User's Guide to Elib - The Emacs Lisp Library

version 0.06

Inge Wallin

last updated 24 Jan 1993

NO WARRANTY

1 Installation

This section describes the installation of Elib, the GNU emacs lisp library. You should install not only the library itself, but also the on-line documentation so that your users will know how to use it. You can create typeset documentation from the file `elib.texinfo` as well as an on-line info file. The following steps are also described in the file `INSTALL` in the source directory.

1.1 Installation of the elisp library

1. Edit the file `Makefile` to reflect the situation at your site. The only thing you will have to change at this stage is the definition of `LISPDIR`. In this directory, a subdirectory with the name `elib` will be created. All elisp files of the library will be copied there when you do the actual installation (see step 2. below). We suggest you use your local elisp directory (usually `/usr/local/lib/elisp` or something similar) for this.
2. Type '`make install`' in the source directory. This will byte-compile all `.el` files of the library and create the subdirectory `elib` in the directory you specified in step 1. It will also copy both the `.el` and the `.elc` files of the library there.

   If you don't want to install the `.el` files but only the `.elc` files (the byte-compiled files), you can type '`make install_elc`' instead of '`make install`'.

   If you only want to create the compiled elisp files, but don't want to install them, you can type '`make elcfiles`' instead. This is what happens if you only type '`make`' without parameters.
3. Edit the file `default.el` in your emacs lisp directory (usually `/usr/gnu/emacs/lisp` or something similar) and enter the contents of the file `elib-startup.el` into it. This file was created from the file `startup_template.el` by the `make` in step 2. It contains an additional entry in the variable `load-path`, which determines the path where emacs looks for elisp files (ending in `.el` or `.elc`).

1.2 Installation of the on-line manual.

1. Create the info file `elib.info` from `elib.texinfo`. If you have the `makeinfo` program, you can do this by running it on `elib.texinfo`. Otherwise you can do it with emacs by running these steps:
   1. Read `elib.texinfo` into an emacs buffer.
   2. Type '`M-X texinfo-format-buffer`'
   3. Save the newly created info file `elib`.
2. Move the info file `elib.info` to your standard info directory. Usually this is is the directory `/usr/gnu/emacs/info` or something similar. (See step 3 above).
3. Edit the file `dir` in the info directory and enter one line to contain a pointer to the info file `elib`. The line can, for instance, look like this:

```
* Elib: (elib.info).     The Emacs Lisp Library.
```

1.3 How to make typeset documentation from elib.texinfo

You can also make a typeset manual from the file `elib.texinfo`. To to this, you must have the `TeX` text formatting program installed. Just follow these steps:

1. If the file `texinfo.tex` is not properly installed in the path given by the environment variable *TEXINPUTS*, get it and put it in the same directory as `elib.texinfo` (the source directory of elib). This file contains macros used by the TeX text formatting program to produce typeset output from a texinfo file. You can get this from, e.g., `prep.ai.mit.edu` in the US or from `isy.liu.se` in Europe.
2. Run TeX by typing '`tex elib.texinfo`'. You might need to do this twice to get all cross references correct. If you have the `texindex` program, you can create a sorted index by typing '`texindex elib.??`' between the two TeX passes. If you don't do this, you still get a typeset manual, but you will not get the index.
3. Convert the resulting device independent file `elib.dvi` to a form which your printer can output and print it. If you have a postscript printer there is a program, `dvi2ps`, which can do this. There is also a program which comes together with TeX, `dvips`, which you can use.

2 What is Elib?

Elib, the GNU emacs lisp library, is a collection of elisp functions which you can use as parts of your own elisp programs. Each file contains functions which have something in common, e.g. they handle a certain data type.

Elib is designed to be both as efficient and as easy to use as possible. Each file in Elib uses the elisp function `provide` to tell emacs when it has been loaded. To use the functions in the file `foo`, you just have to put a line such as:

```
(require 'foo)
```

into your own elisp file. This will cause emacs to load the file `foo.elc` and evaluate the functions in it. This, of course, requires that your system manager has installed Elib properly on your system. See Chapter 1 [Installation], page 5, for more information.

2.1 Contributors to Elib

The following persons have made contributions to GNU Elib.

- Inge Wallin wrote most of the otherwise unattributed functions in Elib as well as all documentation.
- Sebastian Kremer contributed the string functions.
- Thomas Bellman wrote some of the code for AVL trees.
- Per Cederqvist wrote the cookie package and the doubly linked list. The first design of `cookie.el` was made by Inge Wallin.

2.2 Where can I get Elib?

There will probably be a number of sites archiving Elib. Currently the latest release can always be fetched via anonymos ftp from `isy.liu.se`, (IP no. 130.236.1.3) in the directory `pub/gnu/elisp-programs`, or from `ftp.lysator.liu.se` in `pub/emacs`.

3 Container Data Types

Container data types are data types which are used to hold and organize other data. Since lisp is a dynamically typed language, any container data type can hold any other data type or a mix of other data types. This is contrary to the case for `C` or `C++` where all data in a typical container must be of the same type.

As a convention do all names of the functions handling a certain container data type begin in `<type>-`, i.e. the functions implementing the container data type `foo` all start with `foo-`.

3.1 The Stack Data Type

The stack data type provides a simple LIFO stack. There are two implementations of a stack in Elib, one using macros and one using functions. The names of the functions/macros in the two implementations are the same, but the efficiency of using one or the other vary greatly under different circumstances.

The implementation using macros should be used when you want to byte-compile your own elisp program. This will be most efficient since byte-compiling an elisp function using macros has the same effect as using inline code in `C`.

To use the stack data type, put the line

```
(require 'stack-f)
```

in your own elisp source file if you want to use the implementation using functions or

```
(require 'stack-m)
```

if you want to use the implementation using macros. This is the only difference between them, so it is easy to switch between them during debugging.

The following functions are provided by the stack:

`(stack-create)`
: Create a new empty stack.

`(stack-p stack)`
: Return `t` if *stack* is a stack, otherwise return `nil`.

`(stack-push stack element)`
: Push *element* onto *stack*.

`(stack-pop stack)`
: Remove the topmost element from *stack* and return it. If *stack* is empty, return `nil`.

`(stack-empty stack)`
: Return `t` if *stack* is empty, otherwise return `nil`.

`(stack-top stack)`
: Return the top element of *stack*, but don't remove it from the stack. Return `nil` if *stack* is empty.

`(stack-nth stack n)`
: Return the *n*th element of *stack* where the top stack element has number 0. If *stack* is not that long, return `nil`. The element is not removed from the stack.

`(stack-all stack)`
: Return a list of all entries in *stack* with the topmost element first.

`(stack-copy stack)`
: Return a copy of *stack*. All entries in *stack* are also copied.

`(stack-length stack)`
: Return the number of elements in *stack*.

`(stack-clear stack)`
: Remove all elements from *stack*.

3.2 The Queue Data Type

The queue data type provides a simple FIFO queue. There are two implementations of a queue in Elib, one using macros and one using functions. The names of the functions/macros in the two implementations are the same, but the efficiency of using one or the other vary greatly under different circumstances.

The implementation using macros should be used when you want to byte-compile your own elisp program. This will be most efficient since byte-compiling an elisp function using macros has the same effect as using inline code in `C`.

To use the queue data type, put the line

```
(require 'queue-f)
```

in your own elisp source file if you want to use the implementation using functions or

```
(require 'queue-m)
```

if you want to use the implementation using macros. This is the only difference between them, so it is easy to switch between them during debugging.

Not all functions in `queue-m.el` are implemented as macros, only the short ones. This does not make it less recommendable to use the macro version in your compiled code.

The following functions are provided by the queue:

`(queue-create)`
: Create a new empty queue.

`(queue-p queue)`
: Return `t` if *queue* is a queue, otherwise return `nil`.

`(queue-enqueue queue element)`
: Enter *element* last into *queue*.

`(queue-dequeue queue)`
: Remove the first element from *queue* and return it.

`(queue-empty queue)`
: Return `t` if *queue* is empty, otherwise return `nil`.

`(queue-first queue)`
: Return the first element of *queue* or `nil` if it is empty. The element is not removed from the queue.

`(queue-nth queue n)`
: Return the *n*th element of *queue*, where the first element of *queue* has number 0. If the length of *queue* is less than *n*, return `nil`. The element is not removed from the queue.

`(queue-last queue)`
: Return the last element of *queue* or `nil` if it is empty. The element is not removed from the queue.

`(queue-all queue)`
: Return a list of all elements in *queue*. Return `nil` if *queue* is empty. The oldest element in the queue is the first in the list.

`(queue-copy queue)`
: Return a copy of *queue*. All entries in *queue* are also copied.

`(queue-length queue)`
: Return the number of elements in *queue*.

`(queue-clear queue)`
: Remove all elements from *queue*.

3.3 The Doubly Linked List Data Type

The doubly linked list is an efficient data structure if you need to traverse the elements on the list in two directions, and maybe insert new elements in the middle of the list. You can efficiently delete any element, and insert new elements, anywhere on the list.

A doubly linked list (*dll* for short) consists of a number of *nodes*, each containing exactly one *element*. Some of the functions operate directly on the elements, while some manipulate nodes. For instance, all of the functions that let you step forward and backwards in the list handle nodes. Use the function *dll-element* to extract the element of a node.

To use the doubly linked list provided by Elib you must put the line

```
(require 'dll)
```

in your elisp source file.

3.3.1 Creating a Doubly Linked List

`(dll-create)`
: Create an empty doubly linked list.

`(dll-create-from-list list)`
: Given the ordinary lisp list *list*, create a doubly linked list with the same elements.

`(dll-copy dll &optional element-copy-fnc)`
: Return a copy of the doubly linked list *dll*. If optional second argument *element-copy-fnc* is non-`nil` it should be a function that takes one argument, an element, and returns a copy of it. If *element-copy-fnc* is not given the elements themselves are not copied.

3.3.2 Entering elements in a dll

`(dll-enter-first dll element)`
: Add an element first on a doubly linked list.

`(dll-enter-last dll element)`
: Add an element last on a doubly linked list.

`(dll-enter-after dll node element)`
: In the doubly linked list *dll*, insert a node containing *element* after *node*.

`(dll-enter-before dll node element)`
: In the doubly linked list *dll*, insert a node containing *element* before *node*.

3.3.3 Accessing elements of a dll

`(dll-element dll node)`
: Get the element of a *node* in a doubly linked list *dll*.

`(dll-first dll)`
: Return the first element on the doubly linked list *dll*. Return `nil` if the list is empty. The element is not removed.

`(dll-nth dll n)`
: Return the *n*th node from the doubly linked list *dll*. *n* counts from zero. If *dll* is not that long, `nil` is returned. If *n* is negative, return the -(*n*+1)th last element. Thus, `(dll-nth dll 0)` returns the first node, and `(dll-nth dll -1)` returns the last node.

`(dll-last dll)`
: Return the last element on the doubly linked list *dll*. Return `nil` if the list is empty. The element is not removed.

`(dll-next dll node)`
: Return the last element on the doubly linked list *dll*. Return `nil` if the list is empty. The element is not removed.

`(dll-previous dll node)`
: Return the node before *node*, or `nil` if *node* is the first node.

`(dll-all dll)`
: Return all elements on the double linked list *dll* as an ordinary list.

3.3.4 Removing nodes from a dll

`(dll-delete dll node)`
: Delete *node* from the doubly linked list *dll*. Return the element of *node*.

`(dll-delete-first dll)`
: Delete the first *node* from the doubly linked list *dll*. Return the element. Returns `nil` if *dll* was empty.

`(dll-delete-last dll)`
: Delete the last *node* from the doubly linked list *dll*. Return the element. Returns `nil` if *dll* was empty.

`(dll-clear dll)`
: Clear the doubly linked list *dll*, i.e. make it completely empty.

3.3.5 Predicates on a dll

`(dll-p object)`
: Return `t` if *object* is a doubly linked list, otherwise return `nil`.

`(dll-empty dll)`
: Return `t` if the doubly linked list *dll* is empty, `nil` otherwise.

3.3.6 Maps and Filters on a dll

`(dll-map map-function dll)`
: Apply *map-function* to all elements in the doubly linked list *dll*. The function is applied to the first element first.

`(dll-map-reverse map-function dll)`
: Apply *map-function* to all elements in the doubly linked list *dll*. The function is applied to the last element first.

`(dll-filter dll predicate)`
: Remove all elements in the doubly linked list *dll* for which *predicate* returns `nil`.

3.3.7 Miscellaneous dll operations

`(dll-length dll)`
: Returns the number of elements in the doubly linked list *dll*.

`(dll-sort dll predicate)`
: Sort the doubly linked list *dll*, stably, comparing elements using *predicate*. Returns the sorted list. *dll* is modified by side effects. *predicate* is called with two elements of *dll*, and should return `t` if the first element is “less” than the second.

3.3.8 Debugging dll applications

The data structure used by the dll package contains both forward and backward pointers. The primitives in Emacs, such as `print`, know nothing about dlls, so when Emacs tries to print out a dll it will think that it found a circular structure. Fortunately it detects this situation and gives an error message, instead of getting stuck in an eternal loop.

The error message can be quite annoying when you are developing an application that uses dlls. Suppose your code has an error, and you type ‘`(setq debug-on-error t)`’ to try to figure out exactly what the error is. If any function in the backtrace has a dll as an argument, Emacs will abort printing the entire backtrace and only respond with a "Back at top level" message (or something similar, depending on exactly what you are doing) in the echo area.

There are two solutions to this problem: patch your emacs so that it detects circular structures (there have been patches for this floating around the net) or use `dll-debug.el`.

The file `dll-debug.el` implements all of the functionality that are present in `dll.el`, but it uses a normal, singly linked list instead. This makes some operations, like ‘`dll-previous`’, dreadfully slow, but it makes it possible to debug dll applications. `dll-debug.el` also has more built-in sanity tests than `dll.el`.

NOTE: To use the debug package, you must load the library `dll-debug` before you load any of the libraries (such as cookie) or your program that use dll. You must also make sure that you don't load any byte-compiled version of any file that was compiled with the normal dll library. Since it contains some macros very strange results will occur otherwise...

When the debug package is loaded, you simply run your code normally, and any bugs should be easier to trace.

3.4 The Binary Tree Data Type

The binary tree is sometimes an efficient way to store data. When a binary tree is created a compare function is given to the create function (`bintree-create`). This function is used throughout all data entry and deletions into and out of the tree.

To use the binary tree in Elib you must put the line

```
(require 'bintree)
```

in your elisp source file.

The following functions are provided by the binary tree in the library:

`(bintree-create compare-function)`
: Create a new empty binary tree. The argument *compare-function* is a function which compares two instances of the data type which is to be entered into the tree. The call `(compare-function data1 data2)` should return non-`nil` if `data1` is less than `data2`, and `nil` otherwise.

`(bintree-p tree)`
: Return `t` if *tree* is an bintree, otherwise return `nil`.

`(bintree-compare-function tree)`
: Return `compare-function` given to `bintree-create` when *tree* was created.

`(bintree-empty tree)`
: Return `t` if *tree* is empty, otherwise return `nil`.

`(bintree-enter tree data)`
: Enter *data* into *tree*. If there already is a data element which is considered equal to *data* by `compare-function` given to `bintree-create`, the new element will replace the old one in the tree.

`(bintree-delete tree data)`
: Delete the element which is considered equal to *data* by `compare-function` given to `bintree-create`. If there is no matching element within the tree, nothing is done to the tree.

`(bintree-member tree data)`
: Return the element in *tree* which is considered equal to *data* by `compare-function` given to `bintree-create`. If there is no such element in the tree, return `nil`.

`(bintree-map map-function tree)`
: Apply *map-function* to all elements in *tree*. The function is applied in the order in which the tree is sorted.

`(bintree-first tree)`
: Return the first element of *tree*, i.e. the one who is considered first by `compare-function` given to `bintree-create`. If the tree is empty, return `nil`.

`(bintree-last tree)`
: Return the last element of *tree*, i.e. the one who is considered last by `compare-function` given to `bintree-create`. If the tree is empty, return `nil`.

`(bintree-copy tree)`
: Return a copy of *tree*.

`(bintree-flatten tree)`
: Return a sorted list containing all elements of *tree*.

`(bintree-size tree)`
: Return the number of elements in *tree*.

`(bintree-clear tree)`
: Clear *tree*, i.e. make it totally empty.

3.5 The AVL Tree Data Type

The AVL tree data types provides a balanced binary tree. The tree will remain balanced throughout its entire life time, regardless of in which order elements are entered into or deleted from the tree.

Although an AVL tree is not perfectly balanced, it has almost the same performance as if it was. The definition of an AVL tree is that the difference in depth of the two branches of a particular node is at most 1. This criterium is enough to make the performance of searching in an AVL tree very close to a perfectly balanced tree, but will simplify the entering and deleting of data significantly.

All data that is entered into an AVL tree should be of the same type. If they are not, there are no way to compare two elements and this is essential for entering and deleting data from the tree. When a tree is created, a compare function is given to the create function. This function is used throughout the life of the tree in all subsequent insertions and deletions.

To use the Elib AVL tree, you must put the line

```
(require 'avltree)
```

in your elisp source file.

The following functions are provided by the AVL tree in the library:

`(avltree-create compare-function)`
: Create a new empty AVL tree. The argument *compare-function* is a function which compares two instances of the data type which is to be entered into the tree. The call `(compare-function data1 data2)` should return non-`nil` if `data1` is less than `data2`, and `nil` otherwise.

`(avltree-p tree)`
: Return `t` if *tree* is an avltree, otherwise return `nil`.

`(avltree-compare-function tree)`
: Return `compare-function` given to `avltree-create` when *tree* was created.

`(avltree-empty tree)`
: Return `t` if *tree* is empty, otherwise return `nil`.

`(avltree-enter tree data)`
: Enter *data* into *tree*. If there already is a data element which is considered equal to *data* by `compare-function` given to `avltree-create`, the new element will replace the old one in the tree.

`(avltree-delete tree data)`
: Delete the element which is considered equal to *data* by `compare-function` given to `avltree-create`. If there is no matching element within the tree, nothing is done to the tree.

`(avltree-member tree data)`
: Return the element in *tree* which is considered equal to *data* by `compare-function` given to `avltree-create`. If there is no such element in the tree, return `nil`.

`(avltree-map map-function tree)`
: Apply *map-function* to all elements in *tree*. The function is applied in the order in which the tree is sorted.

`(avltree-first tree)`
: Return the first element of *tree*, i.e. the one who is considered first by `compare-function` given to `avltree-create`. If the tree is empty, return `nil`.

`(avltree-last tree)`
: Return the last element of *tree*, i.e. the one who is considered last by `compare-function` given to `avltree-create`. If the tree is empty, return `nil`.

`(avltree-copy tree)`
: Return a copy of *tree*.

`(avltree-flatten tree)`
: Return a sorted list containing all elements of *tree*.

`(avltree-size tree)`
: Return the number of elements in *tree*.

`(avltree-clear tree)`
: Clear *tree*, i.e. make it totally empty.

4 The Cookie package—nodal data in a buffer

If you want to have structured nodal data in a buffer, the cookie package can be a help to you.

Cookie is a package that implements a connection between a dll (a doubly linked list) and the contents of a buffer. Possible uses are `dired` (have all files in a list, and show them), `buffer-list`, `kom-prioritize` (in the LysKOM elisp client) and others. `pcl-cvs.el` uses `cookie.el`.

4.1 Introduction to cookie terminology

The cookie package uses its own terminology. Here are some important definitions.

cookie
: A *cookie* can be any lisp object. When you use the cookie package you specify a pretty-printer, a function that inserts a printable representation of the cookie in the buffer.

collection
: A *collection* consists of a doubly linked list of cookies, a header, a footer and a pretty-printer. It is displayed at a certain point in a certain buffer. (The buffer and point are selected when the collection is created). The header and the footer are constant strings. They appear before and after the cookies. (Currently, once set, they can not be changed).

tin
: A *tin* is an object that contains one cookie. There are functions in this package that given a tin extracts the cookie, or gives the next or previous tin. (All tins are linked together in a doubly linked list. The previous tin is the one that appears before the other in the buffer.) You should not do anything with a tin except pass it to the functions in this package.

Cookie does not affect the mode of the buffer in any way. It merely makes it easy to connect an underlying data representation to the buffer contents.

A collection is a very dynamic thing. You can easily add or delete cookies. You can sort all cookies in a collection (you have to supply a function that compares two cookies). You can apply a function to all cookies in a collection, et c, et c.

Remember that a cookie can be anything. Your imagination is the limit! It is even possible to have another collection as a cookie. In that way some kind of tree hierarchy can be created.

4.2 Coding conventions used in the cookie package

All functions that are intended for external use begin with one of the prefixes '`cookie-`', '`collection-`' or '`tin-`'. The prefix '`icookie-`' is currently used for internal functions and macros. Currently, no global or buffer-local variables are used.

Many functions operate on tins instead of cookies. For most functions, the prefix used should help tell which kind of object the function uses.

Most doc-strings contains an `"Args:"` line that lists the arguments.

4.3 Manipulating the entire collection

`(collection-create buffer pretty-printer &optional header footer pos)`
: Create a collection that is displayed in *buffer*. *buffer* may be a buffer or a buffer name. It is created if it does not exist.

  pretty-printer should be a function that takes one argument, a cookie, and inserts a string representing it in the buffer (at point). The string *pretty-printer* inserts may be empty or span several lines. A trailing newline will always be inserted automatically. The *pretty-printer* should use `insert`, and not `insert-before-markers`.

  Optional third argument *header* is a string that will always be present at the top of the collection. *header* should end with a newline. Optionaly fourth argument *footer* is similar, and will always be inserted at the bottom of the collection.

  Optional fifth argument *pos* is a buffer position, specifying where the collection will be inserted. It defaults to the begining of the buffer. *pos* will probably default to the current value of `(point)` in future releases of Elib, so you should not depend on this default in cases where it matters.

`(collection-empty collection)`
: Return true if there are no cookies in *collection*.

`(collection-length collection)`
: Return the number of cookies in *collection*.

`(collection-list-cookies collection)`
: Return a list of all cookies in *collection*.

4.4 Inserting cookies in the collection

These functions can be used to insert one or more cookies into a collection. The printed representation will immediately and automatically be updated by the cookie package. (It will call the pretty-printer that was specified to `collection-create`).

`(cookie-enter-first collection cookie)`
: Enter *cookie* first in the cookie collection *collection*.

`(cookie-enter-last collection cookie)`
: Enter *cookie* last in the cookie collection *collection*.

`(cookie-enter-after-tin collection tin cookie)`
: Enter *cookie* into *collection*, immediately after *tin*.

`(cookie-enter-before-tin collection tin cookie)`
: Enter *cookie* into *collection*, immediately before *tin*.

`(collection-append-cookies (collection cookie-list))`
: Insert all cookies in the list *cookie-list* last in *collection*.

4.5 Tins and cookies

`(tin-cookie collection tin)`

This function can be used to extract a cookie from *tin*. The collection that *tin* is present in must also be specified as *collection*.

4.6 Deleting cookies

There are a couple of different ways to delete cookies from the collection.

`(tin-delete collection tin)`

Delete *tin* from *collection*. The cookie that is stored in *tin* is returned.

`(cookie-delete-first collection)`

Delete first cookie in *collection* and return it. Returns `nil` if there are no cookies left in *collection*.

`(cookie-delete-last collection)`

Delete last cookie in *collection* and return it. Returns `nil` if there are no cookies left in *collection*.

The following two functions can be used to delete several cookies that fulfills certain criteria.

`(collection-filter-cookies collection predicate &rest extra-args)`

Remove all cookies in *collection* for which *predicate* returns nil. Note that the buffer for *collection* will be current-buffer when *predicate* is called. *predicate* must restore the current buffer before it returns if it changes it.

The *predicate* is called with *cookie* as its first argument. If any *extra-args* are given to `collection-filter-cookies` they will be passed unmodified to *predicate*.

`(collection-filter-tins collection predicate &rest extra-args)`

This is like `collection-filter-cookies`, but *predicate* is called with a tin instead of a cookie.

And finally, a way to delete all cookies in one swift function call:

`(collection-clear collection)`

Remove all cookies in *collection*.

4.7 Collection as a Doubly linked list

The functions in this section treat the collection as a doubly linked list.

`(tin-nth collection n)`

Return the *n*th tin. *n* counts from zero. `nil` is returned if there is less than *n* cookies. If *n* is negative, return the -(*n*+1)th last element. Thus, `(tin-nth dll 0)` returns the first node, and `(tin-nth dll -1)` returns the last node.

Use `tin-cookie` to extract the cookie from the tin (or use `cookie-nth` instead).

`(cookie-nth collection n)`

Like `tin-nth`, but the cookie is returned instead of the tin.

`(tin-next collection tin)`
: Get the next tin. Returns nil if *tin* is `nil` or refers to the last cookie in *collection*.

`(tin-previous collection tin)`
: Get the previous tin. Returns nil if *tin* is `nil` or refers to the first cookie in *collection*.

`(cookie-sort collection predicate)`
: Sort the cookies in *collection*, stably, comparing elements using *predicate*. *predicate* is called with two cookies, and should return '`t`' if the first cookie is *less* than the second.

  All cookies will be refreshed when the sort is complete.

`(cookie-first collection)`
: Return the first cookie in *collection*. The cookie is not removed.

`(cookie-last collection)`
: Return the last cookie in *collection*. The cookie is not removed.

4.8 Scanning the list

`(cookie-map map-function collection &rest map-args)`
: Apply *map-function* to all cookies in *collection*. *map-function* is applied to the first element first. If *map-function* returns non-`nil` the cookie will be refreshed (its pretty-printer will be called once again).

  Note that the buffer for *collection* will be current buffer when *map-function* is called. *map-function* must restore the current buffer to *buffer* before it returns, if it changes it.

  If more than two arguments are given to `cookie-map`, remaining arguments will be passed to *map-function*.

`(cookie-map-reverse map-function collection &rest map-args)`
: Like `cookie-map`, but *map-function* will be applied to the last cookie first.

`(collection-collect-tin collection predicate &rest predicate-args)`
: Select cookies from *collection* using *predicate*. Return a list of all selected tins.

  predicate is a function that takes a cookie as its first argument.

  The tins on the returned list will appear in the same order as in the buffer. You should not rely on in which order *predicate* is called.

  Note that the buffer the *collection* is displayed in is current-buffer when *predicate* is called. *predicate* must restore current-buffer if it changes it.

  If more than two arguments are given to `collection-collect-tin` the remaining arguments will be passed to *predicate*.

`(collection-collect-cookie collection predicate &rest predicate-args)`
: Like `collection-collect-tin`, but a list of cookies is returned.

4.9 Operations that affect the buffer

`(collection-buffer collection)`
: Return the buffer that *collection* is displayed in.

`(collection-refresh collection)`
: Refresh all cookies in *collection*.

  The pretty-printer that was specified when the *collection* was created will be called for all cookies in *collection*.

  Note that `tin-invalidate` is more efficient if only a small number of cookies needs to be refreshed.

`(tin-invalidate collection &rest tins)`
: Refresh some cookies. The pretty-printer for *collection* will be called for all *tins*.

`(collection-set-goal-column collection goal)`
: Set goal-column for *collection*. goal-column is made buffer-local. This function will be obsoleted in the next release of Elib. Instead, there is going to be a function that given a cookie will return a position where the cursor should be stored. The details are not yet decided.

`(tin-goto-previous collection pos arg)`
: Move point to the *arg*th previous cookie. Don't move if we are at the first cookie, or if *collection* is empty. Returns the tin we move to.

`(tin-goto-next collection pos arg)`
: Like `tin-goto-previous`, but move towards the end of the buffer instead.

`(tin-goto collection tin)`
: Move point to *tin*.

`(tin-locate collection pos &optional guess)`
: Return the tin that *pos* (a buffer position) is within.

  pos may be a marker or an integer. *guess* should be a tin that it is likely that *pos* is near.

  If *pos* points before the first cookie, the first cookie is returned. If *pos* points after the last cookie, the last cookie is returned. If *collection* is empty, `nil` is returned.

4.10 Debugging cookie applications

Since the cookie package uses dll, cookie applications can be hard to debug. Fortunately, the same technique can be used here—just load dll-debug prior to loading cookie. See Section 3.3.8 [Debugging dll applications], page 13.

Warning! Don't load a byte-compiled `cookie.elc` that was compiled using dll (as opposed to dll-debug) when you have dll-debug in memory. Your Emacs will be seriously confused.

5 String functions

To use the string functions in Elib you have to put the following line into your elisp source file:

```
(require 'string)
```

The following string functions are provided with Elib.

`(string-replace-match regexp string newtext &optional literal global)`

This function tries to be a string near-equivalent to the elisp function `replace-match`. It returns a string with the first text matched by *regexp* in *string* replaced by *newtext*. If no match is found, `nil` is returned. If optional argument *global* is non-`nil`, all occurances matching *regexp* are replaced instead of only the first one.

If optional argument *literal* is non-`nil`, then *newtext* is inserted exactly as it is. If it is `nil` (which is the default), then the character `\` is treated specially. If a `\` appears in *newtext*, it can start any one of the following sequences:

`\&`	`\&` stands for the entire text being replaced.
`\n`	`\n` stands for the *n*th subexpression in the original regexp. Subexpressions are those expressions grouped inside of `\(...\)`. *n* is a digit.
`\\`	`\\` stands for a single `\` in *newtext*.

Any other character after the `\` will just be copied into the string.

`(string-split pattern string &optional limit)`

Split the string *string* on the regexp *pattern* and return a list of the strings between the matches. If the optional numerical argument *limit* is >= 1, only the first *limit* elements of the list are returned.

For example, the call

```
(string-split "[ \t]+" "Elisp programming is fun.")
```

will return `("Elisp" "programming" "is" "fun.")`, but the call

```
(string-split " " "Elisp programming is fun." 3)
```

will return `("Elisp" "programming" "is")`.

6 Read functions

Elib provides a number of functions for reading data from the minibuffer. To use them in your own elisp programs, put the following line into you source file:

```
(require 'read)
```

The following functions are provided by `read`.

`(read-number &optional prompt default)`
: Read a number from the minibuffer. If optional argument *prompt* is non-`nil`, the user is prompted using *prompt*, otherwise the prompt string `Enter a number:` is used. If optional argument *default* is non-`nil`, it is written within parenthesis after the prompt string. *default* can be either a number or of the type which `(interactive "P")` generates.

`(read-num-range low high &optional prompt show-range)`
: Read a number from the minibuffer. The number returned will be forced to lie between *low* and *high*. If *prompt* is non-`nil`, the user is prompted using *prompt*, otherwise the prompt string `Enter a number:` is used. If *show-range* is non-`nil`, the prompt will show the range within parenthesis to the user.

`(read-silent prompt &optional showchar)`
: Read a string in the minibuffer without echoing. The following characters are special when entering the string:

`DEL`	Delete the last character in the input buffer.
`C-u`	Clear the input buffer.
`RET`	End the reading of the string.
`Newline`	Same as *`RET`*.

If optional argument *showchar* is non-`nil`, one of these characters will be displayed for each character input by the user.

This function is well suited to read a password from the user, but beware of the function `(view-lossage)` which displays the last 100 keystrokes, even hidden ones.

7 Future enhancements

Elib is still under development and needs a number of enhancements to be called fairly complete. Here is a list of wishes of functions and data types which we would like to enter into Elib in future releases:

- Numerical data types such as Floating point numbers, Complex numbers, Arbitrarily long integers, etc. (Perhaps we can get these from the freely distributable elisp package `calc`.)
- Other container data types such as Priority queues, 2-3-trees, Hash tables, Sets, etc.
- Other implementations of old container data types. For instance, are vector implementations of stacks and queues faster than the current ones using cons cells?
- Miscellaneous other small functions.

7.1 Contributions

We are grateful for all donations of code that we can receive. However, your code will be still more useful if you also provide documentation and code to test your new library functions.

8 Reporting bugs

This is an early test release of the GNU emacs lisp library. Undoubtedly there are numerous bugs remaining, both in the elisp source code and in the documentation. If you find a bug in either, please send a bug report to `elib-maintainers@lysator.liu.se`. We will try to be as quick as possible in fixing the bugs and redistributing the fixes.

Table of Contents