Emacs Lisp

Emacs Lisp is a dialect of the Lisp programming language used as a scripting language by Emacs (a text editor family most commonly associated with GNU Emacs and XEmacs). It is used for implementing most of the editing functionality built into Emacs, the remainder being written in C, as is the Lisp interpreter. Emacs Lisp is also termed **Elisp**, although there is also an older, unrelated Lisp dialect with that name.[1]

Users of Emacs commonly write Emacs Lisp code to customize and extend Emacs. Other options include the *Customize* feature that's been in GNU Emacs since version 20. Itself written in Emacs Lisp, Customize provides a set of preferences pages allowing the user to set options and preview their effect in the running Emacs session. When the user saves their changes, Customize simply writes the necessary Emacs Lisp code to the user's config file, which can be set to a special file that only Customize uses, to avoid the possibility of altering the user's own file.

Emacs Lisp can also function as a scripting language, much like the Unix Bourne shell or Perl, by calling Emacs in *batch mode*. In this way it may be called from the command line or via an executable file, and its editing functions, such as buffers and movement commands are available to the program just as in the normal mode. No user interface is presented when Emacs is started in batch mode; it simply executes the passed-in script and exits, displaying any output from the script.

Contents

Emacs Lisp

Emacs logo

Paradigm	Functional, meta, reflective
Family	Lisp
Designed by	Richard Stallman, Guy L. Steele, Jr.
Developer	GNU Project
First appeared	1985
Stable release	26.2 / 12 April 2019
Typing discipline	Dynamic, strong
Scope	Dynamic, optionally lexical
Platform	Emacs
OS	Cross-platform
License	GPLv3
Filename extensions	.el, .elc
Influenced by	
Common Lisp, Maclisp	

Compared to other Lisp dialects

Emacs Lisp is most closely related to Maclisp, with some later influence from Common Lisp.[2] It supports imperative and functional programming methods. Richard Stallman chose Lisp as the extension language for his rewrite of Emacs (the original used Text Editor and Corrector (TECO) as its extension language) because of its powerful features, including the ability to treat functions as data. Although the Common Lisp standard had yet to be formulated, Scheme existed at the time Stallman was rewriting Gosling Emacs into GNU Emacs. He chose not to use it because of its comparatively poor performance on workstations (as opposed to the minicomputers that were Emacs' traditional home), and he wanted to develop a dialect which he thought would be more easily optimized.[3]

The Lisp dialect used in Emacs differs substantially from the more modern Common Lisp and Scheme dialects used for applications programming. A prominent characteristic of Emacs Lisp is in its use of dynamic rather than lexical scope by default (see below). That is, a function may reference local variables in the scope it is called from, but not in the scope where it was defined.

Example

To understand the logic behind Emacs Lisp, it is important to remember that there is an emphasis on providing data structures and features specific to making a versatile text editor over implementing a general-purpose programming language. For example, Emacs Lisp cannot easily read a file a line at a time—the entire file must be read into an Emacs buffer. However, Emacs Lisp provides many features for navigating and modifying buffer text at a sentence, paragraph, or higher syntactic level as defined by modes.

Here follows a simple example of an Emacs extension written in Emacs Lisp. In Emacs, the editing area can be split into separate areas called *windows*, each displaying a different *buffer*. A buffer is a region of text loaded into Emacs' memory (possibly from a file) which can be saved into a text document.

Users can press the default `C-x 2` key binding to open a new window. This runs the Emacs Lisp function `split-window-below`. Normally, when the new window appears, it displays the same buffer as the previous one. Suppose we wish to make it display the next available buffer. In order to do this, the user writes the following Emacs Lisp code, in either an existing Emacs Lisp source file or an empty Emacs buffer:

```
(defun my-split-window-func ()
  (interactive)
  (split-window-below)
  (set-window-buffer (next-window) (other-buffer)))

(global-set-key "\C-x2" 'my-split-window-func )
```

The first statement, `(defun ...)`, defines a new function, `my-split-window-func`, which calls `split-window-below` (the old window-splitting function), then tells the new window to display another (new) buffer. The second statement, `(global-set-key ...)` re-binds the key sequence "C-x 2" to the new function.

This can also be written using the feature called *advice*, which allows the user to create wrappers around existing functions instead of defining their own. This has the advantage of not requiring keybindings to be changed and working wherever the original function is called, as well as being simpler to write but the disadvantage of making debugging more complicated. For this reason, *advice* is not allowed in the source code of GNU Emacs,[4] but if a user wishes, the advice feature can be used in their code to reimplement the above code as follows:

```
(defadvice split-window-below
  (after my-window-splitting-advice first () activate)
  (set-window-buffer (next-window) (other-buffer)))
```

This instructs `split-window-below` to execute the user-supplied code whenever it is called, before executing the rest of the function. Advice can also be specified to execute after the original function, around it —literally wrapping the original, or to conditionally execute the original function based on the results of the advice.

Emacs 24.4 replaces[5] this `defadvice` mechanism with `advice-add`, which is claimed to be more flexible and simpler.[6] The advice above could be reimplemented using the new system as:

```
(defun switch-to-next-window-in-split ()
  (set-window-buffer (next-window) (other-buffer)))

(advice-add 'split-window-below :before #'switch-to-next-window-in-split)
```

These changes take effect as soon as the code is evaluated. It is not necessary to recompile, restart Emacs, or even rehash a configuration file. If the code is saved into an Emacs init file, then Emacs will load the extension the next time it starts. Otherwise, the changes must be reevaluated manually when Emacs is restarted.

Source code

Emacs Lisp code is stored in filesystems as plain text files, by convention with the filename suffix "`.el`". The user's init file is an exception, often appearing as "`.emacs`" despite being evaluated as any Emacs Lisp code. Recent versions of Emacs ("recent" in a 40-year-old program meaning roughly any version released since the mid-1990s) will also load `~/.emacs.el` and `~/.emacs.d/init.el`. Additionally, users may specify any file to load as a config file on the command line, or explicitly state that no config file is to be loaded. When the files are loaded, an interpreter component of the Emacs program reads and parses the functions and variables, storing them in memory. They are then available to other editing functions, and to user commands. Functions and variables can be freely modified and redefined without restarting the editor or reloading the config file.

In order to save time and memory space, much of the functionality of Emacs loads only when required. Each set of optional features shipped with Emacs is implemented by a collection of Emacs code called a package or library. For example, there is a library for highlighting keywords in program source code, and a library for playing the game of Tetris. Each library is implemented using one or more Emacs Lisp source files. Libraries can define one or more *major modes* to activate and control their function.

Emacs developers write certain functions in C. These are *primitives*, also termed *built-in functions* or *subrs*. Although primitives can be called from Lisp code, they can only be modified by editing the C source files and recompiling. In GNU Emacs, primitives are not available as external libraries; they are part of the Emacs executable. In XEmacs, runtime loading of such primitives is possible, using the operating system's support for dynamic linking. Functions may be written as primitives because they need access to external data and libraries not otherwise available from Emacs Lisp, or because they are called often enough that the comparative speed of C versus Emacs Lisp makes a worthwhile difference.

However, because errors in C code can easily lead to segmentation violations or to more subtle bugs, which crash the editor, and because writing C code that interacts correctly with the Emacs Lisp garbage collector is error-prone, the number of functions implemented as primitives is kept to a necessary minimum.

Byte code

Byte-compiling can make Emacs Lisp code execute faster. Emacs contains a compiler which can translate Emacs Lisp source files into a special representation termed bytecode. Emacs Lisp bytecode files have the filename suffix `".elc"`. Compared to source files, bytecode files load faster, occupy less space on the disk, use less memory when loaded, and run faster.

Bytecode still runs more slowly than primitives, but functions loaded as bytecode can be easily modified and re-loaded. In addition, bytecode files are platform-independent. The standard Emacs Lisp code distributed with Emacs is loaded as bytecode, although the matching source files are usually provided for the user's reference as well. User-supplied extensions are typically not byte-compiled, as they are neither as large nor as computationally intensive.

Language features

Notably, the "cl" package implements a fairly large subset of Common Lisp.

Emacs Lisp (unlike some other Lisp implementations) does not do tail-call optimization.[7] Without this, tail recursions can eventually lead to stack overflow.

The apel library aids in writing portable Emacs Lisp code, with the help of the polysylabi platform bridge.

Emacs Lisp is a Lisp-2 meaning that it has a function namespace which is separate from the namespace it uses for other variables.[8]

From dynamic to lexical scoping

Like MacLisp, Emacs Lisp uses dynamic scope, offering static (or lexical) as an option starting from version 24.[9] It can be activated by setting the file local variable `lexical-binding`.[10][11]

In dynamic scoping, if a programmer declares a variable within the scope of a function, it is available to subroutines called from within that function. Originally, this was intended as an optimization; lexical scoping was still uncommon and of uncertain performance. "I asked RMS when he was implementing emacs lisp why it was dynamically scoped and his exact reply was that lexical scope was too inefficient."[12] Dynamic scoping was also meant to provide greater flexibility for user customizations. However, dynamic scoping has several disadvantages. Firstly, it can easily lead to bugs in large programs, due to unintended interactions between variables in different functions. Secondly, accessing variables under dynamic scoping is generally slower than under lexical scoping.[13]

Also, the `lexical-let` macro in the "cl" package does provide effective lexical scope to Emacs Lisp programmers, but while "cl" is widely used, `lexical-let` is rarely used.

References

1. "HEDRICK at RUTGERS (Mngr DEC-20's/Dir LCSR Comp Facility" (1981-12-18). " "information about Common Lisp implementation" " (https://www.cs.cmu.edu/Groups/AI/lang/lisp/doc/history/cl.txt). Letter to "rpg at SU-AI, jonl at MIT-AI". Archived (https://web.archive.org/web/20160920081348/https://www.cs.cmu.edu/Groups/AI/lang/lisp/doc/history/cl.txt) from the original on 2016-09-20. Retrieved 2019-07-28. "We have some experience in Lisp implementation now, since Elisp (the extended implementation of Rutgers/UCI Lisp) is essentially finished."

2. "GNU Emacs Lisp is largely inspired by Maclisp, and a little by Common Lisp. If you know Common Lisp, you will notice many similarities. However, many features of Common Lisp have been omitted or simplified in order to reduce the memory requirements of GNU Emacs. Sometimes the simplifications are so drastic that a Common Lisp user might be very confused. We will occasionally point out how GNU Emacs Lisp differs from Common Lisp." – from the "History" section of the "Introduction" to the Emacs Lisp Manual, as of Emacs 21
3. "So the development of that operating system, the GNU operating system, is what led me to write the GNU Emacs. In doing this, I aimed to make the absolute minimal possible Lisp implementation. The size of the programs was a tremendous concern. There were people in those days, in 1985, who had one-megabyte machines without virtual memory. They wanted to be able to use GNU Emacs. This meant I had to keep the program as small as possible." – from "My Lisp Experiences and the Development of GNU Emacs" (https://www.gnu.org/gnu/rms-lisp.html)
4. "Re: [Emacs-diffs] /srv/bzr/emacs/trunk r111086: gmm-utils.el (gmm-flet" (https://lists.gnu.org/archive/html/emacs-devel/2012-12/msg00146.html). Lists.gnu.org. 2012-12-05. Retrieved 2013-08-18.
5. "NEWS.24.4" (https://www.gnu.org/software/emacs/news/NEWS.24.4).
6. "Porting old advice" (https://www.gnu.org/software/emacs/manual/html_node/elisp/Porting-Old-Advice.html#Porting-Old-Advice).
7. "Appendix C Porting Common Lisp" (https://www.gnu.org/software/emacs/manual/html_node/cl/Porting-Common-Lisp.html). Gnu.org. Retrieved 2019-10-28. "Lisp programmers will want to note that the current Emacs Lisp compiler does not optimize tail recursion"
8. "Google Groups" (https://groups.google.com/forum/#!topic/learn-elisp-for-emacs/RKRDc1Qw3Bs). *groups.google.com*.
9. "Emacs 24.1 released" (http://lists.gnu.org/archive/html/emacs-devel/2012-06/msg00164.html). Lists.gnu.org. Retrieved 2013-08-18.
10. "Lexical binding" (http://lists.gnu.org/archive/html/emacs-devel/2011-04/msg00043.html). Lists.gnu.org. 2011-04-01. Retrieved 2013-08-18.
11. "Dynamic Binding Vs Lexical Binding" (http://www.emacswiki.org/emacs/DynamicBindingVsLexicalBinding#toc8). EmacsWiki. 2013-05-17. Retrieved 2013-08-18.
12. "T" (http://people.csail.mit.edu/gregs/ll1-discuss-archive-html/msg00650.html). People.csail.mit.edu. Retrieved 2013-08-18.
13. Featherston, Sam; Winkler, Susanne (2009-06-02). *Process* (https://books.google.com/books?id=w4R1xtZzTFYC&q=scoping+is+generally+slower+than+under+lexical+scoping.&pg=PA39). Walter de Gruyter. ISBN 978-3-11-021614-1.

External links

- Official website (http://www.gnu.org/software/emacs), GNU Project