Common Lisp Extensions

# Common Lisp Extensions

## Table of Contents

* [GNU Emacs Common Lisp Emulation](#Top)
* [1 Overview](#Overview)
  + [1.1 Usage](#Usage)
  + [1.2 Organization](#Organization)
  + [1.3 Naming Conventions](#Naming-Conventions)
* [2 Program Structure](#Program-Structure)
  + [2.1 Argument Lists](#Argument-Lists)
  + [2.2 Time of Evaluation](#Time-of-Evaluation)
* [3 Predicates](#Predicates)
  + [3.1 Type Predicates](#Type-Predicates)
  + [3.2 Equality Predicates](#Equality-Predicates)
* [4 Control Structure](#Control-Structure)
  + [4.1 Assignment](#Assignment)
  + [4.2 Generalized Variables](#Generalized-Variables)
    - [4.2.1 Setf Extensions](#Setf-Extensions)
    - [4.2.2 Modify Macros](#Modify-Macros)
  + [4.3 Variable Bindings](#Variable-Bindings)
    - [4.3.1 Dynamic Bindings](#Dynamic-Bindings)
    - [4.3.2 Function Bindings](#Function-Bindings)
    - [4.3.3 Macro Bindings](#Macro-Bindings)
  + [4.4 Conditionals](#Conditionals)
  + [4.5 Blocks and Exits](#Blocks-and-Exits)
  + [4.6 Iteration](#Iteration)
  + [4.7 Loop Facility](#Loop-Facility)
    - [4.7.1 Loop Basics](#Loop-Basics)
    - [4.7.2 Loop Examples](#Loop-Examples)
    - [4.7.3 For Clauses](#For-Clauses)
    - [4.7.4 Iteration Clauses](#Iteration-Clauses)
    - [4.7.5 Accumulation Clauses](#Accumulation-Clauses)
    - [4.7.6 Other Clauses](#Other-Clauses)
  + [4.8 Multiple Values](#Multiple-Values)
* [5 Macros](#Macros)
* [6 Declarations](#Declarations)
* [7 Symbols](#Symbols)
  + [7.1 Property Lists](#Property-Lists)
  + [7.2 Creating Symbols](#Creating-Symbols)
* [8 Numbers](#Numbers)
  + [8.1 Predicates on Numbers](#Predicates-on-Numbers)
  + [8.2 Numerical Functions](#Numerical-Functions)
  + [8.3 Random Numbers](#Random-Numbers)
  + [8.4 Implementation Parameters](#Implementation-Parameters)
* [9 Sequences](#Sequences)
  + [9.1 Sequence Basics](#Sequence-Basics)
  + [9.2 Mapping over Sequences](#Mapping-over-Sequences)
  + [9.3 Sequence Functions](#Sequence-Functions)
  + [9.4 Searching Sequences](#Searching-Sequences)
  + [9.5 Sorting Sequences](#Sorting-Sequences)
* [10 Lists](#Lists)
  + [10.1 List Functions](#List-Functions)
  + [10.2 Substitution of Expressions](#Substitution-of-Expressions)
  + [10.3 Lists as Sets](#Lists-as-Sets)
  + [10.4 Association Lists](#Association-Lists)
* [11 Structures](#Structures)
* [12 Assertions and Errors](#Assertions)
* [Appendix A Efficiency Concerns](#Efficiency-Concerns)
  + [A.1 Macros](#Efficiency-Concerns)
  + [A.2 Error Checking](#Efficiency-Concerns)
  + [A.3 Compiler Optimizations](#Efficiency-Concerns)
* [Appendix B Common Lisp Compatibility](#Common-Lisp-Compatibility)
* [Appendix C Porting Common Lisp](#Porting-Common-Lisp)
* [Appendix D Obsolete Features](#Obsolete-Features)
  + [D.1 Obsolete Lexical Binding](#Obsolete-Lexical-Binding)
  + [D.2 Obsolete Macros](#Obsolete-Macros)
  + [D.3 Obsolete Ways to Customize Setf](#Obsolete-Setf-Customization)
* [Appendix E GNU Free Documentation License](#GNU-Free-Documentation-License)
* [Function Index](#Function-Index)
* [Variable Index](#Variable-Index)

Next: [Overview](#Overview), Up: [(dir)](#dir)

## GNU Emacs Common Lisp Emulation

This file documents the GNU Emacs Common Lisp emulation package.

Copyright © 1993, 2001–2016 Free Software Foundation, Inc.

Permission is granted to copy, distribute and/or modify this document under the terms of the GNU Free Documentation License, Version 1.3 or any later version published by the Free Software Foundation; with no Invariant Sections, with the Front-Cover Texts being “A GNU Manual”, and with the Back-Cover Texts as in (a) below. A copy of the license is included in the section entitled “GNU Free Documentation License”.

(a) The FSF's Back-Cover Text is: “You have the freedom to copy and modify this GNU manual.”

* [Overview](#Overview): Basics, usage, organization, naming conventions.
* [Program Structure](#Program-Structure): Arglists, cl-eval-when.
* [Predicates](#Predicates): Type predicates and equality predicates.
* [Control Structure](#Control-Structure): Assignment, conditionals, blocks, looping.
* [Macros](#Macros): Destructuring, compiler macros.
* [Declarations](#Declarations): cl-proclaim, cl-declare, etc.
* [Symbols](#Symbols): Property lists, creating symbols.
* [Numbers](#Numbers): Predicates, functions, random numbers.
* [Sequences](#Sequences): Mapping, functions, searching, sorting.
* [Lists](#Lists): Functions, substitution, sets, associations.
* [Structures](#Structures): cl-defstruct.
* [Assertions](#Assertions): Assertions and type checking.

Appendices

* [Efficiency Concerns](#Efficiency-Concerns): Hints and techniques.
* [Common Lisp Compatibility](#Common-Lisp-Compatibility): All known differences with Steele.
* [Porting Common Lisp](#Porting-Common-Lisp): Hints for porting Common Lisp code.
* [Obsolete Features](#Obsolete-Features): Obsolete features.
* [GNU Free Documentation License](#GNU-Free-Documentation-License): The license for this documentation.

Indexes

* [Function Index](#Function-Index): An entry for each documented function.
* [Variable Index](#Variable-Index): An entry for each documented variable.

Next: [Program Structure](#Program-Structure), Previous: [Top](#Top), Up: [Top](#Top)

## 1 Overview

This document describes a set of Emacs Lisp facilities borrowed from Common Lisp. All the facilities are described here in detail. While this document does not assume any prior knowledge of Common Lisp, it does assume a basic familiarity with Emacs Lisp.

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, it 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. (The separate package EIEIO implements a subset of CLOS functionality. See [Introduction](eieio.html#Top).)
* Other features cannot be implemented without modification to the Emacs Lisp interpreter itself, such as multiple return values, case-insensitive symbols, and complex numbers. This package generally makes no attempt to emulate these features.

This package was originally written by Dave Gillespie, daveg@synaptics.com, as a total rewrite of an earlier 1986 cl.el package by Cesar Quiroz. Care has been taken to ensure that each function is defined efficiently, concisely, and with minimal impact on the rest of the Emacs environment. Stefan Monnier added the file cl-lib.el and rationalized the namespace for Emacs 24.3.

* [Usage](#Usage): How to use this package.
* [Organization](#Organization): The package's component files.
* [Naming Conventions](#Naming-Conventions): Notes on function names.

Next: [Organization](#Organization), Up: [Overview](#Overview)

### 1.1 Usage

This package is distributed with Emacs, so there is no need to install any additional files in order to start using it. Lisp code that uses features from this package should simply include at the beginning:

(require 'cl-lib)

You may wish to add such a statement to your init file, if you make frequent use of features from this package.

Next: [Naming Conventions](#Naming-Conventions), Previous: [Usage](#Usage), Up: [Overview](#Overview)

### 1.2 Organization

The Common Lisp package is organized into four main files:

cl-lib.el

This is the main file, which contains basic functions and information about the package. This file is relatively compact.

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 cl-incf 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 cl-delete-if and cl-assoc.

cl-macs.el

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

There is another file, cl.el, which was the main entry point to this package prior to Emacs 24.3. Nowadays, it is replaced by cl-lib.el. The two provide the same features (in most cases), but use different function names (in fact, cl.el mainly just defines aliases to the cl-lib.el definitions). Where cl-lib.el defines a function called, for example, cl-incf, cl.el uses the same name but without the `cl-' prefix, e.g., incf in this example. There are a few exceptions to this. First, functions such as cl-defun where the unprefixed version was already used for a standard Emacs Lisp function. In such cases, the cl.el version adds a `\*' suffix, e.g., defun\*. Second, there are some obsolete features that are only implemented in cl.el, not in cl-lib.el, because they are replaced by other standard Emacs Lisp features. Finally, in a very few cases the old cl.el versions do not behave in exactly the same way as the cl-lib.el versions. See [Obsolete Features](#Obsolete-Features).

Since the old cl.el does not use a clean namespace, Emacs has a policy that packages distributed with Emacs must not load cl at run time. (It is ok for them to load cl at *compile* time, with eval-when-compile, and use the macros it provides.) There is no such restriction on the use of cl-lib. New code should use cl-lib rather than cl.

There is one more file, cl-compat.el, which defines some routines from the older Quiroz cl.el package that are not otherwise present in the new package. This file is obsolete and should not be used in new code.

Previous: [Organization](#Organization), Up: [Overview](#Overview)

### 1.3 Naming Conventions

Except where noted, all functions defined by this package have the same calling conventions as their Common Lisp counterparts, and names that are those of Common Lisp plus a `cl-' prefix.

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

cl-callf cl-callf2 cl-defsubst  
 cl-letf cl-letf\*

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

cl-evenp cl-oddp cl-minusp  
 cl-plusp cl-endp cl-subst  
 cl-copy-list cl-list\* cl-ldiff  
 cl-rest cl-decf [1] cl-incf [1]  
 cl-acons cl-adjoin [2] cl-pairlis  
 cl-pushnew [1,2] cl-declaim cl-proclaim  
 cl-caaar...cl-cddddr cl-first...cl-tenth  
 cl-mapcar [3]

[1] Only when place is a plain variable name.

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

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

Next: [Predicates](#Predicates), Previous: [Overview](#Overview), Up: [Top](#Top)

## 2 Program Structure

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

* [Argument Lists](#Argument-Lists): &key, &aux, cl-defun, cl-defmacro.
* [Time of Evaluation](#Time-of-Evaluation): The cl-eval-when construct.

Next: [Time of Evaluation](#Time-of-Evaluation), Up: [Program Structure](#Program-Structure)

### 2.1 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.

— Macro: **cl-defun** name arglist body...

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 [Blocks and Exits](#Blocks-and-Exits).

— Macro: **cl-iter-defun** name arglist body...

This form is identical to the regular iter-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 [Blocks and Exits](#Blocks-and-Exits).

— Macro: **cl-defsubst** name arglist body...

This is just like cl-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; cl-defsubst uses a different method (compiler macros) which works in all versions of Emacs, and also generates somewhat more efficient inline expansions. In particular, cl-defsubst arranges for the processing of keyword arguments, default values, etc., to be done at compile-time whenever possible.

— Macro: **cl-defmacro** name arglist body...

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's book Common Lisp, the Language. 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.

— Macro: **cl-function** symbol-or-lambda

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 cl-flet and cl-labels) defined in this package that include arglists in their syntax allow full Common Lisp argument lists.

Note that it is *not* necessary to use cl-defun in order to have access to most CL features in your function. These features are always present; cl-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,

(cl-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.

You can also explicitly specify the keyword argument; it need not be simply the variable name prefixed with a colon. For example,

(cl-defun bar (&key (a 1) ((baz b) 4)))

specifies a keyword :a that sets the variable a with default value 1, as well as a keyword baz that sets the variable b with default value 4. In this case, because baz is not self-quoting, you must quote it explicitly in the function call, like this:

(bar :a 10 'baz 42)

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:

(cl-defun find-thing (thing &rest rest &key need &allow-other-keys)  
 (or (apply 'cl-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 cl-member function. allow-other-keys is used to keep both find-thing and cl-member from complaining about each others' keywords in the arguments.

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:

(cl-defun foo (a b &aux (c (+ a b)) d)  
 body)  
   
 (cl-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 cl-defun and other argument lists as well. In destructuring, any argument variable (var in the above example) 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:

(cl-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 [Declarations](#Declarations)), error checking for wrong number of arguments and invalid keyword arguments is disabled. By default, argument lists are rigorously checked.

Previous: [Argument Lists](#Argument-Lists), Up: [Program Structure](#Program-Structure)

### 2.2 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.

— Macro: **cl-eval-when** (situations...) forms...

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 cl-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 cl-eval-whens, 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 cl-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 cl-eval-whens are nested; consult Steele (second edition) for the gruesome details (and some gruesome examples).

Some simple examples:

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

Note that (cl-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 (cl-eval-when (compile load eval) ...).

Emacs includes two special forms related to cl-eval-when. See [Eval During Compile](elisp.html#Eval-During-Compile). One of these, eval-when-compile, is not quite equivalent to any cl-eval-when construct and is described below.

The other form, (eval-and-compile ...), is exactly equivalent to `(cl-eval-when (compile load eval) ...)'.

— Macro: **eval-when-compile** forms...

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.

— Macro: **cl-load-time-value** 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, cl-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 cl-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: "  
 (cl-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 Oct 31 16:32:28 2012"  
 ", and loaded on: "  
 --temp--))

Next: [Control Structure](#Control-Structure), Previous: [Program Structure](#Program-Structure), Up: [Top](#Top)

## 3 Predicates

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

* [Type Predicates](#Type-Predicates): cl-typep, cl-deftype, and cl-coerce.
* [Equality Predicates](#Equality-Predicates): cl-equalp.

Next: [Equality Predicates](#Equality-Predicates), Up: [Predicates](#Predicates)

### 3.1 Type Predicates

— Function: **cl-typep** object type

Check if object is of type type, where type is a (quoted) type name of the sort used by Common Lisp. For example, (cl-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. (cl-typep object t) is always true. Likewise, the type symbol nil stands for nothing at all, and (cl-typep object nil) is always false.
* The type symbol null represents the symbol nil. Thus (cl-typep object 'null) is equivalent to (null object).
* The type symbol atom represents all objects that are not cons cells. Thus (cl-typep object 'atom) is equivalent to (atom 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 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 cl-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 cl-typep.

— Function: **cl-coerce** object type

This function attempts to convert object to the specified type. If object is already of that type as determined by cl-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, cl-coerce signals an error.

— Macro: **cl-deftype** name arglist forms...

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 cl-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 cl-defmacro except that optional arguments without explicit defaults use \* instead of nil as the “default” default. Some examples:

(cl-deftype null () '(satisfies null)) ; predefined  
 (cl-deftype list () '(or null cons)) ; predefined  
 (cl-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 cl-typecase (see [Conditionals](#Conditionals)) and cl-check-type (see [Assertions](#Assertions)) macros also use type names. The cl-map, cl-concatenate, and cl-merge functions take type-name arguments to specify the type of sequence to return. See [Sequences](#Sequences).

Previous: [Type Predicates](#Type-Predicates), Up: [Predicates](#Predicates)

### 3.2 Equality Predicates

This package defines the Common Lisp predicate cl-equalp.

— Function: **cl-equalp** a b

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 (cl-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 cl-equalp also will not compare strings against vectors of integers.

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. The functions cl-member and cl-assoc use eql, as in Common Lisp. The standard Emacs Lisp functions memq and assq use eq, and the standard memql uses eql.

Next: [Macros](#Macros), Previous: [Predicates](#Predicates), Up: [Top](#Top)

## 4 Control Structure

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

* [Assignment](#Assignment): The cl-psetq form.
* [Generalized Variables](#Generalized-Variables): Extensions to generalized variables.
* [Variable Bindings](#Variable-Bindings): cl-progv, cl-flet, cl-macrolet.
* [Conditionals](#Conditionals): cl-case, cl-typecase.
* [Blocks and Exits](#Blocks-and-Exits): cl-block, cl-return, cl-return-from.
* [Iteration](#Iteration): cl-do, cl-dotimes, cl-dolist, cl-do-symbols.
* [Loop Facility](#Loop-Facility): The Common Lisp loop macro.
* [Multiple Values](#Multiple-Values): cl-values, cl-multiple-value-bind, etc.

Next: [Generalized Variables](#Generalized-Variables), Up: [Control Structure](#Control-Structure)

### 4.1 Assignment

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

— Macro: **cl-psetq** [symbol 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)  
 (cl-psetq x (+ x y) y (\* x y))  
 x  
 => 5  
 y ; y was computed before x was set.  
 => 6

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

cl-psetq always returns nil.

Next: [Variable Bindings](#Variable-Bindings), Previous: [Assignment](#Assignment), Up: [Control Structure](#Control-Structure)

### 4.2 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. For basic information, see [Generalized Variables](elisp.html#Generalized-Variables). This package provides several additional features related to generalized variables.

* [Setf Extensions](#Setf-Extensions): Additional setf places.
* [Modify Macros](#Modify-Macros): cl-incf, cl-rotatef, cl-letf, cl-callf, etc.

Next: [Modify Macros](#Modify-Macros), Up: [Generalized Variables](#Generalized-Variables)

#### 4.2.1 Setf Extensions

Several standard (e.g., car) and Emacs-specific (e.g., window-point) Lisp functions are setf-able by default. This package defines setf handlers for several additional functions:

* Functions from this package:
* cl-rest cl-subseq cl-get cl-getf  
   cl-caaar...cl-cddddr cl-first...cl-tenth
* Note that for cl-getf (as for nthcdr), the list argument of the function must itself be a valid place form.
* General Emacs Lisp functions:
* buffer-file-name getenv  
   buffer-modified-p global-key-binding  
   buffer-name local-key-binding  
   buffer-string mark  
   buffer-substring mark-marker  
   current-buffer marker-position  
   current-case-table mouse-position  
   current-column point  
   current-global-map point-marker  
   current-input-mode point-max  
   current-local-map point-min  
   current-window-configuration read-mouse-position  
   default-file-modes screen-height  
   documentation-property screen-width  
   face-background selected-window  
   face-background-pixmap selected-screen  
   face-font selected-frame  
   face-foreground standard-case-table  
   face-underline-p syntax-table  
   file-modes visited-file-modtime  
   frame-height window-height  
   frame-parameters window-width  
   frame-visible-p x-get-secondary-selection  
   frame-width x-get-selection  
   get-register
* 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 used with setf ((narrow-to-region x (point-max)) in this case).
* A call of the form (substring subplace n [m]), where subplace is itself a valid 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 macro call, in which case the macro is expanded and setf is applied to the resulting form.

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

(setf (aref vec (cl-incf i)) i)

looks like it will evaluate (cl-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.

Previous: [Setf Extensions](#Setf-Extensions), Up: [Generalized Variables](#Generalized-Variables)

#### 4.2.2 Modify Macros

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

— Macro: **cl-psetf** [place form]...

This macro is to setf what cl-psetq is to setq: When several places and forms 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).

— Macro: **cl-incf** place &optional x

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

As with setf, care is taken to preserve the “apparent” order of evaluation. For example,

(cl-incf (aref vec (cl-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 (cl-incf i))  
 (1+ (aref vec (cl-incf i)))) ; wrong!

but rather to something more like

(let ((temp (cl-incf i)))  
 (setf (aref vec temp) (1+ (aref vec temp))))

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

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

— Macro: **cl-decf** place &optional x

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

— Macro: **cl-pushnew** x place &key :test :test-not :key

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 cl-adjoin. See [Lists as Sets](#Lists-as-Sets).

— Macro: **cl-shiftf** place... newvalue

This macro shifts the places 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, (cl-shiftf a b c d) is equivalent to

(prog1  
 a  
 (cl-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.

— Macro: **cl-rotatef** place...

This macro rotates the places left by one in circular fashion. Thus, (cl-rotatef a b c d) is equivalent to

(cl-psetf a b  
 b c  
 c d  
 d a)

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

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

— Macro: **cl-letf** (bindings...) forms...

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 places are saved, the values are stored in them, and then the body forms are executed. Afterwards, the places are set back to their original saved contents. This cleanup happens even if the forms exit irregularly due to a throw or an error.

For example,

(cl-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 cl-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 cl-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 cl-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 cl-letf is to restore the original value of place afterwards.

Note that in this case, and in fact almost every case, place must have a well-defined value outside the cl-letf body. There is essentially only one exception to this, which is place a plain variable with a specified value (such as (a 17) in the above example).

— Macro: **cl-letf\*** (bindings...) forms...

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

— Macro: **cl-callf** function place args...

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, (cl-incf place n) is the same as (cl-callf + place n). Some more examples:

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

Note again that cl-callf is an extension to standard Common Lisp.

— Macro: **cl-callf2** function arg1 place args...

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

The cl-callf and cl-callf2 macros serve as building blocks for other macros like cl-incf, and cl-pushnew. The cl-letf and cl-letf\* macros are used in the processing of symbol macros; see [Macro Bindings](#Macro-Bindings).

Next: [Conditionals](#Conditionals), Previous: [Generalized Variables](#Generalized-Variables), Up: [Control Structure](#Control-Structure)

### 4.3 Variable Bindings

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

See [Modify Macros](#Modify-Macros), for the cl-letf and cl-letf\* forms which are also related to variable bindings.

* [Dynamic Bindings](#Dynamic-Bindings): The cl-progv form.
* [Function Bindings](#Function-Bindings): cl-flet and cl-labels.
* [Macro Bindings](#Macro-Bindings): cl-macrolet and cl-symbol-macrolet.

Next: [Function Bindings](#Function-Bindings), Up: [Variable Bindings](#Variable-Bindings)

#### 4.3.1 Dynamic Bindings

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

— Macro: **cl-progv** symbols values forms...

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 bound to nil. If symbols is shorter than values, the excess values are ignored.

Next: [Macro Bindings](#Macro-Bindings), Previous: [Dynamic Bindings](#Dynamic-Bindings), Up: [Variable Bindings](#Variable-Bindings)

#### 4.3.2 Function Bindings

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

— Macro: **cl-flet** (bindings...) forms...

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 cl-defun form. The function name is defined accordingly but only within the body of the cl-flet, hiding any external definition if applicable.

The bindings are lexical in scope. This means that all references to the named functions must appear physically within the body of the cl-flet form.

Functions defined by cl-flet may use the full Common Lisp argument notation supported by cl-defun; also, the function body is enclosed in an implicit block as if by cl-defun. See [Program Structure](#Program-Structure).

Note that the cl.el version of this macro behaves slightly differently. In particular, its binding is dynamic rather than lexical. See [Obsolete Macros](#Obsolete-Macros).

— Macro: **cl-labels** (bindings...) forms...

The cl-labels form is like cl-flet, except that the function bindings can be recursive. The scoping is lexical, but you can only capture functions in closures if lexical-binding is t. See [Closures](elisp.html#Closures), and [Using Lexical Binding](elisp.html#Using-Lexical-Binding).

Lexical scoping means that all references to the named functions must appear physically within the body of the cl-labels form. References may appear both in the body forms of cl-labels itself, and in the bodies of the functions themselves. Thus, cl-labels can define local recursive functions, or mutually-recursive sets of functions.

A “reference” to a function name is either a call to that function, or a use of its name quoted by quote or function to be passed on to, say, mapcar.

Note that the cl.el version of this macro behaves slightly differently. See [Obsolete Macros](#Obsolete-Macros).

Previous: [Function Bindings](#Function-Bindings), Up: [Variable Bindings](#Variable-Bindings)

#### 4.3.3 Macro Bindings

These forms create local macros and “symbol macros”.

— Macro: **cl-macrolet** (bindings...) forms...

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

Because of the nature of macros, cl-macrolet is always lexically scoped. The cl-macrolet binding will affect only calls that appear physically within the body forms, possibly after expansion of other macros in the body.

— Macro: **cl-symbol-macrolet** (bindings...) forms...

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))  
 (cl-symbol-macrolet ((foo (car bar)))  
 (cl-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 cl-letf or cl-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, such shadowing does not occur, even when lexical-binding is t. (This behavior predates the addition of lexical binding to Emacs Lisp, and may change in future to respect lexical-binding.) At present in this package, only lexical-let and lexical-let\* will shadow a symbol macro. See [Obsolete Lexical Binding](#Obsolete-Lexical-Binding).

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

(cl-defmacro my-dolist ((x list) &rest body)  
 (let ((var (cl-gensym)))  
 (list 'cl-loop 'for var 'on list 'do  
 (cl-list\* 'cl-symbol-macrolet  
 (list (list x (list 'car var)))  
 body))))  
   
 (setq mylist '(1 2 3 4))  
 (my-dolist (x mylist) (cl-incf x))  
 mylist  
 => (2 3 4 5)

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

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

which in turn expands to

(cl-loop for G1234 on mylist do (cl-incf (car G1234)))

See [Loop Facility](#Loop-Facility), for a description of the cl-loop macro. This package defines a nonstandard in-ref loop clause that works much like my-dolist.

Next: [Blocks and Exits](#Blocks-and-Exits), Previous: [Variable Bindings](#Variable-Bindings), Up: [Control Structure](#Control-Structure)

### 4.4 Conditionals

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

— Macro: **cl-case** keyform clause...

This macro evaluates keyform, then compares it with the key values listed in the various clauses. Whichever clause matches the key is executed; comparison is done by eql. If no clause matches, the cl-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 cl-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 C-j, or anything else.

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

— Macro: **cl-ecase** keyform clause...

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

— Macro: **cl-typecase** keyform clause...

This macro is a version of cl-case that checks for types rather than values. Each clause is of the form `(type body...)'. See [Type Predicates](#Type-Predicates), for a description of type specifiers. For example,

(cl-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.

— Macro: **cl-etypecase** keyform clause...

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

Next: [Iteration](#Iteration), Previous: [Conditionals](#Conditionals), Up: [Control Structure](#Control-Structure)

### 4.5 Blocks and Exits

Common Lisp blocks provide a non-local exit mechanism very similar to catch and throw, with lexical scoping. This package actually implements cl-block in terms of catch; however, the lexical scoping allows the byte-compiler to omit the costly catch step if the body of the block does not actually cl-return-from the block.

— Macro: **cl-block** name forms...

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

The cl-block/cl-return-from mechanism is quite similar to the catch/throw mechanism. The main differences are that block names are unevaluated symbols, rather than forms (such as quoted symbols) that evaluate to a tag at run-time; and also that blocks are always lexically scoped. In a dynamically scoped catch, functions called from the catch body can also throw to the catch. This is not an option for cl-block, where the cl-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 lambdas 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 cl-defun and cl-defmacro forms, which do create the implicit block.

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

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

— Macro: **cl-return-from** name [result]

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.

— Macro: **cl-return** [result]

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

— Macro: **cl-tagbody** &rest labels-or-statements

This macro executes statements while allowing for control transfer to user-defined labels. Each element of labels-or-statements can be either a label (an integer or a symbol), or a cons-cell (a statement). This distinction is made before macroexpansion. Statements are executed in sequence, discarding any return value. Any statement can transfer control at any time to the statements that follow one of the labels with the special form (go label). Labels have lexical scope and dynamic extent.

Next: [Loop Facility](#Loop-Facility), Previous: [Blocks and Exits](#Blocks-and-Exits), Up: [Control Structure](#Control-Structure)

### 4.6 Iteration

The macros described here provide more sophisticated, high-level looping constructs to complement Emacs Lisp's basic loop forms (see [Iteration](elisp.html#Iteration)).

— Macro: **cl-loop** forms...

This 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 [Loop Facility](#Loop-Facility). The simple form of loop is described here.

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

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

is exactly equivalent to

(cl-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.)

— Macro: **cl-do** (spec...) (end-test [result...]) forms...

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 cl-psetq form) and the next iteration begins. Once the end-test becomes true, the result forms are evaluated (with the vars still bound to their values) to produce the result returned by cl-do.

The entire cl-do loop is enclosed in an implicit nil block, so that you can use (cl-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 cl-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 that applies the function f to successive pairs of values from the lists foo and bar; it is equivalent to the call (cl-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.

(cl-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)))

— Macro: **cl-do\*** (spec...) (end-test [result...]) forms...

This is to cl-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 cl-psetq.

Here is another way to write the above loop:

(cl-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))

— Macro: **cl-dolist** (var list [result]) forms...

This is exactly like the standard Emacs Lisp macro dolist, but surrounds the loop with an implicit nil block.

— Macro: **cl-dotimes** (var count [result]) forms...

This is exactly like the standard Emacs Lisp macro dotimes, but surrounds the loop with an implicit nil block. 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.

— Macro: **cl-do-symbols** (var [obarray [result]]) forms...

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.

— Macro: **cl-do-all-symbols** (var [result]) forms...

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

See [Mapping over Sequences](#Mapping-over-Sequences), for some more functions for iterating over vectors or lists.

Next: [Multiple Values](#Multiple-Values), Previous: [Iteration](#Iteration), Up: [Control Structure](#Control-Structure)

### 4.7 Loop Facility

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

To remedy this, Common Lisp added a construct called the “Loop Facility” or “loop macro”, with an easy-to-use but very powerful and expressive syntax.

* [Loop Basics](#Loop-Basics): The cl-loop macro, basic clause structure.
* [Loop Examples](#Loop-Examples): Working examples of the cl-loop macro.
* [For Clauses](#For-Clauses): Clauses introduced by for or as.
* [Iteration Clauses](#Iteration-Clauses): repeat, while, thereis, etc.
* [Accumulation Clauses](#Accumulation-Clauses): collect, sum, maximize, etc.
* [Other Clauses](#Other-Clauses): with, if, initially, finally.

Next: [Loop Examples](#Loop-Examples), Up: [Loop Facility](#Loop-Facility)

#### 4.7.1 Loop Basics

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

— Macro: **cl-loop** clauses...

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 cl-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 cl-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 cl-return or cl-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 for additional discussion and examples.

Next: [For Clauses](#For-Clauses), Previous: [Loop Basics](#Loop-Basics), Up: [Loop Facility](#Loop-Facility)

#### 4.7.2 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 cl-loop language.

(cl-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 cl-loop construct. The result is a list of the file names of all the buffers in Emacs's memory. The words for, in, and collect are reserved words in the cl-loop language.

(cl-loop repeat 20 do (insert "Yowsa\n"))

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

(cl-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.

(cl-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.

(cl-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 fors 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 cl-loop constructs if you want nested loops.

Next: [Iteration Clauses](#Iteration-Clauses), Previous: [Loop Examples](#Loop-Examples), Up: [Loop Facility](#Loop-Facility)

#### 4.7.3 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)  
 (cl-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:

(cl-loop for x to 10 collect x)  
 => (0 1 2 3 4 5 6 7 8 9 10)  
 (cl-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 valid 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:

(cl-loop for x in '(1 2 3 4 5 6) collect (\* x x))  
 => (1 4 9 16 25 36)  
 (cl-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.

(cl-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:

(cl-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,

(cl-loop for x in-ref my-list do (cl-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.

(cl-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,

(cl-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 cl-loop. Fortunately, it would rarely if ever be useful to do so. It *is* valid 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

for var being the hash-values of hash-table

This clause iterates over the entries in hash-table with var bound to each key, or value. A `using' clause can bind a second variable to the opposite part.

(cl-loop for k being the hash-keys of h  
 using (hash-values v)  
 do  
 (message "key %S -> value %S" k v))

for var being the key-codes of keymap

for var being the key-bindings of keymap

This clause iterates over the entries in keymap. The iteration does not enter nested keymaps but does enter inherited (parent) keymaps. A using clause can access both the codes and the bindings together.

(cl-loop for c being the key-codes of (current-local-map)  
 using (key-bindings b)  
 do  
 (message "key %S -> binding %S" c b))

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 vectors. 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 “overlays” of a buffer (the clause extents is synonymous with overlays). 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.

for var being the frames

This clause iterates over all Emacs frames. 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. It visits windows in next-window order starting from selected-window (or frame-selected-window if you specify frame). This clause treats the minibuffer window in the same way as next-window does. For greater flexibility, consider using walk-windows instead.

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:

(cl-loop for x on my-list by 'cddr do ...)  
 (cl-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:

(cl-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 cl-psetq).

(cl-loop for x below 5 for y = nil then x collect (list x y))  
 => ((0 nil) (1 1) (2 2) (3 3) (4 4))  
 (cl-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 cl-loop macro is *destructuring*, similar in concept to the destructuring provided by defmacro (see [Argument Lists](#Argument-Lists)). 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.

(cl-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, so for example to process an alist

(cl-loop for (key . value) in '((a . 1) (b . 2))  
 collect value)  
 => (1 2)

Next: [Accumulation Clauses](#Accumulation-Clauses), Previous: [For Clauses](#For-Clauses), Up: [Loop Facility](#Loop-Facility)

#### 4.7.4 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

(cl-loop repeat (1+ n) do ...)  
 (cl-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...)  
 (cl-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 (cl-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.

iter-by iterator

This clause iterates over the values from the specified form, an iterator object. See (see [Generators](elisp.html#Generators)).

Next: [Other Clauses](#Other-Clauses), Previous: [Iteration Clauses](#Iteration-Clauses), Up: [Loop Facility](#Loop-Facility)

#### 4.7.5 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 valid for several accumulation clauses of the same type to accumulate into the same place. From Steele:

(cl-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)

Previous: [Accumulation Clauses](#Accumulation-Clauses), Up: [Loop Facility](#Loop-Facility)

#### 4.7.6 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:

(cl-loop with x = 17 do ...)  
 (let ((x 17)) (cl-loop do ...))  
 (cl-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:

(cl-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)  
 (cl-loop for x below 10  
 if (cl-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)  
 (cl-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 (cl-return form)' (or cl-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 cl-loop, 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 cl-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 that iterate over keymaps, overlays, intervals, frames, windows, and buffers are Emacs-specific extensions.

Previous: [Loop Facility](#Loop-Facility), Up: [Control Structure](#Control-Structure)

### 4.8 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 cl-compiler-macroexpand) or return a list of values. 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 cl-values form, for example, is a synonym for list in Emacs.

— Macro: **cl-multiple-value-bind** (var...) values-form forms...

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

— Macro: **cl-multiple-value-setq** (var...) form

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

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

Next: [Declarations](#Declarations), Previous: [Control Structure](#Control-Structure), Up: [Top](#Top)

## 5 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 [Argument Lists](#Argument-Lists).

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

— Macro: **cl-destructuring-bind** arglist expr forms...

This macro expands to code that 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 cl-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.

— Macro: **cl-define-compiler-macro** name arglist forms...

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:

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

This definition causes (cl-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 cl-member call is left intact. (The actual compiler macro for cl-member optimizes a number of other cases, including common :test predicates.)

— Function: **cl-compiler-macroexpand** form

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 [Macro Bindings](#Macro-Bindings), for descriptions of the cl-macrolet and cl-symbol-macrolet forms for making “local” macro definitions.

Next: [Symbols](#Symbols), Previous: [Macros](#Macros), Up: [Top](#Top)

## 6 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. For example, the byte-code system provides little opportunity to benefit from type information. A few declarations are meaningful when byte compiler optimizations are enabled, as they are by the default. Otherwise these declarations will effectively be ignored.

— Function: **cl-proclaim** decl-spec

This function records a “global” declaration specified by decl-spec. Since cl-proclaim is a function, decl-spec is evaluated and thus should normally be quoted.

— Macro: **cl-declaim** decl-specs...

This macro is like cl-proclaim, except that it takes any number of decl-spec arguments, and the arguments are unevaluated and unquoted. The cl-declaim macro also puts (cl-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 cl-declaim form.)

— Macro: **cl-declare** decl-specs...

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 progns” throughout Lisp syntax, such as function bodies, let bodies, etc. Currently the only declaration understood by cl-declare is special.

— Macro: **cl-locally** declarations... forms...

In this package, cl-locally is no different from progn.

— Macro: **cl-the** type form

cl-the returns the value of form, first checking (if optimization settings permit) that it is of type type. Future byte-compiler optimizations may also make use of this information to improve runtime efficiency.

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 (cl-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 cl-proclaim, cl-declaim, or cl-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 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 (cl-declare (special var1 var2)) is equivalent to (defvar var1) (defvar var2).

In top-level contexts, it is generally better to write (defvar var) than (cl-declaim (special var)), since defvar makes your intentions clearer.

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 function cl-acons is declared inline by this package so that the form (cl-acons key value alist) will expand directly into (cons (cons key value) alist) when it is called in user functions, so as to save function calls.

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.

(cl-declaim (inline foo bar))  
 (cl-eval-when (compile load eval)  
 (cl-proclaim '(inline foo bar)))  
 (defsubst foo (...) ...) ; instead of defun

**Please 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 (cl-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.

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, the speed quality is tied to the byte-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 nil for (safety 3) and to t 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 cl-proclaim or cl-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 byte-compiler. (The cl-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. 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 of the variable byte-compile-warnings for more details.

Next: [Numbers](#Numbers), Previous: [Declarations](#Declarations), Up: [Top](#Top)

## 7 Symbols

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

* [Property Lists](#Property-Lists): cl-get, cl-remprop, cl-getf, cl-remf.
* [Creating Symbols](#Creating-Symbols): cl-gensym, cl-gentemp.

Next: [Creating Symbols](#Creating-Symbols), Up: [Symbols](#Symbols)

### 7.1 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.

— Function: **cl-get** symbol property &optional default

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 cl-get is equivalent to Common Lisp's get.)

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

— Function: **cl-remprop** symbol property

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 cl-remprop for most purposes.)

— Function: **cl-getf** place property &optional default

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) == (cl-getf (symbol-plist sym) prop)

It is valid to use cl-getf as a setf place, in which case its place argument must itself be a valid 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 (cl-getf (symbol-plist sym) prop) val)

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

(cl-incf (cl-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, cl-getf is just a regular function and its place argument can actually be any Lisp expression.

— Macro: **cl-remf** place property

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.

Previous: [Property Lists](#Property-Lists), Up: [Symbols](#Symbols)

### 7.2 Creating Symbols

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

— Function: **cl-gensym** &optional x

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.

(Internally, the variable cl--gensym-counter holds the counter used to generate names. It is initialized with zero and incremented after each use.)

— Function: **cl-gentemp** &optional x

This function is like cl-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.

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 related function defkeyword (to create self-quoting keyword symbols) is not provided.

Next: [Sequences](#Sequences), Previous: [Symbols](#Symbols), Up: [Top](#Top)

## 8 Numbers

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

* [Predicates on Numbers](#Predicates-on-Numbers): cl-plusp, cl-oddp, etc.
* [Numerical Functions](#Numerical-Functions): cl-floor, cl-ceiling, etc.
* [Random Numbers](#Random-Numbers): cl-random, cl-make-random-state.
* [Implementation Parameters](#Implementation-Parameters): cl-most-positive-float, etc.

Next: [Numerical Functions](#Numerical-Functions), Up: [Numbers](#Numbers)

### 8.1 Predicates on Numbers

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

— Function: **cl-plusp** number

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

— Function: **cl-minusp** number

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

— Function: **cl-oddp** integer

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

— Function: **cl-evenp** integer

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

— Function: **cl-digit-char-p** char radix

Test if char is a digit in the specified radix (default is 10). If it is, return the numerical value of digit char in radix.

Next: [Random Numbers](#Random-Numbers), Previous: [Predicates on Numbers](#Predicates-on-Numbers), Up: [Numbers](#Numbers)

### 8.2 Numerical Functions

These functions perform various arithmetic operations on numbers.

— Function: **cl-gcd** &rest integers

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.

— Function: **cl-lcm** &rest integers

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.

— Function: **cl-isqrt** integer

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.

— Function: **cl-floor** number &optional divisor

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

— Function: **cl-ceiling** number &optional divisor

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.

— Function: **cl-truncate** number &optional divisor

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 cl-floor if the argument or quotient is positive, or to cl-ceiling otherwise. The remainder has the same sign as number.

— Function: **cl-round** number &optional divisor

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.

— Function: **cl-mod** number divisor

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

— Function: **cl-rem** number divisor

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

— Function: **cl-parse-integer** string &key start end radix junk-allowed

This function implements the Common Lisp parse-integer function. It parses an integer in the specified radix from the substring of string between start and end. Any leading and trailing whitespace chars are ignored. The function signals an error if the substring between start and end cannot be parsed as an integer, unless junk-allowed is non-nil.

Next: [Implementation Parameters](#Implementation-Parameters), Previous: [Numerical Functions](#Numerical-Functions), Up: [Numbers](#Numbers)

### 8.3 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.

— Function: **cl-random** number &optional state

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 that 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 internal variable cl--random-state, which contains a pre-initialized default random-state object. (Since any number of programs in the Emacs process may be accessing cl--random-state in interleaved fashion, the sequence generated from this will be irreproducible for all intents and purposes.)

— Function: **cl-make-random-state** &optional state

This function creates or copies a random-state object. If state is omitted or nil, it returns a new copy of cl--random-state. This is a copy in the sense that future sequences of calls to (cl-random n) and (cl-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 valid 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 (cl-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.

— Function: **cl-random-state-p** object

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

Previous: [Random Numbers](#Random-Numbers), Up: [Numbers](#Numbers)

### 8.4 Implementation Parameters

This package defines several useful constants having to do with floating-point numbers.

It 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.

— Function: **cl-float-limits**

This function makes sure that the Common Lisp floating-point parameters like cl-most-positive-float have been initialized. Until it is called, these parameters will be nil. If the parameters have already been initialized, the function returns immediately.

The algorithm makes assumptions that will be valid for almost all 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.

— Variable: **cl-most-positive-float**

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.

— Variable: **cl-most-negative-float**

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

— Variable: **cl-least-positive-float**

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.

— Variable: **cl-least-positive-normalized-float**

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 cl-least-positive-float.

— Variable: **cl-least-negative-float**

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

— Variable: **cl-least-negative-normalized-float**

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

— Variable: **cl-float-epsilon**

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.

— Variable: **cl-float-negative-epsilon**

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.

Next: [Lists](#Lists), Previous: [Numbers](#Numbers), Up: [Top](#Top)

## 9 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.

* [Sequence Basics](#Sequence-Basics): Arguments shared by all sequence functions.
* [Mapping over Sequences](#Mapping-over-Sequences): cl-mapcar, cl-map, cl-maplist, etc.
* [Sequence Functions](#Sequence-Functions): cl-subseq, cl-remove, cl-substitute, etc.
* [Searching Sequences](#Searching-Sequences): cl-find, cl-count, cl-search, etc.
* [Sorting Sequences](#Sorting-Sequences): cl-sort, cl-stable-sort, cl-merge.

Next: [Mapping over Sequences](#Mapping-over-Sequences), Up: [Sequences](#Sequences)

### 9.1 Sequence Basics

Many of the sequence functions take keyword arguments; see [Argument Lists](#Argument-Lists). 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, (cl-find x y :key 'car) is similar to (cl-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 eql.

Many functions that 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:

(cl-remove 0 seq :test '=) == (cl-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 cl-mapcar and cl-every, *do* specify exactly the order in which the function is called so side effects are perfectly acceptable in those cases.)

Strings 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, (cl-remove ?A str) may or may not preserve the properties of the characters copied from str into the result.

Next: [Sequence Functions](#Sequence-Functions), Previous: [Sequence Basics](#Sequence-Basics), Up: [Sequences](#Sequences)

### 9.2 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.

— Function: **cl-mapcar** function seq &rest more-seqs

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 cl-mapcar works as a compatible superset of both.

— Function: **cl-map** result-type function seq &rest more-seqs

This function maps function over the argument sequences, just like cl-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 cl-mapcar), or nil (in which case the results are thrown away and cl-map returns nil).

— Function: **cl-maplist** function list &rest more-lists

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

— Function: **cl-mapc** function seq &rest more-seqs

This function is like cl-mapcar, except that the values returned by function are ignored and thrown away rather than being collected into a list. The return value of cl-mapc is seq, the first sequence. This function is more general than the Emacs primitive mapc. (Note that this function is called cl-mapc even in cl.el, rather than mapc\* as you might expect.)

— Function: **cl-mapl** function list &rest more-lists

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

— Function: **cl-mapcan** function seq &rest more-seqs

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

— Function: **cl-mapcon** function list &rest more-lists

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

— Function: **cl-some** predicate seq &rest more-seqs

This function calls predicate on each element of seq in turn; if predicate returns a non-nil value, cl-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 cl-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.

— Function: **cl-every** predicate seq &rest more-seqs

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.

— Function: **cl-notany** predicate seq &rest more-seqs

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.

— Function: **cl-notevery** predicate seq &rest more-seqs

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.

— Function: **cl-reduce** function seq &key :from-end :start :end :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 cl-reduce.

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

(cl-reduce '- '(1 2 3 4))  
 == (- (- (- 1 2) 3) 4) => -8  
 (cl-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 cl-loop macro. In compiled code, cl-loop will be faster since it generates the loop as in-line code with no function calls.

Next: [Searching Sequences](#Searching-Sequences), Previous: [Mapping over Sequences](#Mapping-over-Sequences), Up: [Sequences](#Sequences)

### 9.3 Sequence Functions

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

— Function: **cl-subseq** sequence start &optional end

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's substring function. Note that cl-subseq is the *only* sequence function that allows negative start and end.

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

— Function: **cl-concatenate** result-type &rest seqs

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 (cl-concatenate 'list '(1 2 3)) where the result is identical to an argument.

— Function: **cl-fill** seq item &key :start :end

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

— Function: **cl-replace** seq1 seq2 &key :start1 :end1 :start2 :end2

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 that share storage but are not eq, and the start and end arguments specify overlapping regions, the effect is undefined.

— Function: **cl-remove** item seq &key :test :test-not :key :count :start :end :from-end

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).

— Function: **cl-delete** item seq &key :test :test-not :key :count :start :end :from-end

This deletes all elements of seq that match item. It is a destructive operation. Since Emacs Lisp does not support stretchable strings or vectors, this is the same as cl-remove for those sequence types. On lists, cl-remove will copy the list if necessary to preserve the original list, whereas cl-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 cl-remove-if, cl-remove-if-not, cl-delete-if, and cl-delete-if-not are defined similarly.

— Function: **cl-remove-duplicates** seq &key :test :test-not :key :start :end :from-end

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.

— Function: **cl-delete-duplicates** seq &key :test :test-not :key :start :end :from-end

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

— Function: **cl-substitute** new old seq &key :test :test-not :key :count :start :end :from-end

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.

— Function: **cl-nsubstitute** new old seq &key :test :test-not :key :count :start :end :from-end

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

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

Next: [Sorting Sequences](#Sorting-Sequences), Previous: [Sequence Functions](#Sequence-Functions), Up: [Sequences](#Sequences)

### 9.4 Searching Sequences

These functions search for elements or subsequences in a sequence. (See also cl-member and cl-assoc; see [Lists](#Lists).)

— Function: **cl-find** item seq &key :test :test-not :key :start :end :from-end

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.

— Function: **cl-position** item seq &key :test :test-not :key :start :end :from-end

This function is like cl-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.

— Function: **cl-count** item seq &key :test :test-not :key :start :end

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

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

— Function: **cl-mismatch** seq1 seq2 &key :test :test-not :key :start1 :end1 :start2 :end2 :from-end

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 that 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 (cl-mismatch str1 str2 :key 'upcase), which compares two strings case-insensitively.

— Function: **cl-search** seq1 seq2 &key :test :test-not :key :from-end :start1 :end1 :start2 :end2

This function searches seq2 for a subsequence that matches seq1 (or part of it specified by :start1 and :end1). Only matches that 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.

Previous: [Searching Sequences](#Searching-Sequences), Up: [Sequences](#Sequences)

### 9.5 Sorting Sequences

— Function: **cl-sort** seq predicate &key :key

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's 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 (cl-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. It should only be a simple accessor though, since it's used heavily in the current implementation.

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

— Function: **cl-stable-sort** seq predicate &key :key

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, cl-sort and cl-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 cl-sort in the future.

— Function: **cl-merge** type seq1 seq2 predicate &key :key

This function merges two sequences seq1 and seq2 by interleaving their elements. The result sequence, of type type (in the sense of cl-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.

Next: [Structures](#Structures), Previous: [Sequences](#Sequences), Up: [Top](#Top)

## 10 Lists

The functions described here operate on lists.

* [List Functions](#List-Functions): cl-caddr, cl-first, cl-list\*, etc.
* [Substitution of Expressions](#Substitution-of-Expressions): cl-subst, cl-sublis, etc.
* [Lists as Sets](#Lists-as-Sets): cl-member, cl-adjoin, cl-union, etc.
* [Association Lists](#Association-Lists): cl-assoc, cl-acons, cl-pairlis, etc.

Next: [Substitution of Expressions](#Substitution-of-Expressions), Up: [Lists](#Lists)

### 10.1 List Functions

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

— Function: **cl-caddr** x

This function is equivalent to (car (cdr (cdr x))). Likewise, this package defines all 24 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.

— Function: **cl-first** x

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

— Function: **cl-rest** x

This function is a synonym for (cdr x).

— Function: **cl-endp** x

This function acts like null, but signals an error if x is neither a nil nor a cons cell.

— Function: **cl-list-length** x

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. See also the safe-length function.)

— Function: **cl-list\*** arg &rest others

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

— Function: **cl-ldiff** list sublist

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, (cl-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.

— Function: **cl-copy-list** list

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

— Function: **cl-tree-equal** x y &key :test :test-not :key

This function compares two trees of cons cells. If x and y are both cons cells, their cars and cdrs 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 [Sequences](#Sequences).

Next: [Lists as Sets](#Lists-as-Sets), Previous: [List Functions](#List-Functions), Up: [Lists](#Lists)

### 10.2 Substitution of Expressions

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

— Function: **cl-subst** new old tree &key :test :test-not :key

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 cars and cdrs 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.

— Function: **cl-nsubst** new old tree &key :test :test-not :key

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

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

— Function: **cl-sublis** alist tree &key :test :test-not :key

This function is like cl-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 cars of alist; if it matches, it is replaced by the corresponding cdr.

— Function: **cl-nsublis** alist tree &key :test :test-not :key

This is a destructive version of cl-sublis.

Next: [Association Lists](#Association-Lists), Previous: [Substitution of Expressions](#Substitution-of-Expressions), Up: [Lists](#Lists)

### 10.3 Lists as Sets

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

— Function: **cl-member** item list &key :test :test-not :key

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 [Sequences](#Sequences).

The standard Emacs lisp function member uses equal for comparisons; it is equivalent to (cl-member item list :test 'equal). With no keyword arguments, cl-member is equivalent to memq.

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

— Function: **cl-tailp** sublist list

This function returns t if sublist is a sublist of list, i.e., if sublist is eql to list or to any of its cdrs.

— Function: **cl-adjoin** item list &key :test :test-not :key

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 cl-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.

— Function: **cl-union** list1 list2 &key :test :test-not :key

This function combines two lists that represent sets of items, returning a list that represents the union of those two sets. The resulting list contains all items that appear in list1 or list2, and no others. If an item appears in both list1 and list2 it is 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.

— Function: **cl-nunion** list1 list2 &key :test :test-not :key

This is a destructive version of cl-union; rather than copying, it tries to reuse the storage of the argument lists if possible.

— Function: **cl-intersection** list1 list2 &key :test :test-not :key

This function computes the intersection of the sets represented by list1 and list2. It returns the list of items that appear in both list1 and list2.

— Function: **cl-nintersection** list1 list2 &key :test :test-not :key

This is a destructive version of cl-intersection. It tries to reuse storage of list1 rather than copying. It does *not* reuse the storage of list2.

— Function: **cl-set-difference** list1 list2 &key :test :test-not :key

This function computes the “set difference” of list1 and list2, i.e., the set of elements that appear in list1 but *not* in list2.

— Function: **cl-nset-difference** list1 list2 &key :test :test-not :key

This is a destructive cl-set-difference, which will try to reuse list1 if possible.

— Function: **cl-set-exclusive-or** list1 list2 &key :test :test-not :key

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.

— Function: **cl-nset-exclusive-or** list1 list2 &key :test :test-not :key

This is a destructive cl-set-exclusive-or, which will try to reuse list1 and list2 if possible.

— Function: **cl-subsetp** list1 list2 &key :test :test-not :key

This function checks whether list1 represents a subset of list2, i.e., whether every element of list1 also appears in list2.

Previous: [Lists as Sets](#Lists-as-Sets), Up: [Lists](#Lists)

### 10.4 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.

— Function: **cl-assoc** item a-list &key :test :test-not :key

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 that are not cons cells. (This corresponds to the behavior of assq and assoc in Emacs Lisp; Common Lisp's assoc ignores nils but considers any other non-cons elements of a-list to be an error.)

— Function: **cl-rassoc** item a-list &key :test :test-not :key

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.

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

Two simple functions for constructing association lists are:

— Function: **cl-acons** key value alist

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

— Function: **cl-pairlis** keys values &optional alist

This is equivalent to (nconc (cl-mapcar 'cons keys values)alist).

Next: [Assertions](#Assertions), Previous: [Lists](#Lists), Up: [Top](#Top)

## 11 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.

— Macro: **cl-defstruct** name slots...

The cl-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,

(cl-defstruct person name age sex)

defines a struct type called person that 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, for example:

(cl-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, and 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 that 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 [Declarations](#Declarations).

(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 that 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.

(cl-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.)

(cl-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 [Program Structure](#Program-Structure), 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 that don't specify defaults take their defaults from the slot descriptor. It is valid to include arguments that don't correspond to slot names; these are useful if they are referred to in the defaults for optional, keyword, or &aux arguments that *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.

(cl-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 that 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, cl-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 cl-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:

(cl-defstruct person name (age 0) sex)  
 => person  
 (cl-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 that 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.

(cl-defstruct (person1) name age sex)  
 (cl-defstruct (person2 (:type list) :named) name age sex)  
 (cl-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, cl-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 :includes 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 cl-defstruct facility of this package is entirely compatible with that of Common Lisp.

The cl-defstruct package also provides a few structure introspection functions.

— Function: **cl-struct-sequence-type** struct-type

This function returns the underlying data structure for struct-type, which is a symbol. It returns vector or list, or nil if struct-type is not actually a structure.

— Function: **cl-struct-slot-info** struct-type

This function returns a list of slot descriptors for structure struct-type. Each entry in the list is (name . opts), where name is the name of the slot and opts is the list of slot options given to defstruct. Dummy entries represent the slots used for the struct name and that are skipped to implement :initial-offset.

— Function: **cl-struct-slot-offset** struct-type slot-name

Return the offset of slot slot-name in struct-type. The returned zero-based slot index is relative to the start of the structure data type and is adjusted for any structure name and :initial-offset slots. Signal error if struct struct-type does not contain slot-name.

— Function: **cl-struct-slot-value** struct-type slot-name inst

Return the value of slot slot-name in inst of struct-type. struct and slot-name are symbols. inst is a structure instance. This routine is also a setf place. Can signal the same errors as cl-struct-slot-offset.

Next: [Efficiency Concerns](#Efficiency-Concerns), Previous: [Structures](#Structures), Up: [Top](#Top)

## 12 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.

— Macro: **cl-assert** test-form [show-args string args...]

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, cl-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:

(cl-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.

— Macro: **cl-check-type** form type [string]

This form verifies that form evaluates to a value of type type. If so, it returns nil. If not, cl-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:

(cl-check-type x (integer 1 \*) "a positive integer")

See [Type Predicates](#Type-Predicates), 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.

Next: [Common Lisp Compatibility](#Common-Lisp-Compatibility), Previous: [Assertions](#Assertions), Up: [Top](#Top)

## Appendix A Efficiency Concerns

### A.1 Macros

Many of the advanced features of this package, such as cl-defun, cl-loop, etc., 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 form

(cl-incf i n)

is expanded at compile-time to the Lisp form

(setq i (+ i n))

which is the most efficient ways of doing this operation in Lisp. Thus, there is no performance penalty for using the more readable cl-incf form 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. A loop using cl-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.

— Function: **cl-prettyexpand** form &optional full

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 that appear in the form. The easiest way to use this function is to go to the \*scratch\* buffer and type, say,

(cl-prettyexpand '(cl-loop for x below 10 collect x))

and type C-x C-e immediately after the closing parenthesis; an expansion similar to:

(cl-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 cl-block macro is expanded differently in the interpreter and compiler, so cl-prettyexpand just leaves it alone. The temporary variable G1004 was created by cl-gensym.)

If the optional argument full is true, then *all* macros are expanded, including cl-block, cl-eval-when, and compiler macros. Expansion is done as if form were a top-level form in a file being compiled.

Note that cl-adjoin, cl-caddr, and cl-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.

The Common Lisp standard (as embodied in Steele's book) uses the phrase “it is an error if” to indicate a situation that 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, cl-do variable specifiers are supposed to be lists of one, two, or three forms; extra forms are ignored by this package rather than signaling a syntax error. 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 cl-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 cl-find and cl-member do check their keyword arguments for validity.

### A.3 Compiler Optimizations

Changing the value of byte-optimize from the default t is highly discouraged; many of the Common Lisp macros emit code that can be improved by optimization. In particular, cl-blocks (whether explicit or implicit in constructs like cl-defun and cl-loop) carry a fair run-time penalty; the byte-compiler removes cl-blocks that are not actually referenced by cl-return or cl-return-from inside the block.

Next: [Porting Common Lisp](#Porting-Common-Lisp), Previous: [Efficiency Concerns](#Efficiency-Concerns), Up: [Top](#Top)

## Appendix B Common Lisp Compatibility

The following is a list of all known incompatibilities between this package and Common Lisp as documented in Steele (2nd edition).

The word cl-defun is required instead of defun in order to use extended Common Lisp argument lists in a function. Likewise, cl-defmacro and cl-function are versions of those forms which understand full-featured argument lists. The &whole keyword does not work in cl-defmacro argument lists (except inside recursive argument lists).

The equal predicate does not distinguish between IEEE floating-point plus and minus zero. The cl-equalp predicate has several differences with Common Lisp; see [Predicates](#Predicates).

The cl-do-all-symbols form is the same as cl-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 cl-do-all-symbols to locate any but the default obarray.

The cl-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 cl-values or cl-values-list is always used to return to a cl-multiple-value-bind or other multiple-value receiver; if cl-values is used without cl-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 cl--gensym-counter starts out with zero.

The cl-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 cl-check-type is treated differently.

Next: [Obsolete Features](#Obsolete-Features), Previous: [Common Lisp Compatibility](#Common-Lisp-Compatibility), Up: [Top](#Top)

## Appendix C 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 [Common Lisp Compatibility](#Common-Lisp-Compatibility). 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 to 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 entirely in 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). Traditionally, Emacs Lisp uses dynamic scoping wherein a binding to a variable is visible even inside functions called from the body. See [Dynamic Binding](elisp.html#Dynamic-Binding). Lexical binding is available since Emacs 24.1, so be sure to set lexical-binding to t if you need to emulate this aspect of Common Lisp. See [Lexical Binding](elisp.html#Lexical-Binding).
* Here is an example of a Common Lisp code fragment that would fail in Emacs Lisp if lexical-binding were set to nil:
* (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 (lambda (a) (+ a x)) list))
* With lexical binding, the two functions' usages of x are completely independent. With dynamic binding, 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.
* Internally, this package uses lexical binding so that such problems do not occur. See [Obsolete Lexical Binding](#Obsolete-Lexical-Binding), for a description of the obsolete lexical-let form that emulates a Common Lisp-style lexical binding when dynamic binding is in use.
* 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.
* 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 that 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.
* 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 cl-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 has 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 in Common Lisp where Emacs Lisp uses ?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. However, the EIEIO package (see [Introduction](eieio.html#Top)) does implement some CLOS functionality.
* 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, prog, ldb/dpb, 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.

Next: [GNU Free Documentation License](#GNU-Free-Documentation-License), Previous: [Porting Common Lisp](#Porting-Common-Lisp), Up: [Top](#Top)

## Appendix D Obsolete Features

This section describes some features of the package that are obsolete and should not be used in new code. They are either only provided by the old cl.el entry point, not by the newer cl-lib.el; or where versions with a `cl-' prefix do exist they do not behave in exactly the same way.

* [Obsolete Lexical Binding](#Obsolete-Lexical-Binding): An approximation of lexical binding.
* [Obsolete Macros](#Obsolete-Macros): Obsolete macros.
* [Obsolete Setf Customization](#Obsolete-Setf-Customization): Obsolete ways to customize setf.

Next: [Obsolete Macros](#Obsolete-Macros), Up: [Obsolete Features](#Obsolete-Features)

### D.1 Obsolete Lexical Binding

The following macros are extensions to Common Lisp, where all bindings are lexical unless declared otherwise. These features are likewise obsolete since the introduction of true lexical binding in Emacs 24.1.

— Macro: **lexical-let** (bindings...) forms...

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 (see [Closures](elisp.html#Closures)). 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))  
 (cl-function (lambda (&optional (m 1)) (cl-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 (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.

— Macro: **lexical-let\*** (bindings...) forms...

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

Next: [Obsolete Setf Customization](#Obsolete-Setf-Customization), Previous: [Obsolete Lexical Binding](#Obsolete-Lexical-Binding), Up: [Obsolete Features](#Obsolete-Features)

### D.2 Obsolete Macros

The following macros are obsolete, and are replaced by versions with a `cl-' prefix that do not behave in exactly the same way. Consequently, the cl.el versions are not simply aliases to the cl-lib.el versions.

— Macro: **flet** (bindings...) forms...

This macro is replaced by cl-flet (see [Function Bindings](#Function-Bindings)), which behaves the same way as Common Lisp's flet. This flet takes the same arguments as cl-flet, but does not behave in precisely the same way.

While flet in Common Lisp establishes a lexical function binding, this flet makes a dynamic binding (it dates from a time before Emacs had lexical binding). The result is that flet affects indirect calls to a function as well as calls directly inside the flet form itself.

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.

For those cases where the dynamic scoping of flet is desired, cl-flet is clearly not a substitute. The most direct replacement would be instead to use cl-letf to temporarily rebind (symbol-function 'fun). But in most cases, a better substitute is to use advice, such as:

(defvar my-fun-advice-enable nil)  
 (add-advice 'fun :around  
 (lambda (orig &rest args)  
 (if my-fun-advice-enable (do-something)  
 (apply orig args))))

so that you can then replace the flet with a simple dynamically scoped binding of my-fun-advice-enable.

Note that many primitives (e.g., +) have special byte-compile handling. Attempts to redefine such functions using flet, cl-letf, or advice will fail when byte-compiled.

— Macro: **labels** (bindings...) forms...

This macro is replaced by cl-labels (see [Function Bindings](#Function-Bindings)), which behaves the same way as Common Lisp's labels. This labels takes the same arguments as cl-labels, but does not behave in precisely the same way.

This version of labels uses the obsolete lexical-let form (see [Obsolete Lexical Binding](#Obsolete-Lexical-Binding)), rather than the true lexical binding that cl-labels uses.

Previous: [Obsolete Macros](#Obsolete-Macros), Up: [Obsolete Features](#Obsolete-Features)

### D.3 Obsolete Ways to Customize 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. In Emacs, these are obsolete, replaced by various features of gv.el in Emacs 24.3. See [Adding Generalized Variables](elisp.html#Adding-Generalized-Variables).

— Macro: **define-modify-macro** name arglist function [doc-string]

This macro defines a “read-modify-write” macro similar to cl-incf and cl-decf. You can replace this macro with gv-letplace.

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

(cl-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.

The above incf example could be written using gv-letplace as:

(defmacro incf (place &optional n)  
 (gv-letplace (getter setter) place  
 (macroexp-let2 nil v (or n 1)  
 (funcall setter `(+ ,v ,getter)))))

— Macro: **defsetf** access-fn update-fn

This is the simpler of two defsetf forms, and is replaced by gv-define-simple-setter.

With access-fn the name of a function that 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 that 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 return value of update-fn 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 are:

(defsetf car setcar)  
 (defsetf buffer-name rename-buffer t)

These translate directly to gv-define-simple-setter:

(gv-define-simple-setter car setcar)  
 (gv-define-simple-setter buffer-name rename-buffer t)

— Macro: **defsetf** access-fn arglist (store-var) forms...

This is the second, more complex, form of defsetf. It can be replaced by gv-define-setter.

This form of defsetf is rather like defmacro except for the additional store-var argument. The forms should return a Lisp form that 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 cl-incf that invoke this setf-method will insert temporary variables as needed to make sure the apparent order of evaluation is preserved.

Another standard example:

(defsetf nth (n x) (store)  
 `(setcar (nthcdr ,n ,x) ,store))

You could write this using gv-define-setter as:

(gv-define-setter nth (store n x)  
 `(setcar (nthcdr ,n ,x) ,store))

— Macro: **define-setf-method** access-fn arglist forms...

This is the most general way to create new place forms. You can replace this by gv-define-setter or gv-define-expander.

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 that stores the contents of the store variable into the generalized variable, assuming the temporaries have been bound as described above.
5. A Lisp form that 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. (Note that the setf implementation provided by gv.el does not use this five item format. Its use here is only for backwards compatibility.)

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 cl-gensym. Macros like cl-incf that 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.

Next: [Function Index](#Function-Index), Previous: [Obsolete Features](#Obsolete-Features), Up: [Top](#Top)

## Appendix E GNU Free Documentation License

Version 1.3, 3 November 2008

Copyright © 2000, 2001, 2002, 2007, 2008 Free Software Foundation, Inc.  
 http://fsf.org/  
   
 Everyone is permitted to copy and distribute verbatim copies  
 of this license document, but changing it is not allowed.

1. PREAMBLE

* The purpose of this License is to make a manual, textbook, or other functional and useful document free in the sense of freedom: to assure everyone the effective freedom to copy and redistribute it, with or without modifying it, either commercially or noncommercially. Secondarily, this License preserves for the author and publisher a way to get credit for their work, while not being considered responsible for modifications made by others.
* This License is a kind of “copyleft”, which means that derivative works of the document must themselves be free in the same sense. It complements the GNU General Public License, which is a copyleft license designed for free software.
* We have designed this License in order to use it for manuals for free software, because free software needs free documentation: a free program should come with manuals providing the same freedoms that the software does. But this License is not limited to software manuals; it can be used for any textual work, regardless of subject matter or whether it is published as a printed book. We recommend this License principally for works whose purpose is instruction or reference.

1. APPLICABILITY AND DEFINITIONS

* This License applies to any manual or other work, in any medium, that contains a notice placed by the copyright holder saying it can be distributed under the terms of this License. Such a notice grants a world-wide, royalty-free license, unlimited in duration, to use that work under the conditions stated herein. The “Document”, below, refers to any such manual or work. Any member of the public is a licensee, and is addressed as “you”. You accept the license if you copy, modify or distribute the work in a way requiring permission under copyright law.
* A “Modified Version” of the Document means any work containing the Document or a portion of it, either copied verbatim, or with modifications and/or translated into another language.
* A “Secondary Section” is a named appendix or a front-matter section of the Document that deals exclusively with the relationship of the publishers or authors of the Document to the Document's overall subject (or to related matters) and contains nothing that could fall directly within that overall subject. (Thus, if the Document is in part a textbook of mathematics, a Secondary Section may not explain any mathematics.) The relationship could be a matter of historical connection with the subject or with related matters, or of legal, commercial, philosophical, ethical or political position regarding them.
* The “Invariant Sections” are certain Secondary Sections whose titles are designated, as being those of Invariant Sections, in the notice that says that the Document is released under this License. If a section does not fit the above definition of Secondary then it is not allowed to be designated as Invariant. The Document may contain zero Invariant Sections. If the Document does not identify any Invariant Sections then there are none.
* The “Cover Texts” are certain short passages of text that are listed, as Front-Cover Texts or Back-Cover Texts, in the notice that says that the Document is released under this License. A Front-Cover Text may be at most 5 words, and a Back-Cover Text may be at most 25 words.
* A “Transparent” copy of the Document means a machine-readable copy, represented in a format whose specification is available to the general public, that is suitable for revising the document straightforwardly with generic text editors or (for images composed of pixels) generic paint programs or (for drawings) some widely available drawing editor, and that is suitable for input to text formatters or for automatic translation to a variety of formats suitable for input to text formatters. A copy made in an otherwise Transparent file format whose markup, or absence of markup, has been arranged to thwart or discourage subsequent modification by readers is not Transparent. An image format is not Transparent if used for any substantial amount of text. A copy that is not “Transparent” is called “Opaque”.
* Examples of suitable formats for Transparent copies include plain ASCII without markup, Texinfo input format, LaTeX input format, SGML or XML using a publicly available DTD, and standard-conforming simple HTML, PostScript or PDF designed for human modification. Examples of transparent image formats include PNG, XCF and JPG. Opaque formats include proprietary formats that can be read and edited only by proprietary word processors, SGML or XML for which the DTD and/or processing tools are not generally available, and the machine-generated HTML, PostScript or PDF produced by some word processors for output purposes only.
* The “Title Page” means, for a printed book, the title page itself, plus such following pages as are needed to hold, legibly, the material this License requires to appear in the title page. For works in formats which do not have any title page as such, “Title Page” means the text near the most prominent appearance of the work's title, preceding the beginning of the body of the text.
* The “publisher” means any person or entity that distributes copies of the Document to the public.
* A section “Entitled XYZ” means a named subunit of the Document whose title either is precisely XYZ or contains XYZ in parentheses following text that translates XYZ in another language. (Here XYZ stands for a specific section name mentioned below, such as “Acknowledgements”, “Dedications”, “Endorsements”, or “History”.) To “Preserve the Title” of such a section when you modify the Document means that it remains a section “Entitled XYZ” according to this definition.
* The Document may include Warranty Disclaimers next to the notice which states that this License applies to the Document. These Warranty Disclaimers are considered to be included by reference in this License, but only as regards disclaiming warranties: any other implication that these Warranty Disclaimers may have is void and has no effect on the meaning of this License.

1. VERBATIM COPYING

* You may copy and distribute the Document in any medium, either commercially or noncommercially, provided that this License, the copyright notices, and the license notice saying this License applies to the Document are reproduced in all copies, and that you add no other conditions whatsoever to those of this License. You may not use technical measures to obstruct or control the reading or further copying of the copies you make or distribute. However, you may accept compensation in exchange for copies. If you distribute a large enough number of copies you must also follow the conditions in section 3.
* You may also lend copies, under the same conditions stated above, and you may publicly display copies.

1. COPYING IN QUANTITY

* If you publish printed copies (or copies in media that commonly have printed covers) of the Document, numbering more than 100, and the Document's license notice requires Cover Texts, you must enclose the copies in covers that carry, clearly and legibly, all these Cover Texts: Front-Cover Texts on the front cover, and Back-Cover Texts on the back cover. Both covers must also clearly and legibly identify you as the publisher of these copies. The front cover must present the full title with all words of the title equally prominent and visible. You may add other material on the covers in addition. Copying with changes limited to the covers, as long as they preserve the title of the Document and satisfy these conditions, can be treated as verbatim copying in other respects.
* If the required texts for either cover are too voluminous to fit legibly, you should put the first ones listed (as many as fit reasonably) on the actual cover, and continue the rest onto adjacent pages.
* If you publish or distribute Opaque copies of the Document numbering more than 100, you must either include a machine-readable Transparent copy along with each Opaque copy, or state in or with each Opaque copy a computer-network location from which the general network-using public has access to download using public-standard network protocols a complete Transparent copy of the Document, free of added material. If you use the latter option, you must take reasonably prudent steps, when you begin distribution of Opaque copies in quantity, to ensure that this Transparent copy will remain thus accessible at the stated location until at least one year after the last time you distribute an Opaque copy (directly or through your agents or retailers) of that edition to the public.
* It is requested, but not required, that you contact the authors of the Document well before redistributing any large number of copies, to give them a chance to provide you with an updated version of the Document.

1. MODIFICATIONS

* You may copy and distribute a Modified Version of the Document under the conditions of sections 2 and 3 above, provided that you release the Modified Version under precisely this License, with the Modified Version filling the role of the Document, thus licensing distribution and modification of the Modified Version to whoever possesses a copy of it. In addition, you must do these things in the Modified Version:
  1. Use in the Title Page (and on the covers, if any) a title distinct from that of the Document, and from those of previous versions (which should, if there were any, be listed in the History section of the Document). You may use the same title as a previous version if the original publisher of that version gives permission.
  2. List on the Title Page, as authors, one or more persons or entities responsible for authorship of the modifications in the Modified Version, together with at least five of the principal authors of the Document (all of its principal authors, if it has fewer than five), unless they release you from this requirement.
  3. State on the Title page the name of the publisher of the Modified Version, as the publisher.
  4. Preserve all the copyright notices of the Document.
  5. Add an appropriate copyright notice for your modifications adjacent to the other copyright notices.
  6. Include, immediately after the copyright notices, a license notice giving the public permission to use the Modified Version under the terms of this License, in the form shown in the Addendum below.
  7. Preserve in that license notice the full lists of Invariant Sections and required Cover Texts given in the Document's license notice.
  8. Include an unaltered copy of this License.
  9. Preserve the section Entitled “History”, Preserve its Title, and add to it an item stating at least the title, year, new authors, and publisher of the Modified Version as given on the Title Page. If there is no section Entitled “History” in the Document, create one stating the title, year, authors, and publisher of the Document as given on its Title Page, then add an item describing the Modified Version as stated in the previous sentence.
  10. Preserve the network location, if any, given in the Document for public access to a Transparent copy of the Document, and likewise the network locations given in the Document for previous versions it was based on. These may be placed in the “History” section. You may omit a network location for a work that was published at least four years before the Document itself, or if the original publisher of the version it refers to gives permission.
  11. For any section Entitled “Acknowledgements” or “Dedications”, Preserve the Title of the section, and preserve in the section all the substance and tone of each of the contributor acknowledgements and/or dedications given therein.
  12. Preserve all the Invariant Sections of the Document, unaltered in their text and in their titles. Section numbers or the equivalent are not considered part of the section titles.
  13. Delete any section Entitled “Endorsements”. Such a section may not be included in the Modified Version.
  14. Do not retitle any existing section to be Entitled “Endorsements” or to conflict in title with any Invariant Section.
  15. Preserve any Warranty Disclaimers.
* If the Modified Version includes new front-matter sections or appendices that qualify as Secondary Sections and contain no material copied from the Document, you may at your option designate some or all of these sections as invariant. To do this, add their titles to the list of Invariant Sections in the Modified Version's license notice. These titles must be distinct from any other section titles.
* You may add a section Entitled “Endorsements”, provided it contains nothing but endorsements of your Modified Version by various parties—for example, statements of peer review or that the text has been approved by an organization as the authoritative definition of a standard.
* You may add a passage of up to five words as a Front-Cover Text, and a passage of up to 25 words as a Back-Cover Text, to the end of the list of Cover Texts in the Modified Version. Only one passage of Front-Cover Text and one of Back-Cover Text may be added by (or through arrangements made by) any one entity. If the Document already includes a cover text for the same cover, previously added by you or by arrangement made by the same entity you are acting on behalf of, you may not add another; but you may replace the old one, on explicit permission from the previous publisher that added the old one.
* The author(s) and publisher(s) of the Document do not by this License give permission to use their names for publicity for or to assert or imply endorsement of any Modified Version.

1. COMBINING DOCUMENTS

* You may combine the Document with other documents released under this License, under the terms defined in section 4 above for modified versions, provided that you include in the combination all of the Invariant Sections of all of the original documents, unmodified, and list them all as Invariant Sections of your combined work in its license notice, and that you preserve all their Warranty Disclaimers.
* The combined work need only contain one copy of this License, and multiple identical Invariant Sections may be replaced with a single copy. If there are multiple Invariant Sections with the same name but different contents, make the title of each such section unique by adding at the end of it, in parentheses, the name of the original author or publisher of that section if known, or else a unique number. Make the same adjustment to the section titles in the list of Invariant Sections in the license notice of the combined work.
* In the combination, you must combine any sections Entitled “History” in the various original documents, forming one section Entitled “History”; likewise combine any sections Entitled “Acknowledgements”, and any sections Entitled “Dedications”. You must delete all sections Entitled “Endorsements.”

1. COLLECTIONS OF DOCUMENTS

* You may make a collection consisting of the Document and other documents released under this License, and replace the individual copies of this License in the various documents with a single copy that is included in the collection, provided that you follow the rules of this License for verbatim copying of each of the documents in all other respects.
* You may extract a single document from such a collection, and distribute it individually under this License, provided you insert a copy of this License into the extracted document, and follow this License in all other respects regarding verbatim copying of that document.

1. AGGREGATION WITH INDEPENDENT WORKS

* A compilation of the Document or its derivatives with other separate and independent documents or works, in or on a volume of a storage or distribution medium, is called an “aggregate” if the copyright resulting from the compilation is not used to limit the legal rights of the compilation's users beyond what the individual works permit. When the Document is included in an aggregate, this License does not apply to the other works in the aggregate which are not themselves derivative works of the Document.
* If the Cover Text requirement of section 3 is applicable to these copies of the Document, then if the Document is less than one half of the entire aggregate, the Document's Cover Texts may be placed on covers that bracket the Document within the aggregate, or the electronic equivalent of covers if the Document is in electronic form. Otherwise they must appear on printed covers that bracket the whole aggregate.

1. TRANSLATION

* Translation is considered a kind of modification, so you may distribute translations of the Document under the terms of section 4. Replacing Invariant Sections with translations requires special permission from their copyright holders, but you may include translations of some or all Invariant Sections in addition to the original versions of these Invariant Sections. You may include a translation of this License, and all the license notices in the Document, and any Warranty Disclaimers, provided that you also include the original English version of this License and the original versions of those notices and disclaimers. In case of a disagreement between the translation and the original version of this License or a notice or disclaimer, the original version will prevail.
* If a section in the Document is Entitled “Acknowledgements”, “Dedications”, or “History”, the requirement (section 4) to Preserve its Title (section 1) will typically require changing the actual title.

1. TERMINATION

* You may not copy, modify, sublicense, or distribute the Document except as expressly provided under this License. Any attempt otherwise to copy, modify, sublicense, or distribute it is void, and will automatically terminate your rights under this License.
* However, if you cease all violation of this License, then your license from a particular copyright holder is reinstated (a) provisionally, unless and until the copyright holder explicitly and finally terminates your license, and (b) permanently, if the copyright holder fails to notify you of the violation by some reasonable means prior to 60 days after the cessation.
* Moreover, your license from a particular copyright holder is reinstated permanently if the copyright holder notifies you of the violation by some reasonable means, this is the first time you have received notice of violation of this License (for any work) from that copyright holder, and you cure the violation prior to 30 days after your receipt of the notice.
* Termination of your rights under this section does not terminate the licenses of parties who have received copies or rights from you under this License. If your rights have been terminated and not permanently reinstated, receipt of a copy of some or all of the same material does not give you any rights to use it.

1. FUTURE REVISIONS OF THIS LICENSE

* The Free Software Foundation may publish new, revised versions of the GNU Free Documentation License from time to time. Such new versions will be similar in spirit to the present version, but may differ in detail to address new problems or concerns. See <http://www.gnu.org/copyleft/>.
* Each version of the License is given a distinguishing version number. If the Document specifies that a particular numbered version of this License “or any later version” applies to it, you have the option of following the terms and conditions either of that specified version or of any later version that has been published (not as a draft) by the Free Software Foundation. If the Document does not specify a version number of this License, you may choose any version ever published (not as a draft) by the Free Software Foundation. If the Document specifies that a proxy can decide which future versions of this License can be used, that proxy's public statement of acceptance of a version permanently authorizes you to choose that version for the Document.

1. RELICENSING

* “Massive Multiauthor Collaboration Site” (or “MMC Site”) means any World Wide Web server that publishes copyrightable works and also provides prominent facilities for anybody to edit those works. A public wiki that anybody can edit is an example of such a server. A “Massive Multiauthor Collaboration” (or “MMC”) contained in the site means any set of copyrightable works thus published on the MMC site.
* “CC-BY-SA” means the Creative Commons Attribution-Share Alike 3.0 license published by Creative Commons Corporation, a not-for-profit corporation with a principal place of business in San Francisco, California, as well as future copyleft versions of that license published by that same organization.
* “Incorporate” means to publish or republish a Document, in whole or in part, as part of another Document.
* An MMC is “eligible for relicensing” if it is licensed under this License, and if all works that were first published under this License somewhere other than this MMC, and subsequently incorporated in whole or in part into the MMC, (1) had no cover texts or invariant sections, and (2) were thus incorporated prior to November 1, 2008.
* The operator of an MMC Site may republish an MMC contained in the site under CC-BY-SA on the same site at any time before August 1, 2009, provided the MMC is eligible for relicensing.

### ADDENDUM: How to use this License for your documents

To use this License in a document you have written, include a copy of the License in the document and put the following copyright and license notices just after the title page:

Copyright (C) year your name.  
 Permission is granted to copy, distribute and/or modify this document  
 under the terms of the GNU Free Documentation License, Version 1.3  
 or any later version published by the Free Software Foundation;  
 with no Invariant Sections, no Front-Cover Texts, and no Back-Cover  
 Texts. A copy of the license is included in the section entitled ``GNU  
 Free Documentation License''.

If you have Invariant Sections, Front-Cover Texts and Back-Cover Texts, replace the “with...Texts.” line with this:

with the Invariant Sections being list their titles, with  
 the Front-Cover Texts being list, and with the Back-Cover Texts  
 being list.

If you have Invariant Sections without Cover Texts, or some other combination of the three, merge those two alternatives to suit the situation.

If your document contains nontrivial examples of program code, we recommend releasing these examples in parallel under your choice of free software license, such as the GNU General Public License, to permit their use in free software.

Next: [Variable Index](#Variable-Index), Previous: [GNU Free Documentation License](#GNU-Free-Documentation-License), Up: [Top](#Top)

## Function Index

* [cl-acons](#index-cl_002dacons-176): [Association Lists](#Association-Lists)
* [cl-adjoin](#index-cl_002dadjoin-160): [Lists as Sets](#Lists-as-Sets)
* [cl-assert](#index-cl_002dassert-183): [Assertions](#Assertions)
* [cl-assoc](#index-cl_002dassoc-170): [Association Lists](#Association-Lists)
* [cl-assoc-if](#index-cl_002dassoc_002dif-172): [Association Lists](#Association-Lists)
* [cl-assoc-if-not](#index-cl_002dassoc_002dif_002dnot-173): [Association Lists](#Association-Lists)
* [cl-block](#index-cl_002dblock-37): [Blocks and Exits](#Blocks-and-Exits)
* [cl-caddr](#index-cl_002dcaddr-139): [List Functions](#List-Functions)
* [cl-callf](#index-cl_002dcallf-25): [Modify Macros](#Modify-Macros)
* [cl-callf2](#index-cl_002dcallf2-26): [Modify Macros](#Modify-Macros)
* [cl-case](#index-cl_002dcase-32): [Conditionals](#Conditionals)
* [cl-ceiling](#index-cl_002dceiling-77): [Numerical Functions](#Numerical-Functions)
* [cl-check-type](#index-cl_002dcheck_002dtype-184): [Assertions](#Assertions)
* [cl-coerce](#index-cl_002dcoerce-13): [Type Predicates](#Type-Predicates)
* [cl-compiler-macroexpand](#index-cl_002dcompiler_002dmacroexpand-56): [Macros](#Macros)
* [cl-concatenate](#index-cl_002dconcatenate-108): [Sequence Functions](#Sequence-Functions)
* [cl-copy-list](#index-cl_002dcopy_002dlist-146): [List Functions](#List-Functions)
* [cl-count](#index-cl_002dcount-127): [Searching Sequences](#Searching-Sequences)
* [cl-count-if](#index-cl_002dcount_002dif-132): [Searching Sequences](#Searching-Sequences)
* [cl-count-if-not](#index-cl_002dcount_002dif_002dnot-133): [Searching Sequences](#Searching-Sequences)
* [cl-decf](#index-cl_002ddecf-19): [Modify Macros](#Modify-Macros)
* [cl-declaim](#index-cl_002ddeclaim-58): [Declarations](#Declarations)
* [cl-declare](#index-cl_002ddeclare-59): [Declarations](#Declarations)
* [cl-define-compiler-macro](#index-cl_002ddefine_002dcompiler_002dmacro-55): [Macros](#Macros)
* [cl-defmacro](#index-cl_002ddefmacro-6): [Argument Lists](#Argument-Lists)
* [cl-defstruct](#index-cl_002ddefstruct-178): [Structures](#Structures)
* [cl-defsubst](#index-cl_002ddefsubst-5): [Argument Lists](#Argument-Lists)
* [cl-deftype](#index-cl_002ddeftype-14): [Type Predicates](#Type-Predicates)
* [cl-defun](#index-cl_002ddefun-3): [Argument Lists](#Argument-Lists)
* [cl-delete](#index-cl_002ddelete-112): [Sequence Functions](#Sequence-Functions)
* [cl-delete-duplicates](#index-cl_002ddelete_002dduplicates-118): [Sequence Functions](#Sequence-Functions)
* [cl-delete-if](#index-cl_002ddelete_002dif-115): [Sequence Functions](#Sequence-Functions)
* [cl-delete-if-not](#index-cl_002ddelete_002dif_002dnot-116): [Sequence Functions](#Sequence-Functions)
* [cl-destructuring-bind](#index-cl_002ddestructuring_002dbind-52): [Macros](#Macros)
* [cl-digit-char-p](#index-cl_002ddigit_002dchar_002dp-72): [Predicates on Numbers](#Predicates-on-Numbers)
* [cl-do](#index-cl_002ddo-42): [Iteration](#Iteration)
* [cl-do\*](#index-cl_002ddo_002a-43): [Iteration](#Iteration)
* [cl-do-all-symbols](#index-cl_002ddo_002dall_002dsymbols-47): [Iteration](#Iteration)
* [cl-do-symbols](#index-cl_002ddo_002dsymbols-46): [Iteration](#Iteration)
* [cl-dolist](#index-cl_002ddolist-44): [Iteration](#Iteration)
* [cl-dotimes](#index-cl_002ddotimes-45): [Iteration](#Iteration)
* [cl-ecase](#index-cl_002decase-33): [Conditionals](#Conditionals)
* [cl-endp](#index-cl_002dendp-142): [List Functions](#List-Functions)
* [cl-equalp](#index-cl_002dequalp-15): [Equality Predicates](#Equality-Predicates)
* [cl-etypecase](#index-cl_002detypecase-35): [Conditionals](#Conditionals)
* [cl-eval-when](#index-cl_002deval_002dwhen-9): [Time of Evaluation](#Time-of-Evaluation)
* [cl-evenp](#index-cl_002devenp-71): [Predicates on Numbers](#Predicates-on-Numbers)
* [cl-every](#index-cl_002devery-103): [Mapping over Sequences](#Mapping-over-Sequences)
* [cl-fill](#index-cl_002dfill-109): [Sequence Functions](#Sequence-Functions)
* [cl-find](#index-cl_002dfind-125): [Searching Sequences](#Searching-Sequences)
* [cl-find-if](#index-cl_002dfind_002dif-128): [Searching Sequences](#Searching-Sequences)
* [cl-find-if-not](#index-cl_002dfind_002dif_002dnot-129): [Searching Sequences](#Searching-Sequences)
* [cl-first](#index-cl_002dfirst-140): [List Functions](#List-Functions)
* [cl-flet](#index-cl_002dflet-28): [Function Bindings](#Function-Bindings)
* [cl-float-limits](#index-cl_002dfloat_002dlimits-86): [Implementation Parameters](#Implementation-Parameters)
* [cl-floor](#index-cl_002dfloor-76): [Numerical Functions](#Numerical-Functions)
* [cl-function](#index-cl_002dfunction-7): [Argument Lists](#Argument-Lists)
* [cl-gcd](#index-cl_002dgcd-73): [Numerical Functions](#Numerical-Functions)
* [cl-gensym](#index-cl_002dgensym-66): [Creating Symbols](#Creating-Symbols)
* [cl-gentemp](#index-cl_002dgentemp-67): [Creating Symbols](#Creating-Symbols)
* [cl-get](#index-cl_002dget-62): [Property Lists](#Property-Lists)
* [cl-getf](#index-cl_002dgetf-64): [Property Lists](#Property-Lists)
* [cl-incf](#index-cl_002dincf-18): [Modify Macros](#Modify-Macros)
* [cl-intersection](#index-cl_002dintersection-163): [Lists as Sets](#Lists-as-Sets)
* [cl-isqrt](#index-cl_002disqrt-75): [Numerical Functions](#Numerical-Functions)
* [cl-iter-defun](#index-cl_002diter_002ddefun-4): [Argument Lists](#Argument-Lists)
* [cl-labels](#index-cl_002dlabels-29): [Function Bindings](#Function-Bindings)
* [cl-lcm](#index-cl_002dlcm-74): [Numerical Functions](#Numerical-Functions)
* [cl-ldiff](#index-cl_002dldiff-145): [List Functions](#List-Functions)
* [cl-letf](#index-cl_002dletf-23): [Modify Macros](#Modify-Macros)
* [cl-letf\*](#index-cl_002dletf_002a-24): [Modify Macros](#Modify-Macros)
* [cl-list\*](#index-cl_002dlist_002a-144): [List Functions](#List-Functions)
* [cl-list-length](#index-cl_002dlist_002dlength-143): [List Functions](#List-Functions)
* [cl-load-time-value](#index-cl_002dload_002dtime_002dvalue-11): [Time of Evaluation](#Time-of-Evaluation)
* [cl-locally](#index-cl_002dlocally-60): [Declarations](#Declarations)
* [cl-loop](#index-cl_002dloop-41): [Iteration](#Iteration)
* [cl-loop](#index-cl_002dloop-48): [Loop Basics](#Loop-Basics)
* [cl-macrolet](#index-cl_002dmacrolet-30): [Macro Bindings](#Macro-Bindings)
* [cl-make-random-state](#index-cl_002dmake_002drandom_002dstate-84): [Random Numbers](#Random-Numbers)
* [cl-map](#index-cl_002dmap-96): [Mapping over Sequences](#Mapping-over-Sequences)
* [cl-mapc](#index-cl_002dmapc-98): [Mapping over Sequences](#Mapping-over-Sequences)
* [cl-mapcan](#index-cl_002dmapcan-100): [Mapping over Sequences](#Mapping-over-Sequences)
* [cl-mapcar](#index-cl_002dmapcar-95): [Mapping over Sequences](#Mapping-over-Sequences)
* [cl-mapcon](#index-cl_002dmapcon-101): [Mapping over Sequences](#Mapping-over-Sequences)
* [cl-mapl](#index-cl_002dmapl-99): [Mapping over Sequences](#Mapping-over-Sequences)
* [cl-maplist](#index-cl_002dmaplist-97): [Mapping over Sequences](#Mapping-over-Sequences)
* [cl-member](#index-cl_002dmember-156): [Lists as Sets](#Lists-as-Sets)
* [cl-member-if](#index-cl_002dmember_002dif-157): [Lists as Sets](#Lists-as-Sets)
* [cl-member-if-not](#index-cl_002dmember_002dif_002dnot-158): [Lists as Sets](#Lists-as-Sets)
* [cl-merge](#index-cl_002dmerge-138): [Sorting Sequences](#Sorting-Sequences)
* [cl-minusp](#index-cl_002dminusp-69): [Predicates on Numbers](#Predicates-on-Numbers)
* [cl-mismatch](#index-cl_002dmismatch-134): [Searching Sequences](#Searching-Sequences)
* [cl-mod](#index-cl_002dmod-80): [Numerical Functions](#Numerical-Functions)
* [cl-multiple-value-bind](#index-cl_002dmultiple_002dvalue_002dbind-50): [Multiple Values](#Multiple-Values)
* [cl-multiple-value-setq](#index-cl_002dmultiple_002dvalue_002dsetq-51): [Multiple Values](#Multiple-Values)
* [cl-nintersection](#index-cl_002dnintersection-164): [Lists as Sets](#Lists-as-Sets)
* [cl-notany](#index-cl_002dnotany-104): [Mapping over Sequences](#Mapping-over-Sequences)
* [cl-notevery](#index-cl_002dnotevery-105): [Mapping over Sequences](#Mapping-over-Sequences)
* [cl-nset-difference](#index-cl_002dnset_002ddifference-166): [Lists as Sets](#Lists-as-Sets)
* [cl-nset-exclusive-or](#index-cl_002dnset_002dexclusive_002dor-168): [Lists as Sets](#Lists-as-Sets)
* [cl-nsublis](#index-cl_002dnsublis-155): [Substitution of Expressions](#Substitution-of-Expressions)
* [cl-nsubst](#index-cl_002dnsubst-149): [Substitution of Expressions](#Substitution-of-Expressions)
* [cl-nsubst-if](#index-cl_002dnsubst_002dif-152): [Substitution of Expressions](#Substitution-of-Expressions)
* [cl-nsubst-if-not](#index-cl_002dnsubst_002dif_002dnot-153): [Substitution of Expressions](#Substitution-of-Expressions)
* [cl-nsubstitute](#index-cl_002dnsubstitute-120): [Sequence Functions](#Sequence-Functions)
* [cl-nsubstitute-if](#index-cl_002dnsubstitute_002dif-123): [Sequence Functions](#Sequence-Functions)
* [cl-nsubstitute-if-not](#index-cl_002dnsubstitute_002dif_002dnot-124): [Sequence Functions](#Sequence-Functions)
* [cl-nunion](#index-cl_002dnunion-162): [Lists as Sets](#Lists-as-Sets)
* [cl-oddp](#index-cl_002doddp-70): [Predicates on Numbers](#Predicates-on-Numbers)
* [cl-pairlis](#index-cl_002dpairlis-177): [Association Lists](#Association-Lists)
* [cl-parse-integer](#index-cl_002dparse_002dinteger-82): [Numerical Functions](#Numerical-Functions)
* [cl-plusp](#index-cl_002dplusp-68): [Predicates on Numbers](#Predicates-on-Numbers)
* [cl-position](#index-cl_002dposition-126): [Searching Sequences](#Searching-Sequences)
* [cl-position-if](#index-cl_002dposition_002dif-130): [Searching Sequences](#Searching-Sequences)
* [cl-position-if-not](#index-cl_002dposition_002dif_002dnot-131): [Searching Sequences](#Searching-Sequences)
* [cl-prettyexpand](#index-cl_002dprettyexpand-185): [Efficiency Concerns](#Efficiency-Concerns)
* [cl-proclaim](#index-cl_002dproclaim-57): [Declarations](#Declarations)
* [cl-progv](#index-cl_002dprogv-27): [Dynamic Bindings](#Dynamic-Bindings)
* [cl-psetf](#index-cl_002dpsetf-17): [Modify Macros](#Modify-Macros)
* [cl-psetq](#index-cl_002dpsetq-16): [Assignment](#Assignment)
* [cl-pushnew](#index-cl_002dpushnew-20): [Modify Macros](#Modify-Macros)
* [cl-random](#index-cl_002drandom-83): [Random Numbers](#Random-Numbers)
* [cl-random-state-p](#index-cl_002drandom_002dstate_002dp-85): [Random Numbers](#Random-Numbers)
* [cl-rassoc](#index-cl_002drassoc-171): [Association Lists](#Association-Lists)
* [cl-rassoc-if](#index-cl_002drassoc_002dif-174): [Association Lists](#Association-Lists)
* [cl-rassoc-if-not](#index-cl_002drassoc_002dif_002dnot-175): [Association Lists](#Association-Lists)
* [cl-reduce](#index-cl_002dreduce-106): [Mapping over Sequences](#Mapping-over-Sequences)
* [cl-rem](#index-cl_002drem-81): [Numerical Functions](#Numerical-Functions)
* [cl-remf](#index-cl_002dremf-65): [Property Lists](#Property-Lists)
* [cl-remove](#index-cl_002dremove-111): [Sequence Functions](#Sequence-Functions)
* [cl-remove-duplicates](#index-cl_002dremove_002dduplicates-117): [Sequence Functions](#Sequence-Functions)
* [cl-remove-if](#index-cl_002dremove_002dif-113): [Sequence Functions](#Sequence-Functions)
* [cl-remove-if-not](#index-cl_002dremove_002dif_002dnot-114): [Sequence Functions](#Sequence-Functions)
* [cl-remprop](#index-cl_002dremprop-63): [Property Lists](#Property-Lists)
* [cl-replace](#index-cl_002dreplace-110): [Sequence Functions](#Sequence-Functions)
* [cl-rest](#index-cl_002drest-141): [List Functions](#List-Functions)
* [cl-return](#index-cl_002dreturn-39): [Blocks and Exits](#Blocks-and-Exits)
* [cl-return-from](#index-cl_002dreturn_002dfrom-38): [Blocks and Exits](#Blocks-and-Exits)
* [cl-rotatef](#index-cl_002drotatef-22): [Modify Macros](#Modify-Macros)
* [cl-round](#index-cl_002dround-79): [Numerical Functions](#Numerical-Functions)
* [cl-search](#index-cl_002dsearch-135): [Searching Sequences](#Searching-Sequences)
* [cl-set-difference](#index-cl_002dset_002ddifference-165): [Lists as Sets](#Lists-as-Sets)
* [cl-set-exclusive-or](#index-cl_002dset_002dexclusive_002dor-167): [Lists as Sets](#Lists-as-Sets)
* [cl-shiftf](#index-cl_002dshiftf-21): [Modify Macros](#Modify-Macros)
* [cl-some](#index-cl_002dsome-102): [Mapping over Sequences](#Mapping-over-Sequences)
* [cl-sort](#index-cl_002dsort-136): [Sorting Sequences](#Sorting-Sequences)
* [cl-stable-sort](#index-cl_002dstable_002dsort-137): [Sorting Sequences](#Sorting-Sequences)
* [cl-struct-sequence-type](#index-cl_002dstruct_002dsequence_002dtype-179): [Structures](#Structures)
* [cl-struct-slot-info](#index-cl_002dstruct_002dslot_002dinfo-180): [Structures](#Structures)
* [cl-struct-slot-offset](#index-cl_002dstruct_002dslot_002doffset-181): [Structures](#Structures)
* [cl-struct-slot-value](#index-cl_002dstruct_002dslot_002dvalue-182): [Structures](#Structures)
* [cl-sublis](#index-cl_002dsublis-154): [Substitution of Expressions](#Substitution-of-Expressions)
* [cl-subseq](#index-cl_002dsubseq-107): [Sequence Functions](#Sequence-Functions)
* [cl-subsetp](#index-cl_002dsubsetp-169): [Lists as Sets](#Lists-as-Sets)
* [cl-subst](#index-cl_002dsubst-148): [Substitution of Expressions](#Substitution-of-Expressions)
* [cl-subst-if](#index-cl_002dsubst_002dif-150): [Substitution of Expressions](#Substitution-of-Expressions)
* [cl-subst-if-not](#index-cl_002dsubst_002dif_002dnot-151): [Substitution of Expressions](#Substitution-of-Expressions)
* [cl-substitute](#index-cl_002dsubstitute-119): [Sequence Functions](#Sequence-Functions)
* [cl-substitute-if](#index-cl_002dsubstitute_002dif-121): [Sequence Functions](#Sequence-Functions)
* [cl-substitute-if-not](#index-cl_002dsubstitute_002dif_002dnot-122): [Sequence Functions](#Sequence-Functions)
* [cl-symbol-macrolet](#index-cl_002dsymbol_002dmacrolet-31): [Macro Bindings](#Macro-Bindings)
* [cl-tagbody](#index-cl_002dtagbody-40): [Blocks and Exits](#Blocks-and-Exits)
* [cl-tailp](#index-cl_002dtailp-159): [Lists as Sets](#Lists-as-Sets)
* [cl-the](#index-cl_002dthe-61): [Declarations](#Declarations)
* [cl-tree-equal](#index-cl_002dtree_002dequal-147): [List Functions](#List-Functions)
* [cl-truncate](#index-cl_002dtruncate-78): [Numerical Functions](#Numerical-Functions)
* [cl-typecase](#index-cl_002dtypecase-34): [Conditionals](#Conditionals)
* [cl-typep](#index-cl_002dtypep-12): [Type Predicates](#Type-Predicates)
* [cl-union](#index-cl_002dunion-161): [Lists as Sets](#Lists-as-Sets)
* [define-modify-macro](#index-define_002dmodify_002dmacro-190): [Obsolete Setf Customization](#Obsolete-Setf-Customization)
* [define-setf-method](#index-define_002dsetf_002dmethod-193): [Obsolete Setf Customization](#Obsolete-Setf-Customization)
* [defsetf](#index-defsetf-191): [Obsolete Setf Customization](#Obsolete-Setf-Customization)
* [eval-when-compile](#index-eval_002dwhen_002dcompile-10): [Time of Evaluation](#Time-of-Evaluation)
* [flet](#index-flet-188): [Obsolete Macros](#Obsolete-Macros)
* [labels](#index-labels-189): [Obsolete Macros](#Obsolete-Macros)
* [lexical-let](#index-lexical_002dlet-186): [Obsolete Lexical Binding](#Obsolete-Lexical-Binding)
* [lexical-let\*](#index-lexical_002dlet_002a-187): [Obsolete Lexical Binding](#Obsolete-Lexical-Binding)

Previous: [Function Index](#Function-Index), Up: [Top](#Top)

## Variable Index

* [cl-float-epsilon](#index-cl_002dfloat_002depsilon-93): [Implementation Parameters](#Implementation-Parameters)
* [cl-float-negative-epsilon](#index-cl_002dfloat_002dnegative_002depsilon-94): [Implementation Parameters](#Implementation-Parameters)
* [cl-least-negative-float](#index-cl_002dleast_002dnegative_002dfloat-91): [Implementation Parameters](#Implementation-Parameters)
* [cl-least-negative-normalized-float](#index-cl_002dleast_002dnegative_002dnormalized_002dfloat-92): [Implementation Parameters](#Implementation-Parameters)
* [cl-least-positive-float](#index-cl_002dleast_002dpositive_002dfloat-89): [Implementation Parameters](#Implementation-Parameters)
* [cl-least-positive-normalized-float](#index-cl_002dleast_002dpositive_002dnormalized_002dfloat-90): [Implementation Parameters](#Implementation-Parameters)
* [cl-most-negative-float](#index-cl_002dmost_002dnegative_002dfloat-88): [Implementation Parameters](#Implementation-Parameters)
* [cl-most-positive-float](#index-cl_002dmost_002dpositive_002dfloat-87): [Implementation Parameters](#Implementation-Parameters)