Common Lisp Extensions

For GNU Emacs Lisp

Version 2.02

Dave Gillespie
daveg@synaptics.com

1.1 Overview

Common Lisp is a huge language, and Common Lisp systems tend to be massive and extremely complex. Emacs Lisp, by contrast, is rather minimalist in the choice of Lisp features it offers the programmer. As Emacs Lisp programmers have grown in number, and the applications they write have grown more ambitious, it has become clear that Emacs Lisp could benefit from many of the conveniences of Common Lisp.

The *CL* package adds a number of Common Lisp functions and control structures to Emacs Lisp. While not a 100% complete implementation of Common Lisp, *CL* adds enough functionality to make Emacs Lisp programming significantly more convenient.

Some Common Lisp features have been omitted from this package for various reasons:

- Some features are too complex or bulky relative to their benefit to Emacs Lisp programmers. CLOS and Common Lisp streams are fine examples of this group.
- Other features cannot be implemented without modification to the Emacs Lisp interpreter itself, such as multiple return values, lexical scoping, case-insensitive symbols, and complex numbers. The *CL* package generally makes no attempt to emulate these features.
- Some features conflict with existing things in Emacs Lisp. For example, Emacs' `assoc` function is incompatible with the Common Lisp `assoc`. In such cases, this package usually adds the suffix '`*`' to the function name of the Common Lisp version of the function (e.g., `assoc*`).

The package described here was written by Dave Gillespie, `daveg@synaptics.com`. It is a total rewrite of the original 1986 `cl.el` package by Cesar Quiroz. Most features of the the Quiroz package have been retained; any incompatibilities are noted in the descriptions below. Care has been taken in this version to ensure that each function is defined efficiently, concisely, and with minimal impact on the rest of the Emacs environment.

1.2 Usage

Lisp code that uses features from the *CL* package should include at the beginning:

```
(require 'cl)
```

If you want to ensure that the new (Gillespie) version of *CL* is the one that is present, add an additional `(require 'cl-19)` call:

```
(require 'cl)
(require 'cl-19)
```

The second call will fail (with "`cl-19.el` not found") if the old `cl.el` package was in use.

It is safe to arrange to load *CL* at all times, e.g., in your `.emacs` file. But it's a good idea, for portability, to `(require 'cl)` in your code even if you do this.

1.3 Organization

The Common Lisp package is organized into four files:

`cl.el`
: This is the "main" file, which contains basic functions and information about the package. This file is relatively compact—about 700 lines.

`cl-extra.el`

This file contains the larger, more complex or unusual functions. It is kept separate so that packages which only want to use Common Lisp fundamentals like the `cadr` function won't need to pay the overhead of loading the more advanced functions.

`cl-seq.el`

This file contains most of the advanced functions for operating on sequences or lists, such as `delete-if` and `assoc*`.

`cl-macs.el`

This file contains the features of the packages which are macros instead of functions. Macros expand when the caller is compiled, not when it is run, so the macros generally only need to be present when the byte-compiler is running (or when the macros are used in uncompiled code such as a `.emacs` file). Most of the macros of this package are isolated in `cl-macs.el` so that they won't take up memory unless you are compiling.

The file `cl.el` includes all necessary `autoload` commands for the functions and macros in the other three files. All you have to do is `(require 'cl)`, and `cl.el` will take care of pulling in the other files when they are needed.

There is another file, `cl-compat.el`, which defines some routines from the older `cl.el` package that are no longer present in the new package. This includes internal routines like `setelt` and `zip-lists`, deprecated features like `defkeyword`, and an emulation of the old-style multiple-values feature. See Appendix C [Old CL Compatibility], page 76.

1.4 Installation

Installation of the *CL* package is simple: Just put the byte-compiled files `cl.elc`, `cl-extra.elc`, `cl-seq.elc`, `cl-macs.elc`, and `cl-compat.elc` into a directory on your `load-path`.

There are no special requirements to compile this package: The files do not have to be loaded before they are compiled, nor do they need to be compiled in any particular order.

You may choose to put the files into your main `lisp/` directory, replacing the original `cl.el` file there. Or, you could put them into a directory that comes before `lisp/` on your `load-path` so that the old `cl.el` is effectively hidden.

Also, format the `cl.texinfo` file and put the resulting Info files in the `info/` directory or another suitable place.

You may instead wish to leave this package's components all in their own directory, and then add this directory to your `load-path` and (Emacs 19 only) `Info-directory-list`. Add the directory to the front of the list so the old *CL* package and its documentation are hidden.

1.5 Naming Conventions

Except where noted, all functions defined by this package have the same names and calling conventions as their Common Lisp counterparts.

Following is a complete list of functions whose names were changed from Common Lisp, usually to avoid conflicts with Emacs. In each case, a '*' has been appended to the Common Lisp name to obtain the Emacs name:

`defun*`	`defsubst*`	`defmacro*`	`function*`
`member*`	`assoc*`	`rassoc*`	`get*`
`remove*`	`delete*`	`mapcar*`	`sort*`
`floor*`	`ceiling*`	`truncate*`	`round*`
`mod*`	`rem*`	`random*`	

Internal function and variable names in the package are prefixed by `cl-`. Here is a complete list of functions *not* prefixed by `cl-` which were not taken from Common Lisp:

`member`	`delete`	`remove`	`remq`
`rassoc`	`floatp-safe`	`lexical-let`	`lexical-let*`
`callf`	`callf2`	`letf`	`letf*`
`defsubst*`	`defalias`	`add-hook`	`eval-when-compile`

(Most of these are Emacs 19 features provided to Emacs 18 users, or introduced, like `remq`, for reasons of symmetry with similar features.)

The following simple functions and macros are defined in `cl.el`; they do not cause other components like `cl-extra` to be loaded.

`eql`	`floatp-safe`	`abs`	`endp`
`evenp`	`oddp`	`plusp`	`minusp`
`last`	`butlast`	`nbutlast`	`caar .. cddddr`
`list*`	`ldiff`	`rest`	`first .. tenth`
`member [1]`	`copy-list`	`subst`	`mapcar* [2]`
`adjoin [3]`	`acons`	`pairlis`	`when`
`unless`	`pop [4]`	`push [4]`	`pushnew [3,4]`
`incf [4]`	`decf [4]`	`proclaim`	`declaim`
`add-hook`			

[1] This is the Emacs 19-compatible function, not `member*`.

[2] Only for one sequence argument or two list arguments.

[3] Only if `:test` is `eq`, `equal`, or unspecified, and `:key` is not used.

[4] Only when *place* is a plain variable name.

5 Program Structure

This section describes features of the *CL* package which have to do with programs as a whole: advanced argument lists for functions, and the `eval-when` construct.

5.2 Argument Lists

Emacs Lisp's notation for argument lists of functions is a subset of the Common Lisp notation. As well as the familiar `&optional` and `&rest` markers, Common Lisp allows you to specify default values for optional arguments, and it provides the additional markers `&key` and `&aux`.

Since argument parsing is built-in to Emacs, there is no way for this package to implement Common Lisp argument lists seamlessly. Instead, this package defines alternates for several Lisp forms which you must use if you need Common Lisp argument lists.

`defun*` *name arglist body...* [Special Form]

> This form is identical to the regular `defun` form, except that *arglist* is allowed to be a full Common Lisp argument list. Also, the function body is enclosed in an implicit block called *name*; see Section 7.7 [Blocks and Exits], page 26.

`defsubst*` *name arglist body...* [Special Form]

> This is just like `defun*`, except that the function that is defined is automatically proclaimed `inline`, i.e., calls to it may be expanded into in-line code by the byte compiler. This is analogous to the `defsubst` form in Emacs 19; `defsubst*` uses a different method (compiler macros) which works in all version of Emacs, and also generates somewhat more efficient inline expansions. In particular, `defsubst*` arranges for the processing of keyword arguments, default values, etc., to be done at compile-time whenever possible.

`defmacro*` *name arglist body...* [Special Form]

> This is identical to the regular `defmacro` form, except that *arglist* is allowed to be a full Common Lisp argument list. The `&environment` keyword is supported as described in Steele. The `&whole` keyword is supported only within destructured lists (see below); top-level `&whole` cannot be implemented with the current Emacs Lisp interpreter. The macro expander body is enclosed in an implicit block called *name*.

`function*` *symbol-or-lambda* [Special Form]

> This is identical to the regular `function` form, except that if the argument is a `lambda` form then that form may use a full Common Lisp argument list.

Also, all forms (such as `defsetf` and `flet`) defined in this package that include *arglist*s in their syntax allow full Common Lisp argument lists.

Note that it is *not* necessary to use `defun*` in order to have access to most *CL* features in your function. These features are always present; `defun*`'s only difference from `defun` is its more flexible argument lists and its implicit block.

The full form of a Common Lisp argument list is

```
(var...
 &optional (var initform svar)...
```

```
&rest var
&key ((keyword var) initform svar)...
&aux (var initform)...)
```

Each of the five argument list sections is optional. The *svar*, *initform*, and *keyword* parts are optional; if they are omitted, then '`(var)`' may be written simply '`var`'.

The first section consists of zero or more *required* arguments. These arguments must always be specified in a call to the function; there is no difference between Emacs Lisp and Common Lisp as far as required arguments are concerned.

The second section consists of *optional* arguments. These arguments may be specified in the function call; if they are not, *initform* specifies the default value used for the argument. (No *initform* means to use `nil` as the default.) The *initform* is evaluated with the bindings for the preceding arguments already established; `(a &optional (b (1+ a)))` matches one or two arguments, with the second argument defaulting to one plus the first argument. If the *svar* is specified, it is an auxiliary variable which is bound to `t` if the optional argument was specified, or to `nil` if the argument was omitted. If you don't use an *svar*, then there will be no way for your function to tell whether it was called with no argument, or with the default value passed explicitly as an argument.

The third section consists of a single *rest* argument. If more arguments were passed to the function than are accounted for by the required and optional arguments, those extra arguments are collected into a list and bound to the "rest" argument variable. Common Lisp's `&rest` is equivalent to that of Emacs Lisp. Common Lisp accepts `&body` as a synonym for `&rest` in macro contexts; this package accepts it all the time.

The fourth section consists of *keyword* arguments. These are optional arguments which are specified by name rather than positionally in the argument list. For example,

```
(defun* foo (a &optional b &key c d (e 17)))
```

defines a function which may be called with one, two, or more arguments. The first two arguments are bound to `a` and `b` in the usual way. The remaining arguments must be pairs of the form `:c`, `:d`, or `:e` followed by the value to be bound to the corresponding argument variable. (Symbols whose names begin with a colon are called *keywords*, and they are self-quoting in the same way as `nil` and `t`.)

For example, the call `(foo 1 2 :d 3 :c 4)` sets the five arguments to 1, 2, 4, 3, and 17, respectively. If the same keyword appears more than once in the function call, the first occurrence takes precedence over the later ones. Note that it is not possible to specify keyword arguments without specifying the optional argument `b` as well, since `(foo 1 :c 2)` would bind `b` to the keyword `:c`, then signal an error because `2` is not a valid keyword.

If a *keyword* symbol is explicitly specified in the argument list as shown in the above diagram, then that keyword will be used instead of just the variable name prefixed with a colon. You can specify a *keyword* symbol which does not begin with a colon at all, but such symbols will not be self-quoting; you will have to quote them explicitly with an apostrophe in the function call.

Ordinarily it is an error to pass an unrecognized keyword to a function, e.g., `(foo 1 2 :c 3 :goober 4)`. You can ask Lisp to ignore unrecognized keywords, either by adding the marker `&allow-other-keys` after the keyword section of the argument list, or by specifying an `:allow-other-keys` argument in the call whose value is non-`nil`. If the function uses both `&rest` and `&key` at the same time, the "rest" argument is bound to the keyword list as it appears in the call. For example:

```
(defun* find-thing (thing &rest rest &key need &allow-other-keys)
  (or (apply 'member* thing thing-list :allow-other-keys t rest)
      (if need (error "Thing not found"))))
```

This function takes a `:need` keyword argument, but also accepts other keyword arguments which are passed on to the `member*` function. `allow-other-keys` is used to keep both `find-thing` and `member*` from complaining about each others' keywords in the arguments.

In Common Lisp, keywords are recognized by the Lisp parser itself and treated as special entities. In Emacs, keywords are just symbols whose names begin with colons, which `defun*` has arranged to set equal to themselves so that they will essentially be self-quoting.

As a (significant) performance optimization, this package implements the scan for keyword arguments by calling `memq` to search for keywords in a "rest" argument. Technically speaking, this is incorrect, since `memq` looks at the odd-numbered values as well as the even-numbered keywords. The net effect is that if you happen to pass a keyword symbol as the *value* of another keyword argument, where that keyword symbol happens to equal the name of a valid keyword argument of the same function, then the keyword parser will become confused. This minor bug can only affect you if you use keyword symbols as general-purpose data in your program; this practice is strongly discouraged in Emacs Lisp.

The fifth section of the argument list consists of *auxiliary variables*. These are not really arguments at all, but simply variables which are bound to `nil` or to the specified *initforms* during execution of the function. There is no difference between the following two functions, except for a matter of stylistic taste:

```
(defun* foo (a b &aux (c (+ a b)) d)
  body)

(defun* foo (a b)
  (let ((c (+ a b)) d)
    body))
```

Argument lists support *destructuring*. In Common Lisp, destructuring is only allowed with `defmacro`; this package allows it with `defun*` and other argument lists as well. In destructuring, any argument variable (*var* in the above diagram) can be replaced by a list of variables, or more generally, a recursive argument list. The corresponding argument value must be a list whose elements match this recursive argument list. For example:

```
(defmacro* dolist ((var listform &optional resultform)
                   &rest body)
  ...)
```

This says that the first argument of `dolist` must be a list of two or three items; if there are other arguments as well as this list, they are stored in `body`. All features allowed in regular argument lists are allowed in these recursive argument lists. In addition, the clause '`&whole` *var*' is allowed at the front of a recursive argument list. It binds *var* to the whole list being matched; thus `(&whole all a b)` matches a list of two things, with `a` bound to the first thing, `b` bound to the second thing, and `all` bound to the list itself. (Common Lisp allows `&whole` in top-level `defmacro` argument lists as well, but Emacs Lisp does not support this usage.)

One last feature of destructuring is that the argument list may be dotted, so that the argument list `(a b . c)` is functionally equivalent to `(a b &rest c)`.

If the optimization quality `safety` is set to 0 (see Chapter 9 [Declarations], page 41), error checking for wrong number of arguments and invalid keyword arguments is disabled. By default, argument lists are rigorously checked.

5.3 Time of Evaluation

Normally, the byte-compiler does not actually execute the forms in a file it compiles. For example, if a file contains `(setq foo t)`, the act of compiling it will not actually set `foo` to `t`. This is true even if the `setq` was a top-level form (i.e., not enclosed in a `defun` or other form). Sometimes, though, you would like to have certain top-level forms evaluated at compile-time. For example, the compiler effectively evaluates `defmacro` forms at compile-time so that later parts of the file can refer to the macros that are defined.

`eval-when` (*situations...*) *forms...* [Special Form]

This form controls when the body *forms* are evaluated. The *situations* list may contain any set of the symbols `compile`, `load`, and `eval` (or their long-winded ANSI equivalents, `:compile-toplevel`, `:load-toplevel`, and `:execute`).

The `eval-when` form is handled differently depending on whether or not it is being compiled as a top-level form. Specifically, it gets special treatment if it is being compiled by a command such as `byte-compile-file` which compiles files or buffers of code, and it appears either literally at the top level of the file or inside a top-level `progn`.

For compiled top-level `eval-when`s, the body *forms* are executed at compile-time if `compile` is in the *situations* list, and the *forms* are written out to the file (to be executed at load-time) if `load` is in the *situations* list.

For non-compiled-top-level forms, only the `eval` situation is relevant. (This includes forms executed by the interpreter, forms compiled with `byte-compile` rather than `byte-compile-file`, and non-top-level forms.) The `eval-when` acts like a `progn` if `eval` is specified, and like `nil` (ignoring the body *forms*) if not.

The rules become more subtle when `eval-when`s are nested; consult Steele (second edition) for the gruesome details (and some gruesome examples).

Some simple examples:

```
;; Top-level forms in foo.el:
(eval-when (compile)           (setq foo1 'bar))
(eval-when (load)              (setq foo2 'bar))
(eval-when (compile load)      (setq foo3 'bar))
(eval-when (eval)              (setq foo4 'bar))
(eval-when (eval compile)      (setq foo5 'bar))
(eval-when (eval load)         (setq foo6 'bar))
(eval-when (eval compile load) (setq foo7 'bar))
```

When `foo.el` is compiled, these variables will be set during the compilation itself:

```
foo1  foo3  foo5  foo7      ; 'compile'
```

When `foo.elc` is loaded, these variables will be set:

```
foo2  foo3  foo6  foo7      ; 'load'
```

And if `foo.el` is loaded uncompiled, these variables will be set:

```
foo4  foo5  foo6  foo7      ; 'eval'
```

If these seven `eval-when`s had been, say, inside a `defun`, then the first three would have been equivalent to `nil` and the last four would have been equivalent to the corresponding `setq`s.

Note that `(eval-when (load eval) ...)` is equivalent to `(progn ...)` in all contexts. The compiler treats certain top-level forms, like `defmacro` (sort-of) and `require`, as if they were wrapped in `(eval-when (compile load eval) ...)`.

Emacs 19 includes two special forms related to `eval-when`. One of these, `eval-when-compile`, is not quite equivalent to any `eval-when` construct and is described below. This package defines a version of `eval-when-compile` for the benefit of Emacs 18 users.

The other form, `(eval-and-compile ...)`, is exactly equivalent to '`(eval-when (compile load eval) ...)`' and so is not itself defined by this package.

`eval-when-compile` *forms...* [Special Form]

The *forms* are evaluated at compile-time; at execution time, this form acts like a quoted constant of the resulting value. Used at top-level, `eval-when-compile` is just like '`eval-when (compile eval)`'. In other contexts, `eval-when-compile` allows code to be evaluated once at compile-time for efficiency or other reasons.

This form is similar to the '`#.`' syntax of true Common Lisp.

`load-time-value` *form* [Special Form]

The *form* is evaluated at load-time; at execution time, this form acts like a quoted constant of the resulting value.

Early Common Lisp had a '`#,`' syntax that was similar to this, but ANSI Common Lisp replaced it with `load-time-value` and gave it more well-defined semantics.

In a compiled file, `load-time-value` arranges for *form* to be evaluated when the `.elc` file is loaded and then used as if it were a quoted constant. In code compiled by `byte-compile` rather than `byte-compile-file`, the effect is identical to `eval-when-compile`. In uncompiled code, both `eval-when-compile` and `load-time-value` act exactly like `progn`.

```
(defun report ()
  (insert "This function was executed on: "
          (current-time-string)
          ", compiled on: "
          (eval-when-compile (current-time-string))
          ;; or '#.(current-time-string) in real Common Lisp
          ", and loaded on: "
          (load-time-value (current-time-string))))
```

Byte-compiled, the above defun will result in the following code (or its compiled equivalent, of course) in the `.elc` file:

```
(setq --temp-- (current-time-string))
(defun report ()
  (insert "This function was executed on: "
          (current-time-string)
```

```
", compiled on: "
'"Wed Jun 23 18:33:43 1993"
", and loaded on: "
--temp--))
```

5.4 Function Aliases

This section describes a feature from GNU Emacs 19 which this package makes available in other versions of Emacs.

`defalias` *symbol function* [Function]

This function sets *symbol*'s function cell to *function*. It is equivalent to `fset`, except that in GNU Emacs 19 it also records the setting in `load-history` so that it can be undone by a later `unload-feature`.

In other versions of Emacs, `defalias` is a synonym for `fset`.

6 Predicates

This section describes functions for testing whether various facts are true or false.

6.1 Type Predicates

The *CL* package defines a version of the Common Lisp `typep` predicate.

`typep` *object type* [Function]

Check if *object* is of type *type*, where *type* is a (quoted) type name of the sort used by Common Lisp. For example, `(typep foo 'integer)` is equivalent to `(integerp foo)`.

The *type* argument to the above function is either a symbol or a list beginning with a symbol.

- If the type name is a symbol, Emacs appends '`-p`' to the symbol name to form the name of a predicate function for testing the type. (Built-in predicates whose names end in '`p`' rather than '`-p`' are used when appropriate.)
- The type symbol `t` stands for the union of all types. `(typep object t)` is always true. Likewise, the type symbol `nil` stands for nothing at all, and `(typep object nil)` is always false.
- The type symbol `null` represents the symbol `nil`. Thus `(typep object 'null)` is equivalent to `(null object)`.
- The type symbol `real` is a synonym for `number`, and `fixnum` is a synonym for `integer`.
- The type symbols `character` and `string-char` match integers in the range from 0 to 255.
- The type symbol `float` uses the `floatp-safe` predicate defined by this package rather than `floatp`, so it will work correctly even in Emacs versions without floating-point support.
- The type list `(integer low high)` represents all integers between *low* and *high*, inclusive. Either bound may be a list of a single integer to specify an exclusive limit, or a `*` to specify no limit. The type `(integer * *)` is thus equivalent to `integer`.
- Likewise, lists beginning with `float`, `real`, or `number` represent numbers of that type falling in a particular range.
- Lists beginning with `and`, `or`, and `not` form combinations of types. For example, `(or integer (float 0 *))` represents all objects that are integers or non-negative floats.
- Lists beginning with `member` or `member*` represent objects `eql` to any of the following values. For example, `(member 1 2 3 4)` is equivalent to `(integer 1 4)`, and `(member nil)` is equivalent to `null`.
- Lists of the form `(satisfies predicate)` represent all objects for which *predicate* returns true when called with that object as an argument.

The following function and macro (not technically predicates) are related to `typep`.

`coerce` *object type* [Function]

This function attempts to convert *object* to the specified *type*. If *object* is already of that type as determined by `typep`, it is simply returned. Otherwise, certain types of conversions will be made: If *type* is any sequence type (`string`, `list`, etc.) then *object* will be converted to that type if possible. If *type* is `character`, then strings of length one and symbols with one-character names can be coerced. If *type* is `float`, then integers can be coerced in versions of Emacs that support floats. In all other circumstances, `coerce` signals an error.

`deftype` *name arglist forms...* [Special Form]

This macro defines a new type called *name*. It is similar to `defmacro` in many ways; when *name* is encountered as a type name, the body *forms* are evaluated and should return a type specifier that is equivalent to the type. The *arglist* is a Common Lisp argument list of the sort accepted by `defmacro*`. The type specifier '`(name args...)`' is expanded by calling the expander with those arguments; the type symbol '`name`' is expanded by calling the expander with no arguments. The *arglist* is processed the same as for `defmacro*` except that optional arguments without explicit defaults use `*` instead of `nil` as the "default" default. Some examples:

```
(deftype null () '(satisfies null))    ; predefined
(deftype list () '(or null cons))      ; predefined
(deftype unsigned-byte (&optional bits)
  (list 'integer 0 (if (eq bits '*) bits (1- (lsh 1 bits)))))
(unsigned-byte 8)  ≡  (integer 0 255)
(unsigned-byte)  ≡  (integer 0 *)
unsigned-byte  ≡  (integer 0 *)
```

The last example shows how the Common Lisp `unsigned-byte` type specifier could be implemented if desired; this package does not implement `unsigned-byte` by default.

The `typecase` and `check-type` macros also use type names. See Section 7.6 [Conditionals], page 25. See Chapter 24 [Assertions], page 71. The `map`, `concatenate`, and `merge` functions take type-name arguments to specify the type of sequence to return. See Chapter 14 [Sequences], page 51.

6.2 Equality Predicates

This package defines two Common Lisp predicates, `eql` and `equalp`.

`eql` *a b* [Function]

This function is almost the same as `eq`, except that if *a* and *b* are numbers of the same type, it compares them for numeric equality (as if by `equal` instead of `eq`). This makes a difference only for versions of Emacs that are compiled with floating-point support, such as Emacs 19. Emacs floats are allocated objects just like cons cells, which means that `(eq 3.0 3.0)` will not necessarily be true—if the two `3.0`s were allocated separately, the pointers will be different even though the numbers are the same. But `(eql 3.0 3.0)` will always be true.

The types of the arguments must match, so `(eql 3 3.0)` is still false.

Note that Emacs integers are "direct" rather than allocated, which basically means `(eq 3 3)` will always be true. Thus `eq` and `eql` behave differently only if floating-point numbers are involved, and are indistinguishable on Emacs versions that don't support floats.

There is a slight inconsistency with Common Lisp in the treatment of positive and negative zeros. Some machines, notably those with IEEE standard arithmetic, represent `+0` and `-0` as distinct values. Normally this doesn't matter because the standard specifies that `(= 0.0 -0.0)` should always be true, and this is indeed what Emacs Lisp and Common Lisp do. But the Common Lisp standard states that `(eql 0.0 -0.0)` and `(equal 0.0 -0.0)` should be false on IEEE-like machines; Emacs Lisp does not do this, and in fact the only known way to distinguish between the two zeros in Emacs Lisp is to `format` them and check for a minus sign.

`equalp` *a b* [Function]

This function is a more flexible version of `equal`. In particular, it compares strings case-insensitively, and it compares numbers without regard to type (so that `(equalp 3 3.0)` is true). Vectors and conses are compared recursively. All other objects are compared as if by `equal`.

This function differs from Common Lisp `equalp` in several respects. First, Common Lisp's `equalp` also compares *characters* case-insensitively, which would be impractical in this package since Emacs does not distinguish between integers and characters. In keeping with the idea that strings are less vector-like in Emacs Lisp, this package's `equalp` also will not compare strings against vectors of integers. Finally, Common Lisp's `equalp` compares hash tables without regard to ordering, whereas this package simply compares hash tables in terms of their underlying structure (which means vectors for Lucid Emacs 19 hash tables, or lists for other hash tables).

Also note that the Common Lisp functions `member` and `assoc` use `eql` to compare elements, whereas Emacs Lisp follows the MacLisp tradition and uses `equal` for these two functions. In Emacs, use `member*` and `assoc*` to get functions which use `eql` for comparisons.

7 Control Structure

The features described in the following sections implement various advanced control structures, including the powerful `setf` facility and a number of looping and conditional constructs.

7.1 Assignment

The `psetq` form is just like `setq`, except that multiple assignments are done in parallel rather than sequentially.

`psetq` [*symbol form*]. . . [Special Form]

This special form (actually a macro) is used to assign to several variables simultaneously. Given only one *symbol* and *form*, it has the same effect as `setq`. Given several *symbol* and *form* pairs, it evaluates all the *forms* in advance and then stores the corresponding variables afterwards.

```
(setq x 2 y 3)
(setq x (+ x y)  y (* x y))
x
     ⇒ 5
y                           ; y was computed after x was set.
     ⇒ 15
(setq x 2 y 3)
(psetq x (+ x y)  y (* x y))
x
     ⇒ 5
y                           ; y was computed before x was set.
     ⇒ 6
```

The simplest use of `psetq` is `(psetq x y y x)`, which exchanges the values of two variables. (The `rotatef` form provides an even more convenient way to swap two variables; see Section 7.2.2 [Modify Macros], page 16.)

`psetq` always returns `nil`.

7.2 Generalized Variables

A "generalized variable" or "place form" is one of the many places in Lisp memory where values can be stored. The simplest place form is a regular Lisp variable. But the cars and cdrs of lists, elements of arrays, properties of symbols, and many other locations are also places where Lisp values are stored.

The `setf` form is like `setq`, except that it accepts arbitrary place forms on the left side rather than just symbols. For example, `(setf (car a) b)` sets the car of `a` to `b`, doing the same operation as `(setcar a b)` but without having to remember two separate functions for setting and accessing every type of place.

Generalized variables are analogous to "lvalues" in the C language, where '`x = a[i]`' gets an element from an array and '`a[i] = x`' stores an element using the same notation. Just as certain forms like `a[i]` can be lvalues in C, there is a set of forms that can be generalized variables in Lisp.

7.2.1 Basic Setf

The `setf` macro is the most basic way to operate on generalized variables.

`setf` [*place form*]. . . [Special Form]

This macro evaluates *form* and stores it in *place*, which must be a valid generalized variable form. If there are several *place* and *form* pairs, the assignments are done sequentially just as with `setq`. `setf` returns the value of the last *form*.

The following Lisp forms will work as generalized variables, and so may legally appear in the *place* argument of `setf`:

- A symbol naming a variable. In other words, `(setf x y)` is exactly equivalent to `(setq x y)`, and `setq` itself is strictly speaking redundant now that `setf` exists. Many programmers continue to prefer `setq` for setting simple variables, though, purely for stylistic or historical reasons. The macro `(setf x y)` actually expands to `(setq x y)`, so there is no performance penalty for using it in compiled code.
- A call to any of the following Lisp functions:

```
car                cdr                caar .. cddddr
nth                rest               first .. tenth
aref               elt                nthcdr
symbol-function    symbol-value       symbol-plist
get                get*               getf
gethash            subseq
```

  Note that for `nthcdr` and `getf`, the list argument of the function must itself be a valid *place* form. For example, `(setf (nthcdr 0 foo) 7)` will set `foo` itself to 7. Note that `push` and `pop` on an `nthcdr` place can be used to insert or delete at any position in a list. The use of `nthcdr` as a *place* form is an extension to standard Common Lisp.
- The following Emacs-specific functions are also `setf`-able. (Some of these are defined only in Emacs 19 or only in Lucid Emacs.)

```
buffer-file-name                  marker-position
buffer-modified-p                 match-data
buffer-name                       mouse-position
buffer-string                     overlay-end
buffer-substring                  overlay-get
current-buffer                    overlay-start
current-case-table                point
current-column                    point-marker
current-global-map                point-max
current-input-mode                point-min
current-local-map                 process-buffer
current-window-configuration      process-filter
default-file-modes                process-sentinel
default-value                     read-mouse-position
documentation-property            screen-height
extent-data                       screen-menubar
extent-end-position               screen-width
extent-start-position             selected-window
face-background                   selected-screen
face-background-pixmap            selected-frame
face-font                         standard-case-table
face-foreground                   syntax-table
face-underline-p                  window-buffer
```

```
file-modes                     window-dedicated-p
frame-height                   window-display-table
frame-parameters               window-height
frame-visible-p                window-hscroll
frame-width                    window-point
get-register                   window-start
getenv                         window-width
global-key-binding             x-get-cut-buffer
keymap-parent                  x-get-cutbuffer
local-key-binding              x-get-secondary-selection
mark                           x-get-selection
mark-marker
```

Most of these have directly corresponding "set" functions, like `use-local-map` for `current-local-map`, or `goto-char` for `point`. A few, like `point-min`, expand to longer sequences of code when they are `setf`'d (`(narrow-to-region x (point-max))` in this case).

- A call of the form `(substring` *subplace n* `[`*m*`])`, where *subplace* is itself a legal generalized variable whose current value is a string, and where the value stored is also a string. The new string is spliced into the specified part of the destination string. For example:

```
(setq a (list "hello" "world"))
     ⇒ ("hello" "world")
(cadr a)
     ⇒ "world"
(substring (cadr a) 2 4)
     ⇒ "rl"
(setf (substring (cadr a) 2 4) "o")
     ⇒ "o"
(cadr a)
     ⇒ "wood"
a
     ⇒ ("hello" "wood")
```

  The generalized variable `buffer-substring`, listed above, also works in this way by replacing a portion of the current buffer.

- A call of the form `(apply '`*func* `...)` or `(apply (function` *func*`) ...)`, where *func* is a `setf`-able function whose store function is "suitable" in the sense described in Steele's book; since none of the standard Emacs place functions are suitable in this sense, this feature is only interesting when used with places you define yourself with `define-setf-method` or the long form of `defsetf`.
- A macro call, in which case the macro is expanded and `setf` is applied to the resulting form.
- Any form for which a `defsetf` or `define-setf-method` has been made.

Using any forms other than these in the *place* argument to `setf` will signal an error.

The `setf` macro takes care to evaluate all subforms in the proper left-to-right order; for example,

```
(setf (aref vec (incf i)) i)
```

looks like it will evaluate `(incf i)` exactly once, before the following access to `i`; the `setf` expander will insert temporary variables as necessary to ensure that it does in fact work this way no matter what setf-method is defined for `aref`. (In this case, `aset` would be used and no such steps would be necessary since `aset` takes its arguments in a convenient order.)

However, if the *place* form is a macro which explicitly evaluates its arguments in an unusual order, this unusual order will be preserved. Adapting an example from Steele, given

```
(defmacro wrong-order (x y) (list 'aref y x))
```

the form `(setf (wrong-order a b) 17)` will evaluate *b* first, then *a*, just as in an actual call to `wrong-order`.

7.2.2 Modify Macros

This package defines a number of other macros besides `setf` that operate on generalized variables. Many are interesting and useful even when the *place* is just a variable name.

`psetf` [*place form*]. . . [Special Form]

This macro is to `setf` what `psetq` is to `setq`: When several *place*s and *form*s are involved, the assignments take place in parallel rather than sequentially. Specifically, all subforms are evaluated from left to right, then all the assignments are done (in an undefined order).

`incf` *place* **&optional** *x* [Special Form]

This macro increments the number stored in *place* by one, or by *x* if specified. The incremented value is returned. For example, `(incf i)` is equivalent to `(setq i (1+ i))`, and `(incf (car x) 2)` is equivalent to `(setcar x (+ (car x) 2))`.

Once again, care is taken to preserve the "apparent" order of evaluation. For example,

```
(incf (aref vec (incf i)))
```

appears to increment `i` once, then increment the element of `vec` addressed by `i`; this is indeed exactly what it does, which means the above form is *not* equivalent to the "obvious" expansion,

```
(setf (aref vec (incf i)) (1+ (aref vec (incf i))))   ; Wrong!
```

but rather to something more like

```
(let ((temp (incf i)))
  (setf (aref vec temp) (1+ (aref vec temp))))
```

Again, all of this is taken care of automatically by `incf` and the other generalized-variable macros.

As a more Emacs-specific example of `incf`, the expression `(incf (point) n)` is essentially equivalent to `(forward-char n)`.

`decf` *place* **&optional** *x* [Special Form]

This macro decrements the number stored in *place* by one, or by *x* if specified.

`pop` *place* [Special Form]

This macro removes and returns the first element of the list stored in *place*. It is analogous to `(prog1 (car place) (setf place (cdr place)))`, except that it takes care to evaluate all subforms only once.

`push` *x place* [Special Form]

This macro inserts *x* at the front of the list stored in *place*. It is analogous to `(setf place (cons x place))`, except for evaluation of the subforms.

`pushnew` *x place* **&key** `:test :test-not :key` [Special Form]

This macro inserts *x* at the front of the list stored in *place*, but only if *x* was not `eql` to any existing element of the list. The optional keyword arguments are interpreted in the same way as for `adjoin`. See Section 15.5 [Lists as Sets], page 60.

`shiftf` *place... newvalue* [Special Form]

This macro shifts the *place*s left by one, shifting in the value of *newvalue* (which may be any Lisp expression, not just a generalized variable), and returning the value shifted out of the first *place*. Thus, `(shiftf a b c d)` is equivalent to

```
(prog1
    a
  (psetf a b
         b c
         c d))
```

except that the subforms of *a*, *b*, and *c* are actually evaluated only once each and in the apparent order.

`rotatef` *place...* [Special Form]

This macro rotates the *place*s left by one in circular fashion. Thus, `(rotatef a b c d)` is equivalent to

```
(psetf a b
       b c
       c d
       d a)
```

except for the evaluation of subforms. `rotatef` always returns `nil`. Note that `(rotatef a b)` conveniently exchanges *a* and *b*.

The following macros were invented for this package; they have no analogues in Common Lisp.

`letf` (*bindings...*) *forms...* [Special Form]

This macro is analogous to `let`, but for generalized variables rather than just symbols. Each *binding* should be of the form `(place value)`; the original contents of the *place*s are saved, the *value*s are stored in them, and then the body *form*s are executed. Afterwards, the *place*s are set back to their original saved contents. This cleanup happens even if the *form*s exit irregularly due to a `throw` or an error.

For example,

```
(letf (((point) (point-min))
       (a 17))
  ...)
```

moves “point” in the current buffer to the beginning of the buffer, and also binds `a` to 17 (as if by a normal `let`, since `a` is just a regular variable). After the body exits, `a` is set back to its original value and point is moved back to its original position.

Note that `letf` on `(point)` is not quite like a `save-excursion`, as the latter effectively saves a marker which tracks insertions and deletions in the buffer. Actually, a `letf` of `(point-marker)` is much closer to this behavior. (`point` and `point-marker` are equivalent as `setf` places; each will accept either an integer or a marker as the stored value.)

Since generalized variables look like lists, `let`'s shorthand of using '`foo`' for '`(foo nil)`' as a *binding* would be ambiguous in `letf` and is not allowed.

However, a *binding* specifier may be a one-element list '`(place)`', which is similar to '`(place place)`'. In other words, the *place* is not disturbed on entry to the body, and the only effect of the `letf` is to restore the original value of *place* afterwards. (The redundant access-and-store suggested by the `(place place)` example does not actually occur.)

In most cases, the *place* must have a well-defined value on entry to the `letf` form. The only exceptions are plain variables and calls to `symbol-value` and `symbol-function`. If the symbol is not bound on entry, it is simply made unbound by `makunbound` or `fmakunbound` on exit.

`letf*` (*bindings...*) *forms...* [Special Form]

This macro is to `letf` what `let*` is to `let`: It does the bindings in sequential rather than parallel order.

`callf` *function place args...* [Special Form]

This is the "generic" modify macro. It calls *function*, which should be an unquoted function name, macro name, or lambda. It passes *place* and *args* as arguments, and assigns the result back to *place*. For example, `(incf place n)` is the same as `(callf + place n)`. Some more examples:

```
(callf abs my-number)
(callf concat (buffer-name) "<" (int-to-string n) ">")
(callf union happy-people (list joe bob) :test 'same-person)
```

See Section 7.2.3 [Customizing Setf], page 18, for `define-modify-macro`, a way to create even more concise notations for modify macros. Note again that `callf` is an extension to standard Common Lisp.

`callf2` *function arg1 place args...* [Special Form]

This macro is like `callf`, except that *place* is the *second* argument of *function* rather than the first. For example, `(push x place)` is equivalent to `(callf2 cons x place)`.

The `callf` and `callf2` macros serve as building blocks for other macros like `incf`, `pushnew`, and `define-modify-macro`. The `letf` and `letf*` macros are used in the processing of symbol macros; see Section 7.5.4 [Macro Bindings], page 24.

7.2.3 Customizing Setf

Common Lisp defines three macros, `define-modify-macro`, `defsetf`, and `define-setf-method`, that allow the user to extend generalized variables in various ways.

`define-modify-macro` *name arglist function* [*doc-string*] [Special Form]

This macro defines a "read-modify-write" macro similar to `incf` and `decf`. The macro *name* is defined to take a *place* argument followed by additional arguments described by *arglist*. The call

```
(name place args...)
```

will be expanded to

```
(callf func place args...)
```

which in turn is roughly equivalent to

```
(setf place (func place args...))
```

For example:

```
(define-modify-macro incf (&optional (n 1)) +)
(define-modify-macro concatf (&rest args) concat)
```

Note that `&key` is not allowed in *arglist*, but `&rest` is sufficient to pass keywords on to the function.

Most of the modify macros defined by Common Lisp do not exactly follow the pattern of `define-modify-macro`. For example, `push` takes its arguments in the wrong order, and `pop` is completely irregular. You can define these macros "by hand" using `get-setf-method`, or consult the source file `cl-macs.el` to see how to use the internal `setf` building blocks.

`defsetf` *access-fn update-fn* [Special Form]

This is the simpler of two `defsetf` forms. Where *access-fn* is the name of a function which accesses a place, this declares *update-fn* to be the corresponding store function. From now on,

```
(setf (access-fn arg1 arg2 arg3) value)
```

will be expanded to

```
(update-fn arg1 arg2 arg3 value)
```

The *update-fn* is required to be either a true function, or a macro which evaluates its arguments in a function-like way. Also, the *update-fn* is expected to return *value* as its result. Otherwise, the above expansion would not obey the rules for the way `setf` is supposed to behave.

As a special (non-Common-Lisp) extension, a third argument of `t` to `defsetf` says that the `update-fn`'s return value is not suitable, so that the above `setf` should be expanded to something more like

```
(let ((temp value))
  (update-fn arg1 arg2 arg3 temp)
  temp)
```

Some examples of the use of `defsetf`, drawn from the standard suite of setf methods, are:

```
(defsetf car setcar)
(defsetf symbol-value set)
(defsetf buffer-name rename-buffer t)
```

`defsetf` *access-fn arglist* (*store-var*) *forms...* [Special Form]

This is the second, more complex, form of `defsetf`. It is rather like `defmacro` except for the additional *store-var* argument. The *forms* should return a Lisp form which stores the value of *store-var* into the generalized variable formed by a call to *access-fn* with arguments described by *arglist*. The *forms* may begin with a string which documents the `setf` method (analogous to the doc string that appears at the front of a function).

For example, the simple form of `defsetf` is shorthand for

```
(defsetf access-fn (&rest args) (store)
  (append '(update-fn) args (list store)))
```

The Lisp form that is returned can access the arguments from *arglist* and *store-var* in an unrestricted fashion; macros like `setf` and `incf` which invoke this setf-method will insert temporary variables as needed to make sure the apparent order of evaluation is preserved.

Another example drawn from the standard package:

```
(defsetf nth (n x) (store)
  (list 'setcar (list 'nthcdr n x) store))
```

`define-setf-method` *access-fn arglist forms...* [Special Form]

This is the most general way to create new place forms. When a `setf` to *access-fn* with arguments described by *arglist* is expanded, the *forms* are evaluated and must return a list of five items:

1. A list of *temporary variables*.
2. A list of *value forms* corresponding to the temporary variables above. The temporary variables will be bound to these value forms as the first step of any operation on the generalized variable.
3. A list of exactly one *store variable* (generally obtained from a call to `gensym`).
4. A Lisp form which stores the contents of the store variable into the generalized variable, assuming the temporaries have been bound as described above.
5. A Lisp form which accesses the contents of the generalized variable, assuming the temporaries have been bound.

This is exactly like the Common Lisp macro of the same name, except that the method returns a list of five values rather than the five values themselves, since Emacs Lisp does not support Common Lisp's notion of multiple return values.

Once again, the *forms* may begin with a documentation string.

A setf-method should be maximally conservative with regard to temporary variables. In the setf-methods generated by `defsetf`, the second return value is simply the list of arguments in the place form, and the first return value is a list of a corresponding number of temporary variables generated by `gensym`. Macros like `setf` and `incf` which use this setf-method will optimize away most temporaries that turn out to be unnecessary, so there is little reason for the setf-method itself to optimize.

`get-setf-method` *place* **&optional** *env* [Function]

This function returns the setf-method for *place*, by invoking the definition previously recorded by `defsetf` or `define-setf-method`. The result is a list of five values as described above. You can use this function to build your own `incf`-like modify macros. (Actually, it is better to use the internal functions `cl-setf-do-modify` and `cl-setf-do-store`, which are a bit easier to use and which also do a number of optimizations; consult the source code for the `incf` function for a simple example.)

The argument *env* specifies the "environment" to be passed on to `macroexpand` if `get-setf-method` should need to expand a macro in *place*. It should come from an `&environment` argument to the macro or setf-method that called `get-setf-method`.

See also the source code for the setf-methods for `apply` and `substring`, each of which works by calling `get-setf-method` on a simpler case, then massaging the result in various ways.

Modern Common Lisp defines a second, independent way to specify the `setf` behavior of a function, namely "`setf` functions" whose names are lists `(setf name)` rather than symbols. For example, `(defun (setf foo) ...)` defines the function that is used when `setf` is applied to `foo`. This package does not currently support `setf` functions. In particular, it is a compile-time error to use `setf` on a form which has not already been `defsetf`'d or otherwise declared; in newer Common Lisps, this would not be an error since the function `(setf func)` might be defined later.

7.5 Variable Bindings

These Lisp forms make bindings to variables and function names, analogous to Lisp's built-in `let` form.

See Section 7.2.2 [Modify Macros], page 16, for the `letf` and `letf*` forms which are also related to variable bindings.

7.5.1 Dynamic Bindings

The standard `let` form binds variables whose names are known at compile-time. The `progv` form provides an easy way to bind variables whose names are computed at run-time.

`progv` *symbols values forms. . .* [Special Form]

This form establishes `let`-style variable bindings on a set of variables computed at run-time. The expressions *symbols* and *values* are evaluated, and must return lists of symbols and values, respectively. The symbols are bound to the corresponding values for the duration of the body *forms*. If *values* is shorter than *symbols*, the last few symbols are made unbound (as if by `makunbound`) inside the body. If *symbols* is shorter than *values*, the excess values are ignored.

7.5.2 Lexical Bindings

The *CL* package defines the following macro which more closely follows the Common Lisp `let` form:

`lexical-let` (*bindings. . .*) *forms. . .* [Special Form]

This form is exactly like `let` except that the bindings it establishes are purely lexical. Lexical bindings are similar to local variables in a language like C: Only the code physically within the body of the `lexical-let` (after macro expansion) may refer to the bound variables.

```
(setq a 5)
```

```
(defun foo (b) (+ a b))
(let ((a 2)) (foo a))
     ⇒ 4
(lexical-let ((a 2)) (foo a))
     ⇒ 7
```

In this example, a regular `let` binding of `a` actually makes a temporary change to the global variable `a`, so `foo` is able to see the binding of `a` to 2. But `lexical-let` actually creates a distinct local variable `a` for use within its body, without any effect on the global variable of the same name.

The most important use of lexical bindings is to create *closures*. A closure is a function object that refers to an outside lexical variable. For example:

```
(defun make-adder (n)
  (lexical-let ((n n))
    (function (lambda (m) (+ n m)))))
(setq add17 (make-adder 17))
(funcall add17 4)
     ⇒ 21
```

The call `(make-adder 17)` returns a function object which adds 17 to its argument. If `let` had been used instead of `lexical-let`, the function object would have referred to the global `n`, which would have been bound to 17 only during the call to `make-adder` itself.

```
(defun make-counter ()
  (lexical-let ((n 0))
    (function* (lambda (&optional (m 1)) (incf n m)))))
(setq count-1 (make-counter))
(funcall count-1 3)
     ⇒ 3
(funcall count-1 14)
     ⇒ 17
(setq count-2 (make-counter))
(funcall count-2 5)
     ⇒ 5
(funcall count-1 2)
     ⇒ 19
(funcall count-2)
     ⇒ 6
```

Here we see that each call to `make-counter` creates a distinct local variable `n`, which serves as a private counter for the function object that is returned.

Closed-over lexical variables persist until the last reference to them goes away, just like all other Lisp objects. For example, `count-2` refers to a function object which refers to an instance of the variable `n`; this is the only reference to that variable, so after `(setq count-2 nil)` the garbage collector would be able to delete this instance of `n`. Of course, if a `lexical-let` does not actually create any closures, then the lexical variables are free as soon as the `lexical-let` returns.

Many closures are used only during the extent of the bindings they refer to; these are known as "downward funargs" in Lisp parlance. When a closure is used in this way, regular Emacs Lisp dynamic bindings suffice and will be more efficient than `lexical-let` closures:

```
(defun add-to-list (x list)
  (mapcar (function (lambda (y) (+ x y))) list))
(add-to-list 7 '(1 2 5))
     ⇒ (8 9 12)
```

Since this lambda is only used while `x` is still bound, it is not necessary to make a true closure out of it.

You can use `defun` or `flet` inside a `lexical-let` to create a named closure. If several closures are created in the body of a single `lexical-let`, they all close over the same instance of the lexical variable.

The `lexical-let` form is an extension to Common Lisp. In true Common Lisp, all bindings are lexical unless declared otherwise.

`lexical-let*` (*bindings. . .*) *forms. . .* [Special Form]

This form is just like `lexical-let`, except that the bindings are made sequentially in the manner of `let*`.

7.5.3 Function Bindings

These forms make `let`-like bindings to functions instead of variables.

`flet` (*bindings. . .*) *forms. . .* [Special Form]

This form establishes `let`-style bindings on the function cells of symbols rather than on the value cells. Each *binding* must be a list of the form '`(name arglist forms...)`', which defines a function exactly as if it were a `defun*` form. The function *name* is defined accordingly for the duration of the body of the `flet`; then the old function definition, or lack thereof, is restored.

While `flet` in Common Lisp establishes a lexical binding of *name*, Emacs Lisp `flet` makes a dynamic binding. The result is that `flet` affects indirect calls to a function as well as calls directly inside the `flet` form itself.

You can use `flet` to disable or modify the behavior of a function in a temporary fashion. This will even work on Emacs primitives, although note that some calls to primitive functions internal to Emacs are made without going through the symbol's function cell, and so will not be affected by `flet`. For example,

```
(flet ((message (&rest args) (push args saved-msgs)))
  (do-something))
```

This code attempts to replace the built-in function `message` with a function that simply saves the messages in a list rather than displaying them. The original definition of `message` will be restored after `do-something` exits. This code will work fine on messages generated by other Lisp code, but messages generated directly inside Emacs will not be caught since they make direct C-language calls to the message routines rather than going through the Lisp `message` function.

Functions defined by `flet` may use the full Common Lisp argument notation supported by `defun*`; also, the function body is enclosed in an implicit block as if by `defun*`. See Chapter 5 [Program Structure], page 4.

`labels` (*bindings...*) *forms...* [Special Form]

The `labels` form is a synonym for `flet`. (In Common Lisp, `labels` and `flet` differ in ways that depend on their lexical scoping; these distinctions vanish in dynamically scoped Emacs Lisp.)

7.5.4 Macro Bindings

These forms create local macros and "symbol macros."

`macrolet` (*bindings...*) *forms...* [Special Form]

This form is analogous to `flet`, but for macros instead of functions. Each *binding* is a list of the same form as the arguments to `defmacro*` (i.e., a macro name, argument list, and macro-expander forms). The macro is defined accordingly for use within the body of the `macrolet`.

Because of the nature of macros, `macrolet` is lexically scoped even in Emacs Lisp: The `macrolet` binding will affect only calls that appear physically within the body *forms*, possibly after expansion of other macros in the body.

`symbol-macrolet` (*bindings...*) *forms...* [Special Form]

This form creates *symbol macros*, which are macros that look like variable references rather than function calls. Each *binding* is a list '`(var expansion)`'; any reference to *var* within the body *forms* is replaced by *expansion*.

```
(setq bar '(5 . 9))
(symbol-macrolet ((foo (car bar)))
  (incf foo))
bar
     ⇒ (6 . 9)
```

A `setq` of a symbol macro is treated the same as a `setf`. I.e., `(setq foo 4)` in the above would be equivalent to `(setf foo 4)`, which in turn expands to `(setf (car bar) 4)`.

Likewise, a `let` or `let*` binding a symbol macro is treated like a `letf` or `letf*`. This differs from true Common Lisp, where the rules of lexical scoping cause a `let` binding to shadow a `symbol-macrolet` binding. In this package, only `lexical-let` and `lexical-let*` will shadow a symbol macro.

There is no analogue of `defmacro` for symbol macros; all symbol macros are local. A typical use of `symbol-macrolet` is in the expansion of another macro:

```
(defmacro* my-dolist ((x list) &rest body)
  (let ((var (gensym)))
    (list 'loop 'for var 'on list 'do
          (list* 'symbol-macrolet (list (list x (list 'car var)))
                 body))))

(setq mylist '(1 2 3 4))
```

```
(my-dolist (x mylist) (incf x))
mylist
     ⇒ (2 3 4 5)
```

In this example, the `my-dolist` macro is similar to `dolist` (see Section 7.8 [Iteration], page 27) except that the variable `x` becomes a true reference onto the elements of the list. The `my-dolist` call shown here expands to

```
(loop for G1234 on mylist do
      (symbol-macrolet ((x (car G1234)))
        (incf x)))
```

which in turn expands to

```
(loop for G1234 on mylist do (incf (car G1234)))
```

See Section 7.9 [Loop Facility], page 29, for a description of the `loop` macro. This package defines a nonstandard `in-ref` loop clause that works much like `my-dolist`.

7.6 Conditionals

These conditional forms augment Emacs Lisp's simple `if`, `and`, `or`, and `cond` forms.

`when` *test forms...* [Special Form]

This is a variant of `if` where there are no "else" forms, and possibly several "then" forms. In particular,

```
(when test a b c)
```

is entirely equivalent to

```
(if test (progn a b c) nil)
```

`unless` *test forms...* [Special Form]

This is a variant of `if` where there are no "then" forms, and possibly several "else" forms:

```
(unless test a b c)
```

is entirely equivalent to

```
(when (not test) a b c)
```

`case` *keyform clause...* [Special Form]

This macro evaluates *keyform*, then compares it with the key values listed in the various *clause*s. Whichever clause matches the key is executed; comparison is done by `eql`. If no clause matches, the `case` form returns `nil`. The clauses are of the form

```
(keylist body-forms...)
```

where *keylist* is a list of key values. If there is exactly one value, and it is not a cons cell or the symbol `nil` or `t`, then it can be used by itself as a *keylist* without being enclosed in a list. All key values in the `case` form must be distinct. The final clauses may use `t` in place of a *keylist* to indicate a default clause that should be taken if none of the other clauses match. (The symbol `otherwise` is also recognized in place of `t`. To make a clause that matches the actual symbol `t`, `nil`, or `otherwise`, enclose the symbol in a list.)

For example, this expression reads a keystroke, then does one of four things depending on whether it is an ‘a’, a ‘b’, a RET or LFD, or anything else.

```
(case (read-char)
  (?a (do-a-thing))
  (?b (do-b-thing))
  ((?\r ?\n) (do-ret-thing))
  (t (do-other-thing)))
```

`ecase` *keyform clause...* [Special Form]

This macro is just like `case`, except that if the key does not match any of the clauses, an error is signalled rather than simply returning `nil`.

`typecase` *keyform clause...* [Special Form]

This macro is a version of `case` that checks for types rather than values. Each *clause* is of the form ‘(*type body*...)’. See Section 6.1 [Type Predicates], page 10, for a description of type specifiers. For example,

```
(typecase x
  (integer (munch-integer x))
  (float (munch-float x))
  (string (munch-integer (string-to-int x)))
  (t (munch-anything x)))
```

The type specifier `t` matches any type of object; the word `otherwise` is also allowed. To make one clause match any of several types, use an `(or ...)` type specifier.

`etypecase` *keyform clause...* [Special Form]

This macro is just like `typecase`, except that if the key does not match any of the clauses, an error is signalled rather than simply returning `nil`.

7.7 Blocks and Exits

Common Lisp *blocks* provide a non-local exit mechanism very similar to `catch` and `throw`, but lexically rather than dynamically scoped. This package actually implements `block` in terms of `catch`; however, the lexical scoping allows the optimizing byte-compiler to omit the costly `catch` step if the body of the block does not actually `return-from` the block.

`block` *name forms...* [Special Form]

The *forms* are evaluated as if by a `progn`. However, if any of the *forms* execute `(return-from name)`, they will jump out and return directly from the `block` form. The `block` returns the result of the last *form* unless a `return-from` occurs.

The `block`/`return-from` mechanism is quite similar to the `catch`/`throw` mechanism. The main differences are that block *name*s are unevaluated symbols, rather than forms (such as quoted symbols) which evaluate to a tag at run-time; and also that blocks are lexically scoped whereas `catch`/`throw` are dynamically scoped. This means that functions called from the body of a `catch` can also `throw` to the `catch`, but the `return-from` referring to a block name must appear physically within the *forms* that make up the body of the block. They may not appear within other called functions, although they may appear within macro expansions or `lambda`s in the body. Block names and `catch` names form independent name-spaces.

In true Common Lisp, `defun` and `defmacro` surround the function or expander bodies with implicit blocks with the same name as the function or macro. This does not occur in Emacs Lisp, but this package provides `defun*` and `defmacro*` forms which do create the implicit block.

The Common Lisp looping constructs defined by this package, such as `loop` and `dolist`, also create implicit blocks just as in Common Lisp.

Because they are implemented in terms of Emacs Lisp `catch` and `throw`, blocks have the same overhead as actual `catch` constructs (roughly two function calls). However, Zawinski and Furuseth's optimizing byte compiler (standard in Emacs 19) will optimize away the `catch` if the block does not in fact contain any `return` or `return-from` calls that jump to it. This means that `do` loops and `defun*` functions which don't use `return` don't pay the overhead to support it.

`return-from` *name* [*result*] [Special Form]

This macro returns from the block named *name*, which must be an (unevaluated) symbol. If a *result* form is specified, it is evaluated to produce the result returned from the `block`. Otherwise, `nil` is returned.

`return` [*result*] [Special Form]

This macro is exactly like `(return-from nil result)`. Common Lisp loops like `do` and `dolist` implicitly enclose themselves in `nil` blocks.

7.8 Iteration

The macros described here provide more sophisticated, high-level looping constructs to complement Emacs Lisp's basic `while` loop.

`loop` *forms...* [Special Form]

The *CL* package supports both the simple, old-style meaning of `loop` and the extremely powerful and flexible feature known as the *Loop Facility* or *Loop Macro*. This more advanced facility is discussed in the following section; see Section 7.9 [Loop Facility], page 29. The simple form of `loop` is described here.

If `loop` is followed by zero or more Lisp expressions, then `(loop exprs...)` simply creates an infinite loop executing the expressions over and over. The loop is enclosed in an implicit `nil` block. Thus,

```
(loop (foo)  (if (no-more) (return 72))  (bar))
```

is exactly equivalent to

```
(block nil (while t (foo)  (if (no-more) (return 72))  (bar)))
```

If any of the expressions are plain symbols, the loop is instead interpreted as a Loop Macro specification as described later. (This is not a restriction in practice, since a plain symbol in the above notation would simply access and throw away the value of a variable.)

`do` (*spec...*) (*end-test* [*result...*]) *forms...* [Special Form]

This macro creates a general iterative loop. Each *spec* is of the form

```
(var [init [step]])
```

The loop works as follows: First, each *var* is bound to the associated *init* value as if by a `let` form. Then, in each iteration of the loop, the *end-test* is evaluated; if true, the loop is finished. Otherwise, the body *forms* are evaluated, then each *var* is set to the associated *step* expression (as if by a `psetq` form) and the next iteration begins. Once the *end-test* becomes true, the *result* forms are evaluated (with the *var*s still bound to their values) to produce the result returned by `do`.

The entire `do` loop is enclosed in an implicit `nil` block, so that you can use `(return)` to break out of the loop at any time.

If there are no *result* forms, the loop returns `nil`. If a given *var* has no *step* form, it is bound to its *init* value but not otherwise modified during the `do` loop (unless the code explicitly modifies it); this case is just a shorthand for putting a `(let ((var init)) ...)` around the loop. If *init* is also omitted it defaults to `nil`, and in this case a plain '`var`' can be used in place of '`(var)`', again following the analogy with `let`.

This example (from Steele) illustrates a loop which applies the function `f` to successive pairs of values from the lists `foo` and `bar`; it is equivalent to the call `(mapcar* 'f foo bar)`. Note that this loop has no body *forms* at all, performing all its work as side effects of the rest of the loop.

```
(do ((x foo (cdr x))
     (y bar (cdr y))
     (z nil (cons (f (car x) (car y)) z)))
  ((or (null x) (null y))
   (nreverse z)))
```

`do*` (*spec...*) (*end-test* [*result...*]) *forms...* [Special Form]

This is to `do` what `let*` is to `let`. In particular, the initial values are bound as if by `let*` rather than `let`, and the steps are assigned as if by `setq` rather than `psetq`.

Here is another way to write the above loop:

```
(do* ((xp foo (cdr xp))
      (yp bar (cdr yp))
      (x (car xp) (car xp))
      (y (car yp) (car yp))
      z)
  ((or (null xp) (null yp))
   (nreverse z))
  (push (f x y) z))
```

`dolist` (*var list* [*result*]) *forms...* [Special Form]

This is a more specialized loop which iterates across the elements of a list. *list* should evaluate to a list; the body *forms* are executed with *var* bound to each element of the list in turn. Finally, the *result* form (or `nil`) is evaluated with *var* bound to `nil` to produce the result returned by the loop. The loop is surrounded by an implicit `nil` block.

`dotimes` (*var count* [*result*]) *forms...* [Special Form]

This is a more specialized loop which iterates a specified number of times. The body is executed with *var* bound to the integers from zero (inclusive) to *count* (exclusive), in turn. Then the `result` form is evaluated with *var* bound to the total number of iterations that were done (i.e., `(max 0 count)`) to get the return value for the loop form. The loop is surrounded by an implicit `nil` block.

`do-symbols` (*var* [*obarray* [*result*]]) *forms...* [Special Form]

This loop iterates over all interned symbols. If *obarray* is specified and is not `nil`, it loops over all symbols in that obarray. For each symbol, the body *forms* are evaluated with *var* bound to that symbol. The symbols are visited in an unspecified order. Afterward the *result* form, if any, is evaluated (with *var* bound to `nil`) to get the return value. The loop is surrounded by an implicit `nil` block.

`do-all-symbols` (*var* [*result*]) *forms...* [Special Form]

This is identical to `do-symbols` except that the *obarray* argument is omitted; it always iterates over the default obarray.

See Section 14.2 [Mapping over Sequences], page 52, for some more functions for iterating over vectors or lists.

7.9 Loop Facility

A common complaint with Lisp's traditional looping constructs is that they are either too simple and limited, such as Common Lisp's `dotimes` or Emacs Lisp's `while`, or too unreadable and obscure, like Common Lisp's `do` loop.

To remedy this, recent versions of Common Lisp have added a new construct called the "Loop Facility" or "`loop` macro," with an easy-to-use but very powerful and expressive syntax.

7.9.1 Loop Basics

The `loop` macro essentially creates a mini-language within Lisp that is specially tailored for describing loops. While this language is a little strange-looking by the standards of regular Lisp, it turns out to be very easy to learn and well-suited to its purpose.

Since `loop` is a macro, all parsing of the loop language takes place at byte-compile time; compiled `loop`s are just as efficient as the equivalent `while` loops written longhand.

`loop` *clauses...* [Special Form]

A loop construct consists of a series of *clauses*, each introduced by a symbol like `for` or `do`. Clauses are simply strung together in the argument list of `loop`, with minimal extra parentheses. The various types of clauses specify initializations, such as the binding of temporary variables, actions to be taken in the loop, stepping actions, and final cleanup.

Common Lisp specifies a certain general order of clauses in a loop:

```
(loop name-clause
      var-clauses...
      action-clauses...)
```

The *name-clause* optionally gives a name to the implicit block that surrounds the loop. By default, the implicit block is named `nil`. The *var-clauses* specify what variables should be bound during the loop, and how they should be modified or iterated throughout the course of the loop. The *action-clauses* are things to be done during the loop, such as computing, collecting, and returning values.

The Emacs version of the `loop` macro is less restrictive about the order of clauses, but things will behave most predictably if you put the variable-binding clauses `with`, `for`, and `repeat` before the action clauses. As in Common Lisp, `initially` and `finally` clauses can go anywhere.

Loops generally return `nil` by default, but you can cause them to return a value by using an accumulation clause like `collect`, an end-test clause like `always`, or an explicit `return` clause to jump out of the implicit block. (Because the loop body is enclosed in an implicit block, you can also use regular Lisp `return` or `return-from` to break out of the loop.)

The following sections give some examples of the Loop Macro in action, and describe the particular loop clauses in great detail. Consult the second edition of Steele's *Common Lisp, the Language*, for additional discussion and examples of the `loop` macro.

7.9.2 Loop Examples

Before listing the full set of clauses that are allowed, let's look at a few example loops just to get a feel for the `loop` language.

```
(loop for buf in (buffer-list)
      collect (buffer-file-name buf))
```

This loop iterates over all Emacs buffers, using the list returned by `buffer-list`. For each buffer `buf`, it calls `buffer-file-name` and collects the results into a list, which is then returned from the `loop` construct. The result is a list of the file names of all the buffers in Emacs' memory. The words `for`, `in`, and `collect` are reserved words in the `loop` language.

```
(loop repeat 20 do (insert "Yowsa\n"))
```

This loop inserts the phrase "Yowsa" twenty times in the current buffer.

```
(loop until (eobp) do (munch-line) (forward-line 1))
```

This loop calls `munch-line` on every line until the end of the buffer. If point is already at the end of the buffer, the loop exits immediately.

```
(loop do (munch-line) until (eobp) do (forward-line 1))
```

This loop is similar to the above one, except that `munch-line` is always called at least once.

```
(loop for x from 1 to 100
      for y = (* x x)
      until (>= y 729)
      finally return (list x (= y 729)))
```

This more complicated loop searches for a number `x` whose square is 729. For safety's sake it only examines `x` values up to 100; dropping the phrase '`to 100`' would cause the loop to count upwards with no limit. The second `for` clause defines `y` to be the square of `x` within the loop; the expression after the `=` sign is reevaluated each time through the loop. The `until` clause gives a condition for terminating the loop, and the `finally` clause says what to do when the loop finishes. (This particular example was written less concisely than it could have been, just for the sake of illustration.)

Note that even though this loop contains three clauses (two `for`s and an `until`) that would have been enough to define loops all by themselves, it still creates a single loop rather than some sort of triple-nested loop. You must explicitly nest your `loop` constructs if you want nested loops.

7.9.3 For Clauses

Most loops are governed by one or more `for` clauses. A `for` clause simultaneously describes variables to be bound, how those variables are to be stepped during the loop, and usually an end condition based on those variables.

The word `as` is a synonym for the word `for`. This word is followed by a variable name, then a word like `from` or `across` that describes the kind of iteration desired. In Common Lisp, the phrase `being the` sometimes precedes the type of iteration; in this package both `being` and `the` are optional. The word `each` is a synonym for `the`, and the word that follows it may be singular or plural: '`for x being the elements of y`' or '`for x being each element of y`'. Which form you use is purely a matter of style.

The variable is bound around the loop as if by `let`:

```
(setq i 'happy)
(loop for i from 1 to 10 do (do-something-with i))
i
     ⇒ happy
```

`for` *`var`* `from` *`expr1`* `to` *`expr2`* `by` *`expr3`*

This type of `for` clause creates a counting loop. Each of the three sub-terms is optional, though there must be at least one term so that the clause is marked as a counting clause.

The three expressions are the starting value, the ending value, and the step value, respectively, of the variable. The loop counts upwards by default (*expr3* must be positive), from *expr1* to *expr2* inclusively. If you omit the `from` term, the loop counts from zero; if you omit the `to` term, the loop counts forever without stopping (unless stopped by some other loop clause, of course); if you omit the `by` term, the loop counts in steps of one.

You can replace the word `from` with `upfrom` or `downfrom` to indicate the direction of the loop. Likewise, you can replace `to` with `upto` or `downto`. For example, '`for x from 5 downto 1`' executes five times with `x` taking on the integers from 5 down to 1 in turn. Also, you can replace `to` with `below` or `above`, which are like `upto` and `downto` respectively except that they are exclusive rather than inclusive limits:

```
(loop for x to 10 collect x)
     ⇒ (0 1 2 3 4 5 6 7 8 9 10)
(loop for x below 10 collect x)
     ⇒ (0 1 2 3 4 5 6 7 8 9)
```

The `by` value is always positive, even for downward-counting loops. Some sort of `from` value is required for downward loops; '`for x downto 5`' is not a legal loop clause all by itself.

`for` *`var`* `in` *`list`* `by` *`function`*

This clause iterates *var* over all the elements of *list*, in turn. If you specify the `by` term, then *function* is used to traverse the list instead of `cdr`; it must be a function taking one argument. For example:

```
(loop for x in '(1 2 3 4 5 6) collect (* x x))
     ⇒ (1 4 9 16 25 36)
(loop for x in '(1 2 3 4 5 6) by 'cddr collect (* x x))
     ⇒ (1 9 25)
```

`for` *var* `on` *list* `by` *function*

This clause iterates *var* over all the cons cells of *list*.

```
(loop for x on '(1 2 3 4) collect x)
     ⇒ ((1 2 3 4) (2 3 4) (3 4) (4))
```

With `by`, there is no real reason that the `on` expression must be a list. For example:

```
(loop for x on first-animal by 'next-animal collect x)
```

where `(next-animal x)` takes an "animal" *x* and returns the next in the (assumed) sequence of animals, or `nil` if *x* was the last animal in the sequence.

`for` *var* `in-ref` *list* `by` *function*

This is like a regular `in` clause, but *var* becomes a `setf`-able "reference" onto the elements of the list rather than just a temporary variable. For example,

```
(loop for x in-ref my-list do (incf x))
```

increments every element of `my-list` in place. This clause is an extension to standard Common Lisp.

`for` *var* `across` *array*

This clause iterates *var* over all the elements of *array*, which may be a vector or a string.

```
(loop for x across "aeiou"
      do (use-vowel (char-to-string x)))
```

`for` *var* `across-ref` *array*

This clause iterates over an array, with *var* a `setf`-able reference onto the elements; see `in-ref` above.

`for` *var* `being the elements of` *sequence*

This clause iterates over the elements of *sequence*, which may be a list, vector, or string. Since the type must be determined at run-time, this is somewhat less efficient than `in` or `across`. The clause may be followed by the additional term '`using (index` *var2*`)`' to cause *var2* to be bound to the successive indices (starting at 0) of the elements.

This clause type is taken from older versions of the `loop` macro, and is not present in modern Common Lisp. The '`using (sequence ...)`' term of the older macros is not supported.

`for` *var* `being the elements of-ref` *sequence*

This clause iterates over a sequence, with *var* a `setf`-able reference onto the elements; see `in-ref` above.

`for` *var* `being the symbols [of` *obarray*`]`

This clause iterates over symbols, either over all interned symbols or over all symbols in *obarray*. The loop is executed with *var* bound to each symbol in turn. The symbols are visited in an unspecified order.

As an example,

```
(loop for sym being the symbols
      when (fboundp sym)
      when (string-match "^map" (symbol-name sym))
      collect sym)
```

returns a list of all the functions whose names begin with '`map`'.

The Common Lisp words `external-symbols` and `present-symbols` are also recognized but are equivalent to `symbols` in Emacs Lisp.

Due to a minor implementation restriction, it will not work to have more than one `for` clause iterating over symbols, hash tables, keymaps, overlays, or intervals in a given `loop`. Fortunately, it would rarely if ever be useful to do so. It *is* legal to mix one of these types of clauses with other clauses like `for ... to` or `while`.

`for` *var* `being the hash-keys of` *hash-table*

This clause iterates over the entries in *hash-table*. For each hash table entry, *var* is bound to the entry's key. If you write '`the hash-values`' instead, *var* is bound to the values of the entries. The clause may be followed by the additional term '`using (hash-values` *var2*`)`' (where `hash-values` is the opposite word of the word following `the`) to cause *var* and *var2* to be bound to the two parts of each hash table entry.

`for` *var* `being the key-codes of` *keymap*

This clause iterates over the entries in *keymap*. In GNU Emacs 18 and 19, keymaps are either alists or vectors, and key-codes are integers or symbols. In Lucid Emacs 19, keymaps are a special new data type, and key-codes are symbols or lists of symbols. The iteration does not enter nested keymaps or inherited (parent) keymaps. You can use '`the key-bindings`' to access the commands bound to the keys rather than the key codes, and you can add a `using` clause to access both the codes and the bindings together.

`for` *var* `being the key-seqs of` *keymap*

This clause iterates over all key sequences defined by *keymap* and its nested keymaps, where *var* takes on values which are strings in Emacs 18 or vectors in Emacs 19. The strings or vectors are reused for each iteration, so you must copy them if you wish to keep them permanently. You can add a '`using (key-bindings ...)`' clause to get the command bindings as well.

`for` *var* `being the overlays [of` *buffer*`] ...`

This clause iterates over the Emacs 19 "overlays" or Lucid Emacs "extents" of a buffer (the clause `extents` is synonymous with `overlays`). Under Emacs 18, this clause iterates zero times. If the `of` term is omitted, the current buffer is used. This clause also accepts optional '`from` *pos*' and '`to` *pos*' terms, limiting the clause to overlays which overlap the specified region.

`for` *var* `being the intervals [of` *buffer*`] ...`

This clause iterates over all intervals of a buffer with constant text properties. The variable *var* will be bound to conses of start and end positions, where one start position is always equal to the previous end position. The clause allows of, from, to, and property terms, where the latter term restricts the search to just the specified property. The of term may specify either a buffer or a string. This clause is useful only in GNU Emacs 19; in other versions, all buffers and strings consist of a single interval.

for *var* being the frames
: This clause iterates over all frames, i.e., X window system windows open on Emacs files. This clause works only under Emacs 19. The clause screens is a synonym for frames. The frames are visited in next-frame order starting from selected-frame.

for *var* being the windows [of *frame*]
: This clause iterates over the windows (in the Emacs sense) of the current frame, or of the specified *frame*. (In Emacs 18 there is only ever one frame, and the of term is not allowed there.)

for *var* being the buffers
: This clause iterates over all buffers in Emacs. It is equivalent to 'for *var* in (buffer-list)'.

for *var* = *expr1* then *expr2*
: This clause does a general iteration. The first time through the loop, *var* will be bound to *expr1*. On the second and successive iterations it will be set by evaluating *expr2* (which may refer to the old value of *var*). For example, these two loops are effectively the same:

```
(loop for x on my-list by 'cddr do ...)
(loop for x = my-list then (cddr x) while x do ...)
```

Note that this type of for clause does not imply any sort of terminating condition; the above example combines it with a while clause to tell when to end the loop.

If you omit the then term, *expr1* is used both for the initial setting and for successive settings:

```
(loop for x = (random) when (> x 0) return x)
```

This loop keeps taking random numbers from the (random) function until it gets a positive one, which it then returns.

If you include several for clauses in a row, they are treated sequentially (as if by let* and setq). You can instead use the word and to link the clauses, in which case they are processed in parallel (as if by let and psetq).

```
(loop for x below 5 for y = nil then x collect (list x y))
     ⇒ ((0 nil) (1 1) (2 2) (3 3) (4 4))
(loop for x below 5 and y = nil then x collect (list x y))
     ⇒ ((0 nil) (1 0) (2 1) (3 2) (4 3))
```

In the first loop, y is set based on the value of x that was just set by the previous clause; in the second loop, x and y are set simultaneously so y is set based on the value of x left over from the previous time through the loop.

Another feature of the loop macro is *destructuring*, similar in concept to the destructuring provided by defmacro. The *var* part of any for clause can be given as a list of variables instead of a single variable. The values produced during loop execution must be lists; the values in the lists are stored in the corresponding variables.

```
(loop for (x y) in '((2 3) (4 5) (6 7)) collect (+ x y))
     ⇒ (5 9 13)
```

In loop destructuring, if there are more values than variables the trailing values are ignored, and if there are more variables than values the trailing variables get the value `nil`. If `nil` is used as a variable name, the corresponding values are ignored. Destructuring may be nested, and dotted lists of variables like `(x . y)` are allowed.

7.9.4 Iteration Clauses

Aside from `for` clauses, there are several other loop clauses that control the way the loop operates. They might be used by themselves, or in conjunction with one or more `for` clauses.

`repeat` *integer*

This clause simply counts up to the specified number using an internal temporary variable. The loops

```
(loop repeat n do ...)
(loop for temp to n do ...)
```

are identical except that the second one forces you to choose a name for a variable you aren't actually going to use.

`while` *condition*

This clause stops the loop when the specified condition (any Lisp expression) becomes `nil`. For example, the following two loops are equivalent, except for the implicit `nil` block that surrounds the second one:

```
(while cond forms...)
(loop while cond do forms...)
```

`until` *condition*

This clause stops the loop when the specified condition is true, i.e., non-`nil`.

`always` *condition*

This clause stops the loop when the specified condition is `nil`. Unlike `while`, it stops the loop using `return nil` so that the `finally` clauses are not executed. If all the conditions were non-`nil`, the loop returns `t`:

```
(if (loop for size in size-list always (> size 10))
    (some-big-sizes)
  (no-big-sizes))
```

`never` *condition*

This clause is like `always`, except that the loop returns `t` if any conditions were false, or `nil` otherwise.

`thereis` *condition*

This clause stops the loop when the specified form is non-`nil`; in this case, it returns that non-`nil` value. If all the values were `nil`, the loop returns `nil`.

7.9.5 Accumulation Clauses

These clauses cause the loop to accumulate information about the specified Lisp *form*. The accumulated result is returned from the loop unless overridden, say, by a `return` clause.

`collect` *form*
: This clause collects the values of *form* into a list. Several examples of `collect` appear elsewhere in this manual.

  The word `collecting` is a synonym for `collect`, and likewise for the other accumulation clauses.

`append` *form*
: This clause collects lists of values into a result list using `append`.

`nconc` *form*
: This clause collects lists of values into a result list by destructively modifying the lists rather than copying them.

`concat` *form*
: This clause concatenates the values of the specified *form* into a string. (It and the following clause are extensions to standard Common Lisp.)

`vconcat` *form*
: This clause concatenates the values of the specified *form* into a vector.

`count` *form*
: This clause counts the number of times the specified *form* evaluates to a non-`nil` value.

`sum` *form*
: This clause accumulates the sum of the values of the specified *form*, which must evaluate to a number.

`maximize` *form*
: This clause accumulates the maximum value of the specified *form*, which must evaluate to a number. The return value is undefined if `maximize` is executed zero times.

`minimize` *form*
: This clause accumulates the minimum value of the specified *form*.

Accumulation clauses can be followed by '`into` *var*' to cause the data to be collected into variable *var* (which is automatically `let`-bound during the loop) rather than an unnamed temporary variable. Also, `into` accumulations do not automatically imply a return value. The loop must use some explicit mechanism, such as `finally return`, to return the accumulated result.

It is legal for several accumulation clauses of the same type to accumulate into the same place. From Steele:

```
(loop for name in '(fred sue alice joe june)
      for kids in '((bob ken) () () (kris sunshine) ())
      collect name
      append kids)
     ⇒ (fred bob ken sue alice joe kris sunshine june)
```

7.9.6 Other Clauses

This section describes the remaining loop clauses.

`with` *var* `=` *value*

This clause binds a variable to a value around the loop, but otherwise leaves the variable alone during the loop. The following loops are basically equivalent:

```
(loop with x = 17 do ...)
(let ((x 17)) (loop do ...))
(loop for x = 17 then x do ...)
```

Naturally, the variable *var* might be used for some purpose in the rest of the loop. For example:

```
(loop for x in my-list  with res = nil  do (push x res)
      finally return res)
```

This loop inserts the elements of `my-list` at the front of a new list being accumulated in `res`, then returns the list `res` at the end of the loop. The effect is similar to that of a `collect` clause, but the list gets reversed by virtue of the fact that elements are being pushed onto the front of `res` rather than the end.

If you omit the `=` term, the variable is initialized to `nil`. (Thus the '`= nil`' in the above example is unnecessary.)

Bindings made by `with` are sequential by default, as if by `let*`. Just like `for` clauses, `with` clauses can be linked with `and` to cause the bindings to be made by `let` instead.

`if` *condition clause*

This clause executes the following loop clause only if the specified condition is true. The following *clause* should be an accumulation, `do`, `return`, `if`, or `unless` clause. Several clauses may be linked by separating them with `and`. These clauses may be followed by `else` and a clause or clauses to execute if the condition was false. The whole construct may optionally be followed by the word `end` (which may be used to disambiguate an `else` or `and` in a nested `if`).

The actual non-`nil` value of the condition form is available by the name `it` in the "then" part. For example:

```
(setq funny-numbers '(6 13 -1))
     ⇒ (6 13 -1)
(loop for x below 10
      if (oddp x)
        collect x into odds
        and if (memq x funny-numbers) return (cdr it) end
      else
        collect x into evens
      finally return (vector odds evens))
     ⇒ [(1 3 5 7 9) (0 2 4 6 8)]
(setq funny-numbers '(6 7 13 -1))
     ⇒ (6 7 13 -1)
(loop <same thing again>)
     ⇒ (13 -1)
```

Note the use of `and` to put two clauses into the "then" part, one of which is itself an `if` clause. Note also that `end`, while normally optional, was necessary here to make it clear that the `else` refers to the outermost `if` clause. In the first case, the loop returns a vector of lists of the odd and even values of *x*. In the second case, the odd number 7 is one of the `funny-numbers` so the loop returns early; the actual returned value is based on the result of the `memq` call.

`when` *`condition clause`*
: This clause is just a synonym for `if`.

`unless` *`condition clause`*
: The `unless` clause is just like `if` except that the sense of the condition is reversed.

`named` *`name`*
: This clause gives a name other than `nil` to the implicit block surrounding the loop. The *name* is the symbol to be used as the block name.

`initially [do]` *`forms`*`...`
: This keyword introduces one or more Lisp forms which will be executed before the loop itself begins (but after any variables requested by `for` or `with` have been bound to their initial values). `initially` clauses can appear anywhere; if there are several, they are executed in the order they appear in the loop. The keyword `do` is optional.

`finally [do]` *`forms`*`...`
: This introduces Lisp forms which will be executed after the loop finishes (say, on request of a `for` or `while`). `initially` and `finally` clauses may appear anywhere in the loop construct, but they are executed (in the specified order) at the beginning or end, respectively, of the loop.

`finally return` *`form`*
: This says that *form* should be executed after the loop is done to obtain a return value. (Without this, or some other clause like `collect` or `return`, the loop will simply return `nil`.) Variables bound by `for`, `with`, or `into` will still contain their final values when *form* is executed.

`do` *`forms`*`...`
: The word `do` may be followed by any number of Lisp expressions which are executed as an implicit `progn` in the body of the loop. Many of the examples in this section illustrate the use of `do`.

`return` *`form`*
: This clause causes the loop to return immediately. The following Lisp form is evaluated to give the return value of the `loop` form. The `finally` clauses, if any, are not executed. Of course, `return` is generally used inside an `if` or `unless`, as its use in a top-level loop clause would mean the loop would never get to "loop" more than once.

  The clause '`return` *`form`*' is equivalent to '`do (return` *`form`*`)`' (or `return-from` if the loop was named). The `return` clause is implemented a bit more efficiently, though.

While there is no high-level way to add user extensions to `loop` (comparable to `defsetf` for `setf`, say), this package does offer two properties called `cl-loop-handler` and `cl-loop-for-handler` which are functions to be called when a given symbol is encountered as a top-level loop clause or `for` clause, respectively. Consult the source code in file `cl-macs.el` for details.

This package's `loop` macro is compatible with that of Common Lisp, except that a few features are not implemented: `loop-finish` and data-type specifiers. Naturally, the `for` clauses which iterate over keymaps, overlays, intervals, frames, windows, and buffers are Emacs-specific extensions.

7.10 Multiple Values

Common Lisp functions can return zero or more results. Emacs Lisp functions, by contrast, always return exactly one result. This package makes no attempt to emulate Common Lisp multiple return values; Emacs versions of Common Lisp functions that return more than one value either return just the first value (as in `compiler-macroexpand`) or return a list of values (as in `get-setf-method`). This package *does* define placeholders for the Common Lisp functions that work with multiple values, but in Emacs Lisp these functions simply operate on lists instead. The `values` form, for example, is a synonym for `list` in Emacs.

`multiple-value-bind` (*var*...) *values-form forms*... [Special Form]

This form evaluates *values-form*, which must return a list of values. It then binds the *var*s to these respective values, as if by `let`, and then executes the body *forms*. If there are more *var*s than values, the extra *var*s are bound to `nil`. If there are fewer *var*s than values, the excess values are ignored.

`multiple-value-setq` (*var*...) *form* [Special Form]

This form evaluates *form*, which must return a list of values. It then sets the *var*s to these respective values, as if by `setq`. Extra *var*s or values are treated the same as in `multiple-value-bind`.

The older Quiroz package attempted a more faithful (but still imperfect) emulation of Common Lisp multiple values. The old method "usually" simulated true multiple values quite well, but under certain circumstances would leave spurious return values in memory where a later, unrelated `multiple-value-bind` form would see them.

Since a perfect emulation is not feasible in Emacs Lisp, this package opts to keep it as simple and predictable as possible.

8 Macros

This package implements the various Common Lisp features of `defmacro`, such as destructuring, `&environment`, and `&body`. Top-level `&whole` is not implemented for `defmacro` due to technical difficulties. See Section 5.2 [Argument Lists], page 4.

Destructuring is made available to the user by way of the following macro:

`destructuring-bind` *arglist expr forms...* [Special Form]

This macro expands to code which executes *forms*, with the variables in *arglist* bound to the list of values returned by *expr*. The *arglist* can include all the features allowed for `defmacro` argument lists, including destructuring. (The `&environment` keyword is not allowed.) The macro expansion will signal an error if *expr* returns a list of the wrong number of arguments or with incorrect keyword arguments.

This package also includes the Common Lisp `define-compiler-macro` facility, which allows you to define compile-time expansions and optimizations for your functions.

`define-compiler-macro` *name arglist forms...* [Special Form]

This form is similar to `defmacro`, except that it only expands calls to *name* at compile-time; calls processed by the Lisp interpreter are not expanded, nor are they expanded by the `macroexpand` function.

The argument list may begin with a `&whole` keyword and a variable. This variable is bound to the macro-call form itself, i.e., to a list of the form '`(name args...)`'. If the macro expander returns this form unchanged, then the compiler treats it as a normal function call. This allows compiler macros to work as optimizers for special cases of a function, leaving complicated cases alone.

For example, here is a simplified version of a definition that appears as a standard part of this package:

```
(define-compiler-macro member* (&whole form a list &rest keys)
  (if (and (null keys)
           (eq (car-safe a) 'quote)
           (not (floatp-safe (cadr a))))
      (list 'memq a list)
    form))
```

This definition causes `(member* a list)` to change to a call to the faster `memq` in the common case where *a* is a non-floating-point constant; if *a* is anything else, or if there are any keyword arguments in the call, then the original `member*` call is left intact. (The actual compiler macro for `member*` optimizes a number of other cases, including common `:test` predicates.)

`compiler-macroexpand` *form* [Function]

This function is analogous to `macroexpand`, except that it expands compiler macros rather than regular macros. It returns *form* unchanged if it is not a call to a function for which a compiler macro has been defined, or if that compiler macro decided to punt by returning its `&whole` argument. Like `macroexpand`, it expands repeatedly until it reaches a form for which no further expansion is possible.

See Section 7.5.4 [Macro Bindings], page 24, for descriptions of the `macrolet` and `symbol-macrolet` forms for making "local" macro definitions.

9 Declarations

Common Lisp includes a complex and powerful "declaration" mechanism that allows you to give the compiler special hints about the types of data that will be stored in particular variables, and about the ways those variables and functions will be used. This package defines versions of all the Common Lisp declaration forms: `declare`, `locally`, `proclaim`, `declaim`, and `the`.

Most of the Common Lisp declarations are not currently useful in Emacs Lisp, as the byte-code system provides little opportunity to benefit from type information, and `special` declarations are redundant in a fully dynamically-scoped Lisp. A few declarations are meaningful when the optimizing Emacs 19 byte compiler is being used, however. Under the earlier non-optimizing compiler, these declarations will effectively be ignored.

`proclaim` *decl-spec* [Function]

> This function records a "global" declaration specified by *decl-spec*. Since `proclaim` is a function, *decl-spec* is evaluated and thus should normally be quoted.

`declaim` *decl-specs...* [Special Form]

> This macro is like `proclaim`, except that it takes any number of *decl-spec* arguments, and the arguments are unevaluated and unquoted. The `declaim` macro also puts an `(eval-when (compile load eval) ...)` around the declarations so that they will be registered at compile-time as well as at run-time. (This is vital, since normally the declarations are meant to influence the way the compiler treats the rest of the file that contains the `declaim` form.)

`declare` *decl-specs...* [Special Form]

> This macro is used to make declarations within functions and other code. Common Lisp allows declarations in various locations, generally at the beginning of any of the many "implicit `progn`s" throughout Lisp syntax, such as function bodies, `let` bodies, etc. Currently the only declaration understood by `declare` is `special`.

`locally` *declarations... forms...* [Special Form]

> In this package, `locally` is no different from `progn`.

`the` *type form* [Special Form]

> Type information provided by `the` is ignored in this package; in other words, `(the type form)` is equivalent to *form*. Future versions of the optimizing byte-compiler may make use of this information.
>
> For example, `mapcar` can map over both lists and arrays. It is hard for the compiler to expand `mapcar` into an in-line loop unless it knows whether the sequence will be a list or an array ahead of time. With `(mapcar 'car (the vector foo))`, a future compiler would have enough information to expand the loop in-line. For now, Emacs Lisp will treat the above code as exactly equivalent to `(mapcar 'car foo)`.

Each *decl-spec* in a `proclaim`, `declaim`, or `declare` should be a list beginning with a symbol that says what kind of declaration it is. This package currently understands `special`, `inline`, `notinline`, `optimize`, and `warn` declarations. (The `warn` declaration is an extension of standard Common Lisp.) Other Common Lisp declarations, such as `type` and `ftype`, are silently ignored.

`special` Since all variables in Emacs Lisp are "special" (in the Common Lisp sense), `special` declarations are only advisory. They simply tell the optimizing byte compiler that the specified variables are intentionally being referred to without being bound in the body of the function. The compiler normally emits warnings for such references, since they could be typographical errors for references to local variables.

The declaration `(declare (special` *var1 var2*`))` is equivalent to `(defvar` *var1*`) (defvar` *var2*`)` in the optimizing compiler, or to nothing at all in older compilers (which do not warn for non-local references).

In top-level contexts, it is generally better to write `(defvar` *var*`)` than `(declaim (special` *var*`))`, since `defvar` makes your intentions clearer. But the older byte compilers can not handle `defvar`s appearing inside of functions, while `(declare (special` *var*`))` takes care to work correctly with all compilers.

`inline` The `inline` *decl-spec* lists one or more functions whose bodies should be expanded "in-line" into calling functions whenever the compiler is able to arrange for it. For example, the Common Lisp function `cadr` is declared `inline` by this package so that the form `(cadr` *x*`)` will expand directly into `(car (cdr` *x*`))` when it is called in user functions, for a savings of one (relatively expensive) function call.

The following declarations are all equivalent. Note that the `defsubst` form is a convenient way to define a function and declare it inline all at once, but it is available only in Emacs 19.

```
(declaim (inline foo bar))
(eval-when (compile load eval) (proclaim '(inline foo bar)))
(proclaim-inline foo bar)      ; Lucid Emacs only
(defsubst foo (...) ...)       ; instead of defun; Emacs 19 only
```

Note: This declaration remains in effect after the containing source file is done. It is correct to use it to request that a function you have defined should be inlined, but it is impolite to use it to request inlining of an external function.

In Common Lisp, it is possible to use `(declare (inline ...))` before a particular call to a function to cause just that call to be inlined; the current byte compilers provide no way to implement this, so `(declare (inline ...))` is currently ignored by this package.

`notinline`

The `notinline` declaration lists functions which should not be inlined after all; it cancels a previous `inline` declaration.

`optimize` This declaration controls how much optimization is performed by the compiler. Naturally, it is ignored by the earlier non-optimizing compilers.

The word `optimize` is followed by any number of lists like `(speed 3)` or `(safety 2)`. Common Lisp defines several optimization "qualities"; this package ignores all but `speed` and `safety`. The value of a quality should be an integer from

0 to 3, with 0 meaning "unimportant" and 3 meaning "very important." The default level for both qualities is 1.

In this package, with the Emacs 19 optimizing compiler, the `speed` quality is tied to the `byte-compile-optimize` flag, which is set to `nil` for `(speed 0)` and to `t` for higher settings; and the `safety` quality is tied to the `byte-compile-delete-errors` flag, which is set to `t` for `(safety 3)` and to `nil` for all lower settings. (The latter flag controls whether the compiler is allowed to optimize out code whose only side-effect could be to signal an error, e.g., rewriting `(progn foo bar)` to `bar` when it is not known whether `foo` will be bound at run-time.)

Note that even compiling with `(safety 0)`, the Emacs byte-code system provides sufficient checking to prevent real harm from being done. For example, barring serious bugs in Emacs itself, Emacs will not crash with a segmentation fault just because of an error in a fully-optimized Lisp program.

The `optimize` declaration is normally used in a top-level `proclaim` or `declaim` in a file; Common Lisp allows it to be used with `declare` to set the level of optimization locally for a given form, but this will not work correctly with the current version of the optimizing compiler. (The `declare` will set the new optimization level, but that level will not automatically be unset after the enclosing form is done.)

`warn` This declaration controls what sorts of warnings are generated by the byte compiler. Again, only the optimizing compiler generates warnings. The word `warn` is followed by any number of "warning qualities," similar in form to optimization qualities. The currently supported warning types are `redefine`, `callargs`, `unresolved`, and `free-vars`; in the current system, a value of 0 will disable these warnings and any higher value will enable them. See the documentation for the optimizing byte compiler for details.

10 Symbols

This package defines several symbol-related features that were missing from Emacs Lisp.

10.1 Property Lists

These functions augment the standard Emacs Lisp functions `get` and `put` for operating on properties attached to symbols. There are also functions for working with property lists as first-class data structures not attached to particular symbols.

`get*` *symbol property* **&optional** *default* [Function]

This function is like `get`, except that if the property is not found, the *default* argument provides the return value. (The Emacs Lisp `get` function always uses `nil` as the default; this package's `get*` is equivalent to Common Lisp's `get`.)

The `get*` function is `setf`-able; when used in this fashion, the *default* argument is allowed but ignored.

`remprop` *symbol property* [Function]

This function removes the entry for *property* from the property list of *symbol*. It returns a true value if the property was indeed found and removed, or `nil` if there was no such property. (This function was probably omitted from Emacs originally because, since `get` did not allow a *default*, it was very difficult to distinguish between a missing property and a property whose value was `nil`; thus, setting a property to `nil` was close enough to `remprop` for most purposes.)

`getf` *place property* **&optional** *default* [Function]

This function scans the list *place* as if it were a property list, i.e., a list of alternating property names and values. If an even-numbered element of *place* is found which is `eq` to *property*, the following odd-numbered element is returned. Otherwise, *default* is returned (or `nil` if no default is given).

In particular,

```
(get sym prop)  ≡  (getf (symbol-plist sym) prop)
```

It is legal to use `getf` as a `setf` place, in which case its *place* argument must itself be a legal `setf` place. The *default* argument, if any, is ignored in this context. The effect is to change (via `setcar`) the value cell in the list that corresponds to *property*, or to cons a new property-value pair onto the list if the property is not yet present.

```
(put sym prop val)  ≡  (setf (getf (symbol-plist sym) prop) val)
```

The `get` and `get*` functions are also `setf`-able. The fact that `default` is ignored can sometimes be useful:

```
(incf (get* 'foo 'usage-count 0))
```

Here, symbol `foo`'s `usage-count` property is incremented if it exists, or set to 1 (an incremented 0) otherwise.

When not used as a `setf` form, `getf` is just a regular function and its *place* argument can actually be any Lisp expression.

`remf` *place property* [Special Form]

This macro removes the property-value pair for *property* from the property list stored at *place*, which is any `setf`-able place expression. It returns true if the property was found. Note that if *property* happens to be first on the list, this will effectively do a `(setf place (cddr place))`, whereas if it occurs later, this simply uses `setcdr` to splice out the property and value cells.

10.3 Creating Symbols

These functions create unique symbols, typically for use as temporary variables.

`gensym` **&optional** *x* [Function]

This function creates a new, uninterned symbol (using `make-symbol`) with a unique name. (The name of an uninterned symbol is relevant only if the symbol is printed.) By default, the name is generated from an increasing sequence of numbers, '`G1000`', '`G1001`', '`G1002`', etc. If the optional argument *x* is a string, that string is used as a prefix instead of '`G`'. Uninterned symbols are used in macro expansions for temporary variables, to ensure that their names will not conflict with "real" variables in the user's code.

`*gensym-counter*` [Variable]

This variable holds the counter used to generate `gensym` names. It is incremented after each use by `gensym`. In Common Lisp this is initialized with 0, but this package initializes it with a random (time-dependent) value to avoid trouble when two files that each used `gensym` in their compilation are loaded together. (Uninterned symbols become interned when the compiler writes them out to a file and the Emacs loader loads them, so their names have to be treated a bit more carefully than in Common Lisp where uninterned symbols remain uninterned after loading.)

`gentemp` **&optional** *x* [Function]

This function is like `gensym`, except that it produces a new *interned* symbol. If the symbol that is generated already exists, the function keeps incrementing the counter and trying again until a new symbol is generated.

The Quiroz `cl.el` package also defined a `defkeyword` form for creating self-quoting keyword symbols. This package automatically creates all keywords that are called for by `&key` argument specifiers, and discourages the use of keywords as data unrelated to keyword arguments, so the `defkeyword` form has been discontinued.

12 Numbers

This section defines a few simple Common Lisp operations on numbers which were left out of Emacs Lisp.

12.2 Predicates on Numbers

These functions return `t` if the specified condition is true of the numerical argument, or `nil` otherwise.

`plusp` *number* [Function]

This predicate tests whether *number* is positive. It is an error if the argument is not a number.

`minusp` *number* [Function]

This predicate tests whether *number* is negative. It is an error if the argument is not a number.

`oddp` *integer* [Function]

This predicate tests whether *integer* is odd. It is an error if the argument is not an integer.

`evenp` *integer* [Function]

This predicate tests whether *integer* is even. It is an error if the argument is not an integer.

`floatp-safe` *object* [Function]

This predicate tests whether *object* is a floating-point number. On systems that support floating-point, this is equivalent to `floatp`. On other systems, this always returns `nil`.

12.4 Numerical Functions

These functions perform various arithmetic operations on numbers.

`abs` *number* [Function]

This function returns the absolute value of *number*. (Newer versions of Emacs provide this as a built-in function; this package defines `abs` only for Emacs 18 versions which don't provide it as a primitive.)

`expt` *base power* [Function]

This function returns *base* raised to the power of *number*. (Newer versions of Emacs provide this as a built-in function; this package defines `expt` only for Emacs 18 versions which don't provide it as a primitive.)

`gcd` **&rest** *integers* [Function]

This function returns the Greatest Common Divisor of the arguments. For one argument, it returns the absolute value of that argument. For zero arguments, it returns zero.

`lcm` **&rest** *integers* [Function]

This function returns the Least Common Multiple of the arguments. For one argument, it returns the absolute value of that argument. For zero arguments, it returns one.

`isqrt` *integer* [Function]

This function computes the "integer square root" of its integer argument, i.e., the greatest integer less than or equal to the true square root of the argument.

`floor*` *number* **&optional** *divisor* [Function]

This function implements the Common Lisp `floor` function. It is called `floor*` to avoid name conflicts with the simpler `floor` function built-in to Emacs 19.

With one argument, `floor*` returns a list of two numbers: The argument rounded down (toward minus infinity) to an integer, and the "remainder" which would have to be added back to the first return value to yield the argument again. If the argument is an integer *x*, the result is always the list `(x 0)`. If the argument is an Emacs 19 floating-point number, the first result is a Lisp integer and the second is a Lisp float between 0 (inclusive) and 1 (exclusive).

With two arguments, `floor*` divides *number* by *divisor*, and returns the floor of the quotient and the corresponding remainder as a list of two numbers. If `(floor* x y)` returns `(q r)`, then `q*y + r = x`, with *r* between 0 (inclusive) and *r* (exclusive). Also, note that `(floor* x)` is exactly equivalent to `(floor* x 1)`.

This function is entirely compatible with Common Lisp's `floor` function, except that it returns the two results in a list since Emacs Lisp does not support multiple-valued functions.

`ceiling*` *number* **&optional** *divisor* [Function]

This function implements the Common Lisp `ceiling` function, which is analogous to `floor` except that it rounds the argument or quotient of the arguments up toward plus infinity. The remainder will be between 0 and minus *r*.

`truncate*` *number* **&optional** *divisor* [Function]

This function implements the Common Lisp `truncate` function, which is analogous to `floor` except that it rounds the argument or quotient of the arguments toward zero. Thus it is equivalent to `floor*` if the argument or quotient is positive, or to `ceiling*` otherwise. The remainder has the same sign as *number*.

`round*` *number* **&optional** *divisor* [Function]

This function implements the Common Lisp `round` function, which is analogous to `floor` except that it rounds the argument or quotient of the arguments to the nearest integer. In the case of a tie (the argument or quotient is exactly halfway between two integers), it rounds to the even integer.

`mod*` *number divisor* [Function]

This function returns the same value as the second return value of `floor`.

`rem*` *number divisor* [Function]

This function returns the same value as the second return value of `truncate`.

These definitions are compatible with those in the Quiroz `cl.el` package, except that this package appends '`*`' to certain function names to avoid conflicts with existing Emacs 19 functions, and that the mechanism for returning multiple values is different.

12.9 Random Numbers

This package also provides an implementation of the Common Lisp random number generator. It uses its own additive-congruential algorithm, which is much more likely to give statistically clean random numbers than the simple generators supplied by many operating systems.

`random*` *number* **&optional** *state* [Function]

> This function returns a random nonnegative number less than *number*, and of the same type (either integer or floating-point). The *state* argument should be a `random-state` object which holds the state of the random number generator. The function modifies this state object as a side effect. If *state* is omitted, it defaults to the variable `*random-state*`, which contains a pre-initialized `random-state` object.

`*random-state*` [Variable]

> This variable contains the system "default" `random-state` object, used for calls to `random*` that do not specify an alternative state object. Since any number of programs in the Emacs process may be accessing `*random-state*` in interleaved fashion, the sequence generated from this variable will be irreproducible for all intents and purposes.

`make-random-state` **&optional** *state* [Function]

> This function creates or copies a `random-state` object. If *state* is omitted or `nil`, it returns a new copy of `*random-state*`. This is a copy in the sense that future sequences of calls to `(random* n)` and `(random* n s)` (where *s* is the new random-state object) will return identical sequences of random numbers.
>
> If *state* is a `random-state` object, this function returns a copy of that object. If *state* is `t`, this function returns a new `random-state` object seeded from the date and time. As an extension to Common Lisp, *state* may also be an integer in which case the new object is seeded from that integer; each different integer seed will result in a completely different sequence of random numbers.
>
> It is legal to print a `random-state` object to a buffer or file and later read it back with `read`. If a program wishes to use a sequence of pseudo-random numbers which can be reproduced later for debugging, it can call `(make-random-state t)` to get a new sequence, then print this sequence to a file. When the program is later rerun, it can read the original run's random-state from the file.

`random-state-p` *object* [Function]

> This predicate returns `t` if *object* is a `random-state` object, or `nil` otherwise.

12.10 Implementation Parameters

This package defines several useful constants having to with numbers.

`most-positive-fixnum` [Variable]

This constant equals the largest value a Lisp integer can hold. It is typically `2^23-1` or `2^25-1`.

`most-negative-fixnum` [Variable]

This constant equals the smallest (most negative) value a Lisp integer can hold.

The following parameters have to do with floating-point numbers. This package determines their values by exercising the computer's floating-point arithmetic in various ways. Because this operation might be slow, the code for initializing them is kept in a separate function that must be called before the parameters can be used.

`cl-float-limits` [Function]

This function makes sure that the Common Lisp floating-point parameters like `most-positive-float` have been initialized. Until it is called, these parameters will be `nil`. If this version of Emacs does not support floats (e.g., most versions of Emacs 18), the parameters will remain `nil`. If the parameters have already been initialized, the function returns immediately.

The algorithm makes assumptions that will be valid for most modern machines, but will fail if the machine's arithmetic is extremely unusual, e.g., decimal.

Since true Common Lisp supports up to four different floating-point precisions, it has families of constants like `most-positive-single-float`, `most-positive-double-float`, `most-positive-long-float`, and so on. Emacs has only one floating-point precision, so this package omits the precision word from the constants' names.

`most-positive-float` [Variable]

This constant equals the largest value a Lisp float can hold. For those systems whose arithmetic supports infinities, this is the largest *finite* value. For IEEE machines, the value is approximately `1.79e+308`.

`most-negative-float` [Variable]

This constant equals the most-negative value a Lisp float can hold. (It is assumed to be equal to `(- most-positive-float)`.)

`least-positive-float` [Variable]

This constant equals the smallest Lisp float value greater than zero. For IEEE machines, it is about `4.94e-324` if denormals are supported or `2.22e-308` if not.

`least-positive-normalized-float` [Variable]

This constant equals the smallest *normalized* Lisp float greater than zero, i.e., the smallest value for which IEEE denormalization will not result in a loss of precision. For IEEE machines, this value is about `2.22e-308`. For machines that do not support the concept of denormalization and gradual underflow, this constant will always equal `least-positive-float`.

`least-negative-float` [Variable]

This constant is the negative counterpart of `least-positive-float`.

`least-negative-normalized-float` [Variable]

This constant is the negative counterpart of `least-positive-normalized-float`.

`float-epsilon` [Variable]

This constant is the smallest positive Lisp float that can be added to 1.0 to produce a distinct value. Adding a smaller number to 1.0 will yield 1.0 again due to roundoff. For IEEE machines, epsilon is about `2.22e-16`.

`float-negative-epsilon` [Variable]

This is the smallest positive value that can be subtracted from 1.0 to produce a distinct value. For IEEE machines, it is about `1.11e-16`.

14 Sequences

Common Lisp defines a number of functions that operate on *sequences*, which are either lists, strings, or vectors. Emacs Lisp includes a few of these, notably `elt` and `length`; this package defines most of the rest.

14.1 Sequence Basics

Many of the sequence functions take keyword arguments; see Section 5.2 [Argument Lists], page 4. All keyword arguments are optional and, if specified, may appear in any order.

The `:key` argument should be passed either `nil`, or a function of one argument. This key function is used as a filter through which the elements of the sequence are seen; for example, `(find x y :key 'car)` is similar to `(assoc* x y)`: It searches for an element of the list whose `car` equals `x`, rather than for an element which equals `x` itself. If `:key` is omitted or `nil`, the filter is effectively the identity function.

The `:test` and `:test-not` arguments should be either `nil`, or functions of two arguments. The test function is used to compare two sequence elements, or to compare a search value with sequence elements. (The two values are passed to the test function in the same order as the original sequence function arguments from which they are derived, or, if they both come from the same sequence, in the same order as they appear in that sequence.) The `:test` argument specifies a function which must return true (non-`nil`) to indicate a match; instead, you may use `:test-not` to give a function which returns *false* to indicate a match. The default test function is `:test 'eql`.

Many functions which take *item* and `:test` or `:test-not` arguments also come in `-if` and `-if-not` varieties, where a *predicate* function is passed instead of *item*, and sequence elements match if the predicate returns true on them (or false in the case of `-if-not`). For example:

```
(remove* 0 seq :test '=)   ≡   (remove-if 'zerop seq)
```

to remove all zeros from sequence `seq`.

Some operations can work on a subsequence of the argument sequence; these function take `:start` and `:end` arguments which default to zero and the length of the sequence, respectively. Only elements between *start* (inclusive) and *end* (exclusive) are affected by the operation. The *end* argument may be passed `nil` to signify the length of the sequence; otherwise, both *start* and *end* must be integers, with `0 <= start <= end <= (length seq)`. If the function takes two sequence arguments, the limits are defined by keywords `:start1` and `:end1` for the first, and `:start2` and `:end2` for the second.

A few functions accept a `:from-end` argument, which, if non-`nil`, causes the operation to go from right-to-left through the sequence instead of left-to-right, and a `:count` argument, which specifies an integer maximum number of elements to be removed or otherwise processed.

The sequence functions make no guarantees about the order in which the `:test`, `:test-not`, and `:key` functions are called on various elements. Therefore, it is a bad idea to depend on side effects of these functions. For example, `:from-end` may cause the sequence to be scanned actually in reverse, or it may be scanned forwards but computing a result "as if" it were scanned backwards. (Some functions, like `mapcar*` and `every`, *do* specify exactly the order in which the function is called so side effects are perfectly acceptable in those cases.)

Strings in GNU Emacs 19 may contain "text properties" as well as character data. Except as noted, it is undefined whether or not text properties are preserved by sequence functions. For example, `(remove* ?A str)` may or may not preserve the properties of the characters copied from *str* into the result.

14.2 Mapping over Sequences

These functions "map" the function you specify over the elements of lists or arrays. They are all variations on the theme of the built-in function `mapcar`.

`mapcar*` *function seq* **&rest** *more-seqs* [Function]

This function calls *function* on successive parallel sets of elements from its argument sequences. Given a single *seq* argument it is equivalent to `mapcar`; given *n* sequences, it calls the function with the first elements of each of the sequences as the *n* arguments to yield the first element of the result list, then with the second elements, and so on. The mapping stops as soon as the shortest sequence runs out. The argument sequences may be any mixture of lists, strings, and vectors; the return sequence is always a list.

Common Lisp's `mapcar` accepts multiple arguments but works only on lists; Emacs Lisp's `mapcar` accepts a single sequence argument. This package's `mapcar*` works as a compatible superset of both.

`map` *result-type function seq* **&rest** *more-seqs* [Function]

This function maps *function* over the argument sequences, just like `mapcar*`, but it returns a sequence of type *result-type* rather than a list. *result-type* must be one of the following symbols: `vector`, `string`, `list` (in which case the effect is the same as for `mapcar*`), or `nil` (in which case the results are thrown away and `map` returns `nil`).

`maplist` *function list* **&rest** *more-lists* [Function]

This function calls *function* on each of its argument lists, then on the `cdr`s of those lists, and so on, until the shortest list runs out. The results are returned in the form of a list. Thus, `maplist` is like `mapcar*` except that it passes in the list pointers themselves rather than the `car`s of the advancing pointers.

`mapc` *function seq* **&rest** *more-seqs* [Function]

This function is like `mapcar*`, except that the values returned by *function* are ignored and thrown away rather than being collected into a list. The return value of `mapc` is *seq*, the first sequence.

`mapl` *function list* **&rest** *more-lists* [Function]

This function is like `maplist`, except that it throws away the values returned by *function*.

`mapcan` *function seq* **&rest** *more-seqs* [Function]

This function is like `mapcar*`, except that it concatenates the return values (which must be lists) using `nconc`, rather than simply collecting them into a list.

`mapcon` *function list* **&rest** *more-lists* [Function]

This function is like `maplist`, except that it concatenates the return values using `nconc`.

`some` *predicate seq* **&rest** *more-seqs* [Function]

This function calls *predicate* on each element of *seq* in turn; if *predicate* returns a non-`nil` value, `some` returns that value, otherwise it returns `nil`. Given several sequence arguments, it steps through the sequences in parallel until the shortest one runs out, just as in `mapcar*`. You can rely on the left-to-right order in which the elements are visited, and on the fact that mapping stops immediately as soon as *predicate* returns non-`nil`.

`every` *predicate seq* **&rest** *more-seqs* [Function]

This function calls *predicate* on each element of the sequence(s) in turn; it returns `nil` as soon as *predicate* returns `nil` for any element, or `t` if the predicate was true for all elements.

`notany` *predicate seq* **&rest** *more-seqs* [Function]

This function calls *predicate* on each element of the sequence(s) in turn; it returns `nil` as soon as *predicate* returns a non-`nil` value for any element, or `t` if the predicate was `nil` for all elements.

`notevery` *predicate seq* **&rest** *more-seqs* [Function]

This function calls *predicate* on each element of the sequence(s) in turn; it returns a non-`nil` value as soon as *predicate* returns `nil` for any element, or `t` if the predicate was true for all elements.

`reduce` *function seq* **&key** `:from-end :start :end` [Function]
`:initial-value :key`

This function combines the elements of *seq* using an associative binary operation. Suppose *function* is `*` and *seq* is the list `(2 3 4 5)`. The first two elements of the list are combined with `(* 2 3) = 6`; this is combined with the next element, `(* 6 4) = 24`, and that is combined with the final element: `(* 24 5) = 120`. Note that the `*` function happens to be self-reducing, so that `(* 2 3 4 5)` has the same effect as an explicit call to `reduce`.

If `:from-end` is true, the reduction is right-associative instead of left-associative:

```
(reduce '- '(1 2 3 4))
     ≡ (- (- (- 1 2) 3) 4) ⇒ -8
(reduce '- '(1 2 3 4) :from-end t)
     ≡ (- 1 (- 2 (- 3 4))) ⇒ -2
```

If `:key` is specified, it is a function of one argument which is called on each of the sequence elements in turn.

If `:initial-value` is specified, it is effectively added to the front (or rear in the case of `:from-end`) of the sequence. The `:key` function is *not* applied to the initial value.

If the sequence, including the initial value, has exactly one element then that element is returned without ever calling *function*. If the sequence is empty (and there is no initial value), then *function* is called with no arguments to obtain the return value.

All of these mapping operations can be expressed conveniently in terms of the `loop` macro. In compiled code, `loop` will be faster since it generates the loop as in-line code with no function calls.

14.3 Sequence Functions

This section describes a number of Common Lisp functions for operating on sequences.

`subseq` *sequence start* **&optional** *end* [Function]

This function returns a given subsequence of the argument *sequence*, which may be a list, string, or vector. The indices *start* and *end* must be in range, and *start* must be no greater than *end*. If *end* is omitted, it defaults to the length of the sequence. The return value is always a copy; it does not share structure with *sequence*.

As an extension to Common Lisp, *start* and/or *end* may be negative, in which case they represent a distance back from the end of the sequence. This is for compatibility with Emacs' `substring` function. Note that `subseq` is the *only* sequence function that allows negative *start* and *end*.

You can use `setf` on a `subseq` form to replace a specified range of elements with elements from another sequence. The replacement is done as if by `replace`, described below.

`concatenate` *result-type* **&rest** *seqs* [Function]

This function concatenates the argument sequences together to form a result sequence of type *result-type*, one of the symbols `vector`, `string`, or `list`. The arguments are always copied, even in cases such as `(concatenate 'list '(1 2 3))` where the result is identical to an argument.

`fill` *seq item* **&key** `:start :end` [Function]

This function fills the elements of the sequence (or the specified part of the sequence) with the value *item*.

`replace` *seq1 seq2* **&key** `:start1 :end1 :start2 :end2` [Function]

This function copies part of *seq2* into part of *seq1*. The sequence *seq1* is not stretched or resized; the amount of data copied is simply the shorter of the source and destination (sub)sequences. The function returns *seq1*.

If *seq1* and *seq2* are `eq`, then the replacement will work correctly even if the regions indicated by the start and end arguments overlap. However, if *seq1* and *seq2* are lists which share storage but are not `eq`, and the start and end arguments specify overlapping regions, the effect is undefined.

`remove*` *item seq* **&key** `:test :test-not :key :count :start :end :from-end` [Function]

This returns a copy of *seq* with all elements matching *item* removed. The result may share storage with or be `eq` to *seq* in some circumstances, but the original *seq* will not be modified. The `:test`, `:test-not`, and `:key` arguments define the matching test that is used; by default, elements `eql` to *item* are removed. The `:count` argument specifies the maximum number of matching elements that can be removed (only the leftmost *count* matches are removed). The `:start` and `:end` arguments specify a region in *seq* in which elements will be removed; elements outside that region are not matched or removed. The `:from-end` argument, if true, says that elements should be deleted from the end of the sequence rather than the beginning (this matters only if *count* was also specified).

`delete*` *item seq* **&key** `:test :test-not :key :count :start :end :from-end` [Function]

This deletes all elements of *seq* which match *item*. It is a destructive operation. Since Emacs Lisp does not support stretchable strings or vectors, this is the same as `remove*` for those sequence types. On lists, `remove*` will copy the list if necessary to preserve the original list, whereas `delete*` will splice out parts of the argument list. Compare `append` and `nconc`, which are analogous non-destructive and destructive list operations in Emacs Lisp.

The predicate-oriented functions `remove-if`, `remove-if-not`, `delete-if`, and `delete-if-not` are defined similarly.

`delete` *item list* [Function]

This MacLisp-compatible function deletes from *list* all elements which are `equal` to *item*. The `delete` function is built-in to Emacs 19; this package defines it equivalently in Emacs 18.

`remove` *item list* [Function]

This function removes from *list* all elements which are `equal` to *item*. This package defines it for symmetry with `delete`, even though `remove` is not built-in to Emacs 19.

`remq` *item list* [Function]

This function removes from *list* all elements which are `eq` to *item*. This package defines it for symmetry with `delq`, even though `remq` is not built-in to Emacs 19.

`remove-duplicates` *seq* **&key** `:test :test-not :key :start :end :from-end` [Function]

This function returns a copy of *seq* with duplicate elements removed. Specifically, if two elements from the sequence match according to the `:test`, `:test-not`, and `:key` arguments, only the rightmost one is retained. If `:from-end` is true, the leftmost one is retained instead. If `:start` or `:end` is specified, only elements within that subsequence are examined or removed.

`delete-duplicates` *seq* **&key** `:test :test-not :key :start :end :from-end` [Function]

This function deletes duplicate elements from *seq*. It is a destructive version of `remove-duplicates`.

`substitute` *new old seq* **&key** `:test :test-not :key :count :start :end :from-end` [Function]

This function returns a copy of *seq*, with all elements matching *old* replaced with *new*. The `:count`, `:start`, `:end`, and `:from-end` arguments may be used to limit the number of substitutions made.

`nsubstitute` *new old seq* **&key** `:test :test-not :key :count :start :end :from-end` [Function]

This is a destructive version of `substitute`; it performs the substitution using `setcar` or `aset` rather than by returning a changed copy of the sequence.

The `substitute-if`, `substitute-if-not`, `nsubstitute-if`, and `nsubstitute-if-not` functions are defined similarly. For these, a *predicate* is given in place of the *old* argument.

14.4 Searching Sequences

These functions search for elements or subsequences in a sequence. (See also `member*` and `assoc*`; see Chapter 15 [Lists], page 58.)

`find` *item seq* **&key** `:test :test-not :key :start :end :from-end` [Function]

This function searches *seq* for an element matching *item*. If it finds a match, it returns the matching element. Otherwise, it returns `nil`. It returns the leftmost match, unless `:from-end` is true, in which case it returns the rightmost match. The `:start` and `:end` arguments may be used to limit the range of elements that are searched.

`position` *item seq* **&key** `:test :test-not :key :start :end :from-end` [Function]

This function is like `find`, except that it returns the integer position in the sequence of the matching item rather than the item itself. The position is relative to the start of the sequence as a whole, even if `:start` is non-zero. The function returns `nil` if no matching element was found.

`count` *item seq* **&key** `:test :test-not :key :start :end` [Function]

This function returns the number of elements of *seq* which match *item*. The result is always a nonnegative integer.

The `find-if`, `find-if-not`, `position-if`, `position-if-not`, `count-if`, and `count-if-not` functions are defined similarly.

`mismatch` *seq1 seq2* **&key** `:test :test-not :key :start1 :end1 :start2 :end2 :from-end` [Function]

This function compares the specified parts of *seq1* and *seq2*. If they are the same length and the corresponding elements match (according to `:test`, `:test-not`, and `:key`), the function returns `nil`. If there is a mismatch, the function returns the index (relative to *seq1*) of the first mismatching element. This will be the leftmost pair of elements which do not match, or the position at which the shorter of the two otherwise-matching sequences runs out.

If `:from-end` is true, then the elements are compared from right to left starting at `(1- end1)` and `(1- end2)`. If the sequences differ, then one plus the index of the rightmost difference (relative to *seq1*) is returned.

An interesting example is `(mismatch str1 str2 :key 'upcase)`, which compares two strings case-insensitively.

`search` *seq1 seq2* **&key** `:test :test-not :key :from-end :start1 :end1 :start2 :end2` [Function]

This function searches *seq2* for a subsequence that matches *seq1* (or part of it specified by `:start1` and `:end1`.) Only matches which fall entirely within the region defined by `:start2` and `:end2` will be considered. The return value is the index of the leftmost element of the leftmost match, relative to the start of *seq2*, or `nil` if no matches were found. If `:from-end` is true, the function finds the *rightmost* matching subsequence.

14.5 Sorting Sequences

`sort*` *seq predicate* **&key** `:key` [Function]

This function sorts *seq* into increasing order as determined by using *predicate* to compare pairs of elements. *predicate* should return true (non-`nil`) if and only if its first argument is less than (not equal to) its second argument. For example, `<` and `string-lessp` are suitable predicate functions for sorting numbers and strings, respectively; `>` would sort numbers into decreasing rather than increasing order.

This function differs from Emacs' built-in `sort` in that it can operate on any type of sequence, not just lists. Also, it accepts a `:key` argument which is used to preprocess data fed to the *predicate* function. For example,

```
(setq data (sort data 'string-lessp :key 'downcase))
```

sorts *data*, a sequence of strings, into increasing alphabetical order without regard to case. A `:key` function of `car` would be useful for sorting association lists.

The `sort*` function is destructive; it sorts lists by actually rearranging the `cdr` pointers in suitable fashion.

`stable-sort` *seq predicate* **&key** `:key` [Function]

This function sorts *seq stably*, meaning two elements which are equal in terms of *predicate* are guaranteed not to be rearranged out of their original order by the sort.

In practice, `sort*` and `stable-sort` are equivalent in Emacs Lisp because the underlying `sort` function is stable by default. However, this package reserves the right to use non-stable methods for `sort*` in the future.

`merge` *type seq1 seq2 predicate* **&key** `:key` [Function]

This function merges two sequences *seq1* and *seq2* by interleaving their elements. The result sequence, of type *type* (in the sense of `concatenate`), has length equal to the sum of the lengths of the two input sequences. The sequences may be modified destructively. Order of elements within *seq1* and *seq2* is preserved in the interleaving; elements of the two sequences are compared by *predicate* (in the sense of `sort`) and the lesser element goes first in the result. When elements are equal, those from *seq1* precede those from *seq2* in the result. Thus, if *seq1* and *seq2* are both sorted according to *predicate*, then the result will be a merged sequence which is (stably) sorted according to *predicate*.

15 Lists

The functions described here operate on lists.

15.1 List Functions

This section describes a number of simple operations on lists, i.e., chains of cons cells.

`caddr` *x* [Function]

This function is equivalent to `(car (cdr (cdr x)))`. Likewise, this package defines all 28 `cxxxr` functions where *xxx* is up to four '`a`'s and/or '`d`'s. All of these functions are `setf`-able, and calls to them are expanded inline by the byte-compiler for maximum efficiency.

`first` *x* [Function]

This function is a synonym for `(car x)`. Likewise, the functions `second`, `third`, ..., through `tenth` return the given element of the list *x*.

`rest` *x* [Function]

This function is a synonym for `(cdr x)`.

`endp` *x* [Function]

Common Lisp defines this function to act like `null`, but signalling an error if `x` is neither a `nil` nor a cons cell. This package simply defines `endp` as a synonym for `null`.

`list-length` *x* [Function]

This function returns the length of list *x*, exactly like `(length x)`, except that if *x* is a circular list (where the cdr-chain forms a loop rather than terminating with `nil`), this function returns `nil`. (The regular `length` function would get stuck if given a circular list.)

`last` *x* **&optional** *n* [Function]

This function returns the last cons, or the *n*th-to-last cons, of the list *x*. If *n* is omitted it defaults to 1. The "last cons" means the first cons cell of the list whose `cdr` is not another cons cell. (For normal lists, the `cdr` of the last cons will be `nil`.) This function returns `nil` if *x* is `nil` or shorter than *n*. Note that the last *element* of the list is `(car (last x))`.

`butlast` *x* **&optional** *n* [Function]

This function returns the list *x* with the last element, or the last *n* elements, removed. If *n* is greater than zero it makes a copy of the list so as not to damage the original list. In general, `(append (butlast x n) (last x n))` will return a list equal to *x*.

`nbutlast` *x* **&optional** *n* [Function]

This is a version of `butlast` that works by destructively modifying the `cdr` of the appropriate element, rather than making a copy of the list.

`list*` *arg* **&rest** *others* [Function]

This function constructs a list of its arguments. The final argument becomes the `cdr` of the last cell constructed. Thus, `(list* a b c)` is equivalent to `(cons a (cons b c))`, and `(list* a b nil)` is equivalent to `(list a b)`.

(Note that this function really is called `list*` in Common Lisp; it is not a name invented for this package like `member*` or `defun*`.)

`ldiff` *list sublist* [Function]

If *sublist* is a sublist of *list*, i.e., is `eq` to one of the cons cells of *list*, then this function returns a copy of the part of *list* up to but not including *sublist*. For example, `(ldiff x (cddr x))` returns the first two elements of the list `x`. The result is a copy; the original *list* is not modified. If *sublist* is not a sublist of *list*, a copy of the entire *list* is returned.

`copy-list` *list* [Function]

This function returns a copy of the list *list*. It copies dotted lists like `(1 2 . 3)` correctly.

`copy-tree` *x* **&optional** *vecp* [Function]

This function returns a copy of the tree of cons cells *x*. Unlike `copy-sequence` (and its alias `copy-list`), which copies only along the `cdr` direction, this function copies (recursively) along both the `car` and the `cdr` directions. If *x* is not a cons cell, the function simply returns *x* unchanged. If the optional *vecp* argument is true, this function copies vectors (recursively) as well as cons cells.

`tree-equal` *x y* **&key** `:test :test-not :key` [Function]

This function compares two trees of cons cells. If *x* and *y* are both cons cells, their `car`s and `cdr`s are compared recursively. If neither *x* nor *y* is a cons cell, they are compared by `eql`, or according to the specified test. The `:key` function, if specified, is applied to the elements of both trees. See Chapter 14 [Sequences], page 51.

15.4 Substitution of Expressions

These functions substitute elements throughout a tree of cons cells. (See Section 14.3 [Sequence Functions], page 54, for the `substitute` function, which works on just the top-level elements of a list.)

`subst` *new old tree* **&key** `:test :test-not :key` [Function]

This function substitutes occurrences of *old* with *new* in *tree*, a tree of cons cells. It returns a substituted tree, which will be a copy except that it may share storage with the argument *tree* in parts where no substitutions occurred. The original *tree* is not modified. This function recurses on, and compares against *old*, both `car`s and `cdr`s of the component cons cells. If *old* is itself a cons cell, then matching cells in the tree are substituted as usual without recursively substituting in that cell. Comparisons with *old* are done according to the specified test (`eql` by default). The `:key` function is applied to the elements of the tree but not to *old*.

`nsubst` *new old tree* **&key** `:test :test-not :key` [Function]

This function is like `subst`, except that it works by destructive modification (by `setcar` or `setcdr`) rather than copying.

The `subst-if`, `subst-if-not`, `nsubst-if`, and `nsubst-if-not` functions are defined similarly.

`sublis` *alist tree* **&key** `:test :test-not :key` [Function]

This function is like `subst`, except that it takes an association list *alist* of *old*-*new* pairs. Each element of the tree (after applying the `:key` function, if any), is compared with the `car`s of *alist*; if it matches, it is replaced by the corresponding `cdr`.

`nsublis` *alist tree* **&key** `:test :test-not :key` [Function]

This is a destructive version of `sublis`.

15.5 Lists as Sets

These functions perform operations on lists which represent sets of elements.

`member` *item list* [Function]

This MacLisp-compatible function searches *list* for an element which is `equal` to *item*. The `member` function is built-in to Emacs 19; this package defines it equivalently in Emacs 18. See the following function for a Common-Lisp compatible version.

`member*` *item list* **&key** `:test :test-not :key` [Function]

This function searches *list* for an element matching *item*. If a match is found, it returns the cons cell whose `car` was the matching element. Otherwise, it returns `nil`. Elements are compared by `eql` by default; you can use the `:test`, `:test-not`, and `:key` arguments to modify this behavior. See Chapter 14 [Sequences], page 51.

Note that this function's name is suffixed by '`*`' to avoid the incompatible `member` function defined in Emacs 19. (That function uses `equal` for comparisons; it is equivalent to `(member* item list :test 'equal)`.)

The `member-if` and `member-if-not` functions analogously search for elements which satisfy a given predicate.

`tailp` *sublist list* [Function]

This function returns `t` if *sublist* is a sublist of *list*, i.e., if *sublist* is `eql` to *list* or to any of its `cdr`s.

`adjoin` *item list* **&key** `:test :test-not :key` [Function]

This function conses *item* onto the front of *list*, like `(cons item list)`, but only if *item* is not already present on the list (as determined by `member*`). If a `:key` argument is specified, it is applied to *item* as well as to the elements of *list* during the search, on the reasoning that *item* is "about" to become part of the list.

`union` *list1 list2* **&key** `:test :test-not :key` [Function]

This function combines two lists which represent sets of items, returning a list that represents the union of those two sets. The result list will contain all items which appear in *list1* or *list2*, and no others. If an item appears in both *list1* and *list2* it will be copied only once. If an item is duplicated in *list1* or *list2*, it is undefined whether or not that duplication will survive in the result list. The order of elements in the result list is also undefined.

`nunion` *list1 list2* **&key** `:test :test-not :key` [Function]
This is a destructive version of `union`; rather than copying, it tries to reuse the storage of the argument lists if possible.

`intersection` *list1 list2* **&key** `:test :test-not :key` [Function]
This function computes the intersection of the sets represented by *list1* and *list2*. It returns the list of items which appear in both *list1* and *list2*.

`nintersection` *list1 list2* **&key** `:test :test-not :key` [Function]
This is a destructive version of `intersection`. It tries to reuse storage of *list1* rather than copying. It does *not* reuse the storage of *list2*.

`set-difference` *list1 list2* **&key** `:test :test-not :key` [Function]
This function computes the "set difference" of *list1* and *list2*, i.e., the set of elements that appear in *list1* but *not* in *list2*.

`nset-difference` *list1 list2* **&key** `:test :test-not :key` [Function]
This is a destructive `set-difference`, which will try to reuse *list1* if possible.

`set-exclusive-or` *list1 list2* **&key** `:test :test-not :key` [Function]
This function computes the "set exclusive or" of *list1* and *list2*, i.e., the set of elements that appear in exactly one of *list1* and *list2*.

`nset-exclusive-or` *list1 list2* **&key** `:test :test-not :key` [Function]
This is a destructive `set-exclusive-or`, which will try to reuse *list1* and *list2* if possible.

`subsetp` *list1 list2* **&key** `:test :test-not :key` [Function]
This function checks whether *list1* represents a subset of *list2*, i.e., whether every element of *list1* also appears in *list2*.

15.6 Association Lists

An *association list* is a list representing a mapping from one set of values to another; any list whose elements are cons cells is an association list.

`assoc*` *item a-list* **&key** `:test :test-not :key` [Function]
This function searches the association list *a-list* for an element whose `car` matches (in the sense of `:test`, `:test-not`, and `:key`, or by comparison with `eql`) a given *item*. It returns the matching element, if any, otherwise `nil`. It ignores elements of *a-list* which are not cons cells. (This corresponds to the behavior of `assq` and `assoc` in Emacs Lisp; Common Lisp's `assoc` ignores `nil`s but considers any other non-cons elements of *a-list* to be an error.)

`rassoc*` *item a-list* **&key** `:test :test-not :key` [Function]
This function searches for an element whose `cdr` matches *item*. If *a-list* represents a mapping, this applies the inverse of the mapping to *item*.

`rassoc` *item a-list* [Function]
This function searches like `rassoc*` with a `:test` argument of `equal`. It is analogous to Emacs Lisp's standard `assoc` function, which derives from the MacLisp rather than the Common Lisp tradition.

The `assoc-if`, `assoc-if-not`, `rassoc-if`, and `rassoc-if-not` functions are defined similarly.

Two simple functions for constructing association lists are:

`acons` *key value alist* [Function]

This is equivalent to `(cons (cons key value) alist)`.

`pairlis` *keys values* **&optional** *alist* [Function]

This is equivalent to `(nconc (mapcar* 'cons keys values) alist)`.

16 Hash Tables

A *hash table* is a data structure that maps "keys" onto "values." Keys and values can be arbitrary Lisp data objects. Hash tables have the property that the time to search for a given key is roughly constant; simpler data structures like association lists take time proportional to the number of entries in the list.

`make-hash-table` **&key** `:test :size` [Function]

> This function creates and returns a hash-table object whose function for comparing elements is `:test` (`eql` by default), and which is allocated to fit about `:size` elements. The `:size` argument is purely advisory; the table will stretch automatically if you store more elements in it. If `:size` is omitted, a reasonable default is used.
>
> Common Lisp allows only `eq`, `eql`, `equal`, and `equalp` as legal values for the `:test` argument. In this package, any reasonable predicate function will work, though if you use something else you should check the details of the hashing function described below to make sure it is suitable for your predicate.
>
> Some versions of Emacs (like Lucid Emacs 19) include a built-in hash table type; in these versions, `make-hash-table` with a test of `eq` will use these built-in hash tables. In all other cases, it will return a hash-table object which takes the form of a list with an identifying "tag" symbol at the front. All of the hash table functions in this package can operate on both types of hash table; normally you will never know which type is being used.
>
> This function accepts the additional Common Lisp keywords `:rehash-size` and `:rehash-threshold`, but it ignores their values.

`gethash` *key table* **&optional** *default* [Function]

> This function looks up *key* in *table*. If *key* exists in the table, in the sense that it matches any of the existing keys according to the table's test function, then the associated value is returned. Otherwise, *default* (or `nil`) is returned.
>
> To store new data in the hash table, use `setf` on a call to `gethash`. If *key* already exists in the table, the corresponding value is changed to the stored value. If *key* does not already exist, a new entry is added to the table and the table is reallocated to a larger size if necessary. The *default* argument is allowed but ignored in this case. The situation is exactly analogous to that of `get*`; see Section 10.1 [Property Lists], page 44.

`remhash` *key table* [Function]

> This function removes the entry for *key* from *table*. If an entry was removed, it returns `t`. If *key* does not appear in the table, it does nothing and returns `nil`.

`clrhash` *table* [Function]

> This function removes all the entries from *table*, leaving an empty hash table.

`maphash` *function table* [Function]

> This function calls *function* for each entry in *table*. It passes two arguments to *function*, the key and the value of the given entry. The return value of *function* is ignored; *maphash* itself returns `nil`. See Section 7.9 [Loop Facility], page 29, for an alternate way of iterating over hash tables.

`hash-table-count` *table* [Function]

This function returns the number of entries in *table*. **Warning:** The current implementation of Lucid Emacs 19 hash-tables does not decrement the stored `count` when `remhash` removes an entry. Therefore, the return value of this function is not dependable if you have used `remhash` on the table and the table's test is `eq`. A slower, but reliable, way to count the entries is `(loop for x being the hash-keys of` *`table`* `count t)`.

`hash-table-p` *object* [Function]

This function returns `t` if *object* is a hash table, `nil` otherwise. It recognizes both types of hash tables (both Lucid Emacs built-in tables and tables implemented with special lists.)

Sometimes when dealing with hash tables it is useful to know the exact "hash function" that is used. This package implements hash tables using Emacs Lisp "obarrays," which are the same data structure that Emacs Lisp uses to keep track of symbols. Each hash table includes an embedded obarray. Key values given to `gethash` are converted by various means into strings, which are then looked up in the obarray using `intern` and `intern-soft`. The symbol, or "bucket," corresponding to a given key string includes as its `symbol-value` an association list of all key-value pairs which hash to that string. Depending on the test function, it is possible for many entries to hash to the same bucket. For example, if the test is `eql`, then the symbol `foo` and two separately built strings `"foo"` will create three entries in the same bucket. Search time is linear within buckets, so hash tables will be most effective if you arrange not to store too many things that hash the same.

The following algorithm is used to convert Lisp objects to hash strings:

- Strings are used directly as hash strings. (However, if the test function is `equalp`, strings are `downcase`d first.)
- Symbols are hashed according to their `symbol-name`.
- Integers are hashed into one of 16 buckets depending on their value modulo 16. Floating-point numbers are truncated to integers and hashed modulo 16.
- Cons cells are hashed according to their `car`s; nonempty vectors are hashed according to their first element.
- All other types of objects hash into a single bucket named `"*"`.

Thus, for example, searching among many buffer objects in a hash table will devolve to a (still fairly fast) linear-time search through a single bucket, whereas searching for different symbols will be very fast since each symbol will, in general, hash into its own bucket.

The size of the obarray in a hash table is automatically adjusted as the number of elements increases.

As a special case, `make-hash-table` with a `:size` argument of 0 or 1 will create a hash-table object that uses a single association list rather than an obarray of many lists. For very small tables this structure will be more efficient since lookup does not require converting the key to a string or looking it up in an obarray. However, such tables are guaranteed to take time proportional to their size to do a search.

19 Structures

The Common Lisp *structure* mechanism provides a general way to define data types similar to C's `struct` types. A structure is a Lisp object containing some number of *slots*, each of which can hold any Lisp data object. Functions are provided for accessing and setting the slots, creating or copying structure objects, and recognizing objects of a particular structure type.

In true Common Lisp, each structure type is a new type distinct from all existing Lisp types. Since the underlying Emacs Lisp system provides no way to create new distinct types, this package implements structures as vectors (or lists upon request) with a special "tag" symbol to identify them.

`defstruct` *name slots...* [Special Form]

The `defstruct` form defines a new structure type called *name*, with the specified *slots*. (The *slots* may begin with a string which documents the structure type.) In the simplest case, *name* and each of the *slots* are symbols. For example,

```
(defstruct person name age sex)
```

defines a struct type called `person` which contains three slots. Given a `person` object *p*, you can access those slots by calling `(person-name` *p*`)`, `(person-age` *p*`)`, and `(person-sex` *p*`)`. You can also change these slots by using `setf` on any of these place forms:

```
(incf (person-age birthday-boy))
```

You can create a new `person` by calling `make-person`, which takes keyword arguments `:name`, `:age`, and `:sex` to specify the initial values of these slots in the new object. (Omitting any of these arguments leaves the corresponding slot "undefined," according to the Common Lisp standard; in Emacs Lisp, such uninitialized slots are filled with `nil`.)

Given a `person`, `(copy-person` *p*`)` makes a new object of the same type whose slots are `eq` to those of *p*.

Given any Lisp object *x*, `(person-p` *x*`)` returns true if *x* looks like a `person`, false otherwise. (Again, in Common Lisp this predicate would be exact; in Emacs Lisp the best it can do is verify that *x* is a vector of the correct length which starts with the correct tag symbol.)

Accessors like `person-name` normally check their arguments (effectively using `person-p`) and signal an error if the argument is the wrong type. This check is affected by `(optimize (safety ...))` declarations. Safety level 1, the default, uses a somewhat optimized check that will detect all incorrect arguments, but may use an uninformative error message (e.g., "expected a vector" instead of "expected a `person`"). Safety level 0 omits all checks except as provided by the underlying `aref` call; safety levels 2 and 3 do rigorous checking that will always print a descriptive error message for incorrect inputs. See Chapter 9 [Declarations], page 41.

```
(setq dave (make-person :name "Dave" :sex 'male))
     ⇒ [cl-struct-person "Dave" nil male]
(setq other (copy-person dave))
     ⇒ [cl-struct-person "Dave" nil male]
```

```
(eq dave other)
     ⇒ nil
(eq (person-name dave) (person-name other))
     ⇒ t
(person-p dave)
     ⇒ t
(person-p [1 2 3 4])
     ⇒ nil
(person-p "Bogus")
     ⇒ nil
(person-p '[cl-struct-person counterfeit person object])
     ⇒ t
```

In general, *name* is either a name symbol or a list of a name symbol followed by any number of *struct options*; each *slot* is either a slot symbol or a list of the form '(*slot-name default-value slot-options*...)'. The *default-value* is a Lisp form which is evaluated any time an instance of the structure type is created without specifying that slot's value.

Common Lisp defines several slot options, but the only one implemented in this package is `:read-only`. A non-`nil` value for this option means the slot should not be `setf`-able; the slot's value is determined when the object is created and does not change afterward.

```
(defstruct person
  (name nil :read-only t)
  age
  (sex 'unknown))
```

Any slot options other than `:read-only` are ignored.

For obscure historical reasons, structure options take a different form than slot options. A structure option is either a keyword symbol, or a list beginning with a keyword symbol possibly followed by arguments. (By contrast, slot options are key-value pairs not enclosed in lists.)

```
(defstruct (person (:constructor create-person)
                   (:type list)
                   :named)
  name age sex)
```

The following structure options are recognized.

`:conc-name`
: The argument is a symbol whose print name is used as the prefix for the names of slot accessor functions. The default is the name of the struct type followed by a hyphen. The option `(:conc-name p-)` would change this prefix to `p-`. Specifying `nil` as an argument means no prefix, so that the slot names themselves are used to name the accessor functions.

`:constructor`
: In the simple case, this option takes one argument which is an alternate name to use for the constructor function. The default is `make-`*name*, e.g., `make-person`.

The above example changes this to `create-person`. Specifying `nil` as an argument means that no standard constructor should be generated at all.

In the full form of this option, the constructor name is followed by an arbitrary argument list. See Chapter 5 [Program Structure], page 4, for a description of the format of Common Lisp argument lists. All options, such as `&rest` and `&key`, are supported. The argument names should match the slot names; each slot is initialized from the corresponding argument. Slots whose names do not appear in the argument list are initialized based on the *default-value* in their slot descriptor. Also, `&optional` and `&key` arguments which don't specify defaults take their defaults from the slot descriptor. It is legal to include arguments which don't correspond to slot names; these are useful if they are referred to in the defaults for optional, keyword, or `&aux` arguments which *do* correspond to slots.

You can specify any number of full-format `:constructor` options on a structure. The default constructor is still generated as well unless you disable it with a simple-format `:constructor` option.

```
(defstruct
 (person
  (:constructor nil)   ; no default constructor
  (:constructor new-person (name sex &optional (age 0)))
  (:constructor new-hound (&key (name "Rover")
                                (dog-years 0)
                           &aux (age (* 7 dog-years))
                                (sex 'canine))))
 name age sex)
```

The first constructor here takes its arguments positionally rather than by keyword. (In official Common Lisp terminology, constructors that work By Order of Arguments instead of by keyword are called "BOA constructors." No, I'm not making this up.) For example, `(new-person "Jane" 'female)` generates a person whose slots are `"Jane"`, 0, and `female`, respectively.

The second constructor takes two keyword arguments, `:name`, which initializes the `name` slot and defaults to `"Rover"`, and `:dog-years`, which does not itself correspond to a slot but which is used to initialize the `age` slot. The `sex` slot is forced to the symbol `canine` with no syntax for overriding it.

`:copier`

The argument is an alternate name for the copier function for this type. The default is `copy-`*`name`*. `nil` means not to generate a copier function. (In this implementation, all copier functions are simply synonyms for `copy-sequence`.)

`:predicate`

The argument is an alternate name for the predicate which recognizes objects of this type. The default is *`name`*`-p`. `nil` means not to generate a predicate function. (If the `:type` option is used without the `:named` option, no predicate is ever generated.)

In true Common Lisp, `typep` is always able to recognize a structure object even if `:predicate` was used. In this package, `typep` simply looks for a function called *typename*-p, so it will work for structure types only if they used the default predicate name.

:include
: This option implements a very limited form of C++-style inheritance. The argument is the name of another structure type previously created with defstruct. The effect is to cause the new structure type to inherit all of the included structure's slots (plus, of course, any new slots described by this struct's slot descriptors). The new structure is considered a "specialization" of the included one. In fact, the predicate and slot accessors for the included type will also accept objects of the new type.

If there are extra arguments to the :include option after the included-structure name, these options are treated as replacement slot descriptors for slots in the included structure, possibly with modified default values. Borrowing an example from Steele:

```
(defstruct person name (age 0) sex)
     ⇒ person
(defstruct (astronaut (:include person (age 45)))
  helmet-size
  (favorite-beverage 'tang))
     ⇒ astronaut

(setq joe (make-person :name "Joe"))
     ⇒ [cl-struct-person "Joe" 0 nil]
(setq buzz (make-astronaut :name "Buzz"))
     ⇒ [cl-struct-astronaut "Buzz" 45 nil nil tang]

(list (person-p joe) (person-p buzz))
     ⇒ (t t)
(list (astronaut-p joe) (astronaut-p buzz))
     ⇒ (nil t)

(person-name buzz)
     ⇒ "Buzz"
(astronaut-name joe)
     ⇒ error: "astronaut-name accessing a non-astronaut"
```

Thus, if astronaut is a specialization of person, then every astronaut is also a person (but not the other way around). Every astronaut includes all the slots of a person, plus extra slots that are specific to astronauts. Operations that work on people (like person-name) work on astronauts just like other people.

:print-function
: In full Common Lisp, this option allows you to specify a function which is called to print an instance of the structure type. The Emacs Lisp system offers no hooks into the Lisp printer which would allow for such a feature, so this package simply ignores :print-function.

`:type`
: The argument should be one of the symbols `vector` or `list`. This tells which underlying Lisp data type should be used to implement the new structure type. Vectors are used by default, but `(:type list)` will cause structure objects to be stored as lists instead.

  The vector representation for structure objects has the advantage that all structure slots can be accessed quickly, although creating vectors is a bit slower in Emacs Lisp. Lists are easier to create, but take a relatively long time accessing the later slots.

`:named`
: This option, which takes no arguments, causes a characteristic "tag" symbol to be stored at the front of the structure object. Using `:type` without also using `:named` will result in a structure type stored as plain vectors or lists with no identifying features.

  The default, if you don't specify `:type` explicitly, is to use named vectors. Therefore, `:named` is only useful in conjunction with `:type`.

```
(defstruct (person1) name age sex)
(defstruct (person2 (:type list) :named) name age sex)
(defstruct (person3 (:type list)) name age sex)

(setq p1 (make-person1))
     ⇒ [cl-struct-person1 nil nil nil]
(setq p2 (make-person2))
     ⇒ (person2 nil nil nil)
(setq p3 (make-person3))
     ⇒ (nil nil nil)

(person1-p p1)
     ⇒ t
(person2-p p2)
     ⇒ t
(person3-p p3)
     ⇒ error: function person3-p undefined
```

  Since unnamed structures don't have tags, `defstruct` is not able to make a useful predicate for recognizing them. Also, accessors like `person3-name` will be generated but they will not be able to do any type checking. The `person3-name` function, for example, will simply be a synonym for `car` in this case. By contrast, `person2-name` is able to verify that its argument is indeed a `person2` object before proceeding.

`:initial-offset`
: The argument must be a nonnegative integer. It specifies a number of slots to be left "empty" at the front of the structure. If the structure is named, the tag appears at the specified position in the list or vector; otherwise, the first slot appears at that position. Earlier positions are filled with `nil` by the constructors and ignored otherwise. If the type `:include`s another type, then

`:initial-offset` specifies a number of slots to be skipped between the last slot of the included type and the first new slot.

Except as noted, the `defstruct` facility of this package is entirely compatible with that of Common Lisp.

24 Assertions and Errors

This section describes two macros that test *assertions*, i.e., conditions which must be true if the program is operating correctly. Assertions never add to the behavior of a Lisp program; they simply make "sanity checks" to make sure everything is as it should be.

If the optimization property `speed` has been set to 3, and `safety` is less than 3, then the byte-compiler will optimize away the following assertions. Because assertions might be optimized away, it is a bad idea for them to include side-effects.

`assert` *test-form* [*show-args string args...*] [Special Form]

This form verifies that *test-form* is true (i.e., evaluates to a non-`nil` value). If so, it returns `nil`. If the test is not satisfied, `assert` signals an error.

A default error message will be supplied which includes *test-form*. You can specify a different error message by including a *string* argument plus optional extra arguments. Those arguments are simply passed to `error` to signal the error.

If the optional second argument *show-args* is `t` instead of `nil`, then the error message (with or without *string*) will also include all non-constant arguments of the top-level *form*. For example:

```
(assert (> x 10) t "x is too small: %d")
```

This usage of *show-args* is an extension to Common Lisp. In true Common Lisp, the second argument gives a list of *places* which can be `setf`'d by the user before continuing from the error. Since Emacs Lisp does not support continuable errors, it makes no sense to specify *places*.

`check-type` *form type* [*string*] [Special Form]

This form verifies that *form* evaluates to a value of type *type*. If so, it returns `nil`. If not, `check-type` signals a `wrong-type-argument` error. The default error message lists the erroneous value along with *type* and *form* themselves. If *string* is specified, it is included in the error message in place of *type*. For example:

```
(check-type x (integer 1 *) "a positive integer")
```

See Section 6.1 [Type Predicates], page 10, for a description of the type specifiers that may be used for *type*.

Note that in Common Lisp, the first argument to `check-type` must be a *place* suitable for use by `setf`, because `check-type` signals a continuable error that allows the user to modify *place*.

The following error-related macro is also defined:

`ignore-errors` *forms...* [Special Form]

This executes *forms* exactly like a `progn`, except that errors are ignored during the *forms*. More precisely, if an error is signalled then `ignore-errors` immediately aborts execution of the *forms* and returns `nil`. If the *forms* complete successfully, `ignore-errors` returns the result of the last *form*.

Appendix A Efficiency Concerns

A.1 Macros

Many of the advanced features of this package, such as `defun*`, `loop`, and `setf`, are implemented as Lisp macros. In byte-compiled code, these complex notations will be expanded into equivalent Lisp code which is simple and efficient. For example, the forms

```
(incf i n)
(push x (car p))
```

are expanded at compile-time to the Lisp forms

```
(setq i (+ i n))
(setcar p (cons x (car p)))
```

which are the most efficient ways of doing these respective operations in Lisp. Thus, there is no performance penalty for using the more readable `incf` and `push` forms in your compiled code.

Interpreted code, on the other hand, must expand these macros every time they are executed. For this reason it is strongly recommended that code making heavy use of macros be compiled. (The features labelled “Special Form” instead of “Function” in this manual are macros.) A loop using `incf` a hundred times will execute considerably faster if compiled, and will also garbage-collect less because the macro expansion will not have to be generated, used, and thrown away a hundred times.

You can find out how a macro expands by using the `cl-prettyexpand` function.

`cl-prettyexpand` *form* **&optional** *full* [Function]

This function takes a single Lisp form as an argument and inserts a nicely formatted copy of it in the current buffer (which must be in Lisp mode so that indentation works properly). It also expands all Lisp macros which appear in the form. The easiest way to use this function is to go to the `*scratch*` buffer and type, say,

```
(cl-prettyexpand '(loop for x below 10 collect x))
```

and type *C-x C-e* immediately after the closing parenthesis; the expansion

```
(block nil
  (let* ((x 0)
         (G1004 nil))
    (while (< x 10)
      (setq G1004 (cons x G1004))
      (setq x (+ x 1)))
    (nreverse G1004)))
```

will be inserted into the buffer. (The `block` macro is expanded differently in the interpreter and compiler, so `cl-prettyexpand` just leaves it alone. The temporary variable `G1004` was created by `gensym`.)

If the optional argument *full* is true, then *all* macros are expanded, including `block`, `eval-when`, and compiler macros. Expansion is done as if *form* were a top-level form in a file being compiled. For example,

```
(cl-prettyexpand '(pushnew 'x list))
```

```
     ⊣ (setq list (adjoin 'x list))
(cl-prettyexpand '(pushnew 'x list) t)
     ⊣ (setq list (if (memq 'x list) list (cons 'x list)))
(cl-prettyexpand '(caddr (member* 'a list)) t)
     ⊣ (car (cdr (cdr (memq 'a list))))
```

Note that `adjoin`, `caddr`, and `member*` all have built-in compiler macros to optimize them in common cases.

A.2 Error Checking

Common Lisp compliance has in general not been sacrificed for the sake of efficiency. A few exceptions have been made for cases where substantial gains were possible at the expense of marginal incompatibility. One example is the use of `memq` (which is treated very efficiently by the byte-compiler) to scan for keyword arguments; this can become confused in rare cases when keyword symbols are used as both keywords and data values at once. This is extremely unlikely to occur in practical code, and the use of `memq` allows functions with keyword arguments to be nearly as fast as functions that use `&optional` arguments.

The Common Lisp standard (as embodied in Steele's book) uses the phrase "it is an error if" to indicate a situation which is not supposed to arise in complying programs; implementations are strongly encouraged but not required to signal an error in these situations. This package sometimes omits such error checking in the interest of compactness and efficiency. For example, `do` variable specifiers are supposed to be lists of one, two, or three forms; extra forms are ignored by this package rather than signalling a syntax error. The `endp` function is simply a synonym for `null` in this package. Functions taking keyword arguments will accept an odd number of arguments, treating the trailing keyword as if it were followed by the value `nil`.

Argument lists (as processed by `defun*` and friends) *are* checked rigorously except for the minor point just mentioned; in particular, keyword arguments are checked for validity, and `&allow-other-keys` and `:allow-other-keys` are fully implemented. Keyword validity checking is slightly time consuming (though not too bad in byte-compiled code); you can use `&allow-other-keys` to omit this check. Functions defined in this package such as `find` and `member*` do check their keyword arguments for validity.

A.3 Optimizing Compiler

The byte-compiler that comes with Emacs 18 normally fails to expand macros that appear in top-level positions in the file (i.e., outside of `defun`s or other enclosing forms). This would have disastrous consequences to programs that used such top-level macros as `defun*`, `eval-when`, and `defstruct`. To work around this problem, the *CL* package patches the Emacs 18 compiler to expand top-level macros. This patch will apply to your own macros, too, if they are used in a top-level context. The patch will not harm versions of the Emacs 18 compiler which have already had a similar patch applied, nor will it affect the optimizing Emacs 19 byte-compiler written by Jamie Zawinski and Hallvard Furuseth. The patch is applied to the byte compiler's code in Emacs' memory, *not* to the `bytecomp.elc` file stored on disk.

The Emacs 19 compiler (for Emacs 18) is available from various Emacs Lisp archive sites such as `archive.cis.ohio-state.edu`. Its use is highly recommended; many of the

Common Lisp macros emit code which can be improved by optimization. In particular, `block`s (whether explicit or implicit in constructs like `defun*` and `loop`) carry a fair run-time penalty; the optimizing compiler removes `block`s which are not actually referenced by `return` or `return-from` inside the block.

Appendix B Common Lisp Compatibility

Following is a list of all known incompatibilities between this package and Common Lisp as documented in Steele (2nd edition).

Certain function names, such as `member`, `assoc`, and `floor`, were already taken by (incompatible) Emacs Lisp functions; this package appends '`*`' to the names of its Common Lisp versions of these functions.

The word `defun*` is required instead of `defun` in order to use extended Common Lisp argument lists in a function. Likewise, `defmacro*` and `function*` are versions of those forms which understand full-featured argument lists. The `&whole` keyword does not work in `defmacro` argument lists (except inside recursive argument lists).

In order to allow an efficient implementation, keyword arguments use a slightly cheesy parser which may be confused if a keyword symbol is passed as the *value* of another keyword argument. (Specifically, `(memq :`*`keyword rest-of-arguments`*`)` is used to scan for `:`*`keyword`* among the supplied keyword arguments.)

The `eql` and `equal` predicates do not distinguish between IEEE floating-point plus and minus zero. The `equalp` predicate has several differences with Common Lisp; see Chapter 6 [Predicates], page 10.

The `setf` mechanism is entirely compatible, except that setf-methods return a list of five values rather than five values directly. Also, the new "`setf` function" concept (typified by `(defun (setf foo) ...)`) is not implemented.

The `do-all-symbols` form is the same as `do-symbols` with no *obarray* argument. In Common Lisp, this form would iterate over all symbols in all packages. Since Emacs obarrays are not a first-class package mechanism, there is no way for `do-all-symbols` to locate any but the default obarray.

The `loop` macro is complete except that `loop-finish` and type specifiers are unimplemented.

The multiple-value return facility treats lists as multiple values, since Emacs Lisp cannot support multiple return values directly. The macros will be compatible with Common Lisp if `values` or `values-list` is always used to return to a `multiple-value-bind` or other multiple-value receiver; if `values` is used without `multiple-value-...` or vice-versa the effect will be different from Common Lisp.

Many Common Lisp declarations are ignored, and others match the Common Lisp standard in concept but not in detail. For example, local `special` declarations, which are purely advisory in Emacs Lisp, do not rigorously obey the scoping rules set down in Steele's book.

The variable `*gensym-counter*` starts out with a pseudo-random value rather than with zero. This is to cope with the fact that generated symbols become interned when they are written to and loaded back from a file.

The `defstruct` facility is compatible, except that structures are of type `:type vector :named` by default rather than some special, distinct type. Also, the `:type` slot option is ignored.

The second argument of `check-type` is treated differently.

Appendix C Old CL Compatibility

Following is a list of all known incompatibilities between this package and the older Quiroz `cl.el` package.

This package's emulation of multiple return values in functions is incompatible with that of the older package. That package attempted to come as close as possible to true Common Lisp multiple return values; unfortunately, it could not be 100% reliable and so was prone to occasional surprises if used freely. This package uses a simpler method, namely replacing multiple values with lists of values, which is more predictable though more noticeably different from Common Lisp.

The `defkeyword` form and `keywordp` function are not implemented in this package.

The `member`, `floor`, `ceiling`, `truncate`, `round`, `mod`, and `rem` functions are suffixed by '`*`' in this package to avoid collision with existing functions in Emacs 18 or Emacs 19. The older package simply redefined these functions, overwriting the built-in meanings and causing serious portability problems with Emacs 19. (Some more recent versions of the Quiroz package changed the names to `cl-member`, etc.; this package defines the latter names as aliases for `member*`, etc.)

Certain functions in the old package which were buggy or inconsistent with the Common Lisp standard are incompatible with the conforming versions in this package. For example, `eql` and `member` were synonyms for `eq` and `memq` in that package, `setf` failed to preserve correct order of evaluation of its arguments, etc.

Finally, unlike the older package, this package is careful to prefix all of its internal names with `cl-`. Except for a few functions which are explicitly defined as additional features (such as `floatp-safe` and `letf`), this package does not export any non-'`cl-`' symbols which are not also part of Common Lisp.

C.1 The `cl-compat` package

The *CL* package includes emulations of some features of the old `cl.el`, in the form of a compatibility package `cl-compat`. To use it, put `(require 'cl-compat)` in your program.

The old package defined a number of internal routines without `cl-` prefixes or other annotations. Call to these routines may have crept into existing Lisp code. `cl-compat` provides emulations of the following internal routines: `pair-with-newsyms`, `zip-lists`, `unzip-lists`, `reassemble-arglists`, `duplicate-symbols-p`, `safe-idiv`.

Some `setf` forms translated into calls to internal functions that user code might call directly. The functions `setnth`, `setnthcdr`, and `setelt` fall in this category; they are defined by `cl-compat`, but the best fix is to change to use `setf` properly.

The `cl-compat` file defines the keyword functions `keywordp`, `keyword-of`, and `defkeyword`, which are not defined by the new *CL* package because the use of keywords as data is discouraged.

The `build-klist` mechanism for parsing keyword arguments is emulated by `cl-compat`; the `with-keyword-args` macro is not, however, and in any case it's best to change to use the more natural keyword argument processing offered by `defun*`.

Multiple return values are treated differently by the two Common Lisp packages. The old package's method was more compatible with true Common Lisp, though it used heuristics that caused it to report spurious multiple return values in certain cases. The `cl-compat` package defines a set of multiple-value macros that are compatible with the old CL package; again, they are heuristic in nature, but they are guaranteed to work in any case where the old package's macros worked. To avoid name collision with the "official" multiple-value facilities, the ones in `cl-compat` have capitalized names: `Values`, `Values-list`, `Multiple-value-bind`, etc.

The functions `cl-floor`, `cl-ceiling`, `cl-truncate`, and `cl-round` are defined by `cl-compat` to use the old-style multiple-value mechanism, just as they did in the old package. The newer `floor*` and friends return their two results in a list rather than as multiple values. Note that older versions of the old package used the unadorned names `floor`, `ceiling`, etc.; `cl-compat` cannot use these names because they conflict with Emacs 19 built-ins.

Appendix D Porting Common Lisp

This package is meant to be used as an extension to Emacs Lisp, not as an Emacs implementation of true Common Lisp. Some of the remaining differences between Emacs Lisp and Common Lisp make it difficult to port large Common Lisp applications to Emacs. For one, some of the features in this package are not fully compliant with ANSI or Steele; see Appendix B [Common Lisp Compatibility], page 75. But there are also quite a few features that this package does not provide at all. Here are some major omissions that you will want watch out for when bringing Common Lisp code into Emacs.

- Case-insensitivity. Symbols in Common Lisp are case-insensitive by default. Some programs refer to a function or variable as `foo` in one place and `Foo` or `FOO` in another. Emacs Lisp will treat these as three distinct symbols.

  Some Common Lisp code is written in all upper-case. While Emacs is happy to let the program's own functions and variables use this convention, calls to Lisp builtins like `if` and `defun` will have to be changed to lower-case.

- Lexical scoping. In Common Lisp, function arguments and `let` bindings apply only to references physically within their bodies (or within macro expansions in their bodies). Emacs Lisp, by contrast, uses *dynamic scoping* wherein a binding to a variable is visible even inside functions called from the body.

  Variables in Common Lisp can be made dynamically scoped by declaring them `special` or using `defvar`. In Emacs Lisp it is as if all variables were declared `special`.

  Often you can use code that was written for lexical scoping even in a dynamically scoped Lisp, but not always. Here is an example of a Common Lisp code fragment that would fail in Emacs Lisp:

  ```
  (defun map-odd-elements (func list)
    (loop for x in list
          for flag = t then (not flag)
          collect (if flag x (funcall func x))))

  (defun add-odd-elements (list x)
    (map-odd-elements (function (lambda (a) (+ a x))) list))
  ```

  In Common Lisp, the two functions' usages of `x` are completely independent. In Emacs Lisp, the binding to `x` made by `add-odd-elements` will have been hidden by the binding in `map-odd-elements` by the time the `(+ a x)` function is called.

  (This package avoids such problems in its own mapping functions by using names like `cl-x` instead of `x` internally; as long as you don't use the `cl-` prefix for your own variables no collision can occur.)

  See Section 7.5.2 [Lexical Bindings], page 21, for a description of the `lexical-let` form which establishes a Common Lisp-style lexical binding, and some examples of how it differs from Emacs' regular `let`.

- Common Lisp allows the shorthand `#'x` to stand for `(function x)`, just as `'x` stands for `(quote x)`. In Common Lisp, one traditionally uses `#'` notation when referring to the name of a function. In Emacs Lisp, it works just as well to use a regular quote:

  ```
  (loop for x in y by #'cddr collect (mapcar #'plusp x))  ; Common Lisp
  ```

```
(loop for x in y by 'cddr collect (mapcar 'plusp x))    ; Emacs Lisp
```

When `#'` introduces a `lambda` form, it is best to write out `(function ...)` longhand in Emacs Lisp. You can use a regular quote, but then the byte-compiler won't know that the `lambda` expression is code that can be compiled.

```
(mapcar #'(lambda (x) (* x 2)) list)            ; Common Lisp
(mapcar (function (lambda (x) (* x 2))) list)   ; Emacs Lisp
```

Lucid Emacs supports `#'` notation starting with version 19.8.

- The "backquote" feature uses a different syntax in Emacs Lisp.

```
(defmacro foo (v &rest body) `(let ((,v 0)) @,body))  ; Common Lisp
(defmacro foo (v &rest body) (` (let (((, v) 0)) (@, body)))  ; Emacs
```

- Reader macros. Common Lisp includes a second type of macro that works at the level of individual characters. For example, Common Lisp implements the quote notation by a reader macro called `'`, whereas Emacs Lisp's parser just treats quote as a special case. Some Lisp packages use reader macros to create special syntaxes for themselves, which the Emacs parser is incapable of reading.

  The lack of reader macros, incidentally, is the reason behind Emacs Lisp's unusual backquote syntax. Since backquotes are implemented as a Lisp package and not built-in to the Emacs parser, they are forced to use a regular macro named `` ` `` which is used with the standard function/macro call notation.

- Other syntactic features. Common Lisp provides a number of notations beginning with `#` that the Emacs Lisp parser won't understand. For example, '`#| ... |#`' is an alternate comment notation, and '`#+lucid (foo)`' tells the parser to ignore the `(foo)` except in Lucid Common Lisp.

- Packages. In Common Lisp, symbols are divided into *packages*. Symbols that are Lisp built-ins are typically stored in one package; symbols that are vendor extensions are put in another, and each application program would have a package for its own symbols. Certain symbols are "exported" by a package and others are internal; certain packages "use" or import the exported symbols of other packages. To access symbols that would not normally be visible due to this importing and exporting, Common Lisp provides a syntax like `package:symbol` or `package::symbol`.

  Emacs Lisp has a single namespace for all interned symbols, and then uses a naming convention of putting a prefix like `cl-` in front of the name. Some Emacs packages adopt the Common Lisp-like convention of using `cl:` or `cl::` as the prefix. However, the Emacs parser does not understand colons and just treats them as part of the symbol name. Thus, while `mapcar` and `lisp:mapcar` may refer to the same symbol in Common Lisp, they are totally distinct in Emacs Lisp. Common Lisp programs which refer to a symbol by the full name sometimes and the short name other times will not port cleanly to Emacs.

  Emacs Lisp does have a concept of "obarrays," which are package-like collections of symbols, but this feature is not strong enough to be used as a true package mechanism.

- Keywords. The notation `:test-not` in Common Lisp really is a shorthand for `keyword:test-not`; keywords are just symbols in a built-in `keyword` package with the special property that all its symbols are automatically self-evaluating. Common Lisp programs often use keywords liberally to avoid having to use quotes.

In Emacs Lisp a keyword is just a symbol whose name begins with a colon; since the Emacs parser does not treat them specially, they have to be explicitly made self-evaluating by a statement like `(setq :test-not ':test-not)`. This package arranges to execute such a statement whenever `defun*` or some other form sees a keyword being used as an argument. Common Lisp code that assumes that a symbol `:mumble` will be self-evaluating even though it was never introduced by a `defun*` will have to be fixed.

- The `format` function is quite different between Common Lisp and Emacs Lisp. It takes an additional "destination" argument before the format string. A destination of `nil` means to format to a string as in Emacs Lisp; a destination of `t` means to write to the terminal (similar to `message` in Emacs). Also, format control strings are utterly different; `~` is used instead of `%` to introduce format codes, and the set of available codes is much richer. There are no notations like `\n` for string literals; instead, `format` is used with the "newline" format code, `~%`. More advanced formatting codes provide such features as paragraph filling, case conversion, and even loops and conditionals.

  While it would have been possible to implement most of Common Lisp `format` in this package (under the name `format*`, of course), it was not deemed worthwhile. It would have required a huge amount of code to implement even a decent subset of `format*`, yet the functionality it would provide over Emacs Lisp's `format` would rarely be useful.
- Vector constants use square brackets in Emacs Lisp, but `#(a b c)` notation in Common Lisp. To further complicate matters, Emacs 19 introduces its own `#(` notation for something entirely different—strings with properties.
- Characters are distinct from integers in Common Lisp. The notation for character constants is also different: `#\A` instead of `?A`. Also, `string=` and `string-equal` are synonyms in Emacs Lisp whereas the latter is case-insensitive in Common Lisp.
- Data types. Some Common Lisp data types do not exist in Emacs Lisp. Rational numbers and complex numbers are not present, nor are large integers (all integers are "fixnums"). All arrays are one-dimensional. There are no readtables or pathnames; streams are a set of existing data types rather than a new data type of their own. Hash tables, random-states, structures, and packages (obarrays) are built from Lisp vectors or lists rather than being distinct types.
- The Common Lisp Object System (CLOS) is not implemented, nor is the Common Lisp Condition System.
- Common Lisp features that are completely redundant with Emacs Lisp features of a different name generally have not been implemented. For example, Common Lisp writes `defconstant` where Emacs Lisp uses `defconst`. Similarly, `make-list` takes its arguments in different ways in the two Lisps but does exactly the same thing, so this package has not bothered to implement a Common Lisp-style `make-list`.
- A few more notable Common Lisp features not included in this package: `compiler-let`, `tagbody`, `prog`, `ldb/dpb`, `parse-integer`, `cerror`.
- Recursion. While recursion works in Emacs Lisp just like it does in Common Lisp, various details of the Emacs Lisp system and compiler make recursion much less efficient than it is in most Lisps. Some schools of thought prefer to use recursion in Lisp over other techniques; they would sum a list of numbers using something like

  ```
  (defun sum-list (list)
    (if list
  ```

```
        (+ (car list) (sum-list (cdr list)))
      0))
```

where a more iteratively-minded programmer might write one of these forms:

```
(let ((total 0)) (dolist (x my-list) (incf total x)) total)
(loop for x in my-list sum x)
```

While this would be mainly a stylistic choice in most Common Lisps, in Emacs Lisp you should be aware that the iterative forms are much faster than recursion. Also, Lisp programmers will want to note that the current Emacs Lisp compiler does not optimize tail recursion.

Function Index

A

B

C

D

E

F

G

H

I

L

M

N

O

P

R

S

T

U

W

Variable Index

*

F

L

M

Table of Contents