Evolution of Emacs Lisp

STEFAN MONNIER, Université de Montréal, Canada
MICHAEL SPERBER, Active Group GmbH, Germany

While Emacs proponents largely agree that it is the world's greatest text editor, it is almost as much a Lisp machine disguised as an editor. Indeed, one of its chief appeals is that it is *programmable* via its own programming language, Elisp (or Emacs Lisp), a Lisp in the classic tradition. In this article, we present the history of this language over its more than 30 years of evolution. Its core has remained remarkably stable since its inception in 1985, in large part to preserve compatibility with the many third-party packages providing a multitude of extensions. Still, Elisp has evolved and continues to do so.

Despite the fact that it is closely tied to a concrete editor, Elisp has spawned multiple implementations—in Emacs itself but also in variants of Emacs, such as XEmacs, Edwin, and even the window manager Sawfish. Through competing implementations as well as changes in maintainership, it has picked up outside influences over the years, most notably from Common Lisp.

Important aspects of Elisp have been shaped by concrete *requirements* of the editor it supports, such as the *buffer-local variables* that tie bindings to editing contexts, as well as implementation constraints. These requirements led to the choice of a Lisp dialect as Emacs's language in the first place, specifically its simplicity and dynamic nature: Loading additional Emacs packages or changing the ones in place occurs frequently, and having to restart the editor in order to re-compile or re-link the code would be unacceptable. Fulfilling this requirement in a more static language would have been difficult at best.

One of Lisp's chief characteristics is its malleability through its uniform syntax and the use of macros. This has allowed the language to evolve much more rapidly and substantively than the evolution of its core would suggest, by letting Emacs packages provide new surface syntax. In particular, Elisp can be customized to look much like Common Lisp, and additional packages provide multiple-dispatch object systems, legible regular expressions, programmable pattern matching constructs, generalized variables, and more. Still, the core has also evolved, albeit slowly. Most notably, it acquired support for lexical scoping.

The timeline of Elisp development is closely tied to the projects and people who have shaped it over the years: We document Elisp history through its predecessors, Mocklisp and MacLisp, its early development up to the "Emacs schism" and the fork of Lucid Emacs, the development of XEmacs, and the subsequent rennaissance of Emacs development.

CCS Concepts: • **Social and professional topics**; • **Professional topics**; • **History of computing**; • **History of programming languages**;

Additional Key Words and Phrases: history of programming languages, Lisp, Emacs Lisp

ACM Reference Format:
Stefan Monnier and Michael Sperber. 2018. Evolution of Emacs Lisp. 1, 1 (September 2018), 36 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

Authors' addresses: Stefan Monnier, Université de Montréal, C.P. 6128, succ. centre-ville, Montréal, QC, H3C 3J7, Canada, monnier@iro.umontreal.ca; Michael Sperber, Active Group GmbH, Hechinger Str. 12/1, Tübingen, Germany, sperber@deinprogramm.de.

XXXX-XXXX/2018/9-ART
https://doi.org/10.1145/nnnnnnn.nnnnnnn

Contents

1 INTRODUCTION

Elisp is the extension language of the Emacs text editor. In this sense, it is just a side-project of Emacs and might be overlooked as a programming language. But Emacs itself comes with more than a million lines of Elisp code, and yet more Elisp is distributed separately from Emacs in various Emacs Lisp Package Archives (ELPA). If you additionally consider that the majority of Emacs users have likely written a few lines of Elisp in their configuration file, it is clearly one of the most widely used dialects of Lisp.

1.1 Organization

Elisp has evolved in multiple strands and implementations over the years, and thus its evolution did not happen along a single timeline. Moreover, some aspects evolved over long periods of time. To avoid excessive interleaving of those aspects, we have organized the top-level structure of the paper into chronological eras. Within an era, as a new topic is introduced, we usually follow that topic chronologically to its conclusion, even if that means going beyond the era where it started.

We trace the overall evolution of Elisp in the software projects that implemented it. Its predecessors EINE (1976) and Multics Emacs (1978) were themselves written in Lisp. Unix Emacs (also known as Gosling Emacs), which appeared in 1981, was written in C but included its own Lisp-like language. We briefly describe those predecessors in Section 2.

Emacs as we know it today itself started in early 1985. Section 3 describes the driving motivation for the early design and implementation of Elisp. The following Section 4 traces the evolution of the base language design, and Section 5 its implementation. Development continued at a high pace until about 1991. Around that time, its development slowed down and was overtaken by Lucid Emacs, later renamed XEmacs, whose development of Elisp

we describe in Section 6. Eventually, development of Emacs picked up again, and both co-evolved until about 2007. We describe the relevant aspects of that evolution in Section 7. After 2007, XEmacs development slowed down, and we describe this post-XEmacs period in Section 8.

Elisp was re-implemented in several other projects outside this successions. We briefly touch upon those projects in Section 9. Our conclusions are in Section 10.

2 PREHISTORY

While Emacs's original inception was as a set of macros for the TECO editor, it had no high-level extension language. Arguably, the strongest influences for Elisp were Gosling Emacs's Mock Lisp and MacLisp.

2.1 MacLisp

Several early incarnations of Emacs were written in Lisp, notably on the Lisp Machine (called *EINE* for "EINE Is Not EMACS") by Dan Weinreb and in MacLisp [17, 20] by Bernie Greenberg in 1978 [24]. The ensuing possibilities for extending the editor were attractive to Richard Stallman, who wanted to write a new widely available version. As a result, Elisp is a direct descendant of MacLisp.

2.2 Gosling Emacs

Unix Emacs, written by James Gosling in 1980/1981 [12], preserved in the history books as *Gosling Emacs*, was one of the immediate predecessors of Emacs. It featured an extension language called *MLisp* or *Mock Lisp*, which bears visual resemblance to Elisp. MLisp featured function definitions via `defun`, as well as many built-in functions (such as `eolp`, `forward-character`, `save-excursion`) whose names survive in Elisp. Emacs contained some backwards-compatibility support for MLisp until Emacs 21.2 in 2001.

MLisp was a quite limited language: It lacked cons cells and lists. MLisp did have dynamic binding and local variables, but a peculiar mechanism for passing arguments: There were no named parameters. Instead, a program would invoke the `arg` function: For example, `(arg 1)` would access the first argument. Moreover, argument expressions were essentially evaluated in a call-by-name fashion by `arg`, and evaluation happened in the dynamic environment of the callee.

3 EARLY HISTORY

Following Greenberg's Multics Emacs, Richard Stallman decided to write a (for him) second version of Emacs, which included Elisp from the start. As Greenberg's Emacs required a high-performance Lisp compiler to run efficiently, Richard Stallman decided to reimplement the basis for the new Emacs in C, with an integrated Lisp interpreter. Buffer manipulation and redisplay were written in efficient C, higher-level functionality in Lisp.

The initial design of Elisp was motivated by the pervasive requirement of *extensibility*; users "must be able to redefine each character" [25]. TECO and Gosling Emacs featured small languages that were either too obscure or too weak to support this vision. Consequently, Richard Stallman took inspiration from MacLisp, and Elisp started as a real programming language with powerful abstraction facilities, thus foregoing Greenspun's tenth rule.

Moreover, Richard Stallman, made the design of Elisp embody and showcase the ideals of Free Software. For example, not only is it legal to get and modify the source code, but every effort is made to encourage the end-user to do so. The resulting requirements had a profound influence on the Elisp language:

- The language should be accessible to a wide audience, so that as many people as possible can adapt Emacs to their own needs, without being dependent on the availability of someone with a technical expertise. This can be seen concretely in the inclusion in Emacs of the *Introduction to Programming in Emacs Lisp* tutorial [6] targeting users with no programming experience. This has been a strong motivation to keep Elisp on the minimalist side and to resist incorporation of many Common Lisp features.
- It should be easy for the end-user to find the relevant code in order to modify Emacs's behavior. This has driven the development of elements such as the *docstrings* (see Section 4.4) and more generally the self-documenting aspect of the language. It also imposes constraints on the evolution of the language: the use of some facilities, such as *advice*, is discouraged because it makes the code more opaque.
- Emacs should be easily portable to as many platforms as possible. This largely explains why Elisp is still using a fairly naive mark&sweep garbage collector, and why its main execution engine is a simple byte-code interpreter.

4 BASE LANGUAGE DESIGN

The base language of Elisp started out as a straightforward variant of MacLisp [17, 20].

Many basic special forms are identical: `defun`, `defvar`, `defmacro`, `let`, `let*`, `cond`, `if`, `function`, `catch`, `throw`, `unwind-protect`. So are basic data structures: symbols, `nil`, cons cells, arrays, and many familiar Lisp functions.

The language used dynamic scoping where the latest binding of a variable is simply stored in the *value slot* of the corresponding symbol object, resulting in simple and efficient variable lookups and let-bindings. Symbols could be used to refer to variables. In particular, the `set` function, which performed assignment, accepted the *name* of a variable as its argument. A symbol had an associated *property list*, essentially a dictionary mapping property names (also symbols) to values.

Like MacLisp, Elisp was (and still is) a Lisp-2 [26]. The namespaces for functions and "ordinary values" are separate, and to call a function bound to a variable, a program must use `funcall`. Also, symbols can be used as function values.

Some specialized control structures were missing in the original Elisp, among them the `do` loop construct, and the accompanying `return` and `go` forms for non-local control transfer.

This section discusses some notable differences from and additions to MacLisp. Lambda expressions worked slightly differently, as discussed in Section 4.1. Section 4.2 discusses the string representation (absent in MacLisp). MacLisp-style backquote reader syntax was only added later to Elisp—see Section 4.3. A feature supporting easy extensibility was self-documentation through *docstrings*, described in Section 4.4. Section 4.5 describes how Emacs ties functions to interactive commands. Section 4.6 describes Elisp's improvements over MacLisp's error handling. Section 4.7 describes buffer-local variables, which tie the language to the editor. One important pragmatic aspect of the use of the language was extensibility of existing code via hooks, discussed in Section 4.8.

4.1 Lambda

Interestingly (unlike MacLisp), `lambda` was not technically part of the Elisp language until around 1991 when it was added as a macro, early during the development of Emacs-19. In Emacs-18, anonymous functions were written as quoted values of the form:

```
'(lambda (..ARGS..) ..BODY..)
```

While the `lambda` macro has made this quote unnecessary for almost 30 years now, many instances of this practice still occur in Elisp code, even though it prevents byte-compilation of the body.

Somewhat relatedly, only in 1993 did Lucid Emacs 19.8 import the `#'...` reader shorthand for `(function ...)` from MacLisp. Emacs followed suit the year after.

4.2 Strings

Elisp included support for string objects from the beginning of course. Originally, they were just byte arrays.

In 1992, during the early development of Emacs-19, this basic type was extended by Joseph Arceneaux with support for *text-properties*: each char of a string can be annotated with a set of properties, which maps property names to values, where property names can be any symbol. This can be used to carry information such as color and font to use when displaying the various parts of the string.

XEmacs added a similar, though incompatible, feature to their strings at around the same time, called *extents*. The incompatibility was due to a fundamental disagreement about how those annotations should be propagated when strings are manipulated: XEmacs's extents cover a contiguous span of characters and they are objects in their own right with their own identity, so they need to be duplicated when a substring is selected, with the new extents not necessarily covering the same characters, and concatenation may end up bringing two "identical" extents next to each other but cannot merge them into one. Richard Stallman felt like this was introducing undesired complexities, and decided that Emacs's text properties should not have identity and should apply to the characters of the string so there is never a question of splitting or merging text properties.

With buffer-local variables 4.7, this is one of the very rare cases where the core Elisp language has been extended with a feature that specifically caters to the needs of the Emacs text editor, to keep track of rendering information. It is more generally useful, of course, but turns strings into a much fancier datatype than in most other languages.

Around 1994, support for arbitrary character sets was added to Emacs and XEmacs, which required distinguishing between bytes and characters, and hence changing its string objects. Characters are represented with a variable number of bytes, similar to `utf-8`, favoring a compact representation at the cost of a slower random access. In practice, random access to strings is fairly rare, so it works rather well, but it does impose a significant constraint: there is an indirection between the string object and the bytes it holds because the `aset` primitive which replaces one of the string's characters with another by side-effect will sometimes change the string's length in bytes, which requires to relocate the string's bytes elsewhere.

In XEmacs, strings have a "modified-tick" that is bumped every time a string is modified in-place. This count can be retrieved with the new function `string-modified-tick`. The XEmacs codebase never contained a user for this function, so the motivation for this is unclear. It is a straightforward parallel to the `buffer-modified-tick` function that tracks modifications to buffers, and which has many uses.

4.3 Backquote

Curiously, Elisp did not until 1994 include the reader syntax for quasiquotation common to many other Lisps, including MacLisp [1]. Instead, the special forms that came with quasiquotation—backquote (`` ` ``), unquote (`,`) and unquote-splicing (`,@`) were the names of special forms. Thus, what is usually written as:

```
`(a b ,c)
```

was written in early Elisp as:

```
(` (a b (, c)))
```

The releases of Emacs-19.28 and XEmacs 19.12 in 1994 finally added the proper reader support to generate the latter version from the first. The reader had to rely on a heuristic for that, though, because `(`a)` is valid in both syntaxes but means different things: It could either denote an old-style backquote expression (denoting the backquoted symbol `a`) or a single-element list containing the new-style backquoted symbol `a`.

So the old format, though obsolete, was still supported, and the new format was only recognized in some cases; more specifically the new unquote was only recognized within a new backquote, and the new backquote was only recognized if it occurred within a new backquote or if it did not immediately follow an open parenthesis.

Seeing how uses of old-style backquotes were not going away, in 2007 Emacs-22.2 introduced explicit tests and warnings to bring attention to uses of old-style backquotes, while still keeping the actual behavior unchanged.

Then in 2012 with the release of Emacs-24.1, the behavior was changed so that the old format is only recognized if it follows a parenthesis and is followed by a space. The main motivation for this change was the introduction of the `pcase` macro where patterns can also use the backquote syntax and are commonly placed right after a parenthesis so they were otherwise mistaken for an old-style backquote syntax.

Finally in 2018 during the development of Emacs-27 the old-style backquote syntax was removed.

4.4 Docstrings

An important feature of Emacs from the start was the idea of *self-documentation* [25]. To that end, every definition form in Elisp can include documentation in the form of a string after the signature:

```
(defun ignore (&rest _ignore)
  "Do nothing and return nil.
This function accepts any number of arguments, but ignores them."
  nil)
```

This *docstring* can be retrieved in various ways, in particular through the user interface. This idea was apparently first introduced in the original Emacs implementation in TECO, then adapted to Elisp. Many languages have since adopted them, notably Common Lisp [22] and Clojure.

4.5 Interactive functions

The most direct method to make a function implemented in Elisp accessible to a user is to provide a keybinding for it. This is not realistic for all functions, however—there are too many of them.

A user can also invoke an function by typing `M-x` *function-name*. This only makes sense for a certain subset of all defined functions, namely *interactive* functions, which are specially marked with an `interactive` form at the beginning of the body, like this:

```
(defun forward-symbol (arg)
  "Move point to the next position that is the end of a symbol.
A symbol is any sequence of characters that are in either the
```

```
word constituent or symbol constituent syntax class.
With prefix argument ARG, do it ARG times if positive, or move
backwards ARG times if negative."
  (interactive "^p")
  (if (natnump arg)
      (re-search-forward "\\(\\sw\\|\\s_\\)+" nil 'move arg)
    (while (< arg 0)
      (if (re-search-backward "\\(\\sw\\|\\s_\\)+" nil 'move)
          (skip-syntax-backward "w_"))
      (setq arg (1+ arg)))))
```

The `interactive` form also has an optional string operand that regulates how such a function receives its arguments. (It can also be an expression that is evaluated to produce a list of arguments.) In the above example, `p` means that the function accepts a *prefix argument*: The user can type `ESC` *number* or `C-u` (to specify powers of 4) prior to invoking the function, and the number will be passed to the function as an argument, in this case as `arg`. (The `^` (a more recent addition) has to do with region selection with a pressed shift key.)

4.6 Non-local exits

Very early on, Elisp featured a good set of primitives to handle non-local exits. Additionally to the `catch` and `throw` primitives inherited from MacLisp, it came with `unwind-protect`, which was inherited from TECO, as well as an error handling system.

MacLisp's error-handling system was quite primitive, and had poor separation of signalling and handling [21]. Elisp instead featured a *condition system* inspired from the Lisp Machine although it did not cover all the features of the Lisp Machine system, such as the ability to recover from an error.

The `signal` function takes an `ERROR-SYMBOL` classifying the exceptional situation and an additional `DATA` argument. An error symbol is a symbol with an `error-conditions` property that is a list of condition names. For example, the following invocation states that a `pixmap` parameter is bound to an invalid argument, and should have fulfilled the predicate `stipple-pixmap-p`:

```
(signal 'wrong-type-argument (list #'stipple-pixmap-p pixmap))
```

An invocation of `signal` escapes to the nearest use of `condition-case` in the call stack, which dispatches on the condition name. These condition names can be used to dispatch on the condition with the `condition-case` form which specifies a handler. Here is an example:

```
(condition-case err
    (key-binding (this-command-keys))
  (wrong-type-argument
    <handle>))
```

This evaluates `(key-binding ...)`. If a `wrong-type-argument` is signalled during evaluation, `<handle>` is evaluated, with `err` bound to a pair consisting of the error symbol and the data argument passed to `signal`.

Emacs comes with a set of standard errors, establishing a protocol between signaling and handling code.

4.7 Buffer-local variables

Buffer-local variables are the prominent feature marking Elisp as the extension language of a text editor. In Emacs, *buffers* are objects that store the contents of a file while it's being edited. In addition to the file's content, buffers store auxiliary information such as the name of the associated file, the rules to use to highlight its contents, the editing mode to use, the character encoding used etc. Additionally, there is a global reference managed by Emacs called the *current buffer*, that determines the implicit target of editing operations.

Making a variable *buffer-local* associates the value of the variable with the current buffer. An assignment to a buffer-local variable only affects dereferences with the same current buffer in effect. Any variable can be made local to any buffer, so variables can take different values in different buffers. For example, the variable `buffer-file-name` keeps the name of the file associated with the corresponding buffer. This variable typically has a different value in every buffer. In the absence of such a buffer-local assignment, a variable is said to have its *default* or *global* value.

Variables can be both buffer-local and dynamically bound at the same time:

```
(let ((buffer-file-name "/home/rms/.emacs"))
  (with-current-buffer "some-other-buffer"
    buffer-file-name))
```

This example will not return `"/home/rms/.emacs"` but the buffer-local value of `buffer-file-name` in the buffer `"some-other-buffer"` instead because `with-current-buffer` temporarily changes which buffer is current.

The presence of buffer-local values significantly complicates the implementation of looking up, setting, and let-binding a variable, so the code is optimized for the "normal" case of variables that have not been made local to any buffer.

Over the years many bugs were fixed in the implementation of let-binding buffer-local variables. The most famous one was fixed in Emacs-21.1. This bug affected code like

```
(let ((buffer-file-name "/home/rms/.emacs"))
  ...
  (set-buffer other-buffer)
  ...)
```

where the current buffer was different when the let-binding is entered from when it is left[1]; the bug was that this code ended up "restoring" the value of `buffer-file-name` into the wrong buffer when exiting the `let`.

4.8 Hooks

One important aspect of the extensibility Richard Stallman originally conceived for Emacs was the ability to make existing functions run additional code without having to change them, so as to extend their behavior. Emacs supports this at well-defined points called *hooks*.

A hook is simply a variable bound to a list of functions. The `add-hook` function adds a function to such a list by reassigning the variable. (This makes use of the ability to refer to a variable by its name symbol.)

Hooks are not a core language feature, but their use has been a pervasive convention in Emacs from the start. In particular, many libraries run a hook when they are loaded to allow customization. Also, modes generally run hooks to allow modifying their behavior.

[1] `set-buffer` is like `with-current-buffer` except that it is not scoped, so it takes effect until the next `set-buffer`.

The old TECO version of Emacs also allowed attaching hooks to variable changes [25], but this feature was not provided in Elisp because Richard Stallman considered it a misfeature, which could make it difficult to debug the code. Yet this very feature was finally added to Elisp in 2018 in the form of *variable watchers*, though they are ironically mostly meant to be used as debugging aides.

Of course, authors do not always have the foresight to place hooks where users need them, so in 1992, the `advice.el` package was added to Emacs-19, providing a `defadvice` macro duplicating a design available in MacLisp and Lisp Machines, that allowed attaching code to functions even if they do not run hooks.

The way the `defadvice` macro gave access to function arguments did not work with lexical scoping; furthermore time had shown that most of the non-core features of `defadvice` were very seldom used, and even more seldom used properly. So in late 2012 a new package `nadvice.el` was developed which provides the same core features but with a much simpler design that tries to make better use of existing language features. It was released as part of Emacs-24.4.

4.9 I/O

One of the ways Elisp distinguishes itself from most other programming languages is in its treatment of input/output: rather than follow the usual design based on some kind of file or stream object and primitives like `open`/`read`/`write`/`close`, Elisp only offers coarser access to files via two primitive functions `insert-file-contents` and `write-region` which transfer file contents between a file and a buffer. So all file manipulation takes place by reading the file into a buffer, performing the desired manipulation in this buffer, and then writing the result back into the file.

Since this approach does not extend naturally to interaction with external processes or remote hosts, these are handled in a completely different way: there are primitive functions that spawn a sub-process or open up a connection to a remote host and return a so-called *process* object. These objects behave a bit like streams, with `process-send-string` corresponding to the traditional `write`, but where the traditional `read` is replaced by execution of a callback whenever data is received from the sub-process or the remote host.

5 BASE LANGUAGE IMPLEMENTATION

At least initially, the Elisp language was defined by its sole implementation. Various design aspects had impact on the users of the language, and this section discusses some of them. Section 5.1 describes the byte-code interpreter. Section 5.2 discusses tail-call optimization. Section 5.2.1 touches on issues related to bootstrapping. Section 5.3 describes the essential data representation of the implementation. The following sections touch on other issues related to memory management: Section 5.4 describes the evolution of scanning GC roots. Section 5.5 describes changes in XEmacs to simplify heap management. Section 5.6 describes efforts in Emacs and XEmacs to reduce the tag bits. Improvements to the GC algorithm are described in Section 5.7. Section 5.8 describes the image-dumping mechanism in Emacs and XEmacs and how it changed over time. Sections 5.9 and 5.10 document debugging and profiling efforts, respectively. Finally, Section 5.11 describes recent efforts to add JIT compilation to Elisp.

5.1 Byte-code interpreter

Emacs has two execution engines for Elisp: The first is a very simple interpreter written in C operating directly on the S-expression representation of the code. Some time before the

release of Emacs-16.56 in July 1985, a `byte-code` function was added, which interpreted its string argument as a sequence of stack-based byte codes to execute, along with a compiler, written in Elisp, which translated Elisp code to that byte-code language.

While Elisp is basically a "run of the mill" programming language, just with some specific functions tailored to the particular use of a text editor, this byte-code language is much less standard since it includes many byte codes corresponding to Elisp primitives such as `forward-char`, `insert`, or `current-column`.

While a systematic study would likely reveal that the operations that should deserve their own byte-code in modern Elisp code are quite different, the byte-code language of Emacs has changed very little over the years and is still essentially the same as that of 1985. The main changes were those made to support lexical scoping; see Section 8.1.

5.2 Tail-call optimization

Elisp does not optimize away tail calls. With Scheme being familiar to many Elisp developers, this is a disappointment for many. In 1991, Jamie Zawinski added an `unbind_all` instruction to the Lucid Emacs byte-code engine (which appears in both Emacs and XEmacs to this day) that was intended to support tail-call optimization, but never implemented the optimization itself.

There have been two patches developed independently and submitted to Emacs maintainers in late 2012 (by Troels Nielsen first and Chris Gay two months later) to optimize away tail calls in lexically-scoped byte-compiled code, but so far none of them have made it into an official release.

The main reason for that is partly a chicken-and-eggs problem. Tail-call optimization (TCO) is largely incompatible with dynamic scoping, so it was basically inapplicable until the introduction of lexical scoping in 2012; furthermore, function calls are relatively expensive in the current implementation of Elisp. Together, these two factors created a coding style which favored the use of iteration over recursive definitions, which in turn makes TCO rarely beneficial on existing code.

This said, there were other objections to those patches:

- They only affected byte-compiled code, and while it is expected that most code is byte-compiled before being executed, it's very common to run non-compiled Elisp code, especially during development or debugging. So if Elisp code started to rely on TCO, it would tend to cause problems when interpreted. One alternative is to always byte-compile the code and get rid of the Elisp interpreter, but that's a change that would have much further consequences.
- The available implementations of TCO affect the behavior not only in terms of reducing the stack allocation and increasing performance, but it would also eliminate some activation frames from stack backtraces, which would be detrimental to debugging. This is particularly true for tail-calls which are made to other functions, such as non-recursive tail-calls.

5.2.1 Bootstrap. Since the Elisp compiler is itself written in Elisp, it requires a form of bootstrap. Until 2002 during the development of Emacs-21, Emacs's development was done using the revision control system RCS [30], and changes were installed by logging remotely into an FSF-owned machine and performing the commits there. This machine also kept the (last) compiled form of the Elisp files and so naturally provided the needed compiled files for bootstrap. When development moved to CVS [2] to ease collaboration between contributors, another solution was needed.

Of course, since Emacs also comes with a simple direct Elisp interpreter, bootstrapping is not very difficult, but several changes were needed nevertheless because the previous use of pre-compiled files had hidden some circular dependencies. Of those, the only one that was not removed in Emacs-21.1 is that some Elisp code relies on the fact that some functions are *autoloaded*, and some of that code is used to build the file that contains all those `autoload` declarations. So instead, the solution found was to keep a pre-built copy of that file in the CVS repository.

5.3 Data representation and memory management

Emacs started with a data representation of its boxed data based on 32bit words. Those used a 7bit tag located in the most significant bits, 1 extra "mark" bit (see below), and 24bit of immediate data: either an integer or a pointer. Not all the possible 128 tags were actually used, but the 24 remaining bits where amply sufficient to represent the needed pointers and integers for the typical available memory of the machines of that time.

The memory management used a simple mark&sweep garbage collection algorithm, and allocated objects by blocs of 4KB dedicated to a particular kind of objects: one bloc each for cons cells, floats, symbols, markers, and strings, all other objects were allocated directly with `malloc`. To avoid fragmentation in the blocs of strings, those were compacted during each GC. The "mark" bit in each 32bit box was not used for the object contained in the 32bit box. Instead it was used so that cons cells could occupy only 2 words: the extra bit needed to store the mark&sweep's *markbit* of each cons cell was stored in the "mark" bit of the first word (i.e. of the `car`).

Over time, this tagging scheme became problematic, since it limited to 16 MB the size of the Lisp heap and to 8MB the size of files that could be edited. The limit on file size is fundamentally linked to the maximum representable integer since that is how buffer positions are represented.

So in 1995, with the release of Emacs-19.29, the scheme was tweaked so that the tag was reduced to 3 bits, pushing the maximum file size to a more comfortable 128MB and the maximum heap size to 256MB. To reduce the tag to 3 bits, the less important object types were placed into two groups: one group using the `Lisp_Misc` tag and another using the `Lisp_Vector` tag. The `Lisp_Misc` tag was used for objects that could share the same heap size as Lisp markers (6 words), and hence be allocated from the same 4KB bloc. For some of those objects, it imposed a slight waste of space, which was justified by the fact that objects using the `Lisp_Vector` tag had other extra costs: 2 words of header, plus the overhead of having each object by allocated directly by `malloc`.

XEmacs implemented a similar change in 1993 with Lucid Emacs 19.8, giving Elisp 28-bit integers and a 28-bit address space, and merged that data representation of Emacs 19.30 with the release of XEmacs 19.13, also in 1995.

5.4 Scanning the stack

Until Emacs-21, the mark phase of the GC was precise: the global roots were explicitly registered, the GC knew all the types of Lisp objects and where were the fields that could contain references, and the roots from the stack were also explicitly registered into a singly linked list itself directly allocated on the stack.

The cost of properly registering/unregistering stack references was perceived to be high: it slowed down execution, both directly by adding administrative code and indirectly by preventing some variables from being kept in registers, and it was a source of bugs, especially since some code tried to be clever and avoid registering local references under the assumption

that the GC could not be triggered at that particular point. Similarly, the relocation of strings was a frequent source of hard-to-track bugs because only the references known to the GC were properly updated, so the programmer had to be careful not to keep unboxed or unregistered references to a string at any point where a GC was possible.

To try and address those concerns, for Emacs-21.1, Gerd Möllmann changed the string compaction code and implemented a conservative stack scan. Strings are split into the string object itself, of fixed size and non-relocatable, and the relocatable string data to which the code (almost) never keeps a direct reference. In order to find out if a given word found on the stack might be a potentially valid reference to a Lisp object, it keeps a memory map that records which regions of the memory contains which kinds of Lisp objects. This conservative stack scanning could be used either in addition to the singly linked list of registered references, as a kind of debugging aide, or replace it altogether.

Thanks to the interactive nature of Emacs and its opportunistic GC strategy which ensures that the GC is often run when the stack is almost empty, the slower conservative stack scanning and the potential false positives it introduces have not been a problem. The maintenance of the memory map, implemented as a red-black tree, was hence the main cost of this new stack scanning, which proved competitive with the previous scheme. The previous scheme was kept in use on some rare systems until Emacs-25.1, where all the register/unregister code of stack references could finally be removed.

5.5 Heap management in XEmacs

XEmacs has kept precise GC, and implemented a number of improvements in its memory management. In particular, Markus Kaltenbach and Marcus Crestani implemented a scanning algorithm that used the memory-layout descriptors that had been added to support the portable dumper. (See Section 5.8.) Marcus Crestani also replaced the allocator: The new, much simplified allocator eliminated the distinction between objects that were allocated in blocks and those allocated via `malloc`, and would make the decision individually based on size.

In the process of these GC improvements, Crestani and Michael Sperber also added *ephemerons* to Elisp [13].

5.6 Squeezing the tag bits

Of course, a limit of 256MB was not actually comfortable.

XEmacs added a "minimal-tagbits" configure option with the release of XEmacs 21.0 in 1998, yielding 31-bit integers and a 1GB maximum file size. The mark bits moved to the object headers, which were also added to cons cells, making them 3-word objects. The mark bit was removed from the boxes, and the number of type tags was reduced to four: records (heap-allocated objects, characters, even and odd fixnums.) This became default with XEmacs 21.2 in 2002.

During the development of Emacs-22 in 2007, its tagging scheme was similarly reworked: The mark bit was removed, and the tag bits were moved to the least significant bits, allowing the Lisp heap to grow as large as the full address space allowed. Moreover, the *markbit* of cons cells was moved to a separate bitmap stored alongside each bloc of cons cells, which required allocating those blocs on 4KB alignment boundaries. This avoided the use of one extra word per cons cell just to store the markbit, as was done previously for floats. This same bitmap scheme was of course also used for floats, thus reducing the typical heap size of floats from 96 bits to 64 bits. Reducing the heap size of floats to 64bits and avoiding the use of 3 words for cons cells was not just motivated by thriftiness but rather by the need to

enforce that all objects be aligned on a multiple of 8, so as to free the least significant 3 bits for use as tag bits.

At that point, there was no illusion that a maximum file size of 256MB was sufficient, but at the same time, the design of Emacs made it basically impossible to view files larger than 4GB or edit files larger than 2GB (on a 32bit system) anyway, no matter which tagging scheme we used, so there was not a lot of room for improvement.

In Emacs-23.2 the tagging scheme was tweaked to use 2 tags for integers, hence pushing the maximum file size to 512MB.

To cover the remaining space, a new compilation option `--with-wide-int` was introduced in Emacs-24.1 to make the boxed data use 64bit on 32bit systems. This imposes a significant extra cost in terms of space and time but makes it possible to edit files up to about 2GB. When this compilation option is used, tag bits are placed in the most significant bits again, so that the 32bit of pointers can be extracted at no cost at all.

In Emacs-24.3, the allocation of objects using the `Lisp_Vector` tag (which is used for many more object types than just vectors) was modified: instead of calling `malloc` for each such objects, they are now allocated from "vector blocs". The motivation was not that `malloc` was too slow, but that the implementation of our conservative stack scanning keeps track of every part of the Lisp heap allocated with `malloc` in a balanced tree, so every such `malloc` costs us an $O(N \log N)$ operation plus a heap allocation of an extra tree node, which was very costly for small objects both in time and space. The reason why it took until Emacs-24.3 to fix this performance issue is that objects using the `Lisp_Vector` tag were historically not used in large numbers in early Elisp code. There were two factors that changed this situation: first, over time the style of Elisp coding evolved and the use of `cl.el`'s `defstruct` (which internally represents those objects as vectors) became much more common, and second the closures used in the lexical-scoping feature of Emacs-24.1 also use the `Lisp_Vector` tag.

In Emacs-24.4, the representation of objects using the `Lisp_Vector` tag (which is used for many more object types than just vectors) was improved so as to reduce their header from 2 words down to a single word.

In Emacs-27.1, the object representation was changed again: the distinction between `Lisp_Misc` and `Lisp_Vector` was dropped by making all objects use the `Lisp_Vector` representation since it had been improved sufficiently to be competitive with the special-cased `Lisp_Misc` representation.

5.7 New GC algorithms

During the learly years of Emacs, the main complaints from users about the simple mark&sweep algorithm were the GC pauses. These were solved very simply in Emacs-19.31 by removing the messages that indicated when GC was in progress. Since then complaints about the performance of the GC have been rare. Most of them have to do with the amount of time wasted in the GC during initialization phases, where a lot of data is allocated without generating much garbage.

Over time, there have been several attempts to replace the GC.

Some time between 1996 and 1999, while waiting for Emacs-21 development to start to be able to add his new redisplay engine, Gerd Möllman worked on a generational incremental mostly-copying GC based on a read barrier implemented using the operating system's VM primitives, but it was never completed nor even actually integrated into Emacs's code. Gerd gave up on this work when he realized that it was infringing on a patent and hence wouldn't be distributable with Emacs anyway.

During 2003, Dave Love worked on replacing Emacs's GC with Boehm's conservative GC. The effort went far enough to get a usable Emacs but it was never completed, mostly because the early performance results were disappointing. It also showed that such a replacement is non-trivial. The first issue is that various parts of Emacs's code assume that a collection can only occur during Elisp code execution and not during heap allocation, and of course those assumptions are not explicitly recorded in the code. The second issue is that Emacs currently implements various forms of ad-hoc weak references which need to be adapted to the more standard forms of weak references.

In XEmacs, GC pauses continued to be a perceived problem. In 2005, Marcus Crestani developed an incremental collector for XEmacs, again using a VM-based write barrier. It was released with XEmacs 21.5.21 in 2005 [7] and soon was turned on by default. It eliminates GC pauses from the user experience, and its asymptotic performance is competitive with the old collector.

5.8 Image dumping

An important feature of Elisp has been the `dump-emacs` function, which can be used to store a heap image of the running Emacs into a file, which can later be restored. This is crucial for Emacs's usability, as it allows the editor to start up quickly, without having to load and run all initialization code every time.

The implementation of `dump-emacs` started out as a set of highly platform-specific C files that implemented a function called `unexec`. The `unexec` function turns the running process back into an executable file. However, the implementations of `unexec` have been hard to write and required frequent maintenance. For example, in 2016 the Glibc maintainers decided to make internal changes to their `malloc` implementation which broke this functionality.

For XEmacs, in 1999, Olivier Galibert (based on initial work by Kyle Jones) started writing a *portable dumper* that would serialize just the heap of the running XEmacs into a file, which could later be `mmap`ed. Galibert added explicit memory-layout descriptors for all Lisp types to the system, which would later also benefit the new incremental garbage collector. (See Section 5.7.) This required pervasive changes to the C code, and so it took until 2001 for the first version of the portable dumper to be released.

The Glibc announcement re-ignited interest in implementing a more portable way to dump and restore Emacs's heap. Some experiments were first made to dump the heap in the form of a "normal" byte-compiled Elisp file. While this solution proved easy to implement, the time to load such a dumped heap remained too high to be acceptable. So the experiment only resulted in the implementation of some improvements to the performance of the code to read byte-compiled code. In parallel, Daniel Colascione worked on a different approach more like that of XEmacs's portable dumper. The code was basically finished, but Emacs maintainers so far have not accepted it into the official release. The main reason for that reluctance is that it trades off the brittle platform-dependence of the old unexec code for more code which partly duplicates some of the GC's code.

5.9 Debugging

Debugging support was added very early to Elisp: Emacs-16.56 already included the *backtrace debugger*, which suspends execution at the time a *condition* is signaled, showing the current stack backtrace and letting you examine the state of the application and place breakpoints before pursuing execution.

For most developers, this is still the main debugger for Elisp. Of course, it has seen various improvements over time, most of them affecting only the user interface. The main exceptions

are: in 1995 (for Emacs-19.31), it was refined so that the debugger is only invoked for some conditions, making it possible for developers to keep this debugger enabled all the time, without impeding normal use, and in 2012 (for Emacs-24.1) it was improved so as to be able to execute code in the context of any activation frame, which was necessary to allow access to lexically scoped variables.

In 1998, Daniel Laliberte developed another Elisp debugger, called Edebug. It was included into Emacs a few years later during the early development of Emacs-19. This one works without any special support in the interpreter; instead it instruments the Elisp source code you want to debug, such that running this code lets you step through this code and displays the various values returned by the evaluation of each step. We do not know from where Daniel took this idea, but Edebug was probably the inspiration for the Portable Scheme Debugger [14], and the same basic technique was used (and significantly improved) in SML/NJ [31].

One interesting feature of Edebug is that in the presence of arbitrary user-defined macros, it is generally impossible to correctly instrument source code since Edebug cannot guess which arguments to a macro are normal Elisp expressions and which ones play a different role. By default Edebug works around this difficulty by leaving arguments to unknown macros non-instrumented, which is safe but suboptimal. To improve on this default behavior, the macro author can annotate its macro with a *debug specification* which describes the role of each argument using a kind of grammar formalism, so Edebug can know which parts should be instrumented.

5.10 Profiling

In 1992, during early development of Emacs-19, Boaz Ben-Zvi implemented the `profile.el` package which implemented a fairly simple Elisp profiler all in Elisp. This implementation was based on instrumenting a set of user-specified Elisp functions by modifying their bodies in-place to keep track of time spent in those functions.

In 1994, Barry A. Warsaw implemented the `elp.el` package which took a similar approach but without modifying functions's bodies, which was brittle and inconvenient and only worked for functions defined in Elisp. Instead it replaced the instrumented functions with wrappers which counted the number of calls, along with the execution time, and internally called the original function's definition. This package was included in Emacs-19.29 and made `profile.el` obsolete. Its implementation was significantly reworked for Emacs-24.4 to make use of the new `nadvice.el` package in order to add/remove instrumentation instead of doing it in its own ad-hoc way.

Ben Wing added an Elisp profiler to XEmacs 19.14 in 1996 by instrumenting entry points in the byte-code interpreter. In profiling mode, the byte-code interpreter provides timing information about function calls and allocation.

In early 2011, Tomohiro Matsuyama started implementing in Emacs's C code a sampling-based profiler for Elisp. He finished the implementation as part of Google's Summer of Code of 2012, and it was included in Emacs-24.3. The main advantage of this profiler compared to `elp.el` is that it does not require instrumentation, and it collects (partial) stack traces. This means that not only the user does not need to know beforehand which functions might be involved but it can show a an actual call tree.

5.11 JIT compilation

The existing implementations of Elisp are relatively inefficient, all being based on fairly naive interpretation techniques.

5.11.1 First attempt. In 2004, Matthew Mundell developed the first JIT compiler for Elisp. This took byte-compiled Elisp code and used the GNU Lightning library to turn it into native machine code on the fly. The speedup obtained reached a factor of about 2 in the best case, which was rather disappointing, so it was not included into Emacs since the extra maintenance burden was not considered justified. An important reason for the disappointing performance is that it only removed the immediate interpretation overhead, but did not affect function calls nor was it able to remove redundant type checks.

5.11.2 Second attempt. Around 2012, Burton Samograd developed a second JIT compiler for Elisp. This took a similar path, but using GNU Libjit instead. It was very simplistic, turning each byte-code into a call to a C function. The resulting performance was not more impressive, the author measuring a 25% speedup on a `raytracer.el` test.

5.11.3 Third attempt. In 2016, Nickolas Lloyd developed the third JIT compiler for Elisp, again based on GNU Libjit and based on a similar approach. It improved on Burton's implementation by open-coding most common byte-codes instead, which avoided many C function calls, but it obtained comparable results most likely because Libjit is not very good at optimizing its code and C function calls aren't that costly. but did get to the point of being stable enough to be used globally.

5.11.4 Fourth attempt. In 2018, Tom Tromey took another stab at it, again using GNU Libjit. Compared to Nickolas, it focuses exclusively on code using lexical binding, which is likely to benefit more, and it implements additional optimizations by getting rid at compile-time of all the manipulation of the Lisp stack. In the best case it reaches a speed up factor of 3.5, which is not ideal, but is a bit more respectable. It can be used globally but is still a very naive JIT compiler which requires further work to try and avoid pathological behavior in some situations where the JIT compilation is performed too eagerly, leading to significant slowdowns.

The discussion whether this last JIT compiler will be integrated as an experimental feature into Emacs-27.1 is still on-going.

6 XEMACS PERIOD

In 1991, Lucid Inc., a software development company based in Menlo Park, Carlifornia, started a project called *Energize*. Energize was to be a C/C++ integrated development environment based on Emacs [10]. Lucid decided to use Emacs as the central component of Energize. At the time, the current version of Emacs was 18, which was still essentially a textual application. The then upcoming version of Emacs, Emacs 19, was to have a graphical user interface and many other features that the developers at Lucid considered essential for the development of Energize. However, at the time that Lucid needed Emacs 19, a release was not in sight.[2]

While Lucid at first tried to support and thus speed up the development of Emacs 19, the required cooperation between Lucid and the Free Software Foundation soon broke down. As a result, Lucid forked Emacs development, creating its own Emacs variant *Lucid Emacs*.[3] Jamie Zawinski was the primary developer of Lucid Emacs. In 1994, Lucid went bankrupt. Sun subsequently wanted to ship Lucid Emacs with their operating system, and ended up financing some of the continued development of Lucid Emacs, and effected a name change to the current *XEmacs*.

[2]The first official release of Emacs 19, Emacs 19.28, came out in on November 1, 1994.
[3]The first release of Lucid Emacs came out in April, 1992.

The focus of Lucid Emacs was on providing a proper graphical user interface. As a result, most of the changes to Elisp in Lucid Emacs / XEmacs were to support the move from a TTY-based purely textual model to a graphical model.

6.1 Event and keymap representations

One significant departure from Emacs in Lucid Emacs was the representation of keymaps: Emacs, to this day, uses a transparent S-expression representation for keymaps. In early Emacs in a keymap was simply a two-element list whose car is the symbol `keymap` and whose second element is either a vector indexed by character code or an *association list*. The current representation for keymaps in Emacs is richer but follows largely the same design—here is an example [15]:

```
(keymap
 (3 keymap
    ;; C-c C-z
    (26 . run-lisp))
 (27 keymap
     ;; ‘C-M-x’, treated as ‘<ESC> C-x’
     (24 . lisp-send-defun))
 ;; This part is inherited from ‘lisp-mode-shared-map’.
 keymap
 ;; <DEL>
 (127 . backward-delete-char-untabify)
 (27 keymap
     ;; ‘C-M-q’, treated as ‘<ESC> C-q’
     (17 . indent-sexp)))
```

This creates problems with software evolution: While Emacs offered constructors and mutators for keymaps, Emacs code could, in principle, just use the tools for manipulating S-expressions for creating them—`cons`, `rplaca`, `rplacd` etc. Emacs-19 tried to keep such code working to some extent by merely extending the previous list representation. Eventually however such code would break in the face of representation changes, but would not immediately trigger an error. Furthermore, the support for inheritance was fundamentally flawed until Emacs-24.1 where it was extended to multiple-inheritance, but at the cost of a fairly significant and delicate rewrite of the code.

The developers Lucid Emacs of Lucid Emacs foresaw these problems, and instead made keymaps into an opaque datatype, forbidding manipulation via the S-expression primitives and giving themselves a lot more implementation freedom. This allowed Lucid Emacs to evolve more rapidly the representations of keymaps to cater to a richer set of input events (including mouse events, for instance).

This step reflected a general difference in philosophy. Lucid Emacs also used opaque data types for case tables and input events, both of which retain transparent representations in Emacs to this day.

6.2 Character representation

Another instance of a change in representation happened with the release of XEmacs 20, the first release of XEmacs with support for MULE (Multi-Lingual Emacs).

Previous versions of Emacs and XEmacs were inherently tied to an 8-bit representation of characters. Moreover, they had used strings not only for representing text but also for

representing key sequences. In strings, the high bit represented "meta," basically restricting Emacs to ASCII. Characters outside of strings could have more modifiers in higher bits.

This situation was no longer tenable when multi-language support came to XEmacs. The work on MULE [19] predates widespread adoption of Unicode, and at the time XEmacs adopted MULE (around 1994), a number of other text encodings were still in use. The MULE character representation encoded a character as an integer that represented two numbers, in the high and low bits respectively: One represented the national character set, the other the associated codepoint.

To enforce the separation between characters and their associated encodings, XEmacs 20 made characters a separate data type. XEmacs had functions to convert between a character and its numerical representation (`make-char` and `char-int`). Generally, Elisp allows programs to mostly handle text as strings, and avoid manipulating the numerical representation. Making characters an opaque type additionally discouraged the practice.

6.3 C FFI

As the Elisp runtime was written in C, it was always possible to add new Elisp functions written in C to the system. Those C functions could also call Elisp functions.

However, functions written in C originally lacked the dynamic nature of Elisp, as they had to be linked into the Emacs executable. Starting in 1998, J. Kean Johnston added facilities to XEmacs (released with version 21.2 in 1999), which allowed Elisp code to build and dynamically load shared libraries (called *modules*) written in C into running editor and call the functions defined therein. Starting in 2002 with XEmacs 21.5.5, a number of such modules were distributed with XEmacs, among them bindings for existing C libraries such as Zlib, Ldap, PostgreSQL.

Even with modules in place, developers still had to create wrappers to make existing C libraries accessible in Elisp. In 2005, Zajcev Evgeny wrote an FFI for SXEmacs [28], a fork of XEmacs. This FFI allows loading and calling existing C libraries directly, without intervening wrappers, by declaring the type signatures of C functions in Elisp. For example, the FFI allows using Curl like this:

```
(ffi-load "libcurl.so")
(setq curl:curl_escape
      (ffi-defun '(function c-string c-string int) "curl_escape"))
(let* ((url "http://foo.org/please escape this<$!=3>")
       (str (ffi-create-fo 'c-string url))
       (len (ffi-create-fo 'int (length url)))
       (result (ffi-call-function curl:curl_escape str len))
  (ffi-get result))
```

Richard Stallman refused to incorporate XEmacs's FFI into Emacs for fear that it would open up a backdoor with which developers would be able to legally circumvent the GNU General Public License (GPL) and thus link Emacs's own code with code that does not abide by these licensing terms. After many years of pressure on this issue (not just within the Emacs project, since this affected several other GNU projects, most notably GCC), a solution was agreed to, which was to implement an FFI that would only accept to load libraries that came with a special symbol attesting that this library is compatible with the GPL. As a result, after a very long wait, 2016 finally saw the release of Emacs-25.1 with an FFI comparable in functionality to that of XEmacs. So far, we do not know of any publicly available package which makes use of this new functionality, sadly. But rumors indicate that

it has been used in a few private projects, either to link Emacs with another language or to extend Emacs with ad-hoc functionality implemented in C for performance reasons.

6.4 Aliases

During the development of XEmacs 19.12, which was 1995, the first official release of Emacs 19 appeared, Emacs-19.28. Emacs had implemented some XEmacs functionality, notably the support for multiple open GUI windows. XEmacs had called these windows "screens," while Emacs called them "frames." Compatibility with Emacs was an important goal for the XEmacs developers at the time. Consequently, they renamed the associated functionality.

To preserve compatibility for Elisp code written for previous versions of XEmacs, XEmacs introduced forms `define-obsolete-functional-alias` and `define-obsolete-variable-alias`. The byte-code compiler would emit warnings if these aliases were used, but still compile the code.

Emacs had long had a `defalias` form to declare function aliases, on which the `define-obsolete-function-alias` functionality could be based.[4] XEmacs 19.12 added a corresponding primitive form for variable aliases, `defvaralias`, and functions `variable-alias` and `indirect-variable` to examine the alias chains.

These additions were only merged into Emacs-22.1 in 2007.

7 EMACS/XEMACS CO-EVOLUTION

Some aspects of Elisp evolved in both Emacs and XEmacs, with both versions borrowing design and code from the other.

7.1 Performance improvements

Both Emacs and XEmacs made various performance improvements, most of which were merged back and forth between the two versions.

Jamie Zawinski and Hallvard Furuseth wrote a new optimizing byte-compiler for Lucid Emacs, which, after some initial resistance, was merged into the Emacs codebase by 1992.

Also around 1992, during the early development of Emacs-19, the implementation of the byte-code interpreter was rewritten, and the result ended up in both in Emacs and XEmacs. As part of this rewrite, new object type for byte-compiled Elisp functions was introduced; before that, a byte-compiled function looked like:

```
(lambda (..ARGS..) (byte-code "..." ...))
```

Then in Emacs-19.29 in an attempt to speed up loading of Elisp packages as well as reduce the memory use of Emacs processes, a mechanism was added so that documentation strings as well as byte-code could be lazily fetched from compiled Elisp files. This can introduce problems if the file is modified while Emacs is running, so while this feature is always used for documentation strings it's very rarely used for byte-code.

XEmacs added a just-in-time optimization pass to the byte code. This would perform some validity checks ahead (eliding them from the actual execution), pre-compute stack use, make byte-code jumps relative (saving a register), and optimize relative jumps with short offsets. This effectively created an alternative byte-code dialect, which XEmacs would convert back to the "standard" representation on demand. During the XEmacs 19 and much of the XEmacs 20 cycle, developers avoided changing the byte-code format to make byte-code files interchangeable between Emacs and XEmacs. While no significant changes

[4] Curiously, `defalias` was elided from the Emacs code base in 1986 and reintroduced in 1993.

were made to the byte-code format in XEmacs, the two instruction sets eventually drifted and became incompatible.

In late 2009, the Emacs byte-code interpreter was modified by Tom Tromey to implement token threading using GCC's *computed goto* feature, when available. A patch for this feature had been submitted in May 2004 by Jaeyoun Chung, but the speed improvement was not even measured at that time so it had not raised much enthusiasm. Tom's implementation was no better, and the speed up was a meager 5% but he pushed stronger for its inclusion, which only happened with Emacs-24.3, in 2012. The reason why the speed improvement is a bit disappointing was not really investigated, but the general consensus is that the byte-code interpreter is simply not very optimized, so the relative cost of the `switch` is not as high as it could (or, arguably, should) be.

7.2 Unicode

As Unicode [29] became universally adopted, Emacs and XEmacs both supported the standard. Emacs 21.1 supported the `utf-8` coding-system, but it was not unified with other charsets. In 2001, Emacs 22.1 introduced a form of unification between the Unicode charset and several other charsets. XEmacs did the same in 2001 with release 21.4.

As Unicode evolved a universal text representation and supplanted many of the earlier encodings,, Emacs and XEmacs both started efforts to replace the internal MULE representation by Unicode altogether. This appeared in Emacs 23 (2007) and XEmacs 21.5 (starting about 2010 in a separate branch). As a result, the integer representation of a character in both Emacs and XEmacs is its Unicode scalar value.

7.3 Bignums

Somewhat surprisingly for Lisp, Elisp had no support for arbitrarily large integers (*bignums*) for many years. Integer range was restricted by word size on the underlying machine, and representation changes over time have affected the exact range available in Elisp. As a result, various functions dealing with numbers beyond the fixnum range had to implement workarounds. Notable are `file-attributes` (which may use a pair of two fixnums for inode numbers, device numbers, user id, and group id, and may use a float for the file size) and `current-time`, which returns a list of numbers to encode the time. Another place where the limited range of integers has caused friction has been in the fact that it also limits the maximum size of file that can be edited.

Moreover, Elisp was used for more and more applications beyond text editing, and also had to implement workarounds. As a result, Calc, an advanced calculator and computer algebra tool, which has shipped with the Emacs distribution since 2001, had to implement bignum arithmetic in Lisp.

Jerry James added bignums to XEmacs 21.5.18 in 2004, using the GMP library [11]. In Emacs, Gerd Möllman started work on adding support for bignums via GMP around October 2001, but never finished it. It's only in August 2018 that Tom Tromey, with the help of Paul Eggert and several other developers, finally added support for bignums to Emacs (again, using GMP).

The support for bignums in XEmacs includes arbitrary-precision integers, rationals, and floating point numbers and is optional at build time, so while it is fairly complete, XEmacs's Elisp programs still cannot rely on bignum support. Consequently, `file-attributes`, `current-time`, and Calc still do not take advantage of bignums.

In contrast, Emacs's bignum support is currently restricted to arbitrary-precision integers but the feature is provided unconditionally by bundling the `mini-gmp` library with Emacs for

those systems where GMP is not installed. The lack of support for rationals and arbitrary-precision floats is only a reflection of the lack of interest for these features. The support was made unconditional so that the code does not need to keep alternate code paths for when bignums are not available. As a result, `file-attributes` and Calc have been modified to use native bignums (but not `current-time` which would need arbitrary-precision rationals or floating points).

The introduction of bignums raises some design issues in Elisp, as previously integers were always unboxed. This meant that the fast `eq` only behaved differently from `eql` on floating point numbers. As a result, Elisp could assume that, if two integers represented the same number, `eq` would return true on them. Bignums are heap-allocated, so the same is not necessarily true for two bignums. In XEmacs, `eq` can return `nil` in this case, and this seems to have caused no serious problems. This issue is still being discussed with Emacs.

7.4 Terminal-local and frame-local variables, specifiers

In 1995, Emacs-19.29 added the ability to have *frames* on several different X11 servers at the same time. XEmacs evolved similarly. This led to a requirement that certain aspects of display should be local to a frame or an output device. Emacs calls GUI windows *frames* and as with buffers, Emacs maintains a reference to the *current frame* which determines the implicit target of GUI operations. XEmacs also maintained *devices* as part of the display context, to distinguish, say, between different TTYs and different X11 servers.

As buffer-local variables already allow settings that are sentitive to context, Emacs furthered the analog by adding the notion of *terminal-local* variables, which are variables which take different values depending on the X11 server (the *terminal*) to which the current frame belongs. The set of terminal-local variables is small and predefined in the C code; they are mostly used internally to keep track of things like keyboard state; there is no way for Elisp programs to create others.

XEmacs chose a different route: Starting in 1995, Ben Wing (working from a prototype by Chuck Thompson) implemented *specifiers*. which are objects that manage properties that depend on a generalized notion of *display context* [32]. The first prototype implementation was released with XEmacs 19.12.

A specifier's value (its *instance*) depends on its *locale*, which can be buffer, a window, a frame, a device, or a MULE character set, or certain properties of these. For example, `default-toolbar` is a specifier that could be created with:

```
(defvar default-toolbar (make-specifier 'toolbar))
```

(The `'toolbar` is a type argument that the specifier system uses for validation.)

Elisp code could retrieve a specifier's value (its *instantiator*) like this:

```
(specifier-instance default-toolbar)
```

Code can modify a specifier with a value that is buffer-local as follows:

```
(set-specifier default-toolbar <value> (current-buffer))
```

Emacs-20 in 1998 added the ability to set variable to a frame-local value. Contrary to terminal-local variables, any variable can be made frame-local, and additionally, a variable can be both frame-local and buffer-local at the same time.

In 2008, during the development of Emacs-23.1 several bugs were found and fixed in corner case interactions between let-bindings and buffer-local and frame-local variables (for example, when a variable is made buffer-local between the moment a let-binding is entered

and when it is left), and at that occasion it was decided that variables should not be allowed to be both buffer-local and frame-local.

The work on those bugs made it clear that the implementation of buffer-local and frame-local bindings was too hard to follow, so in 2010 the implementation was reworked to make the different possible states more explicit in the code, and at the same time it was decided that frame-local variables should be deprecated: while buffer-local variables are used extensively in Elisp and replacing them with explicit accesses to fields or properties of buffer objects would make Elisp code heavier, frame-local variables were not in widespread use and could easily be replaced by more traditional use of accessors to frame properties, making it hard to justify the extra complexity in the implementation. So in 2012 with the release of Emacs-24.1, it became impossible to let-bind frame-local variables any more, and in 2018 with the release of Emacs-26.1 frame-local variables have been removed altogether.

8 POST-XEMACS

Between 1991 and 2001, Emacs improved rather slowly compared to XEmacs. But starting around 2001, Emacs's pace picked up again. In 2008 Richard Stallman stepped down (again) from the maintainership of Emacs, and the new maintainers have proved more eager to make Elisp evolve, whereas XEmacs started to lose momentum starting about 2010.

This section discusses some notable evolution of the design of Elisp during that time: lexical scoping (Section 8.1), Common Lisp compatibility (Section 8.4), generalized variables (Section 8.5), object-oriented programming (Section 8.6), native support for objects (Section 8.7), generators (Section 8.8), concurrency (Section 8.9), inline functions (Section 8.10) and various attempts at module systems (Section 8.11).

8.1 Lexical scoping

While Scheme was already about to get its second revision when Richard Stallman started to work on GNU Emacs, and he obviously knew about Scheme, being developed in a nearby office, Elisp was mostly derived from MacLisp and used exclusively dynamic scoping. Dynamic scoping is used extensively to add "hidden" configuration options to existing functions without passing explicit arguments [25].

Eventually, lexical scoping became the established standard in the Lisp family in both Common Lisp and Scheme. So of course, the question of adding lexical scoping to Elisp has been brought up many times.

The first implementation appeared quite early, in the form of the `lexical-let` macro, which was part of the new `cl.el` by Dave Gillespie introduced in 1993 and which performed a local form of closure-conversion. While this macro was used in many packages, it was never considered as a good solution to the problem of providing lexical scoping. The somewhat long name was likely a factor, but the reason was more probably due to the fact that the code generated by the macro was less efficient than equivalent dynamically-scoped code and was more difficult to debug because the backtrace-based debugger showed you the gory details of the macro-expansion rather than the corresponding source. For these reasons, `lexical-let` was only used in those particular cases where lexical scoping was really beneficial.

Dynamic scoping had two main drawbacks in practice:

- The lack of closures. Some packages circumvented the lack of closures by building lambda expressions on the fly with constructs like `‘(lambda (x) (+ x ’,y))`, which suffered from various problems such as the fact that macros within that closure were expanded late, and its code was not seen by the byte-compiler. Emacs-23.1

introduced the curry operator `apply-partially` to cover similar use cases without those drawbacks.
- The global visibility of variable names, requiring more care with the choice of local names. The convention followed in Emacs to name all global variables with a package-specific prefix works well to avoid name conflicts, except in the presence of higher-order functions, like `reduce`, and it was also problematic in a few other cases such as in the byte-compiler: in order to emit warnings about the use of undeclared variables, the byte-compiler just tested whether that variable was already known to Emacs, which always returned true for those variables locally bound by one of the functions on the call stack, such as the functions in the byte-compiler itself. So some code was made uglier with long local variable names in order not to interfere with other local bindings. Worse: these "solutions" were never really complete.

The only fully satisfactory solution to the desire for lexical scoping in Elisp was that it should be the scoping used by default by all binding constructs, as is the case in Common Lisp. But at the same time, there was a non-negotiable need to preserve compatibility with existing Elisp code, although some limited breakage for rare situations could be tolerated.

The vast majority of existing Elisp code was (and still is) agnostic to the kind of scoping used in the sense that either dynamic or lexical scoping gives the same result in almost all circumstances. This was true of early Elisp code and has become even more true over time as the byte-compiler started to warn about references to undeclared variables. Warning about unused variables would have probably pushed even more Elisp code to be agnostic. But in any case, it seemed clear that despite the above, the majority of Elisp packages relied somewhere on dynamic scoping. So while there was hope to be able to switch Elisp to use lexical scoping, it was not clear how to find the few places where dynamic scoping is needed so as to avoid breaking too many existing packages.

In 2001, Matthias Neubauer implemented a code analysis that, instead of trying to find the places where dynamic scoping is needed, tries to find those bindings for which lexical scoping would not change the resulting semantics [18]. This tool could have been used to mechanically convert Elisp packages to a lexically scoped version of Elisp, while preserving the semantics. The plan with this approach was to facilitate moving Elisp code to Scheme eventually, but the overall project was too large to ever be realized.

Around 2002, Miles Bader started working on a branch of Emacs with support for lexical scoping. His approach to the problem was to have two languages: an Elisp with dynamic scope and another with lexical scope. Each file was tagged to indicate which language was to be used, and in turn, each function was tagged with which language it was using, so functions using dynamic scope could seamlessly call functions with lexical scope and vice versa. This way, old code would keep working exactly as before and any new code that wanted to benefit from lexical scoping would simply have to add the corresponding "`-*-lexical-binding:t -*-`" at the beginning of the file.

The two languages were sufficiently similar that the new lexically scoped variant only required minor changes to the existing interpreter. But the changes needed to support this new language in the byte-compiler were more problematic, causing progress on this branch to be slow. This branch was kept up-to-date with the main Emacs development but the work was never completed.

It was only in 2010 that Stefan Monnier's student Igor Kuzmin worked on a summer project in which he tried to solve the problem differently: instead of directly adding support for lexical scoping and closures to the single-pass byte-compiler code (which required significant

changes to the code), the idea was to implement a separate pass to perform closure conversion as a pre-processing step. This freed the closure conversion from the constraints imposed by the design of the single-pass byte-compiler, making it much easier to implement, and it also significantly reduced the amount of changes needed in the byte-compiler, thus reducing the risk of introducing regressions.

Two years later, Emacs-24.1 was released with support for lexical scoping based on Miles Bader's *lexbind* branch combined with Igor's closure conversion. The main focus at that point was to:

- minimize the changes to the existing code to limit incompatibility with existing Elisp packages;
- make sure performance of existing code was not affected by the new feature;
- provide reliable support for the new lexical scoping mode, though not necessarily with the best performance.

Changes to the byte code were introduced as part of the lexical scoping feature that appeared in 2012 in Emacs-24.1, but were actually developed much earlier, probably around 2003. Until the introduction of lexical-scoping, the stack-based byte-code only used its stack in the most simple way, and did not include any stack operation beyond `pop`/`dup`/`exch`, so to better support lexical scoping where the lexical variables are stored on the stack, several byte-codes were added to index directly into the stack, to modify stack slots, and to discard several stack elements at once.

Performance of the new lexical scoping mode proved to be competitive with the performance of the dynamic scoping mode except for its interaction with the `catch`, `condition-case`, and `unwind-protect` primitives whose underlying byte-codes were a poor fit, requiring run-time construction of Elisp code to propagate the lexical context into the body of those constructs. So in Emacs-24.4, new byte codes were introduced and the byte-code compiler was modified to be able to make use of them. Nowadays, code compiled using lexical scoping is generally expected to be marginally faster than if compiled with dynamic scoping.

8.2 Eager macro-expansion

The exact time at which a macro is expanded has never been clearly specified in Elisp. Until Emacs-24, macro-expansion usually took place as late as possible for interpreted code, whereas for byte-compiled code, macro-expansion always took place during byte-compilation, with some notable exceptions where the code was "hidden" from the byte-code compiler. In the byte-code compiler, the macro-expansion was also done "lazily" in that it was done on the fly during the single pass of compilation.

In order to implement the separate closure conversion phase for Emacs-24, this had to be changed so that the code is macro-expanded in a separate phase before closure conversion and the actual byte-compilation, using a new `macroexpand-all` function. This caused some visible differences in corner cases where some macros ended up expanded in code which earlier was eliminated by optimizations before getting to the point of macro-expansion, but in practice this did not cause any serious regression.

This use of the new `macroexpand-all` function was made yet a bit more prevalent in Emacs-24.3 which applies it when loading a non-compiled file, so that macro-expansion now happens "eagerly" when loading a file rather than lazily when Emacs actually runs the code. This eager macro-expansion occasionally bumps into problematic dependencies (typically in files which were never compiled), so it fails gracefully: if an error is signaled during the macro-expansion that takes place while loading a file, Emacs just aborts the

macro-expansion and continues with the non-expanded code as in the past, though not without duly notifying the user about the problem.

Emacs-25.1 additionally fine-tuned these macro-expansion phases (both while loading a file and while compiling them) according to the section 3.2.3.1 of the Common Lisp HyperSpec [22], so as to improve the handling of macros that expand to both definitions and uses of those definitions.

8.3 Pcase

While working on the lexical-binding feature, Stefan Monnier grew increasingly frustrated with the shape of the code used to traverse the abstract syntax tree, littered with `car`, `cdr` carrying too little information, compared to the kind of code he would write for that in statically typed functional languages with algebraic datatypes.

So he started working on a pattern matching construct inspired by those languages. Before embarking on this project, he looked for existing libraries providing this kind of functionality, finding many of them for Common Lisp and Scheme, but none of them satisfying his expectations: either the generated code was not considered efficient enough, or the code seemed too difficult to port to Elisp, or the set of accepted patterns was too limited and not easily extensible.

So the `pcase.el` package was born, being first released as part of Emacs-24.1, and used extensively in the part of the byte-code compiler providing support for lexical binding.

Additionally to the `pcase` macro itself that provides a superset of Common Lisp's `case` macro, this package also provides the `pcase-let` macro, which uses the same machinery and supports the same patterns in order to deconstruct objects, but where it is allowed to assume that the pattern does match and hence can skip all the tests, leaving only the operations that extract data.

After the release of Emacs-24.1, Stefan was made aware of Racket's `match` construct [8], which somehow eluded his earlier search for existing pattern matching macros and whose design makes it easy to define new patterns. The implementation of Racket's `match` could not be easily reused in Elisp because it relies too much on the compiler's efficient handling of locally defined functions, but `pcase.el` was improved to follow some of the design of Racket's `match`. The new version appeared in Emacs-25.1 and the main resulting novelty was the introduction of `pcase-defmacro` which can define new patterns in a modular way, often using the new low-level pattern `app`.

8.4 CL-lib

While the core of Elisp has evolved very slowly over the years, the evolution of other Lisps (mostly Scheme and Common Lisp) has put pressure to try and add various extensions to the language. As it turns out, Elisp, to a first approximation, can be seen as a subset of Common Lisp, so already in 1986 Cesar Quiroz wrote a `cl.el` package which provided various Common Lisp facilities implemented as macros.

Richard Stallman never wanted Elisp to morph into Common Lisp, but he saw the value of offering such facilities, so this `cl.el` package was included fairly early on into Emacs, and has been one of the most popular packages, used by a large proportion of Elisp packages. Yet, Richard did not want to impose `cl.el` onto any Emacs user, so he imposed a policy where the use of `cl.el` was restricted *within* Emacs itself. More specifically, Elisp packages bundled with Emacs were restricted to limit their use of `cl.el` in such a way that `cl.el` never needed to be loaded during a normal editing session. Concretely, this meant that the only features of `cl.el` that could be used were: macros and inlined functions.

The reasons why Richard did not want to use `cl.el` and turn Elisp into Common Lisp are not completely clear, but the following elements seem to have been part of the motivation:

(1) Common Lisp was considered a very large language back then, so in all likelihood it would have taken a significant effort to really make Elisp into a reasonably complete implementation of Common Lisp.
(2) Many aspects of Common Lisp design did not make consensus, as evidenced by the divide between Common Lisp and of Scheme. Richard disliked several aspects of Common Lisp's design, such as the use of keyword arguments, especially in low-level primitives like `mapcar`.
(3) Some aspects of Common Lisp's design can incur an important efficiency cost, and Emacs already carried the stigma of *eight megabytes and constantly swapping*, so there were good reasons to try and not make Elisp's efficiency any worse.
(4) Keeping Elisp small meant that users could participate in its development without having to learn all of Common Lisp. When inclusion of Common Lisp features was discussed, Richard would often point out the cost in terms of the need for more, and more complex, documentation.
(5) The implementation of `cl.el` was fairly invasive, redefining some core Elisp functions.
(6) Finally, turning Elisp into Common Lisp would imply a loss of control, in that Emacs would be somewhat bound to Common Lisp's evolution and would have to follow the decisions of the designers of Common Lisp on most aspects.

Over the years, the importance of the first two points has waned to some extent. Also the popularity of the `cl.el` package, as well as the relentless pressure from Emacs contributors asking for more Common Lisp features has also reduced the relevance of the third point.

XEmacs took the easy route on this and loaded `cl.el` into the standard XEmacs image, starting with XEmacs 19.14 in 1996. Emacs instead took a longer road, where over the years, various macros and functions from `cl.el` were found to be sufficiently popular to move them into Elisp proper:

1997 The release of Emacs-20.1 sees the move of the macros `when` and `unless` as well as the functions `caar`, `cadr`, `cdar`, and `cddr`.
2001 Emacs-21.1 includes the hash-table functions, reimplemented in C, as well as the Common Lisp concept of *keywords* (though only as objects, not as arguments). Additionally the macros `dolist`, `dotimes`, `push`, and `pop` are also added to Elisp, which introduced some difficulties: in `cl.el` those macros included extra functionality which relied on parts of `cl.el` which we did not want to move to Elisp proper, specifically `block`/`return` and generalized references. For that reason the macros added to Elisp do not actually replace those of `cl.el`; instead when `cl.el` is loaded, it overrides the original macros with its own version.
2007 Emacs-22.1 adds `delete-dups`, which provides a subset of `cl.el`'s `delete-duplicates`.
2012 Emacs-24.1 adds `macroexpand-all` and lexical scoping, which obsoletes `cl.el`'s `lexical-let`.
2013 Emacs-24.3 adds compiler macros, `setf` and generalized references.
2018 To the `cXXr` functions incorporated in Emacs-20.1, Emacs-26.1 adds the remaining `cXXXr` functions. The resistance against those was mostly one of style, since they tend to lead to poorly readable code.

During the development of Emacs-24.3 the issue of better integration of the `cl.el` package came up again. The main point of pressure was the desire to use `cl.el` *functions* within packages bundled with Emacs. The main resistance from Richard Stallman was coming from

the last point above, but this time, a compromise was found: replace the `cl.el` package with a new package (called `cl-lib.el`) which provides the same facilities but with names which all use the `cl-` prefix. This way, the `cl-lib.el` package doesn't turn Elisp into the Common Lisp language, but instead provides Common Lisp facilities under its own namespace, leaving Elisp free to evolve in its own way.

On that occasion, some details of `cl.el`'s implementation were reworked to be less invasive. The main aspect was that `cl.el` redefined Emacs's macro-expansion wholesale with its own implementation, which incorporated support for `lexical-let`, `flet`, and `symbol-macrolet`, so this was reworked in `cl-lib.el` to provide those features while still using the standard macro-expansion code.

To encourage adoption of this new library, a forward compatibility version of `cl-lib.el` for use on older Emacs and XEmacs versions was released at the same time as Emacs-24.3. Despite the annoyance of having to use a `cl-` prefix, which caused some resistance to this new library, the change has been surprisingly successful if we look at the proportion of new packages which use `cl-lib.el` instead of `cl.el`.

8.5 Generalized variables

To facilitate the move to `cl-lib.el`, some frequently used functionality from `cl.el` was moved directly to Elisp. The most visible one is the support for *generalized variables*, also variously known as *places*, *generalized references*, or *lvalues*. A generalized variable is a form that can be used as an expression and an updateable reference. The concept comes from Common Lisp, and the Emacs implementation originally a part of `cl.el`. In both Common Lisp and Emacs, a number of special forms take generalized variables as operands—in particular, `setf`, which treats a generalized variable as a reference and sets its value.

In Common Lisp, macros accepting a place can ask `get-setf-expansion` to turn it into a list of five elements:

(*VARS VALS STORE-VAR STORE-FORM ACCESS-FORM*)

such that for example `(push` *`EXP PLACE`*`)` turns into:

```
(let ((v EXP))
  (let* (VARS = VALS)
    (let ((STORE-VAR (cons v ACCESS-FORM)))
      STORE-FORM)))
```

This imposes a fairly rigid structure which, while general enough to adapt to most needs, can be burdensome and leads to verbose code with a lot of plumbing, both in the implementation of places and in the implementation of macros which take places as arguments.

The original `cl.el` code followed this Common Lisp design. But when implementing the support for `setf` and friends in Elisp, a fresh new implementation of the concept was used. The reasons for this new implementation were:

- NIH syndrome: the existing code was hard to follow.
- The previous code made use of internal helper functions from `cl.el` which we wanted to keep in `cl.el`, so some significant massaging was needed anyway.
- the implementor of *cl-lib* considered this part of Common Lisp's design ugly.

So the reimplementation uses a different design: instead of a five element list, a *place* maps to a single higher-order function. This higher-order function takes as its sole argument a *DO* function of two arguments, the *ACCESS-FORM* and the *STORE-FUNCTION*. For example, the `push` macro could be naively implemented as:

```
(defmacro push (EXP PLACE)
  `(let ((x ,EXP))
     ,(funcall (gv-get-place-function PLACE)
               (lambda (ACCESS-FORM STORE-FUNCTION)
                 (funcall STORE-FUNCTION `(cons x ,ACCESS-FORM))))))
```

instead, Elisp provides a macro `gv-letplace` which lets us write the above as:

```
(defmacro push (EXP PLACE)
  `(let ((x ,EXP))
     ,(gv-letplace (ACCESS-FORM STORE-FUNCTION) PLACE
        (funcall STORE-FUNCTION `(cons x ,ACCESS-FORM)))))
```

This design generally leads to cleaner and simpler code, and we can easily provide backward compatibility wrappers for most of Common Lisp's primitives.

Any 5-tuple representation of a place can easily be turned into a corresponding higher-order function. The reverse is not true, however, so this design precludes compatibility with Common Lisp and `cl.el`'s `get-setf-expansion`, which must produce the five values described above. Breaking compatibility with `get-setf-expansion` was of course a downside, but in practice this function was almost never used outside of `cl.el` itself so very few packages were impacted by this incompatibility.

8.6 Object-oriented programming

While `cl.el` early on (originally developed in 1986 by Cesar Quiroz , and included in Emacs-18.51 in 1988) provided compatibility with Common Lisp's `defstruct`, including the ability to define new structs as extensions/subtypes of others, thus providing a limited form of inheritance, actual support for object-oriented programming in the form of method dispatch has been historically limited in Emacs.

The first real step in that direction was the development of EIEIO by Eric Ludlam around the end of 1995, beginning of 1996. The official name "Enhanced Implementation of Emacs Interpreted Objects" hints at the earlier existence of some "Emacs Interpreted Objects" package but in reality the acronym came before its expansion. EIEIO started as an experiment to try use an object system in Emacs, first following a model like that of C++, but very soon switching to a CLOS-inspired model.

EIEIO is an implementation of a subset of CLOS. Its development was mostly driven by actual needs more than as an end in itself: the original motivation was to try and play with an object request broker, then a widget toolkit, and later switched to providing support for the CEDET package. It included support for most of CLOS's `defclass`, as well as support for a subset of `defmethod`, more specifically it was limited to single-dispatch methods, dispatching on the first argument, and it could only dispatch based on types of `defclass` objects. It also had incomplete support for method combinations, only allowing `:before` and `:after` methods but not `:around` nor any user-defined additional qualifiers.

EIEIO spent most of its life as part of the CEDET package (a package providing IDE-like features) before being integrated into Emacs-23.2 in 2010, along with most of CEDET. Use of EIEIO within Emacs stayed fairly limited, partly for reasons of inertia, but also because EIEIO suffered some of the same problems as `cl.el` in that it was not "namespace clean".

At the end of 2014, Stefan Monnier started to try and clean up EIEIO so as to be able to use it in more parts of Emacs. The intention was basically to add a "`cl-`" prefix as was done for `cl-lib` (because it was perceived that an "`eieio-`" prefix would be too verbose to be

popular), but there was also a desire to improve the `defmethod` with support for `:around` methods and dispatch on other types than those defined with `defclass`.

It became quickly evident that the implementation of method dispatch needed a complete overhaul: rather than constructing combined methods up-front and memoizing the result, as in typical CLOS implementations, EIEIO's dispatch and `call-next-method` did all their work dynamically, relying on dynamically-scoped variables to preserve state in a way that was both brittle and somewhat inefficient.

So, instead of improving EIEIO's `defmethod`, a completely new version of CLOS's `defmethod` was implemented in the new `cl-generic.el` package, which appeared in Emacs-25.1. The main immediate downside was that the idea to cleanup the rest of EIEIO (which implements `defclass` objects) ended up forgotten along the way. The implementation is not super efficient, but it's already several times faster than the previous one in EIEIO. This package provides largely the same featureset as CLOS's `defmethod`, except for some important differences:

(1) Method combinations cannot be specified per method like in CLOS, but instead new method combinations can be added globally by adding appropriate methods to `cl-generic-combine-methods`. This seemed like a good idea, but there is no known user of this feature at this time, not even internal.
(2) The set of supported specializers is not hard-coded. Instead, they can be defined in a modular way via the notion of *generalizer* inspired from [23]. This is used both internally (to define all the standard specializers) as well as in some external packages, most notably in EIEIO to support dispatching on `defclass` types.

The main motivation for the first difference above was that CLOS's support for method combinations seemed too complex: the cost of implementation was not justified by the expected use of the feature, so it was replaced by a much simpler mechanism.

As for the second difference, it was made necessary by the need to dispatch on EIEIO objects even though `cl-generic.el` could not depend on EIEIO since it was not clean enough. There were additional motivations for it, though: not only it was clearly desirable to be able to define new specializers, but it also made the implementation of the main specializers cleaner, and most importantly it seemed like an interesting problem to solve.

Existing code that could make use of this new machinery, required dispatching on contextual information (i.e. on the current state) rather than only on arguments. Consequently, `cl-generic.el` also adds support to its `cl-defmethod` for pseudo-arguments of the form "`&context (`*`EXP SPECIALIZER`*`)`". This is used for methods which are only applicable in specific contexts, such as in specific major modes or in frames using a particular kind of GUI.

The implementation of `cl-generic.el` was accompanied by an extension of the on-line help system so as to be able to give information not just about Elisp variables, functions, and faces but also other kinds of named elements, starting with types. And to go along with that, the implementation of `cl-defstruct` was improved to better preserve information about the type hierarchy so that the on-line help system can be used to traverse it. This started as an attempt to adapt to `cl-generic.el` the EIEIO facilities to explore interactively EIEIO objects and methods, but is more modular and better integrated with the rest of Emacs's on-line help system.

8.7 Actual objects

While Emacs-25's `cl-generic.el` introduced object-oriented programming facilities into Elisp, objects (whether defined via `cl-lib`'s `cl-defstruct` or EIEIO's `defclass`) were still represented as vectors and hence couldn't be reliably distinguished from vectors, for example to pretty-print them.

This was addressed in Emacs-26 by the introduction of the `make-record` primitive and corresponding new object type. Those *records* are implemented just like the vectors used previously, except that their tag indicates that they should be treated as records instead of vectors, and that by convention the first field of a record is supposed to contain a type descriptor, which can be just a *symbol*.

The main complexity introduced by this change was the need for a new syntax to print and read those new objects, as well as the incompatibility between the printed representation of objects using the old vector-based encoding and those using the new encoding.

8.8 Generators

With the success of Python's and Javascript's iterators and generators, some Emacs users felt like Elisp was lacking in abstraction, so in 2015, Daniel Colascione developed the `generator.el`, which was included into Emacs-25.1. It makes it easy and convenient to write generators using macros `iter-lambda` and `iter-yield`. Its implementation is based on a kind of local conversion to continuation-passing style (CPS) and hence relies extensively on the use of lexical scoping, to work around the fact that Elisp does not directly provide something like `call/cc` to access underlying continuations. It only handles a (relatively large) subset of Elisp, because CPS conversion of forms like `unwind-protect` cannot be defined in general in Elisp.

8.9 Concurrency

Elisp is a fundamentally sequential language, and it relies very heavily on side-effects to a global state. Yet, its use in an interactive program has inevitably lead to a desire for concurrency to try and improve responsiveness. So concurrency appeared very early on: already in Emacs-16.56 Emacs included support for asynchronous processes, i.e. the execution of separate programs whose output was processed by so-called *process filters* whenever the Elisp execution engine is idly waiting for the next user command.

While this very limited form of cooperative concurrency was slightly improved in 1994's Lucid Emacs 19.9 and 1996's Emacs-19.31 by adding native support for timers (they were earlier implemented as an asynchronous process sending Emacs output at the requested time), it has been the only form of concurrency available for most of Emacs's life.

Adding true shared-memory concurrency to Elisp is very problematic because of the pervasive reliance on a shared state in all of existing Elisp code, but the limited existing support was limiting even in those cases where it was technically sufficient: many Elisp packages which interact with external programs block many more times than really necessary, simply because in order to avoid it the coder needs to write their code in a *continuation-passing style*, which interacts poorly with dynamic scoping and requires significant surgery to retro-fit to existing code.

So, shared-memory concurrency was largely considered as inapplicable to Elisp. Nevertheless, in November 2008, Giuseppe Scrivano posted a first naive attempt at adding threads to Elisp. This effort did not go much further, but it inspired Tom Tromey to try its own luck at the game. In 2010, he started to work on adding shared-memory cooperative concurrency

primitives like `make-thread` to Emacs. Interaction with the implementation of dynamic scoping, which is based on a global state for speed, required experimentation with various approaches. Correctly handling buffer-local and frame-local bindings without a complete rewrite was particularly painful and many approaches were abandoned simply because it was too difficult to keep them up-to-date with the evolving Emacs codebase.

This result of this was finally released in 2018, as part of Emacs-26.1. Context switches still only take place at a few known points where Elisp is idle (or via explicit calls to `thread-yield`). The current implementation of this feature makes context switches take time proportional to the current stack depth, because the dynamic bindings of the old thread need to be saved and removed, after which the dynamic bindings of the new thread need to be restored. Earlier implementation approaches tried to avoid this expensive form of context switching, by making global variable lookups a bit more expensive instead, but these required much more extensive and delicate changes to existing code, so while they may still be good options for the future, this first implementation favored a simpler and safer approach.

Over the years, other approaches to concurrency and parallelism have been developed as Elisp packages, most notably the `async.el` package developed in 2012 which runs Elisp code in parallel in a separate Emacs subprocess.

8.10 Inline functions

Function calls are fairly expensive in Elisp, and their semantics involves looking up the current definition in the global name space, so function inlining is at the same time important for performance and semantically visible.

So during the development of the new byte-code compiler for Emacs-19, a new `defsubst` macro was added which works like `defun` except that it annotates the function so the byte-code compiler inlines it whenever it can. This inlining was fairly naive, but worked both for compiled and non-compiled functions (by either inlining the function's body into the source code or into the generated byte-code of the caller).

The new `cl.el` package by Dave Gillespie included in 1993 introduced a new form of inlining in the form of the `defsubst*` macro, which is almost identical to `defsubst` from the outside, except for including support for Common Lisp extensions like keyword arguments, but its implementation is different and does not correctly preserve the semantics of the dynamically scoped formal arguments, which are instead replaced by substitution with the actual arguments.

Of course, additionally to those two, Elisp came with a third way to implement an "inlinable function", namely by defining it as a macro.

In late 2014, while working to adapt the `cl-lib` library to the changes in EIEIO and `cl-defstruct` objects, Stefan Monnier became frustrated by the redundancy between `cl-typep`'s definition and its compiler macro. The `cl-typep` function takes two arguments and tests if the first is a value of the type specified by the second. The function definition takes care of the general case where the type argument is only known at run-time, but the compiler-macro is important to optimize for example `(cl-typep x 'integer)` into `(integerp x)`. While this optimization could arguably be performed automatically by a sufficiently sophisticated compiler, the Elisp compiler is much too naive for that. So he developed the new macro `define-inline`, which was included in Emacs-25.1. It lets one define a function in such a way that from its definition the macro can extract both a normal function body and a corresponding compiler-macro.

8.11 Module system

While Elisp was designed from the beginning as a real programming language rather than a tiny ad-hoc extension language, it was not designed for "programming in the large" as witnessed by the lack of module or namespace system.

Yet, the Elisp side of Emacs is now a rather large system, so in order to avoid name conflicts, Elisp relies on a poor man's namespace system, as mentioned in Sec. 8.1, where code loosely follows a convention where global functions and variables belonging to package ¡*pkg*¿ will define identifiers starting with a `<pkg>-` prefix (and use a prefix of `<pkg>--` in order to indicate that this identifier should be considered an internal definition).

There have been many attempts to remedy this situation by providing support for some form of namespacing:

- In May 2011, Christopher Wellons developed the Fakespace package which lets one define private variables and functions and then prevents them from escaping into the global namespace. Non-private definitions still rely on the usual package-prefix naming convention to avoid conflicts.
- In October 2012, Chris Barrett developed the Namespaces package which provides an extensive set of new macros to define namespaces, define functions and variables in those namespaces, and use them from other namespaces.
- In March 2013, Wilfred Hughes developed the proof-of-concept `with-namespace` macro which basically adds the specified namespace prefix to all the elements defined in its body.
- At the same time, Yann Hodique developed the proof-of-concept Codex package which instead tries to provide a functionality similar to Common Lisp's packages, where each package has its own *obarray*.
- In early 2014, Artur Malabarba developed the Names package, which takes the approach of `with-namespace`, but doing a much more thorough job.
- In 2015, the same Artur Malabarba developed the Nameless package which takes a completely different approach: it does not provide any new Elisp construct and instead focuses on making Emacs *hide* the package prefixes from the user while working on the code.

To date, no namespacing facility has been incorporated into Emacs, nor seen much use in other packages. The last time this subject was (hotly) debated among Emacs maintainers and contributors, around 2013 following a blog post by Nic Ferrier, no consensus appeared, but other than inertia one of the main arguments in favor of the status quo was that Elisp's poor man's namespacing makes cross-referencing easier: any simple textual search can be used to find definitions and uses of any global function or variable, including filesystem-wide searches or even web searches, whereas all the alternatives introduce names that need to be interpreted relative to their context forcing the reliance on an IDE that understands the particular namespacing used when browsing the code.

9 ALTERNATIVE IMPLEMENTATIONS

Implementation of Elisp have not been confined to Emacs and its derivatives. Two implementations—Edwin and JEmacs—are notable for running Elispcode on editors implemented independently from Emacs. Moreover, a Common Lisp package emulates Emacs Lisp, and Guile Scheme also comes with support for the Elisp language. These implementations all aim at running existing Elisp code in alternative environments, and consequently feature no significant language changes.

9.1 Edwin

Edwin is the editor that ships with MIT Scheme [16]. Its user interface is based on that of Emacs. Edwin is implemented completely in Scheme, and Scheme is its native extension language. Additionally, Matthew Birkholz implemented an Elisp interpreter in Scheme that was able to run substantial Elisp packages [3] at the time, among them the Gnus news reader.

9.2 Librep

In 1993, John Harper started working on an embeddable implementation of Elisp called Librep, which is most famously used as the extension language of the Sawfish window manager. While Librep started as a Lisp dialect that was mostly compatible with Elisp, it has since significantly diverged, including a module system, lexical scoping, tail-call elimination, and first-class continuations.

9.3 Elisp in Common Lisp

In 1999, Sam Steingold also implemented the Elisp language as a Common Lisp package [27]. He was motivated by the hope of moving Emacs to work with a Common Lisp language instead of Elisp. His `elisp.lisp` package does not attempt to reimplement the library of functions provided in Emacs to manipulate buffers and other related objects, so it focuses on the "pure" Elisp language; but it was able to run the non-UI parts of the Emacs Calendar, which provides sophisticated functions to manipulate and convert dates between many historical calendars.

9.4 JEmacs

JEmacs [5] is an editor that ships with Kawa Scheme [4]. JEmacs comes with support for running some Elisp code. Its implementation (written partly in Java and partly in Scheme) works by translating Elisp code to Scheme, and running the result.

9.5 Guile

Guile Scheme [9] was conceived as the universal extension language of the GNU project, with the specific intention of replacing Elispin Emacs at one point. This has not happened (yet), but Guile does ship with a fairly complete implementation of Elisp that translates Elisp programs to Guile's intermediate language. It is used in the Guile-Emacs system, which is a work-in-progress modification of Emacs where the Elisp engine is provided by Guile.

9.6 Emacs-Ejit

In 2013, Nic Ferrier implemented in Elisp a compiler from Elisp to Javascript. This was designed so as to be able to write complete web sites all in Elisp, using the Elnode Elisp package to do the server-side processing and using Emacs-Ejit to write the client-side code in Elisp as well. It does not really aim to run any Elisp package in your browser, so its runtime library only provides a small subset of Elisp's standard primitives.

10 CONCLUSION

Many steps of Elisp's evolution have been the result of efforts of single individuals, driven by a specific purpose, yet it has managed to keep an arguably sane and cohesive overall design, thanks on the one hand to a maintainership which was more interested in improving the text editor than the language and kept an eye on the longer term, and on the other to the

willingness to break backward compatibility in specific cases, in order to gradually address problems encountered over time.

Like Lisp in general the development of Emacs has seen plenty of discord, with unbridgeable personal differences, heated debates, and forks along the way. However, througout these differences, Elisp has steered a remarkably stable course of conservative development and gradual extension. It has mostly grown by slowly incorporating popular features from other languages, both in the language itself and in its implementation. But it has also come up with its own features, such as docstrings, buffer-local variables, and the addition of text-properties to strings.

Possibly, the organic growth of all the Elisp packages developed within the community of Emacs users and developers, whose composability relies on social mechanisms, has provided enough cohesive force to keep balkanization at bay for more than 30 years. Given the amount of changes it went through over the last decade, we are looking forward to the Elisp of the next 30 years.

10.1 Acknowledgments

Emacs and Elisp are the result of the contribution of an impressive number of individuals. We thank them all for their contributions of course. With respect to this article, while the efforts put into maintaining the revision history of Emacs through the various revision systems it has used have been very helpful, we'd like to thank also Lars Brinkhoff for his archiving work at https://github.com/larsbrinkhoff/emacs-history which fills some of the holes of the early life of Emacs.

REFERENCES

[1] A. Bawden. Quasiquotation in Lisp. In O. Danvy, editor, *Proceedings of the ACM SIGPLAN Workshop on PEPM Partial Evaluation and Semantics-Based Program Manipulation PEPM '99*, pages 4–12, San Antonio, Texas, USA, Jan. 1999.

[2] B. Berliner. Cvs ii: Parallelizing software development. In *USENIX Winter 1990 Technical Conference*, volume 341, page 352, 1990.

[3] M. Birkholz. Emacs Lisp in Edwin Scheme. Technical Report A.I. Memo No. TR-1451, Massachusetts Institute of Technology, Sept. 1993.

[4] P. Bothner. The Kawa Scheme language, 1999. URL https://www.gnu.org/software/kawa/index.html.

[5] P. Bothner. JEmacs - the Java/Scheme-based Emacs. *Free Software Magazine*, Mar. 2002. URL http://jemacs.sourceforge.net/JEmacs-FSM.html.

[6] R. J. Chassell. *An Introduction to Programming in Emacs Lisp*. Free Software Foundation, Boston, Massachusetts, Emacs version 26.1 edition, 2018. URL https://www.gnu.org/software/emacs/manual/html_mono/eintr.html.

[7] M. Crestani. A new garbage collector for XEmacs. Master's thesis, Universität Tübingen, 2005. URL http://crestani.de/xemacs/pdf/thesis-newgc.pdf.

[8] M. Flatt and PLT. *The Racket Reference*. PLT, v.7.0 edition, Aug. 2018. URL https://docs.racket-lang.org/reference/index.html.

[9] *Guile Reference Manual*. Free Software Foundation, July 2018. URL https://www.gnu.org/software/guile/manual/.

[10] R. P. Gabriel. Letter to Chris DiBona and Tim O'Reilly. Web page. URL https://www.dreamsongs.com/DiBona-OReillyLetter.html.

[11] GMP. The GNU Multiple Precision arithmetic library. URL https://gmplib.org/.

[12] J. Gosling. *Unix Emacs*. Carnegie-Mellon University, 1981.

[13] B. Hayes. Ephemerons: A new finalization mechanism. In *Proceedings of the 12th ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications*, OOPSLA '97, pages 176–183, New York, NY, USA, 1997. ACM.

[14] P. Kellomäki. PSD - a portable scheme debugger. *SIGPLAN Lisp Pointers*, VI(1):15–23, jan 1993. URL https://dl.acm.org/citation.cfm?doid=173770.173772.

[15] B. Lewis, D. LaLiberte, R. Stallman, and GNU Manual Group. *GNU Emacs Lisp Reference Manual*. Free Software Foundation, 51 Franklin St, Fifth Floor, Boston, MA 02110-1301, 3.1 edition, 2018. URL https://www.gnu.org/software/emacs/manual/elisp.html.
[16] *MIT/GNU Scheme*. Massachusetts Institute of Technology, 2014. URL https://www.gnu.org/software/mit-scheme/documentation/mit-scheme-user/index.html. Version 9.4.
[17] D. A. Moon. *MACLISP Reference Manual*. Project MAC - M.I.T., Cambridge, Massachusetts, Apr. 1974. Revision 0.
[18] M. Neubauer and M. Sperber. Down with Emacs Lisp: Dynamic scope analysis. In *International Conference on Functional Programming*, pages 38–49, Florence, Italy, Sept. 2001. URL http://www.deinprogramm.de/sperber/papers/dynamic-scope-analysis.pdf.
[19] K. Ohmaki. Open source software research activities in AIST towards secure open systems. In *7th IEEE International Symposium on High Assurance Systems Engineering*, pages 37–41, Oct 2002.
[20] K. M. Pitman. The revised Maclisp manual. Technical Report Technical Report 295, Laboratory for Computer Science, MIT, Cambridge, Massachusetts, 1983. URL http://www.maclisp.info/pitmanual/. Saturday Morning Edition.
[21] K. M. Pitman. Condition handling in the Lisp language family. In A. Romanovsky, C. Dony, J. Knudsen, and A. Tripathi, editors, *Advances in Exception Handling Techniques*, volume 2022 of *Lecture Notes in Computer Science*. Springer, 2001. URL http://www.nhplace.com/kent/Papers/Condition-Handling-2001.html.
[22] K. M. Pitman. Common Lisp HyperSpec, 2005. URL http://www.lispworks.com/documentation/HyperSpec/Front/index.htm.
[23] C. Rhodes, J. Moringen, and D. Lichteblau. Generalizers: New metaobjects for generalized dispatch. In *European Lisp Symposium*, 2014. URL http://research.gold.ac.uk/9924/1/els-specializers.pdf.
[24] R. Stallman. My Lisp experiences and the development of GNU Emacs. Speech transcript, Oct. 2002. URL https://www.gnu.org/gnu/rms-lisp.en.html. International Lisp Conference.
[25] R. M. Stallman. EMACS: The extensible, customizable self-documenting display editor. In *Proceedings of the ACM SIGPLAN SIGOA Symposium on Text Manipulation*, pages 147–156, New York, NY, USA, 1981. ACM. URL https://www.gnu.org/software/emacs/emacs-paper.html.
[26] G. L. Steele, Jr. and R. P. Gabriel. The evolution of Lisp. In *The Second ACM SIGPLAN Conference on History of Programming Languages*, HOPL-II, pages 231–270, New York, NY, USA, 1993. ACM.
[27] S. Steingold. Load Emacs-Lisp files into Common Lisp, 1999. URL https://sourceforge.net/p/clocc/hg/ci/default/tree/src/cllib/elisp.lisp.
[28] SXEmacs. SXEmacs – redefining Emacs. Web site. URL http://www.sxemacs.org/.
[29] The Unicode Consortium. The Unicode Standard. Technical Report Version 6.0.0, Unicode Consortium, Mountain View, CA, 2011. URL http://www.unicode.org/versions/Unicode6.0.0/.
[30] W. F. Tichy. RCS - a system for version control. *Software Practice&Experience*, 15(7), 1985. URL https://www.gnu.org/software/rcs/tichy-paper.pdf. Purdue University.
[31] A. Tolmach and A. W. Appel. Debugging Standard ML without reverse engineering. In *Conference on Lisp and Functional Programming*, pages 1–12, 1990. URL https://dl.acm.org/citation.cfm?id=91564.
[32] B. Wing, B. Lewis, D. LaLiberte, and R. Stallman. *XEmacs Lisp Reference Manual*, version 3.3 edition, 1998. For XEmacs Version 21.0.

Received August 2018