interval lo hi tag)
    (values lo hi tag))
  ((interval-tag (bounds tag))
    tag))
```

`tagged-interval` is a subtype of `interval`; i.e. values of type `tagged-interval` are also considered to be of type `interval`. Thus, `tagged-interval` inherits all function definitions from `interval` and defines a new function `interval-tag` just for `tagged-interval` values. Here is some code explaining the usage of `tagged-interval`:

```
(define ti (make-tagged-interval 4.0 9.0 'inclusive))
(tagged-interval? ti)       ⇒ #t
(interval? ti)              ⇒ #t
(interval-length ti)        ⇒ 5.0
(interval-tag ti)           ⇒ inclusive
(interval-tag interval-obj)
⇒ [error] not an instance of type tagged-interval: #interval:((1.0 . 9.5))
```

Constructors of extended types, such as `make-tagged-interval` return multiple values: all the parameters for a super-constructor call and one additional value (the last value) representing the data provided by the extended type. In the example above, `make-tagged-interval` returns three values: `lo`, `hi`, and `tag`. After the constructor `make-tagged-interval` is called, the super-constructor is invoked with arguments `lo` and `hi`. The result of `make-tagged-interval` is a `tagged-interval` object consisting of two state values contained in a list: one for the supertype `interval` (consisting of the bounds `(lo . hi)`) and one for the subtype `tagged-interval` (consisting of the tag). This can also be seen when displaying a `tagged-interval` value:

```
ti ⇒ #<tagged-interval (4.0 . 9.0) inclusive>
```

This is also the reason why function `interval-tag` gets access to two unwrapped values, `bounds` and `tag`: one (`bounds`) corresponds to the value associated with type `interval`, and the other one (`tag`) corresponds to the value associated with type `tagged-interval`.

58.3 Type introspection

LispKit defines a type tag for every different type of object. The type of objects created with the `make-type` facility are represented with unintered symbols whose string representation matches the type label provided to `make-type`. Enum and record objects come with a corresponding type tag as well. All standard, built-in types use interned symbols as type tags. Procedure `type-of` returns the type tag associated with the value of the argument of `type-of`.

(type-of *object*) procedure

Returns a list of type tags, i.e. symbols, associated with the type of *object*. The type tags in the list are sorted, starting with the most specific type. If no type can be determined, `type-of` returns the empty list.

Type tags of custom types are returned as the first value of `make-type`. Type tags of records can be retrieved via `record-type-tag` from the corresponding record type object. Similarly, type tags of enum objects are accessible via `enum-type-type-tag`. Type tags of native types are typically exported by the libraries providing access to the type. All standard, built-in types are represented by the following interned symbols: `void`, `end-of-file`, `null`, `boolean`, `symbol`, `fixnum`, `bignum`, `integer`, `rational`, `flonum`, `real`, `complex`, `number`, `char`, `string`, `bytevector`, `pair`, `list`, `box`, `mpair`, `array`, `vector`, `gvector`, `values`, `procedure`, `parameter`, `promise`, `environment`, `hashtable`, `port`, `input-port`, `output-port`, `record-type`, and `error`. For undefined values `#f` is returned.

58.4 Type management

(make-type *type-label*) procedure

Based on a string or symbol *type-label*, creates a new, unique type, and returns a type tag representing the new type as well as five values dealing with this new type:

1. The first value is a type tag, i.e. an uninterned symbol representing the new type which has the same string representation as *type-label*.
2. The second value is a unary procedure returning a new object of the new type which is wrapping the argument of the procedure (i.e. the internal representation of the new type).
3. The third value is a type test predicate which accepts one argument and returns `#t` if the argument is of the new type, and `#f` otherwise.
4. The fourth value is a procedure which takes one object of the new type and returns its internal representation (that was passed to the procedure returned as the second value).
5. The fifth value is a type generator procedure (similar to `make-type`), a function that takes a type label and returns five values representing a new subtype of the new type.

type-label has two uses: it is used for debugging purposes and determines the string representation of the corresponding type tag. It is shown when an object's textual representation is used. In particular, calling the third procedure (the type de-referencing function) will result in an error message exposing the type label if the argument is of a different type than expected.

(define-type *name name?* ((*make-name x* ...) *e* ...) *func* ...) syntax
(define-type *name name?* ((*make-name x* ...) *e* ...) *ref func* ...)

Defines a new standalone type *name* consisting of a type test predicate *name?*, a constructor *make-name*, and an optional function *ref* used to unwrap values of type *name*. *ref* is optional and normally not needed since functions *func* can be declared such that the unwrapping happens implicitly. All functions *func* defined via `define-type` take an object (usually called `self`) of the defined type as their first argument.

There are two ways to declare a function as part of `define-type`: one providing access to `self` directly, and one only providing access to the unwrapped data value:

- `((` *name-func self y* ... `)` *expr* ... `)` provides access directly to *self* (which is a value of type *name*), and
- `((` *name-func* `(` *repr* `)` *y* ... `)` *expr* ... `)` provides access only to the unwrapped data `repr`.

(define-type (*name super*) *name?* ((*make-name x* ...) *e* ...) *func* ...) syntax
(define-type (*name super*) *name?* ((*make-name x* ...) *e* ...) *ref func* ...)

This variant of `define-type` defines a new extensible type *name* extending supertype *super*, which also needs to be an extensible type. A new extensible type *name* comes with a type test predicate *name?*, a constructor *make-name*, and an optional function *ref* used to unwrap values of type *name*. *ref* is optional and normally not needed since functions *func* can be declared such that the unwrapping happens implicitly. All functions *func* defined via `define-type` take an object (usually called `self`) of the defined type as their first argument.

There are two ways to declare a function as part of `define-type`: one providing access to `self` directly, and one providing access to the unwrapped data values (one for each type in the supertype chain):

- `((` *name-func self y* ... `)` *expr* ... `)` provides access directly to *self* (which is a value of type *name*), and
- `((` *name-func* `(` *repr* ... `)` *y* ... `)` *expr* ... `)` provides access only to the unwrapped data values `repr`.

Constructors of extended types return multiple values: all the parameters for a super-constructor call and one additional value (the last value) representing the data provided by the extended type.

obj object

The supertype of all extensible types defined via `define-type`. The type tag of `obj` can be retrieved via `(type-of obj)`.

(obj-type-tag *etype*) object

Returns the type tag associated with the supertype of all extensible types `obj`.

(extensible-type? *obj*) procedure

Returns `#t` if *obj* is an instance of an extensible type. For example, `(extensible-type? obj)` returns `#t`.

(extensible-type-tag *etype*) procedure

Returns the type tag associated with the extensible type `etype` defined by the `define-type` form.

59 LispKit Vector

Vectors are heterogeneous data structures whose elements are indexed by a range of integers. A vector typically occupies less space than a list of the same length, and a randomly chosen element can be accessed in constant time vs. linear time for lists.

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

Two vectors are `equal?` if they have the same length, and if the values in corresponding slots of the vectors are `equal?` .

A vector can be *mutable* or *immutable*. Trying to change the state of an *immutable vector*, e.g. via `vector-set!` will result in an error being raised.

Vectors are written using the notation `#(obj ...)` . For example, a vector of length 3 containing the number zero in element 0, the list (1 2 3 4) in element 1, and the string "Lisp" in element 2 can be written as follows: `#(0 (1 2 3 4) "Lisp")` .

Vector constants are self-evaluating, so they do not need to be quoted in programs. Vector constants, i.e. vectors created with a vector literal, are *immutable*.

LispKit also supports *growable vectors* via library `(lispkit gvector)` . As opposed to regular vectors, a growable vector does not have a fixed size and supports adding and removing elements. While a growable vector does not satisfay the `vector?` predicate, this library also accepts growable vectors as parameters whenever a vector is expected. Use predicate `mutable-vector?` for determining whether a vector is either a regular mutable vector or a growable vector.

59.1 Predicates

(vector? *obj*) procedure
Returns `#t` if *obj* is a regular vector; otherwise returns `#f` . This function returns `#f` for growable vectors; see library `(lispkit gvector)` .

(mutable-vector? *obj*) procedure
Returns `#t` if *obj* is either a mutable regular vector or a growable vector (see library `(lispkit gvector)`); otherwise returns `#f` .

(immutable-vector? *obj*) procedure
Returns `#t` if *obj* is an immutable vector; otherwise returns `#f` .

(vector= *eql vector* ...) procedure
Procedure `vector=` is a generic comparator for vectors. Vectors *a* and *b* are considered equal by `vector=` if their lengths are the same, and for each respective elements *ai* and *bi*, `(eql ai bi)` evaluates to true. *eql* is always applied to two arguments.

If there are only zero or one vector argument, `#t` is automatically returned. The dynamic order in which comparisons of elements and of vectors are performed is unspecified.

```
(vector= eq? #(a b c d) #(a b c d))  ⇒  #t
(vector= eq? #(a b c d) #(a b d c))  ⇒  #f
(vector= = #(1 2 3 4 5) #(1 2 3 4))  ⇒  #f
(vector= = #(1 2 3 4) #(1.0 2.0 3.0 4.0))  ⇒  #t
(vector= eq?) ⇒  #t
(vector= eq? '#(a))  ⇒  #t
```

59.2 Constructors

(make-vector *k*) procedure
(make-vector *k fill*)

Returns a newly allocated vector of *k* elements. If a second argument is given, then each element is initialized to *fill*. Otherwise the initial contents of each element is unspecified.

(vector *obj* ...) procedure
Returns a newly allocated mutable vector whose elements contain the given arguments. It is analogous to `list`.

```
(vector 'a 'b 'c)  ⇒  #(a b c)
```

(immutable-vector *obj* ...) procedure
Returns a newly allocated immutable vector whose elements contain the given arguments in the given order.

(list->vector *list*) procedure
The `list->vector` procedure returns a newly created mutable vector initialized to the elements of the list *list* in the order of the list.

```
(list->vector '(a b c))  ⇒  #(a b c)
```

(list->immutable-vector *list*) procedure
The `list->vector` procedure returns a newly created immutable vector initialized to the elements of the list *list* in the order of the list.

(string->vector *str*) procedure
(string->vector *str start*)
(string->vector *str start end*)

The `string->vector` procedure returns a newly created mutable vector initialized to the elements of the string *str* between *start* and *end* (i.e. including all characters from index *start* to index *end*-1).

```
(string->vector "ABC")  ⇒  #(#\A #\B #\C)
```

(vector-copy *vector*) procedure
(vector-copy *vector mutable*)
(vector-copy *vector start*)
(vector-copy *vector start end*)
(vector-copy *vector start end mutable*)

Returns a newly allocated copy of the elements of the given vector between *start* and *end*, but excluding the element at index *end*. The elements of the new vector are the same (in the sense of `eqv?`) as the elements of the old.

mutable is a boolean argument. If it is set to `#f`, an immutable copy of *vector* will be created. The type of the second argument of `vector-copy` is used to disambiguate between the second and third version of the function. An exact integer will always be interpreted as *start*, a boolean value will always be interpreted as *mutable*.

```
(define a #(1 8 2 8))          ; a may be immutable
(define b (vector-copy a))     ; creates a mutable copy of a
(vector-set! b 0 3)            ; b is mutable
b  ⇒  #(3 8 2 8)
(define c (vector-copy a #f)) ; creates an immutable copy of a
(vector-set! c 0 3)  ⇒  error  ; error, since c is immutable
(define d (vector-copy b 1 3))
d  ⇒  #(8 2)
```

(vector-append *vector ...*) procedure

Returns a newly allocated mutable vector whose elements are the concatenation of the elements of the given vectors.

```
(vector-append #(a b c) #(d e f))  ⇒  #(a b c d e f)
```

(vector-concatenate *vector xs*) procedure

Returns a newly allocated mutable vector whose elements are the concatenation of the elements of the vectors in *xs*. *xs* is a proper list of vectors.

```
(vector-concatenate '(#(a b c) #(d) #(e f)))  ⇒  #(a b c d e f)
```

(vector-map *f vector1 vector2 ...*) procedure

Constructs a new mutable vector of the shortest size of the vector arguments *vector1*, *vector2*, etc. Each element at index *i* of the new vector is mapped from the old vectors by `(f (vector-ref vector1 i) (vector-ref vector2 i) ...)`. The dynamic order of the application of f is unspecified.

```
(vector-map + #(1 2 3 4 5) #(10 20 30 40))  ⇒  #(11 22 33 44)
```

(vector-map/index *f vector1 vector2 ...*) procedure

Constructs a new mutable vector of the shortest size of the vector arguments *vector1*, *vector2*, etc. Each element at index *i* of the new vector is mapped from the old vectors by `(f i (vector-ref vector1 i) (vector-ref vector2 i) ...)`. The dynamic order of the application of f is unspecified.

```
(vector-map/index (lambda (i x y) (cons i (+ x y))) #(1 2 3) #(10 20 30))
⇒  #((0 . 11) (1 . 22) (2 . 33))
```

(vector-sort *pred vector*) procedure
(vector-sort *pred vector start*)
(vector-sort *pred vector start end*)

Procedure `vector-sort` returns a new vector containing the elements of *vector* in sorted order using *pred* as the “less than” predicate. If *start* and *end* are given, they indicate the sub-vector that should be sorted.

```
(vector-sort < (vector 7 4 9 1 2 8 5))
⇒  #(1 2 4 5 7 8 9)
```

59.3 Iterating over vectors

(vector-for-each *f vector1 vector2 …*) procedure

`vector-for-each` implements a simple vector iterator: it applies *f* to the corresponding list of parallel elements from *vector1 vector2* … in the range *[0, length)*, where *length* is the length of the smallest vector argument passed. In contrast with `vector-map`, *f* is reliably applied to each subsequent element, starting at index 0, in the vectors.

```
(vector-for-each (lambda (x) (display x) (newline))
                 #("foo" "bar" "baz" "quux" "zot"))
⇒
foo
bar
baz
quux
zot
```

(vector-for-each/index *f vector1 vector2 …*) procedure

`vector-for-each/index` implements a simple vector iterator: it applies *f* to the index *i* and the corresponding list of parallel elements from *vector1 vector2* … in the range *[0, length)*, where *length* is the length of the smallest vector argument passed. The only difference to `vector-for-each` is that `vector-for-each/index` always passes the current index as the first argument of *f* in addition to the elements from the vectors *vector1 vector2* ….

```
(vector-for-each/index
  (lambda (i x) (display i)(display ": ")(display x)(newline))
  #("foo" "bar" "baz" "quux" "zot"))
⇒
0: foo
1: bar
2: baz
3: quux
4: zot
```

59.4 Managing vector state

(vector-length *vector*) procedure

Returns the number of elements in *vector* as an exact integer.

(vector-ref *vector k*) procedure

The `vector-ref` procedure returns the contents of element *k* of *vector*. It is an error if *k* is not a valid index of *vector*.

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

(vector-set! *vector k obj*) procedure

The `vector-set!` procedure stores *obj* in element *k* of *vector*. It is an error if *k* is not a valid index of *vector*.

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

```
  ⇒  #(0 ("Sue" "Sue") "Anna")
(vector-set! '#(0 1 2) 1 "doe")
  ⇒  error    ;; constant/immutable vector
```

(vector-swap! ***vector j k*****)** procedure

The `vector-swap!` procedure swaps the element *j* of *vector* with the element *k* of *vector*.

59.5 Destructive vector operations

Procedures which operate only on a part of a vector specify the applicable range in terms of an index interval [*start*; *end*[; i.e. the *end* index is always exclusive.

(vector-copy! ***to at from*****)** procedure
(vector-copy! ***to at from start*****)**
(vector-copy! ***to at from start end*****)**

Copies the elements of vector *from* between *start* and *end* to vector *to*, starting at *at*. The order in which elements are copied is unspecified, except that if the source and destination overlap, copying takes place as if the source is first copied into a temporary vector and then into the destination. *start* defaults to 0 and *end* defaults to the length of *vector*.

It is an error if *at* is less than zero or greater than the length of *to*. It is also an error if `(- (vector-length to) at)` is less than `(- end start)`.

```
(define a (vector 1 2 3 4 5))
(define b (vector 10 20 30 40 50)) (vector-copy! b 1 a 0 2)
b =⇒ #(10 1 2 40 50)
```

(vector-fill! ***vector fill*****)** procedure
(vector-fill! ***vector fill start*****)**
(vector-fill! ***vector fill start end*****)**

The `vector-fill!` procedure stores *fill* in the elements of *vector* between *start* and *end*. *start* defaults to 0 and *end* defaults to the length of *vector*.

```
(define a (vector 1 2 3 4 5))
(vector-fill! a ’smash 2 4)
a  ⇒  #(1 2 smash smash 5)
```

(vector-reverse! ***vector*****)** procedure
(vector-reverse! ***vector start*****)**
(vector-reverse! ***vector start end*****)**

Procedure `vector-reverse!` destructively reverses the contents of *vector* between *start* and *end*. *start* defaults to 0 and *end* defaults to the length of *vector*.

```
(define a (vector 1 2 3 4 5))
(vector-reverse! a)
a  ⇒  #(5 4 3 2 1)
```

(vector-sort! ***pred vector*****)** procedure
(vector-sort! ***pred vector start*****)**
(vector-sort! ***pred vector start end*****)**

Procedure `vector-sort!` destructively sorts the elements of *vector* using the "less than" predicate *pred* between the indices *start* and *end*. Default for *start* is 0, for *end* it is the length of the vector.

```
(define a (vector 7 4 9 1 2 8 5))
(vector-sort! < a)
a  ⇒  #(1 2 4 5 7 8 9)
```

(vector-map! *f vector1 vector2* ...) procedure

Similar to `vector-map` which maps the various elements into a new vector via function *f*, procedure `vector-map!` destructively inserts the mapped elements into *vector1*. The dynamic order in which *f* gets applied to the elements is unspecified.

```
(define a (vector 1 2 3 4))
(vector-map! + a #(10 20 30))
a  ⇒  #(11 22 33 4)
```

(vector-map/index! *f vector1 vector2* ...) procedure

Similar to `vector-map/index` which maps the various elements together with their index into a new vector via function *f*, procedure `vector-map/index!` destructively inserts the mapped elements into *vector1*. The dynamic order in which *f* gets applied to the elements is unspecified.

```
(define a (vector 1 2 3 4))
(vector-map/index! (lambda (i x y) (cons i (+ x y))) a #(10 20 30))
a  ⇒  #((0 . 11) (1 . 22) (2 . 33) 4)
```

59.6 Converting vectors

(vector->list *vector*) procedure
(vector->list *vector start*)
(vector->list *vector start end*)

The `vector->list` procedure returns a newly allocated list of the objects contained in the elements of *vector* between *start* and *end* in the same order line in *vector*.

```
(vector->list '#(dah dah didah))  ⇒  (dah dah didah)
(vector->list '#(dah dah didah) 1 2)  ⇒  (dah)
```

(vector->string *vector*) procedure
(vector->string *vector start*)
(vector->string *vector start end*)

The `vector->string` procedure returns a newly allocated string of the objects contained in the elements of *vector* between *start* and *end*. This procedure preserves the order of the characters. It is an error if any element of vector between *start* and *end* is not a character.

```
(vector->string #(#\1 #\2 #\3)  ⇒  "123"
```

60 LispPad AppleScript

Library `(lisppad applescript)` exports procedures for invoking *Automator workflows* and *AppleScript* scripts and subroutines from Scheme code. Since LispPad runs in a sandbox and scripts and subroutines are executed outside of the sandbox, this will enable direct integrations with other macOS applications supporting AppleScript or Automator such as Mail, Safari, Music, etc.

60.1 Script authorization

The script authorization mechanism of macOS is unfortunately a bit cumbersome, requiring the installation of the *Automator* and *AppleScript* files in a particular directory specifically for LispPad. `(system-directory 'application-scripts)` returns a list of directories in which scripts are accessible by LispPad. This includes typically the directory:

```
/Users/username/Library/Application Scripts/net.objecthub.LispPad
```

This directory can be opened on macOS's Finder via:

```
(open-file (car (system-directory 'application-scripts)))
```

Scripts need to be copied to this directory.

60.2 Script integration

As an example, the following script defines two AppleScript subroutines `safariFrontURL` and `setSafariFrontURL`. The AppleScript code also displays an error if the script is run overall as its only role is to make subroutines accessible to LispPad. Such scripts are written using Apple's *Script Editor* application and need to be stored in a directory accesible by LispPad as explained above.

```
on safariFrontURL()
  tell application "Safari" to return URL of front document
end safariFrontURL

on setSafariFrontURL(newUrl)
  tell application "Safari" to set URL of front document to newUrl
end setSafariFrontURL

on run
  display alert "Do not run this script. It provides AppleScript sub-routines to LispPad."
end run
```

Assuming that the script was saved in a file at path:

```
/Users/username/Library/Application Scripts/net.objecthub.LispPad/AccessSafari.scpt
```

it is now possible to load the script via procedure `applescript` into an *AppleScript* object from which the various subroutines can be accessed:

```
(import (lisppad applescript))
(define script (applescript "AccessSafari.scpt"))
```

It is possible to run the whole script via procedure `execute-applescript`:

```
(execute-applescript script)
```

The execution of scripts is always synchronous, so the procedure call to `execute-applescript` terminates only when the execution of the script terminates. When executed, the script above will always display an alert since it was not made to be executed.

It is not possible to pass parameters via `execute-applescript` or receive results. This can be achieved by calling subroutines with procedure `apply-applescript-proc`. The following code will invoke subroutine `safariFrontURL` from the script above and return the URL of the current frontmost Safari window:

```
(apply-applescript-proc script "safariFrontURL" '())
```

The third argument of procedure `apply-applescript-proc` is a list of parameters for the subroutine. The following code will set the URL of the frontmost Safari window to "https://www.lisppad.app".

```
(apply-applescript-proc script "setSafariFrontURL" '("https://www.lisppad.app"))
```

Library `(lisppad applescript)` provides a means to quickly create Scheme functions matching AppleScript subroutines. This is shown in the following code:

```
(define safari-front-url (applescript-proc script "safariFrontURL"))
(define set-safari-front-url! (applescript-proc script "setSafariFrontURL"))
(display (safari-front-url))
(newline)
(set-safari-front-url! "https://www.lisppad.app")
```

60.3 Exchanging data

This is how library `(lisppad applescript)` is mapping data types when exchanging data between Scheme and AppleScript:

Scheme datatype	AppleScript datatype
void	null
boolean	boolean
fixnum	int32
flonum	double
proper list	list
string	unicode text
date-time	date

If data of other data types is attempted to be exchanged, it might lead to failures or the data might get dropped or omitted.

60.4 API

(applescript? *obj*) procedure

Returns `#t` if *obj* is an AppleScript object, `#f` otherwise.

(applescript *path*) procedure

Loads and compiles the AppleScript file at *path* returning an AppleScript object that can be used to execute the script or subroutines defined by the script.

(applescript-path *script*) procedure

Returns the file path from which the AppleScript object *script* was created.

(execute-applescript *script*) procedure

Executes the given AppleScript script. The execution is synchronous and `execute-applescript` will only return once *script* has been executed.

(apply-applescript-proc *script name args*) procedure

Invokes the subroutine *name* defined by AppleScript *script* with the arguments *args*. *name* is a string, *script* is an AppleScript object, and *args* is a list of arguments passed on to the subroutine. `apply-applescript-proc` returns the result returned by the subroutine, i.e. the execution of the subroutine is synchronous.

(applescript-proc *script name*) procedure

Returns a Scheme procedure for subroutine *name* defined in AppleScript *script*. *name* is a string and *script* is an AppleScript object. `applescript-proc` is defined in the following way:

```
(define (applescript-proc script name)
  (lambda args
    (apply apply-applescript-proc script name args)))
```

61 LispPad Draw Map

Library `(lisppad draw map)` provides an API for creating map snapshots. A map snapshot encapsulates a map image and provides a procedure for mapping locations to points on the image. This makes it possible to draw on top of the image based on locations (lat/longs). Here is a typical use case for this library:

```
(import (lispkit draw)
        (lisppad location)
        (lisppad draw map))
(define d
  (let*
    (; Determine the current location
     (center (current-location))
     ; Show a 1km box around the center
     (area   (size 1000 1000))
     ; Create a map snapshot of 500x500 points
     (snapsh (make-map-snapshot center area (size 500 500) 'satellite))
     ; Determine the points on the map image for the center
     (pt     (map-snapshot-point snapsh center)))
    ; Create a drawing of the map with highlighted center
    (drawing
      ; Draw the map at the origin of the drawing
      (draw-image (map-snapshot-image snapsh) (point 0 0))
      ; Highlight the center with a red circle
      (set-fill-color red)
      (fill-ellipse
        (rect (point (- (point-x pt) 4) (- (point-y pt) 4))
              (size 8 8))))))
```

The body of the `let*` form first draws the image and then layers a red ellipse on top. This is done in the context of a drawing, which can then be turned into an image and saved.

(map-snapshot? *obj*) procedure

Returns `#t` if *obj* is a map snapshot object; otherwise `#f` is returned.

(make-map-snapshot *center dist size*) procedure
(make-map-snapshot *center dist size type*)
(make-map-snapshot *center dist size type poi*)
(make-map-snapshot *center dist size type poi bldng*)

Creates a new map snapshot object which represents a rectangular area of a map whose center is the location *center*. Locations are created and managed via library `(lisppad location)`. *dist* describes the width and height of the map region. If is either a `lat-long-span` object or a size object, as defined by library `(lispkit draw)`. `lat-long-span` objects describe a width and height in terms of a north-to-south and east-to-west distance measured in degrees. size objects are interpreted as width and height measured in meters. *size* is a size object describing the dimensions of the image in points. *type* is a symbold that indicates the map type. Supported are:

- `standard` : Street map that shows the position of all roads and some road names.
- `satellite` : Satellite imagery of the area.
- `satellite-flyover` : Satellite image of the area with flyover data where available.

- `hybrid` : Satellite image of the area with road and road name information layered on top.
- `hybrid-flyover` : Hybrid satellite image with flyover data where available.
- `standard-muted` : Street map where the underlying map details are less emphasized to make custom data on top stand out more.

poi is a list of symbols indicating the categories for which point of interests are highlighted on the map. The following categories are supported:

`airport`, `amusement-park`, `aquarium`, `atm`, `bakery`, `bank`, `beach`, `brewery`, `cafe`, `campground`, `car-rental`, `ev-charger`, `fire-station`, `fitness-center`, `supermarket`, `gas-station`, `hospital`, `hotel`, `laundry`, `library`, `marina`, `movie-theater`, `museum`, `national-park`, `nightlife`, `park`, `parking`, `pharmacy`, `police`, `post-office`, `public-transport`, `restaurant`, `restroom`, `school`, `stadium`, `store`, `theater`, `university`, `winery`, `zoo` .

bldng is a boolean parameter (default is `#f`) indicating whether to show buildings or not.

(map-snapshot-image *msh*) procedure
Given a map snapshot object *msh*, procedure `map-snapshot-image` returns an image of the map encapsulated by *msh*.

(map-snapshot-point *msh loc*) procedure
(map-snapshot-point *msh lat long*)

Given a map snapshot object *msh*, procedure `map-snapshot-point` returns a point on the image of the map that matches the given location *loc*, or the location derived from the given latitude *lat* and longitude *long*.

(lat-long-span *latspan longspan*) procedure
Returns a new `lat-long-span` object from the given latitudal (north-to-south) and longitudal (east-to-west) distances *latspan* and *longspan* measured in degrees.

62 LispPad Location

Library `(lisppad location)` implements procedures for geocoding and reverse geocoding and provides representations of *locations* (latitude, longitude, altitude) and *places* (structured representation of addresses).

62.1 Locations

A *location* consists of a latitude, a longitude, and an optional altitude. Locations are represented as lists of two or three flonum values.

(location? *obj*) procedure
Returns `#t` if the given expression *obj* is a valid location; returns `#f` otherwise.

(location *latitude longitude*) procedure
(location *latitude longitude altitude*)

Creates a location for the given *latitude*, *longitude*, and *altitude*. This procedure fails with an error if any of the provided arguments are not flonum values.

(current-location) procedure
Returns the current device location. If a device location can't be determined, the procedure returns `#f`. The procedure also returns `#f` if the user did not authorize the device to reveal the location to LispPad.

(location-latitude *loc*) procedure
Returns the latitude of location *loc*.

(location-longitude *loc*) procedure
Returns the longitude of location *loc*.

(location-altitude *loc*) procedure
Returns the altitude of location *loc*. Since altitudes are optional, procedure `location-altitude` returns `#f` if the altitude is undefined.

(location-distance *loc1 loc2*) procedure
Returns the distance between location *loc1* and location *loc2* in meters.

62.2 Places

A *place* is a structured representation describing a place on Earth. Its main components are address components, but a place might also provide meta-information such as the timezone of the place or the ISO country code. Library `(lispkit date-time)` provides more functionality to deal with such metadata. Places are represented as lists of one to ten strings in the following order:

1. ISO country code
2. Country
3. Region (a part of the country; e.g. State, Bundesland, Kanton, etc.)
4. Administrational region (a part of the region; e.g. County, Landkreis, etc.)

5. Postal code
6. City
7. Locality (a part of the city; e.g. District, Stadtteil, etc.)
8. Street
9. Street number
10. Time zone

Note that all components are optional. An optional component is represented as `#f`.

(place? *obj*) procedure

Returns `#t` if the given expression *obj* is a valid place; returns `#f` otherwise.

(place *code*) procedure
(place *code country*)
(place *code country region*)
(place *code country region admin*)
(place *code country region admin zip*)
(place *code country region admin zip city*)
(place *code country region admin zip city locality*)
(place *code country region admin zip city locality street*)
(place *code country region admin zip city locality street nr*)
(place *code country region admin zip city locality street nr tz*)

Returns a location for the given components of a place. Each component is either `#f` (= undefined) or a string.

(place-country-code *pl*) procedure
Returns the country code for place *pl* as a string or `#f` if the country code is undefined.

(place-country *pl*) procedure
Returns the country for place *pl* as a string or `#f` if the country is undefined.

(place-region *pl*) procedure
Returns the region for place *pl* as a string or `#f` if the region is undefined.

(place-admin *pl*) procedure
Returns the administrational region for place *pl* as a string or `#f` if the administrational region is undefined.

(place-postal-code *pl*) procedure
Returns the postal code for place *pl* as a string or `#f` if the postal code is undefined.

(place-city *pl*) procedure
Returns the city for place *pl* as a string or `#f` if the city is undefined.

(place-locality *pl*) procedure
Returns the locality for place *pl* as a string or `#f` if the locality is undefined.

(place-street *pl*) procedure
Returns the street for place *pl* as a string or `#f` if the street is undefined.

(place-street-number *pl*) procedure
Returns the street number for place *pl* as a string or `#f` if the street number is undefined.

(place-timezone *pl*) procedure
Returns the timezone for place *pl* as a string or `#f` if the timezone is undefined.

62.3 Geocoding

(geocode *obj*)
(geocode *obj locale*) procedure

Returns a list of locations for the given place or address. *obj* is either a valid place representation or it is an address string. *locale* is a symbol representing a locale, which is used to interpret the given place or address. `geocode` signals an error if the geocoding operation fails (e.g. if there is no network access).

```
(geocode "Brandschenkestrasse 110, Zürich" 'de_CH)
⇒ ((47.3654121 8.5247038))
```

(reverse-geocode *loc*)
(reverse-geocode *loc locale*)
(reverse-geocode *lat long*)
(reverse-geocode *lat long locale*) procedure

Returns a list of places for the given location. *loc* is a valid location. *lat* and *long* describe latitude and longitude as flonums directly. *locale* is a symbol representing a locale. It is used for the place representations returned by `reverse-geocode`.

```
(reverse-geocode (location 47.36541 8.5247) 'en_US)
⇒ (("CH" "Switzerland" "Zürich" "Zürich"
    "8002" "Zürich" "Brunau"
    "Brandschenkestrasse" "110" "Europe/Zurich"))
```

(place->address *pl*) procedure
Formats a place as an address. For this operation to succeed, it is important that the country code of the place *pl* is set as it is used to determine the address format.

```
(define pl (car (reverse-geocode (location 47.36541 8.5247) 'de_CH)))
pl ⇒ (("CH" "Schweiz" "Zürich" "Zürich"
       "8002" "Zürich" "Brunau"
       "Brandschenkestrasse" "110" "Europe/Zurich"))
(display (place->address pl))
⇒
Brandschenkestrasse 110
8002 Zürich
Schweiz
```

(address->place *str*)
(address->place *str locale*) procedure

Parses the given address string *str* into a place (or potentially multiple possible places) and returns this as a list of places. *locale* is a symbol representing a locale. It is used for the place representations returned by `address->place`.

```
(address->place "Brandschenkestrasse 110, Zürich")
⇒ (("CH" "Switzerland" "Zürich" "Zürich"
    "8002" "Zürich" "Brunau"
    "Brandschenkestrasse" "110" "Europe/Zurich"))
```

63 LispPad Speech

Library `(lisppad speech)` provides a speech synthesis API which parses text and converts it into audible speech. The conversion is based on factors like the language, the *voice*, and a range of parameters which are all aggregated by *speaker* objects.

63.1 Speech synthesis

(speak *text*) procedure
(speak *text speaker*)

Speaks the given string *text* using with the *speaker* object providing all speech synthesis parameters. If *speaker* is not provided, the value of parameter object `current-speaker` is used.

(phonemes *text*) procedure
(phonemes *text speaker*)

Converts the given natural language string *text* into a string of phonemes using the given *speaker*. If *speaker* is not provided, the value of parameter object `current-speaker` is used.

Speakers can be configured to speak phonemes instead of natural language via procedure `speaker-interpret-phonemes!`.

63.2 Speakers

A *speaker* is an object defining speech synthesis parameters. There is a *current speaker* which is used by default, unless a speaker is explicitly specified for the various procedures that require a speaker parameter.

A speaker object has the following components:

- an immutable voice,
- a mutable speaking rate,
- a mutable speaking volume,
- a flag determining whether the speaker interprets text or phonemes,
- a flag determining how numbers are interpreted, as well as
- a speaking pitch.

current-speaker parameter object
Defines the *current speaker*, which is used as a default by all functions for which the speaker argument is optional. If there is no current speaker, this parameter is set to `#f`.

(speaker? *obj*) procedure
Returns `#t` if *obj* is a speaker object; otherwise `#f` is returned.

(make-speaker) procedure
(make-speaker *voice*)

Returns a new speaker for the given *voice*. If *voice* is not provided, a default voice, specified at the operating system level, is being used. Speakers are stateful objects which can be configured with a number of procedures: `set-speaker-rate!` , `set-speaker-volume!` , `set-speaker-interpret-phonemes!` , `set-speaker-interpret-numbers!` , and `set-speaker-pitch!` .

(speaker-voice) procedure
(speaker-voice *speaker*)

Returns the voice of *speaker*. If *speaker* is not provided, the parameter object `current-speaker` is used.

(speaker-rate) procedure
(speaker-rate *speaker*)

Returns the speaking rate of *speaker*. If *speaker* is not provided, the parameter object `current-speaker` is used.

(set-speaker-rate! *rate*) procedure
(set-speaker-rate! *rate speaker*)

Sets the speaking rate of *speaker* to number *rate*. If *speaker* is not provided, the parameter object `current-speaker` is used.

(speaker-volume) procedure
(speaker-volume *speaker*)

Returns the volume of *speaker* as a flonum ranging from 0.0 to 1.0. If *speaker* is not provided, the parameter object `current-speaker` is used.

(set-speaker-volume! *volume*) procedure
(set-speaker-volume! *volume speaker*)

Sets the volume of *speaker* to number *volume* which is a flonum between 0.0 and 1.0. If *speaker* is not provided, the parameter object `current-speaker` is used.

(speaker-interpret-phonemes) procedure
(speaker-interpret-phonemes *speaker*)

Returns `#t` if *speaker* interprets phonemes instead of natural language text. If *speaker* is not provided, the parameter object `current-speaker` is used.

(set-speaker-interpret-phonemes! *phoneme?*) procedure
(set-speaker-interpret-phonemes! *phoneme? speaker*)

If boolean argument *phoneme?* is `#f` , *speaker* is configured to interpret natural language. If *phoneme?* is set to any other value, the *speaker* is interpreting phonemes instead. If *speaker* is not provided, the parameter object `current-speaker` is used.

(speaker-interpret-numbers) procedure
(speaker-interpret-numbers *speaker*)

Returns `#t` if *speaker* interprets numbers as a natural language speaker would do ("100" is spoken as "hundred"). If it returns `#f` , *speaker* decomposes numbers into a sequence of digits and speaks them individually ("100" is spoken as "one zero zero"). If *speaker* is not provided, the parameter object `current-speaker` is used.

(set-speaker-interpret-numbers! *natural?*) procedure
(set-speaker-interpret-numbers! *natural? speaker*)

Sets the number interpretation of *speaker* to boolean *natural?*. If *natural?* is `#t` *speaker* will interpret numbers as a natural language speaker would do ("100" is spoken as "hundred"). If *natural?* is `#f` , *speaker* decomposes numbers into a sequence of digits and speaks them individually ("100" is spoken as "one zero zero"). If *speaker* is not provided, the parameter object `current-speaker` is used.

(speaker-pitch) procedure
(speaker-pitch *speaker*)

Returns the pitch of *speaker* as a pair of two flonums: the car is the base of the pitch, and the cdr is the modulation of the pitch. If *speaker* is not provided, the parameter object `current-speaker` is used.

(set-speaker-pitch! *pitch*) procedure
(set-speaker-pitch! *pitch speaker*)

Sets the pitch of *speaker* to the pair of flonums *pitch* whose car is the base of the pitch, and the cdr is the modulation of the pitch. If *speaker* is not provided, the parameter object `current-speaker` is used.

63.3 Voices

Voices are provided by the operating system and library `(lispkit speech)` does not have an explicit representation as objects. Symbols are used as identifiers for voices. For example, `com.apple.speech.synthesis.voice.Alex` refers to the default US voice.

A voice has the following characteristics:

- Name (string)
- Age (fixnum)
- Gender (`male` or `female`)
- Locale (symbol, e.g. `en_US`)

Library `(lispkit system)` provides means to handle *locales*, including language and country codes.

(voice) procedure
(voice *name*)
(voice *id*)

Returns a symbol identifying the voice specified by the arguments of `voice`. If no argument is provided, an indentifier for the default voice is returned. If a *name* string is provided, then an identifier for a voice whose name is *name* is returned, or `#f` if no such voice exists. If an *id* symbol is provided, then an identifier for a voice whose identifier matches *id* is returned, or `#f` if no such voice exists.

(available-voices) procedure
(available-voices *lang*)
(available-voices *lang gender*)

Returns a list of symbols identifying voices matching the given language filter *lang* and gender filter *gender*. Both *lang* and *gender* are symbols. *lang* should either be a language or locale identifier. It can also be set to `#f` if only a gender filter is needed. *gender* should either be symbol `male` or `female`.

```
(available-voices 'en)
⇒ (com.apple.speech.synthesis.voice.Alex com.apple.speech.synthesis.voice.daniel
 ↪  com.apple.speech.synthesis.voice.fiona com.apple.speech.synthesis.voice.Fred
 ↪  com.apple.speech.synthesis.voice.karen com.apple.speech.synthesis.voice.moira
 ↪  com.apple.speech.synthesis.voice.rishi com.apple.speech.synthesis.voice.samantha
 ↪  com.apple.speech.synthesis.voice.tessa com.apple.speech.synthesis.voice.veena)
(available-voices (locale "en" "GB"))
⇒ (com.apple.speech.synthesis.voice.daniel)
```

(available-voice? *obj*) procedure
Returns `#t` if *obj* is a symbol identifying an available voice, otherwise `#f` is returned. This procedure fails if *obj* is neither a symbol nor the value `#f`.

(voice-name *voice*) procedure

Returns the name of the voice identified by symbol *voice*.

(voice-age *voice*) procedure

Returns the age of the voice identified by symbol *voice*.

(voice-gender *voice*) procedure

Returns the gender of the voice identified by symbol *voice*.

(voice-locale *voice*) procedure

Returns the locale of the voice identified by symbol *voice*.

64 LispPad System macOS

Library `(lisppad system macos)` defines an API for retrieving information of the LispPad application and user environment as well as scripting the LispPad user interface on macOS. Procedures that match the same functionality on iOS are also available via library `(lisppad system)`.

Library `(lisppad system macos)` provides functionality primarily for managing LispPad windows: new windows can be created, properties of existing windows can be changed, and the content of existing windows can be accessed and modified. There is also support for making use of simple dialogs, e.g. for displaying messages, asking the user to make a choice, or for letting the user choose a file or directory in a load or save panel.

64.1 Files

(project-directory) procedure
Returns the path to the project directory as defined in the preferences of LispPad. `project-directory` returns `#f` if no project directory was explicitly set.

(icloud-directory) procedure
Returns the path to the iCloud directory. `icloud-directory` returns `#f` if iCloud synchronization is disabled.

64.2 Windows

`(lisppad system macos)` does not provide a data structure for modeling references to LispPad windows. Instead, it uses integer ids as references. Two different types of windows can be managed:

- *Edit windows* are used for editing text, and
- *Graphics windows* are used for displaying drawings created via library `(lispkit draw)`.

Other types of windows are currently not accessible via library `(lisppad system macos)`.

(open-document *path*) procedure
Opens a document stored in a file at path *path*. Only documents that LispPad is able to open are supported.

(edit-windows) procedure
Returns an association list containing all open edit windows. Each open window has an entry of the form (*window id* . *window title*). For example, the result of invoking `(edit-windows)` could look like this: `((106102873393392 . "LispKit Libraries.md") (106377751319520 . "Untitled"))`.

(graphics-windows) procedure
Returns an association list containing all open graphics windows. Each open window has an entry of the form (*window id* . *window title*). For example, the result of invoking `(graphics-windows)` could look like this: `((106102873393789 . "My Drawing") (106377751899571 . "Untitled Drawing"))`.

(window-name *win*) procedure
Returns the name of the window with window id *win*.

(set-window-name! *win name*) procedure
Sets the name of the window with window id *win* to string *name*.

(window-position *win*) procedure
Returns the position of the window with window id *win*. The position of a window is the upper left corner of its title bar represented as a point.

(set-window-position! *win pos*) procedure
Sets the position of the window with window id *win* to point *pos*. The position of a window is the upper left corner of its title bar.

(window-size *win*) procedure
Returns the size of the window with window id *win*. The size of a window consists of its width and height represented as a size.

(set-window-size! *win size*) procedure
Sets the size of the window with window id *win* to size *size*. The size of a window consists of its width and height.

(close-window *win*) procedure
Closes the window with window id *win*.

64.3 Edit Windows

(make-edit-window *str pos size*) procedure
Creates a new edit window containing *str* as its textual content. The window's initial position is *pos* and its size is *size*.

(edit-window-text *win*) procedure
Returns the textual content of the edit window with the given window id *win*.

(insert-edit-window-text! *win str*) procedure
(insert-edit-window-text! *win str start*)
(insert-edit-window-text! *win str start end*)

Inserts a string *str* replacing text between *start* and *end* for the edit window with window id *win*. It *start* is not provided, *start* is considered to be 0 (i.e. the text is inserted at the beginning). If *end* is not provided, it is considered to be the length of the text contained in the edit window *win*.

(edit-window-text-length *win*) procedure
Returns the length of the text contained in the edit window with window id *win*.

64.4 Graphics Windows

A graphics window displays a *drawing* in a canvas of a given minimum *drawing size* (different from the *window size*). A *scaling factor* can be utilized to adjust the size of the drawing to the window. Furthermore, the *background color* of the drawing can be freely adjusted. There is a *graphics window label* at the bottom of graphic windows which can be used to display a line of text.

(make-graphics-window *drawing dsize*) procedure
(make-graphics-window *drawing dsize title*)

(make-graphics-window *drawing dsize title pos*)
(make-graphics-window *drawing dsize title pos size*)

Creates a new graphics window for drawing *drawing*. *dsize* refers to the size of the drawing. *title* is the window title of the new window, *pos* is its initial position, and *size* corresponds to the initial size of the graphics window.

(use-graphics-window *drawing dsize title*) procedure
(use-graphics-window *drawing dsize title pos*)
(use-graphics-window *drawing dsize title pos size*)
(use-graphics-window *drawing dsize title pos size ignore*)

This is almost equivalent to function `make-graphics-window`. The main difference consists in `use-graphics-window` reusing an existing graphics window if there is one open with the given title. If there is no window whose title matches *title*, a new graphics window will be created. If a window exists already and boolean argument *ignore* is set to `#t`, the existing window's position and size will not be updated.

(update-graphics-window *win*) procedure
This function forces the graphics window with window id *win* to redraw its content. Currently, graphics windows are only guaranteed to redraw automatically after executing a command in the session window which was used to create the drawing object.

(graphics-window-drawing *win*) procedure
Returns the drawing associated with the graphics window with window id *win*.

(set-graphics-window-drawing! *win drawing*) procedure
Sets the drawing associated with the graphics window with window id *win* to *drawing*.

(graphics-window-label *win*) procedure
Each graphics window has a label at the bottom of the window. This label can be arbitrarily modified, and e.g. used as a caption. `graphics-window-label` returns the label of the graphics window with window id *win*.

(set-graphics-window-label! *win str*) procedure
Each graphics window has a label at the bottom of the window. The label of graphics window *win* can be set via function `set-graphics-window-label!` to string *str*.

(graphics-window-background *win*) procedure
Returns the background color for the graphics window with id *win*. By default, the background color is white.

(set-graphics-window-background! *win color*) procedure
Sets the background color for the graphics window with id *win* to *color*.

(drawing-size *win*) procedure
Returns the size of the drawing associated with graphics window *win*. Please note that this is not the *window size* of *win*.

(set-drawing-size! *win size*) procedure
Sets the size of the drawing associated with graphics window *win* to *size*. Please note that this is not setting the *window size* of *win*.

(drawing-scale *win*) procedure
Returns the scaling factor used for showing the drawing in graphics window with id *win*. By default, the scaling factor is 1.0.

(set-drawing-scale! *win factor*) procedure
Sets the scaling factor used for showing the drawing in graphics window with id *win* to *factor*.

64.5 Navigation

(show-message-panel *title*) procedure
(show-message-panel *title str*)
(show-message-panel *title str button*)

Shows a message panel within the current session window. *title* refers to the panel title, *str* is the message to be displayed_, and *button* is the label of the confirmation button.

(show-choice-panel *title str*) procedure
(show-choice-panel *title str yes*)
(show-choice-panel *title str yes no*)

Shows a choice panel within the current session window. *title* refers to the panel title, *str* is the question to be asked, and *yes* and *no* refer to the two labels of the buttons for users to choose. The function returns `#t` if the user clicked on the "yes button".

(show-load-panel *prompt*) procedure
(show-load-panel *prompt folders*)
(show-load-panel *prompt folders filetypes*)

Displays a load panel within the current session window together with the given *prompt* message. *folders* is a boolean argument; it should be set to `#t` if the user is required to select a folder. *filetypes* is a list of suffixes of selectable file types.

(show-save-panel *prompt*) procedure
(show-save-panel *prompt filetypes*)

Displays a save panel within the current session window together with the given *prompt* message. *filetypes* is a list of suffixes of selectable file types.

(show-help *name*) procedure
Shows documentation for identifier *name* (string), if available.

(search-help *str*) procedure
Searches all documentation for string *str* and shows all matches.

64.6 Sessions

(session-id) procedure
Returns a unique fixnum (within LispPad) identifying the session.

(session-name) procedure
Returns the name of the LispPad session which executes this function.

(session-display *obj*) procedure
(session-display *obj bold?*)
(session-display *obj bold? col*)

Displays value *obj* in the current session in color *col* and in bold if *bold?* is true.

(session-write *obj*) procedure
(session-write *obj bold?*)
(session-write *obj bold? col*)

Writes the value *obj* into the current session in color *col* and in bold if *bold?* is true.

(session-log *time sev str*) procedure
(session-log *time sev str tag*)

Logs the message *str* with severity *sev* at the given timestamp *time* (a double value) in the session log. *sev* is one of the following symbols: `debug`, `info`, `warn`, `error`, or `fatal`.

64.7 Environment

(screen-size) procedure
(screen-size *win*)

Returns the screen size of the screen on which window *win* is displayed. If argument *win* is omitted, function `screen-size` will return the size of the main screen.

(dark-mode?) procedure
Return `#t` if the session window of the LispPad session which executes this function is rendered in *dark mode*; returns `#f` otherwise.

65 LispPad System iOS

Library `(lisppad system ios)` implements a simple API for LispPad Go-specific system procedures as well as functionality for scripting the *LispPad Go* application. Procedures that match the same functionality on macOS are also available via library `(lisppad system)`.

65.1 Files

(project-directory) procedure
Returns the path to the documents directory of LispPad Go, where local files are stored.

(icloud-directory) procedure
Returns the path to the iclouds directory of LispPad Go, where files are stored that are synchronized via iCloud. This procedure returns `#f` if iCloud synchronization is not enabled.

(icloud-list) procedure
Returns a list of file paths of all LispPad-related files that are synchronized via iCloud. The paths are relative to the iCloud directory which can be obtained via procedure `icloud-directory`.

(preview-file *path*) procedure
Shows a preview of the content of the file at the given file path *path*. Supported are many different types of files, including text files, images, PDF files, spreadsheets, etc.

(share-file *path*) procedure
Shows a file share panel for the file at file path *path*, allowing the user to share the file with another application.

(open-in-files-app *path*) procedure
Opens the Files app at the given file path *path*.

65.2 Images

(save-bitmap-in-library *img*) procedure
Saves the given bitmap-based image *img* in the photo library of the user. The first time this procedure gets invoked, it asks the user for permission to access the photo library.

(load-bitmaps-from-library) procedure
(load-bitmaps-from-library *max*)
(load-bitmaps-from-library *max filter*)

Opens an image selector showing the images of the user's photo library. The user can select up to *max* images from the photo library. These are returned by procedure `load-bitmaps-from-library` as a list of images. *filter* is an image filter for narrowing down the types of images that are shown. *filter* has either the form of:

- a symbol, indicating a type of images (e.g. `bursts`, `panoramas`, `videos`),
- `(not` *filter* `)`, indicating the inverse of *filter*,

- `(and` *filter* … `)` , indicating the conjunction of the given filters, or
- `(or` *filter* … `)` , indicating the disjunction of the given filters.

The following image type tags, expressed as a symbol, are supported: `bursts` , `cinematic-videos` , `depth-effect-photos` , `images` , `live-photos` , `panoramas` , `screen-recordings` , `screenshots` , `slomo-videos` , `timelapse-videos` , and `videos` . The default is `images` .

(load-bytevectors-from-library *obj*) procedure
Opens an image selector showing the images of the user's photo library. The user can select up to *max* images from the photo library. These are returned by procedure `load-bitmaps-from-library` as a list of bytevectors. *filter* is an image filter for narrowing down the types of images that are shown, as documented for procedure `load-bitmaps-from-library` . `load-bytevectors-from-library` is useful if one is dealing with videos or other types of data that are not supported natively.

65.3 Navigation

(show-preview-panel *obj*) procedure
(show-preview-panel *obj type*)

Shows a preview of *obj* when possible. Supported are the following types of data: strings (textual data), bytevectors (binary data), styled text, images, and drawings. For strings and bytevectors it is important that parameter *type* is used to narrow down the type of content. *type* is a string representing a "file extension". Supported are at least: `"png"` , `"jpg"` , `"jpeg"` , `"gif"` , `"bmp"` , `"tif"` , `"tiff"` , `"text"` , `"txt"` , `"markdown"` , `"md"` , `"html"` , `"rtf"` , and `"rtfd"` . Other *type* extensions might work.

(show-share-panel *obj*) procedure
(show-share-panel *obj type*)

Shows a panel for sharing *obj* with other applications when possible. Supported are the following types of data: strings (textual data), bytevectors (binary data), styled text, images, and drawings. For strings and bytevectors it is important that parameter *type* is used to narrow down the type of content. *type* is a string representing a "file extension". Supported are at least: `"png"` , `"jpg"` , `"jpeg"` , `"gif"` , `"bmp"` , `"tif"` , `"tiff"` , `"text"` , `"txt"` , `"markdown"` , `"md"` , `"html"` , `"rtf"` , and `"rtfd"` . Other *type* extensions might work.

(show-load-panel *prompt folders filetypes*) procedure
Displays a file load panel with the given *prompt* message. *folders* is a boolean argument; it should be set to `#t` if the user is required to select a folder. *filetypes* is a list of suffixes of selectable file types.

(show-save-panel *prompt*) procedure
(show-save-panel *prompt path*)
(show-save-panel *prompt path locked*)

Displays a file save panel with the given *prompt* message. *path* might refer to a pre-selected file. Boolean argument *locked* determines if the folder (provided via *path*) may be changed or not.

(show-interpreter-tab *tab*) procedure
(show-interpreter-tab *tab canvas*)

LispPad Go has three interpreter views: the console view, the log view, and the canvas view. Procedure `show-interpreter-tab` enables programmatic navigation between these three views. *tab* is one of the following three symbols: `console` , `log` , and `canvas` . If *tab* is `canvas` , then a second argument *canvas* can be provided referring to a canvas to select.

(show-help *name*) procedure
Shows documentation for identifier *name* (symbol or string), if available.

65.4 Canvases

A canvas shows a drawing in the canvas view of the interpreter. The following parameters can be configured for every canvas: the drawing, the size of the canvas, a scale factor (default is 1.0), and an optional background color. Canvases are identified by a fixnum identifier.

(make-canvas *drawing size*) procedure
(make-canvas *drawing size name*)
(make-canvas *drawing size name color*)

Creates a new canvas of *size* showing *drawing*. String *name* is used to identify the canvas in the user interface. If *name* is not provided, a unqiue name is generated. *color* is the background color of the new canvas. `make-canvas` returns a canvas identifier (fixnum), which is used to refer to canvases in the API.

(use-canvas *drawing size*) procedure
(use-canvas *drawing size name*)
(use-canvas *drawing size name color*)

Creates or reuses a canvas of *size* showing *drawing*. If there is already an existing canvas of the same *name*, it is reused and reconfigured. Otherwise, a new canvas is created. *color* is the background color of the canvas. `use-canvas` returns a canvas identifier (fixnum), which is used to refer to canvases in the API.

(close-canvas *canvas*) procedure
Closes *canvas*. *canvas* is either a canvas identifier (fixnum) or it is a name of a canvas (string). `close-canvas` returns `#t` if a canvas was closed, otherwise `#f` is returned.

(canvas-name *canvas*) procedure
Returns the name of *canvas* as a string. *canvas* is either a canvas identifier (fixnum) or it is a name of a canvas (string). It is an error if no matching canvas was found.

(set-canvas-name! *canvas name*) procedure
Sets the name of *canvas* to string *name*. *canvas* is either a canvas identifier (fixnum) or it is a name of a canvas (string). It is an error if no matching canvas was found.

(canvas-size *canvas*) procedure
Returns the size of *canvas*. *canvas* is either a canvas identifier (fixnum) or it is a name of a canvas (string). It is an error if no matching canvas was found.

(set-canvas-size! *canvas size*) procedure
Sets the size of *canvas* to *size*. *canvas* is either a canvas identifier (fixnum) or it is a name of a canvas (string). It is an error if no matching canvas was found.

(canvas-scale *canvas*) procedure
Returns the scale factor used by *canvas*. The default is 1.0. *canvas* is either a canvas identifier (fixnum) or it is a name of a canvas (string). It is an error if no matching canvas was found.

(set-canvas-scale! *canvas scale*) procedure
Sets the scale factor used by *canvas* to number *scale*. *canvas* is either a canvas identifier (fixnum) or it is a name of a canvas (string). It is an error if no matching canvas was found.

(canvas-background *canvas*) procedure
Returns the background color of *canvas* if one was defined, or `#f` if no background color was set. *canvas* is either a canvas identifier (fixnum) or it is a name of a canvas (string). It is an error if no matching canvas was found.

(set-canvas-background! *canvas color*) procedure

Sets the background color of *canvas* to *color* (color or `#f`). *canvas* is either a canvas identifier (fixnum) or it is a name of a canvas (string). It is an error if no matching canvas was found.

(canvas-drawing *canvas*) procedure

Returns the drawing shown by *canvas*. *canvas* is either a canvas identifier (fixnum) or it is a name of a canvas (string). It is an error if no matching canvas was found.

(set-canvas-drawing! *canvas drawing*) procedure

Sets the background color of *canvas* to *drawing*. *canvas* is either a canvas identifier (fixnum) or it is a name of a canvas (string). It is an error if no matching canvas was found.

65.5 Sessions

(session-id) procedure

Returns a unique fixnum (within LispPad Go) identifying the interpreter session. The number changes when the interpreter is reset or the application is restarted.

(session-name) procedure

Returns the name of the LispPad session which executes this function. This name is customizable on macOS. On iOS, the name is generated using the session id.

(session-log *time sev str*) procedure
(session-log *time sev str tag*)

Logs the message *str* with severity *sev* at the given timestamp *time* (a double value) in the session log. *sev* is one of the following symbols: `debug`, `info`, `warn`, `error`, or `fatal`.

65.6 Environment

(screen-size) procedure

Returns the screen size of the screen of the device running LispPad Go.

(dark-mode?) procedure

Return `#t` if the device on which LispPad Go is running is using *dark mode*; returns `#f` otherwise.

66 LispPad Turtle

Library `(lisppad turtle)` implements a simple graphics pane (a graphics window on macOS, a canvas on iOS) for displaying turtle graphics. The library supports one graphics pane per LispPad session which gets initialized by invoking `init-turtle`. `init-turtle` will create a new turtle and display its drawing on a graphics pane. If there is already an existing graphics pane with the given title, it will be reused. Turtle graphics can be reset via `reset-turtle`.

As opposed to library `(lispkit draw turtle)`, this library supports an *indicator*, i.e. a symbol that shows where the turtle is currently located and whether the pen is up or down. By default, an arrow is used as an indicator. A yellow arrow indicates that the pen is down, a white translucent arrow indicates that the pen is up.

Once `init-turtle` was called, the following functions can be used to move the turtle across the plane:

- `(indicator-on)` : Show the turtle indicator
- `(indicator-off)` : Hide the turtle indicator
- `(pen-up)` : Lifts the turtle
- `(pen-down)` : Drops the turtle
- `(pen-color color)` : Sets the current color of the turtle
- `(pen-size size)` : Sets the size of the turtle pen
- `(home)` : Moves the turtle back to the origin
- `(move x y)` : Moves the turtle to position `(x, y)`
- `(heading angle)` : Sets the angle of the turtle (in radians)
- `(turn angle)` : Turns the turtle by the given angle (in radians)
- `(left angle)` : Turn left by the given angle (in radians)
- `(right angle)` : Turn right by the given angle (in radians)
- `(forward distance)` : Moves forward by `distance` units drawing a line if the pen is down
- `(backward distance)` : Moves backward by `distance` units drawing a line if the pen is down
- `(arc angle radius)` : Turns the turtle by the given angle (in radians) and draws an arc with `radius` around the current turtle position.

This library provides a simplified, interactive version of the API provided by library `(lispkit draw turtle)`.

66.1 Setup

(init-turtle) procedure
(init-turtle scale)
(init-turtle scale title)

Initializes a new turtle and displays its drawing in a graphics pane (a graphics window on macOS, a canvas on iOS). `init-turtle` gets two optional arguments: `scale` and `title`. `scale` is a scaling factor which determines the size of the turtle drawing. `title` is a string that defines the name of the graphics pane used by the turtle graphics. It also acts as the identify of the turtle graphics pane; i.e. it won't be possible to have two sessions with the same name but a different graphics pane.

(reset-turtle) procedure

Closes the graphics pane and resets the turtle library.

(turtle-drawing) procedure

Returns the drawing associated with the current turtle.

(turtle-indicator) procedure

Returns the turtle indicator that is currently used.

66.2 Indicators

Turtle indicators are defined and configured via record `<indicator>`. Instances are created using `make-indicator`. `empty-indicator` is a predefined indicator showing nothing. `arrow-indicator` is a constructor for creating arrow indicators of different sizes.

indicator-type-tag object

Symbol representing the `indicator` type. The `type-for` procedure of library `(lispkit type)` returns this symbol for all indicator objects.

(indicator? *obj*) procedure

Returns `#t` if *obj* is a turtle indicator object; returns `#f` otherwise.

(make-indicator *shape width stroke-color up-color down-color*) procedure

Returns a new turtle indicator of given *shape* and stroke *width*. *stroke-color* is the color used to draw the shape, *up-color* is the color used to fill the shape when the pen is up, and *down-color* is the color used to fill the shape when the pen is down.

empty-indicator object

An indicator which is not drawing anything. This is used to disable indicators fully.

(make-arrow-indicator) procedure
(make-arrow-indicator *size*)
(make-arrow-indicator *size width*)
(make-arrow-indicator *size width stroke-color*)
(make-arrow-indicator *size width stroke-color up-color*)
(make-arrow-indicator *size width stroke-color up-color down-color*)

Returns a new arrow indicator. *size* is the size of the arrow shape (12 is the default), *width* is the stroke width used to draw the shape, *stroke-color* is the color used to draw the shape, *up-color* is the color used to fill the shape when the pen is up, and *down-color* is the color used to fill the shape when the pen is down.

66.3 Drawing

(indicator-on) procedure

Show the turtle indicator.

(indicator-off) procedure

Hide the turtle indicator.

(pen-up) procedure

Lifts the turtle from the plane. Subsequent `forward` and `backward` operations don't lead to lines being drawn. Only the current coordinates are getting updated.

(pen-down) procedure

Drops the turtle onto the plane. Subsequent `forward` and `backward` operations will lead to lines being drawn.

(pen-color *color*) procedure

Sets the drawing color of the turtle to *color*. *color* is a color object as defined by library `(lispkit draw)` .

(pen-size *size*) procedure

Sets the pen size of the turtle to *size*. The pen size corresponds to the width of lines drawn by `forward` and `backward` .

(home) procedure

Moves the turtle to its home position.

(move *x y*) procedure

Moves the turtle to the position described by the coordinates *x* and *y*.

(heading *angle*) procedure

Sets the heading of the turtle to *angle*. *angle* is expressed in terms of degrees.

(turn *angle*) procedure

Adjusts the heading of the turtle by *angle* degrees.

(right *angle*) procedure

Adjusts the heading of the turtle by *angle* degrees.

(left *angle*) procedure

Adjusts the heading of the turtle by -*angle* degrees.

(forward *distance*) procedure

Moves the turtle forward by *distance* units drawing a line if the pen is down.

(backward *distance*) procedure

Moves the turtle backward by *distance* units drawing a line if the pen is down.

(arc *angle radius*) procedure

Turns the turtle by the given *angle* (in radians) and draws an arc with *radius* around the current turtle position if the pen is down.

67 SRFI Libraries

LispPad supports a broad range of libraries standardized and published via the SRFI process. The following libraries come pre-packaged with LispPad:

- SRFI 1: List Library
- SRFI 2: AND-LET* - an AND with local bindings, a guarded LET* special form
- SRFI 6: Basic String Ports
- SRFI 8: receive - Binding to multiple values
- SRFI 9: Defining Record Types
- SRFI 11: Syntax for receiving multiple values
- SRFI 14: Character-set library
- SRFI 16: Syntax for procedures of variable arity
- SRFI 17: Generalized set!
- SRFI 18: Multithreading support
- SRFI 19: Time Data Types and Procedures
- SRFI 23: Error reporting mechanism
- SRFI 26: Notation for Specializing Parameters without Currying
- SRFI 27: Sources of Random Bits
- SRFI 28: Basic Format Strings
- SRFI 31: A special form rec for recursive evaluation
- SRFI 33: Integer Bitwise-operation Library
- SRFI 34: Exception Handling for Programs
- SRFI 35: Conditions
- SRFI 39: Parameter objects
- SRFI 41: Streams
- SRFI 46: Basic Syntax-rules Extensions
- SRFI 48: Intermediate Format Strings
- SRFI 51: Handling rest list
- SRFI 54: Formatting
- SRFI 55: require-extension
- SRFI 63: Homogeneous and Heterogeneous Arrays
- SRFI 64: A Scheme API for test suites
- SRFI 69: Basic hash tables
- SRFI 87: `=>` in case clauses
- SRFI 95: Sorting and Merging
- SRFI 98: An interface to access environment variables
- SRFI 101: Purely Functional Random-Access Pairs and Lists
- SRFI 102: Procedure Arity Inspection
- SRFI 111: Boxes
- SRFI 112: Environment inquiry
- SRFI 113: Sets and bags
- SRFI 118: Simple adjustable-size strings
- SRFI 121: Generators
- SRFI 125: Intermediate hash tables
- SRFI 128: Comparators

- SRFI 129: Titlecase procedures
- SRFI 132: Sort Libraries
- SRFI 133: Vector Library
- SRFI 134: Immutable Deques
- SRFI 135: Immutable Texts
- SRFI 137: Minimal Unique Types
- SRFI 141: Integer division
- SRFI 142: Bitwise Operations
- SRFI 143: Fixnums
- SRFI 144: Flonums
- SRFI 145: Assumptions
- SRFI 146: Mappings
- SRFI 149: Basic syntax-rules Template Extensions
- SRFI 151: Bitwise Operations
- SRFI 152: String Library
- SRFI 154: First-class dynamic extents
- SRFI 155: Promises
- SRFI 158: Generators and Accumulators
- SRFI 161: Unifiable Boxes
- SRFI 162: Comparators sublibrary
- SRFI 165: The Environment Monad
- SRFI 166: Monadic Formatting
- SRFI 167: Ordered Key Value Store
- SRFI 173: Hooks
- SRFI 174: POSIX Timespecs
- SRFI 175: ASCII Character Library
- SRFI 177: Portable keyword arguments
- SRFI 180: JSON
- SRFI 189: Maybe and Either: optional container types
- SRFI 194: Random data generators
- SRFI 195: Multiple-value boxes
- SRFI 196: Range Objects
- SRFI 204: Wright-Cartwright-Shinn pattern matcher
- SRFI 208: NaN procedures
- SRFI 209: Enums and Enum Sets
- SRFI 210: Procedures and Syntax for Multiple Values
- SRFI 214: Flexvectors
- SRFI 215: Central log exchange
- SRFI 216: SICP Prerequisites
- SRFI 217: Integer sets
- SRFI 219: Define higher-order lambda
- SRFI 221: Generator/accumulator sub-library
- SRFI 222: Compound objects
- SRFI 223: Generalized binary search procedures
- SRFI 224: Integer mappings
- SRFI 227: Optional Arguments
- SRFI 228: Composing Comparators
- SRFI 229: Tagged procedures
- SRFI 230: Atomic Operations
- SRFI 232: Flexible curried procedures
- SRFI 233: INI files
- SRFI 235: Combinators