1.1 Generalities

This section tells you want you need to know about the routines in the file `cl.el`, so that you can use them. The distribution also includes this documentation and perhaps a few example files.

1.1.1 License, Availability, Maintenance

These files are covered by the GNU Emacs General Public License (if you don't know its terms, try `C-h C-c`) and the statement of no warranty (again, you can type `C-h C-w` if you don't know its terms) also applies to them.

I, Cesar Quiroz, the original author of the software described here, will appreciate hearing about bug reports (and fixes), suggestions and comments, applying both to the code and to this documentation. I don't promise to act on those communications, but whenever they might conduce to improvements of this software, I will make those improvements available to the general community through the Free Software Foundation. You can reach me at the following net addresses:

> quiroz@seneca.cs.rochester.edu quiroz@rochester.arpa {allegra | seismo | ... } ! rochester ! quiroz CSNET: Cesar Quiroz at node Rochester

1.1.2 Purpose and Limitations

These routines were written with two purposes in mind:

1. To make programming in Emacs Lisp even more compatible with Common Lisp. Indeed, my original motivation was to have a `do` macro.
2. To facilitate to novice Lisp programmers a chance to practice with features commonly found only in expensive implementations of Lisp.

In order to satisfy these purposes, the routines were written in such a way that it is practical to use them inside Emacs: no effort was given to implement features that could slow down the host Emacs unnecessarily nor require recoding of the Emacs Lisp interpreter.

For instance, no support is given to floating point arithmetic.

So, I have tried to implement a subset of the functionality of Common Lisp. Whatever is implemented, has syntactic and semantic properties like the equally named features of Common Lisp, but not all the relevant features have been implemented (see Section 1.12 [To Do], page 10, for some suggestions). When deciding what to include, I have tried to strike a balance between these constraints:

1. Keep it simple, I didn't want to spend much time in this idea.
2. Keep it compatible with Common Lisp.
3. Keep it flexible, so my code doesn't oppose a better implementation (when this looked hard, I just didn't implement the feature).
4. Keep an eye on the intended use of Emacs Lisp: to support an advanced editor. I don't expect that more arithmetic support will be as conducive to this goal as having better iterations and conditionals will.

For background, the reference is "Common Lisp: The Language" by Guy Steele Jr. (Digital Press, 1984). For all the features described here you can assume that the intent has been to provide the syntax and semantics of the features of like name in the book. For the most part, this documentation will concentrate on how my routines *fail* to implement Common Lisp faithfully.

1.1.2.1 Specific Limitations

Emacs Lisp and Common Lisp differ enough to make some of the emulation difficult, expensive or nearly impractical. Some specific limitations are stated here:

1. Common Lisp is lexically scoped (mostly), while Emacs Lisp is dynamically scoped. Things like `block`, `return`, `tagbody` are then practically impossible to imitate correctly (in principle, rewriting `eval`, `apply` and a couple of other functions would suffice, problem is that such rewriting amounts to a new interpreter on top of the old.) Things like '`implicit-blocks`', '`implicit-tagbodies`' and the like have not been implemented at all. Where they are needed, the most you can assume is that I tried to put '`implicit-progns`' around places where it made sense.
2. Emacs Lisp's `lambda` does not support all the possible argument markers. Similarly, `defmacro` doesn't support automatic destructuring of the calls. An approximation to a keyword-based calling style was implemented, mainly for the sake of `defstruct`, but is not general enough.
3. Emacs Lisp supports arithmetic only on integers.
4. Emacs Lisp doesn't support many of the basic types of Common Lisp. In particular, there are no arrays beyond vectors and strings (although these ones are compatible), characters are essentially small integers, etc.
5. There are no declarations in Emacs Lisp (in the sense of Common Lisp's `declare`, `proclaim`, ...) nor there is a explicit lattice of types. These limitations could be effectively overcome from Lisp code (to a extent), but I don't see them as a very pressing need, if a need at all in Emacs Lisp. `defstruct` can be used to generate new types that can be recognized at runtime.
6. The Emacs Lisp reader is not programmable. The syntax it accepts is almost standard, but it preempts '?' as a dispatching macro of sorts. The `format` function is incompatible with Common Lisp. There isn't a 'quasi-constant' notation (the usual `backquote` of Common Lisp). None of these differences causes any problems when writing Emacs Lisp (although the lack of backquoting is felt sorely), but they oppose a Common Lisp emulation.

1.1.3 Loading and Compiling

The file `cl.el` provides the '`cl`' feature, you can use this to test whether these routines have been loaded, or to load them from your code (by means of `(require 'cl)`). The compiled file is a little larger than 50K bytes.

If you need to recompile the sources, make sure you load them first on the Emacs that will do the recompilation. This is because many syntactic characteristics (like the special forms) have been implemented as macros and you need to make sure that macros are known to the compiler before they are used.

These extensions work correctly when interpreted in a GNU Emacs of version 17.64 or beyond. Compiling them requires a more recent byte compiler, preferably one strictly younger than version 18.XX.

1.1.4 On Line Help

The routines in this file have documentation strings, so you can (and should) consult them for the exact syntax allowed. That information is not repeated in this manual. Some of the routines are also documented explicitly in the Common Lisp reference, their doc-strings begin with '`[cl]`' to represent this fact.

The rest (those without the '`[cl]`' mark) are auxiliary functions or macros used by the rest of the implementation. They are not constrained by any standard and are advertised only in as much as they can be useful in other applications.

Each of the following sections contains a subsection called 'Features Provided'. It lists briefly the user-visible features of this implementation. In its entries, names by themselves refer to functions. Macros and variables are identified by a 'MACRO' or a 'VARIABLE' ahead of their names.

1.2 Symbols

The most important omission is that of a *packages* mechanism. (For a possible implementation, see Section 1.12 [To Do], page 10) Whenever a Common Lisp function expects a package, I have substituted an obarray. There is a hack to have pseudo-keywords, see below.

There are two other notorious omissions over which I haven't lost any sleep. The first is the lack of a `remprop` function, which could be easily provided if needed. The second is the lack of ways to modify the print name of a symbol. This one would probably be good only to introduce strange bugs, so I don't miss it.

1.2.1 Features Provided

`VARIABLE *gensym-index*`
`VARIABLE *gensym-prefix*`
`VARIABLE *gentemp-index*`
`VARIABLE *gentemp-prefix*`
: These variables are used to keep the state of the generator of new names. Better leave them alone.

`gensym`
`gentemp`
: These do the same as the Common Lisp names of like names.

`MACRO defkeyword`
`keyword-of`
`keywordp`
: These provide the pseudo-keywords implementation.

1.2.2 Keywords

The lack of packages makes it difficult to implement keywords correctly. I have provided a macro `defkeyword` that takes a symbol and makes sure it evaluates to itself. (So, it is like

`defconst`.) If your programs ever need keywords, put a bunch of calls to `defkeyword` at the beginning of your code, so when loaded they will be in effect.

The (standard) predicate `keywordp` tests to see if the given symbol's name begins with a colon and then ensures that it evaluates to itself.

The function `keyword-of` takes a symbol and returns a keyword of like name.

```
 (keyword-of 'foo)
:foo
 (keyword-of ':bar)
:bar
```

This feature was added mainly to support `defstruct` and the tests of the sequence functions. It is fragile and easy to fool.

1.2.3 New Symbols

A common need (especially when writing macros) is to be able to invent new names for things. I provide the `gensym` and `gentemp` functions. The global state needed is kept in the variables `*gentemp-index*`, `*gentemp-prefix*`, `*gensym-index*` and `*gensym-prefix*`. Changing them, especially the index ones, is a very bad idea. I am not providing the Common Lisp default prefixes ('G' for `gensym` and 'T' for `gentemp`) because of debugging paranoia. My default prefixes are harder to come by when giving sane names to things.

1.3 Lists

Lists (indeed, conses) are so deeply involved in Lisp that there seems to be little need to justify improving the list handling of a Lisp.

Common Lisp, however, is a rather huge Lisp. I haven't provided all the functions in the chapter of lists, mainly because some of them could be implemented correctly only if keyword arguments were supported. That explains why I haven't rushed to provide `subst`, `sublis`, etc. Also, that explains the rather temporary nature of the implementation of `member` and `adjoin`. I will welcome any efforts to extend this work.

1.3.1 Features Provided

`endp`
`list*`
`list-length`

Very standard utilities. List* has proven especially useful to overcome the lack of a real `backquote`. In addition, things that usually required the relatively clumsy

```
(cons 'a (cons 'b oldlist))
(append (list a b) oldlist)
```

can now be simply put:

```
(list* 'a 'b oldlist)
```

See also `acons`.

`member` Another well known function. Supports test with `eql` only.

`acons`
`pairlis` These two are part of the standards association lists implementation. I am leaving `sublis` as an exercise for the reader.

`adjoin` Done mainly for the sake of `pushnew`.

`butlast`
`last`
`ldiff` Occasionally useful ways to access the last cons or a specified tail of a list. I don't remember why there isn't a `tailp` here.

`c[ad][ad][ad][ad]r, up to four a's or d's`
These 28 functions (and their setf inverses) have been provided once and for all. Many packages contributed to Emacs Lisp contain macros that implement some of these, I think this code will make most of them unnecessary.

`first`
`rest`
`second`
`third`
`fourth`
`fifth`
`sixth`
`seventh`
`eighth`
`ninth`
`tenth` More standard accessors (and their setf inverses). Not particularly useful but easy to provide.

`setnth`
`setnthcdr`
These functions are non-standard. They are here for `defsetf` purposes only, but they might be useful on their own.

1.4 Sequences

Sequences are partly supported in Emacs Lisp (see, for instance, the `elt` function). This limited support is compatible with Common Lisp, so it should be easy to extend. However, the lack of keyword arguments makes many of the functions impossible so far (but, as mentioned below, a basic framework for that extension is provided here).

The functionality really provided here is given by the functions (essentially, predicates) `every`, `some`, `notevery`, `notany`. I have found them useful countless times, so I thought to provide them before anything else.

That still leaves many omissions, though.

1.4.1 Features Provided

`every`
`notany`
`notevery`
`some` Extremely useful functions. If your favorite Lisp doesn't have them, you are missing a lot.

`setelt` A setf-inverse to `elt`.

`add-to-klist`
`build-klist`
`extract-from-klist`
A *klist* is just an alist whose keys are keywords. I based the pseudo-keyword argument support of `defstruct` on this idea, but their best fit is here, as they could help to write the remaining sequence-handling functions (`find`, `substitute`, ...) that I didn't provide for the lack of a good keyword arguments mechanism.

`elt-satisfies-if-not-p`
`elt-satisfies-if-p`
`elt-satisfies-test-p`
`elts-match-under-klist-p`
The Common Lisp book defines some of the semantics of sequence functions in terms of satisfaction of certain tests. These predicates provide that functionality, but I haven't integrated them with the rest of the extensions. However, I thought it was better to include them anyway, as they can serve somebody else as a starting point.

1.5 Conditionals

An elementary incompatibility prevents us from producing true Common Lisp here. The `if` forms are different. In Emacs Lisp, `if` can take any number of subforms, there being a *condition* form, a *then* form, and after them any number of *else* subforms, which are executed in an implicit `progn`. Moreover, that style is widely used in the Emacs sources, so I thought most impractical to break with it to support Common Lisp's `if` (where only one *else* form is tolerated). For the most part, I use `cond` almost always, so it doesn't bother me much. If you use single-branch `if`s often, consider `when` or `unless` as alternatives.

`case` and `ecase` are a convenient way to write things that usually end up in a very baroque `cond`.

1.5.1 Features Provided

`MACRO case`
`MACRO ecase`
`MACRO unless`
`MACRO when`
The standard stuff, completely implemented.

1.6 Iterations

Having a `do` macro was my original motivation. The alternatives in standard Emacs Lisp are either expensive (recursion) or correspond directly to the expansion of my macros:

```
(macroexpand '
 (do ((i 0) (j 1 (+ 1 j)))
     ((> j (foo i)) (cons i bar))
   (setq i (baz i j))))

(let ((i 0) (j 1))
  (while (not (> j (foo i)))
    (setq i (baz i j))
    (psetq j (+ 1 j)))
  (cons i bar))
```

So I prefer to leave to the macros the problem of remembering the details right.

The incompatibilities are due to the problems already discussed (see Section 1.1 [Generalities], page 1, for more details).

If you write Emacs Lisp code often, you will find enough uses for these. Examples are cooking up a translation table to move `C-s` out of the way of multiplexers, switches, concentrators and similar fauna, or building keymaps.

1.6.1 Features Provided

`MACRO do`
`MACRO do*`
`MACRO dolist`
`MACRO dotimes`
: The standard, but for the lack of implicit blocks.

`MACRO loop`
: The basic standard one, not the fancy one. As per the book, warns you about atomic entries at the surface of the macro (to guarantee that the fancy `loop` macros can be made standard later).

`MACRO do-all-symbols`
`MACRO do-symbols`
: These operate on obarrays, the default is the current one.

1.7 Multiple Values

The multiple values mechanism covers (simply and elegantly, in my opinion) various common needs:

1. The case where a function returns a composite value, that has to be assembled in the callee and disassembled in the caller. An example is `pair-with-newsyms`.
2. The case where a function might cheaply compute redundant information that is useful to the caller only eventually. For instance, routines that compute quotients and remainders together, whose callers might be more often interested in just receiving the quotient.

3. The case of functions that usually return a useful value, but might need to elaborate on occasion (say, returning a reason code too).

The general idea is that one such function *always* returns the extra values, but only callers that are aware of this ability receive them. Unaware callers just receive the first value.

I think my implementation is pretty much complete. I am imposing no limits on the number of multiple values a function may return, so I am not providing the constant `multiple-values-limit`. You can assume multiple values are bound by the memory size only.

1.7.1 Features Provided

`values`
`values-list`
: These are the forms that produce multiple values.

`MACRO multiple-value-bind`
`MACRO multiple-value-call`
`MACRO multiple-value-list`
`MACRO multiple-value-prog1`
`MACRO multiple-value-setq`
: These are the forms that receive multiple values.

`VARIABLE *mvalues-count*`
`VARIABLE *mvalues-values*`
: Used by the implementation. Don't touch them!

1.8 Integer Arithmetic

I have provided most of the functions that are supposed to act on integers. Of those that take arbitrary numbers, I have implemented those that have a reasonable implementation if restricted to integers only, although some more could be added (like a restricted form of `expt`).

Being a little worried about the bad fame that affects some implementations of the '%' C operator, I have taken perhaps unnecessary precautions whenever integer division is concerned (see the function `safe-idiv`). This should be of interest only when dividing numbers that might be negative, but I have preferred here to be safe rather than fast.

1.8.1 Features Provided

`abs`
`signum` The usual.

`gcd`
`lcm` The usual.

`isqrt` A rather annoying function. Only use I can think of: to cut short a prime number sieve.

`evenp`
`oddp`
`plusp`
`minusp` A few predicates that use to come handy.

`ceiling`
`floor`
`round`
`truncate`
`mod`
`rem` The intention is to give everybody his preferred way to divide integers. I have tried not to depend on the unreliable semantics of C's integer division, I hope I got it right. Read the code when in doubt.

1.9 Generalized Variables

This implementation has many limitations. Take a look to see if you want to overcome them, the fixes might prove unnecessarily expensive for Emacs purposes. The ones I am clearly aware of:

1. Common Lisp suggests an underlying mechanism (the setf-methods) to implement generalized variables. I have used ad-hoc ideas that gave me a rather trivial implementation that suffers from some inflexibility. As a result, `defsetf` only admits the simplest form and there is no `define-modify-macro` nor there are functions to handle the (nonexistent) setf-methods.
2. I haven't implemented (I was uninterested) `getf` and friends. This shouldn't be hard.

In addition to providing this mechanism, I have written `defsetf`s for almost every accessor I thought of. There is room for improvement here, as Emacs Lisp provides many types of its own (buffers, windows, keymaps, syntax tables, . . .) for which pairs of accessors and mutators could be defined.

If you want to check whether a function has a setf-inversor, look at the property ':setf-update-fn' of its name. This is a characteristic of my implementation, not mandated by Common Lisp, so you shouldn't use it in code, but only to determine interactively what can be setf'd.

1.9.1 Features Provided

`MACRO setf`
`MACRO psetf`
Almost complete implementation. `Setf` should handle `apply` inside itself and not in a `defsetf`, but the difference is so minute I feel lazy about fixing this. `Psetf` is the version where the assignments occur in parallel.

`MACRO defsetf`
Very sketchy implementation. I will appreciate if somebody puts some time in implementing the whole works of setf-methods and such.

`MACRO incf`
`MACRO decf`
The usual and standard.

`MACRO pop`
`MACRO push`
`MACRO pushnew`
Should be the usual, but I haven't had the time to test them properly.

`MACRO rotatef`
`MACRO shiftf`

Very fancy. Good for implementing history rings and such. To swap two values, the following forms are equivalent:

```
(rotatef a b)
(psetf a b b a)
(psetq a b b a)  ;not good for anything but variables
```

1.10 Structures

I haven't had the time to construct a complete implementation of structures, but the part provided should stand on its own for many purposes. I am not supporting 'BOA constructors', nor typed slots (the `:type`, `:named` and `:initial-offset` options), nor explicit representational types. The rest should be pretty much complete. See the example file `fractions.el` for an idea of how complete the implementation is, and for exercises.

When writing these functions, I noticed I was incurring in lots of auxiliaries. I used dollar signs in their names, in the hope that this could prevent clashes with user functions. In retrospect, I should have done it in the other sections, too.

1.10.1 Features Provided

`MACRO defstruct`

Create records (a la C structs) and use them as types in your programs. Almost completely standard.

`make$structure$instance`

This non-standard function implements most of the 'guts' of the 'make-' constructors. It can be used as an example of the pseudo keyword-arguments.

1.11 Miscellanea

1.11.1 Features Provided

`MACRO psetq`

A parallel-assignments version of `setq`, makes the expansions of `do` and `do*` be very similar, as they should. Otherwise used to swap two values, now superseded by `rotatef`.

`duplicate-symbols-p`
`pair-with-newsyms`
`reassemble-argslists`
`unzip-list`
`zip-lists`

These are utilities I find useful when parsing a call or generating code inside a macro. Non standard.

1.12 To Do

No doubt many people will like to extend the functionality of these routines. When considering doing so, please try and do it in such a way that your implementation of a subset

of the functionality of Common Lisp is not inimical with a more extensive or more correct one. For definiteness, ask yourself the questions:

- Will my code run under a correct implementation of Common Lisp?
- Will a correct implementation of Common Lisp run if my code is loaded?

The first question tests the pertinence of your extensions. The second tries to discover "extensions" that prevent correct implementations of other features. Please tell me if you notice a case in which my code fails to pass any of those tests.

The next subsections propose some more extensions. I hope that they are attempted by people learning Lisp, as a way to enhance their understanding. Of course, experts are also admitted.

1.12.1 Keyword Arguments

Some effort has been done to handle keywords almost right. For instance, a structure constructor (see Section 1.10 [Structures], page 10) can be invoked with keyword arguments.

Look for the functions whose names have a '`klist`' in them. They were written to facilitate parsing calls with keyword arguments, but I haven't done a complete implementation yet. (Note that `member`, `assoc` and perhaps some other function, have to be implemented independently of the general framework. More details by Email if you want to try your hand at this.)

1.12.2 Mapping Functions

There is enough support to write `maplist`, `mapl`, etc. Emacs Lisp already provides some of the mapping functions, the trick now is to code the rest in a very efficient manner, so there will be an incentive to use `maplist` over an explicit iteration. I have a draft implementation, but I don't have the time to test it now.

1.12.3 Complete the current implementation

Some of the features described above are only a partial implementation of the Common Lisp features. Things that cry for a more complete form:

`defsetf` Only the simplest format is supported. The most general one is needed too. Also, try to get `define-setf-method` and `get-setf-method` to work.

`define-modify-macro`
Same as above. The modify-macros provided are all ad hoc.

`defstruct`
I think my version recognizes all the options and then proceeds to ignore most of them. Making sure that at least good error messages are produced would be nice. Also, what about BOA constructors?

There are other places where your programming ingenuity would help us all. For instance, `subst`, `sublis` and the like could be easily provided in the *lists* section. (I haven't done it because I wanted to have the keyword arguments stuff first.)

1.12.4 Hash Tables

A very simple implementation of hash tables would admit only strings as keys. For each string and a given number of buckets (a prime is desirable here), add the numeric values of all (or of a reasonable subset) of the characters and assign the bucket whose index is the remainder of the sum modulo the (prime) number of buckets.

A more convenient implementation can then be based on using `prin1-to-string` on an arbitrary Lisp object and using the output string as a key. This should make it easy to write `sxhash`. Remember that one needs to ensure that `(equal x y)` should imply that `(= (sxhash x) (sxhash y))`; and also that the keys are state-less (so you can write them to a file and read them back later).

Don't forget to provide a `defsetf` for `gethash`.

1.12.5 Packages

Packages should be easy to implement on top of a good hash table implementation, either by using it directly or by reusing some shared code. Don't worry too much about efficiency: package qualification has no run-time cost, only read- and print-time costs.

The difficult thing is to integrate it correctly. You have to replace the built-in functions `read` and `write`. This is not as bad as writing a programmable reader, but still a pain. For starters, your routines could remember the default definitions of the above mentioned functions:

```
(setf def-reader  (symbol-function 'read))
(setf def-printer (symbol-function 'print))
...
```

And then your specialized functions could just use `apply` to exercise the default ones, intercepting their activity in time to do the package qualification. You might have to do this to `prin1`, `prin1-to-string` and friends.

1.12.6 Streams and Files

This is the first "To Do" that might require doing some C programming. The purpose is to construct an efficient byte stream abstraction that will allow Streams and Files to be handled. Think of stdio, not Unix I/O, because Emacs already runs under other operating systems. Also, think of doing it in a way that can be generalized easily (for instance, streams kept in memory without a file behind, streams as an interface to a windowing system, etc.) Of course, the intended syntax is that of Common Lisp.

1.12.7 Reader and Printer

The Emacs Lisp reader (the C function `Fread`) is not reentrant nor programmable. It could be fixed as Lisp Code, but that is probably uglily expensive (as bad as redoing `eval` and `apply` to support lexical scoping). Doing this extension is probably a bad idea: a Common Lisp reader is incompatible with Emacs Lisp code (because of the '`?\`' constructions) and the most important rule to keep in mind is that this code is running under Emacs, so the host shouldn't be burdened too much with these emulations. Same goes for a more complete printer (a Common Lisp `format` would be incompatible with the Emacs Lisp one).

1.12.8 Backquote

Even if the reader is not made programmable nor reentrant, a backquoting mechanism could come handy. You need to study the way the current reader does `quote` and hack from there. This might be a more worthwhile extension than the complete rewrite of the reader.

1.12.9 Wild Ideas

Perhaps there is a way to implement `block`, `tagbody`, `return` and friends in spite of the dynamic scoping of Emacs Lisp. By this, I mean an adequate definition that preserves remotely the original intent and still provides a sensible set of constructs. Other dynamically scoped Lisps have these features, so implementing them is not necessarily impossible.

In the same spirit of these extensions would be to provide a port of something like Flavors (was there a PD version from Maryland, for Franz perhaps?) and then rephrase the language of major and minor modes in an Object Oriented paradigm.

Also, the rather gross `loop` macros that are out there in many Lisp systems could be helpful to some people (but then think of a `lisp-indent-hook` that handles them properly).