Evolution of Emacs Lisp

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

Shepherd: Brent Hailpern, IBM Research, USA

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. Emacs Lisp is 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, Emacs Lisp has evolved and continues to do so.

Important aspects of Emacs Lisp have been shaped by concrete requirements of the editor it supports 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 alongside new functions. In particular, Emacs Lisp 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 Emacs Lisp development is closely tied to the projects and people who have shaped it over the years: We document Emacs Lisp 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 → History of programming languages**.

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

ACM Reference Format:
Stefan Monnier and Michael Sperber. 2020. Evolution of Emacs Lisp. *Proc. ACM Program. Lang.* 4, HOPL, Article 74 (June 2020), 55 pages. https://doi.org/10.1145/3386324

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.

2475-1421/2020/6-ART74
https://doi.org/10.1145/3386324

Contents

1 INTRODUCTION

Emacs is a text editor originally developed by Richard Stallman, who is also the founder of the Free Software Foundation (FSF) and who launched the GNU Project.

Emacs Lisp 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 Emacs Lisp code, and yet more Emacs Lisp 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 Emacs Lisp in their configuration file, it is arguably one of the most widely used dialects of Lisp.

1.1 The Authors

Neither of us was around when Emacs and Emacs Lisp were designed, but we are two researchers in programming languages with a keen interest and many years of experience in (X)Emacs and Emacs Lisp, which made us well positioned to write this article.

Stefan Monnier has been (and is still) a core contributor to Emacs since 1999, was head maintainer from 2008 to 2015, and he supervised student projects on the Emacs Lisp implementation in Emacs.

Michael Sperber has been a core contributor to XEmacs since 1994. Most of his work was done from 1995 to 2003, when he was a research assistant at the University of Tübingen. Apart from his own work on the XEmacs codebase, he supervised several student projects on the Emacs Lisp implementation in XEmacs, including a co-authored paper on the analysis of dynamic scope [Neubauer and Sperber 2001].

1.2 Paper Organization

Emacs Lisp has evolved in several 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.

Emacs as we know it today itself started in 1984. Section 3 describes the driving motivation for the early design and implementation of Emacs Lisp. 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 impact on Emacs Lisp 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, to finally conclude in Section 9.

Emacs Lisp was re-implemented in several other projects outside this succession. We briefly touch upon those projects in Appendix A. Appendix B contains the transcript of an interview with Joseph Arcenaux, the initial maintainer of Emacs 19.

2 EMACS HISTORY

While in theory Emacs Lisp exists independently from Emacs, its design, implementation, and history are inextricably linked to that of Emacs, so we present here a short overview of the important events in Emacs's life.

2.1 Emacs's Early History

Emacs's original inception was as a set of macros and keybindings for the TECO text editor. TECO was written by Dan Murphy starting in 1962 [Murphy 2009], then at the MIT AI lab, and featured a primitive programming language. In the mid-1970s Richard Stallman, then also at MIT, added to TECO a "real-time edit mode" (then a novelty) that updated the display of the text being edited as typing happened [Greenberg 1996]. This mode allowed mapping key strokes to little TECO programs, and users would typically configure TECO to bind printing characters to self-insert, and control characters to perform other tasks such as navigation to other parts of the text via so-called *macros*. Guy Steele coordinated among the users, creating a common set of keybindings and macros, which Richard Stallman evolved into the first version of Emacs [Moon et al. 1978; Seibel 2009]. ("Emacs" stands for "editor macros.") Note that TECO and Emacs were not completely separate entities in the sense that TECO's extension language was also Emacs's extension language and

that they evolved together, where TECO's extension language was regularly extended to provide facilities then used by Emacs [Stallman 2002].

This original Emacs grew popular in the lab and soon started to be reimplemented in various other systems. Several of those incarnations of Emacs were written in Lisp, notably *EINE* (recursive acronym for "EINE Is Not Emacs") on the Lisp Machine by Dan Weinreb, followed by *ZWEI* (for "ZWEI was EINE Initially") by Dan Weinreb and Mike McMahon [Weinreb 1979], and *Multics Emacs* by Bernie Greenberg in 1978 [Greenberg 1996; Stallman 2002] written in MacLisp [Moon 1974; Pitman 1983]. Given Lisp's dynamic nature, it was only natural that those systems used Lisp as their extension language.

Unix Emacs, written by James Gosling in 1980/1981 [Gosling 1981] and preserved in the history books as *Gosling Emacs*, was implemented in C and it featured an extension language called *MLisp* or *Mock Lisp*, which bears visual resemblance to Lisp, but was a lesser member of the Lisp family in the sense that it lacked data structures like *cons* cells and was generally too limited to be usable as an implementation language.

Richard Stallman, who had worked on ZWEI [Stallman 2018b], liked the possibilities offered for extending the editor by having Lisp be both the extension language and the implementation language. But high-performance Lisp compilers were not widely available, so he decided to write a (for him) second version of Emacs which would use Lisp as its extension language, like Greenberg's Multics Emacs, but where the implementation would be partly written in Lisp and partly in C (including buffer manipulation primitives, the redisplay code, and a Lisp interpreter) so as to be more widely available.

In 1984, Richard Stallman took Gosling Emacs as a starting point for this endeavor, and began replacing the Mock Lisp interpreter and data structures of Gosling Emacs with a new interpreter for the language that is now Emacs Lisp, adapting internal data structures to those of this Emacs Lisp interpreter. At that time, Gosling Emacs was distributed commercially by Unipress under the name *Unipress Emacs*, and Unipress ordered Richard Stallman to stop distributing his version of Emacs, which forced him to replace or rewrite the rest of Emacs's code as well, such as the code responsible for the redisplay. It also motivated him to invent the GNU General Public License [Tai 2001], to try and ensure that users of his code would never have to go through such an experience.

2.2 Design Goals

The initial design of Emacs Lisp was motivated by the pervasive requirement of *extensibility*: users "must be able to redefine each character" [Stallman 1981]. 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 Emacs Lisp started right from the beginning as a full-featured programming language with powerful abstraction facilities, thus foregoing Greenspun's tenth rule, a popular epigram on programming that came up in 1993: "Any sufficiently complicated C or Fortran program contains an ad-hoc, informally-specified, bug-ridden, slow implementation of half of Common Lisp." [Greenspun 2003]

Moreover, Richard Stallman made the design of Emacs embody and showcase the ideals of Free Software. For example, not only is it permitted 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 Emacs Lisp language. Retrospectively, those can be summarized as:

- 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 technical expertise. For example, the *Introduction to Programming in Emacs Lisp* tutorial [Chassell 2018] targets users with no programming experience. This has been a strong

motivation to keep Emacs Lisp 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* (Section 4.8) 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* (Section 4.7), is discouraged because it makes the code more opaque.
- Emacs should be easily portable to as many platforms as possible. This largely explains why Emacs Lisp still runs in a simple byte-code interpreter.

2.3 The Great Schism

In 1988, Lucid Inc., a software development company founded by Richard Gabriel and based in Menlo Park, California, started a project called *Energize* [Gabriel 2019; Gabriel et al. 1990]. Energize was to be a C/C++ integrated development environment based on Emacs [Gabriel 1999]. 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. For Energize, Lucid needed a graphical user interface and the ability to add a large number of annotations to program source code.

Lucid hired Joseph Arcenaux, the principal Emacs developer at that time, to add these features for Lucid while continuing to work on the upcoming release of Emacs 19. It soon became clear that the respective goals Lucid and the Free Software Foundation had for Emacs 19 were incompatible, and the required cooperation between Lucid and the Free Software Foundation broke down (Appendix B).

As a result, Lucid forked Emacs development, creating its own Emacs variant *Lucid Emacs*, whose primary developer and maintainer was Jamie Zawinski.

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*. Sun eventually lost interest in XEmacs, which continued as an open-source community effort.

2.4 Timeline

Table 1 contains a timeline of significant milestones in the development of Emacs and Lucid Emacs/XEmacs. It also includes a few milestones from *Epoch*, a predecessor to Lucid Emacs that had support for multiple windows.

For the time until 2007, the timeline in this table borrows much from Jamie Zawinski's timeline [Zawinski 2007]. According to the `etc/NEWS.1-17` file, Emacs versions counted from 1.1 to 1.12 and then switched to 13 which was the first version announced publicly on usenet.

2.5 Development Model

This section describes the development organization of Emacs and Lucid Emacs/XEmacs, respectively.

Emacs. Emacs is a Free Software project, developed by a loosely connected set of volunteers doing it mostly in their spare time, and several of them do not consider themselves computer professionals. Development is not really organized, in the sense that volunteers work on what they find appealing rather than follow some agreed upon plan, and the overall direction of evolution is in effect controlled by the *head maintainer(s)* by accepting or rejecting contributions, and via discussions in mailing lists. At times, the Free Software Foundation paid the maintainer a salary, but this is not currently the case.

Table 1. Emacs development timeline

Date	Version	Maintainer	Notes
1985-03	Emacs 13	Richard Stallman	Earliest recorded release
1985-07	Emacs 16.56	Richard Stallman	Oldest version still available Gosling Emacs code expunged
1987-09	Emacs 18.49	Richard Stallman	
1987	Epoch	Alan M. Carroll	Forks from Emacs 18.49
1988-12	Epoch 1.0	Alan M. Caroll, Simon Kaplan	
1989-08	Emacs 18.55	Richard Stallman	
1990	Emacs 19	Joe Arcenaux	Forks from Emacs 18.55 Text properties, Sec. 4.11 `advice.el`, Sec. 4.7
1990-04	Lucid Emacs 19.0	Jamie Zawinski	Extents, Sec. 4.11
1990-08	Epoch 4.0	Marc Andreesen	
1992-10	Emacs 18.59	Richard Stallman	Last release of Emacs 18
1993-05	Emacs 19.7 beta	Jim Blandy	First public beta of Emacs 19 `lambda` macro, Sec. 4.3
1993-05	Emacs 19.8 beta	Jim Blandy	
1993-09	Lucid Emacs 19.8	Jamie Zawinski	4-bit tags, Sec 5.4 Merged Epoch and Emacs 19.8
1994-05	Emacs 19.23 beta	Richard Stallman	
1994-05	Lucid Emacs 19.10	Jamie Zawinski	
1994-09	XEmacs 19.11	Chuck Thompson, Ben Wing	
1994-11	Emacs 19.28	Richard Stallman	Proper backquote, Sec. 4.2 First real release of Emacs 19
1995-06	Emacs 19.29	Richard Stallman	3-bit tags, Sec. 5.4
1995	XEmacs 20	Steve Baur	Work on MULE support begins
1995-09	XEmacs 19.13	Chuck Thompson, Ben Wing	Merged Emacs 19.30
1996-08	Emacs 19.34	Richard Stallman	Last release of Emacs 19
1997-02	XEmacs 20.0	Steve Baur	Custom library, Sec. 7.2
1997-03	XEmacs 19.15	Steve Baur	
1997-09	Emacs 20.1	Richard Stallman	First release with MULE
1997-10	XEmacs 19.16	Steve Baur	
1998-02	XEmacs 20.4	Steve Baur	First stable release with MULE Packages shipped separately
1998-07	XEmacs 21.0	Steve Baur	1-bit tags
2001-04	XEmacs 21.4	Stephen Turnbull	
2001-10	Emacs 21.1	Gerd Möllmann	New redisplay engine
2007-06	Emacs 22.1	Richard Stallman	Last bits of Mock Lisp removed
2009-07	Emacs 23.1	Chong Yidong, Stefan Monnier	Unicode internally, Sec. 7.3
2012-06	Emacs 24.1	Chong Yidong, Stefan Monnier	Lexical scoping, Sec. 8.1
2013-03	Emacs 24.3	Chong Yidong, Stefan Monnier	CL-Lib, Sec. 8.4
2014-10	Emacs 24.4	Chong Yidong, Stefan Monnier	`nadvice.el`, Sec. 4.7
2016-09	Emacs 25.1	John Wiegley, Eli Zaretskii	
2018-05	Emacs 26.1	John Wiegley, Eli Zaretskii	Records, Sec. 4.5 Threads, Sec 8.9

Richard Stallman was the original maintainer and still participates. Even though several other people have been official maintainers at various points in time, he always acted as a "default maintainer" when no one wanted to step up and he has followed Emacs's development very closely even when he was not officially the maintainer. Also, as Richard Stallman is the founder of the Free Software Foundation, Emacs is part of the FSF's GNU project and is linked to the FSF by its use of FSF's resources to host the code and own the copyright. The link with the FSF is of course particularly strong because of Stallman's role in Emacs.

The organization of Emacs's development is not really codified, but in practice the role of the maintainer is a balancing act between the desire to maintain a sane code base, to promote Free Software, to be attractive to its users, and to encourage contributions. There is also more structure to the organization: maintenance is really shared among a handful of people who earned each other's respect by the weight of their past contributions and who can then make more or less any change they like to the code, and accept or reject external contributions, as long as nobody complains. In case of disagreement between those core contributors the maintainer has the final say. This structure probably appeared around 1991 and until 2000 was embodied in the fact that the core contributors were members of a private *emacs-core* mailing-list and were the only ones with direct access to the code. Since late 2000, when the CVS server became public and the *emacs-core* private mailing-list was replaced by the public *emacs-devel* mailing-list, this has become much more blurry: technically, more than a hundred contributors have direct write access to the code now and nowhere is it ever written or said which of those is or is not considered a core contributor at any given point it time. Instead the system functions by self-regulation: every time someone installs a change without prior permission, someone else may or may not complain about the change's quality, which can lead to requiring further changes or even a complete revert, so over time contributors learn to judge on their own what makes a change acceptable.

Lucid Emacs/XEmacs. Lucid Emacs was initially managed by the respective maintainers, who integrated patches sent by contributors, and published tarballs for releases on ftp sites.

This changed when Steve Baur took over as maintainer in 1996: He imported the source code into a publically accessible CVS repository on the SourceForge platform for open-source software. Steve Baur then established a *review board* whose members received commit access to the repository, and whose job was to review and apply patches via a special mailing list, `xemacs-patches`, which exists to this day.

For XEmacs 19.15, Steve Baur also split off many Emacs Lisp packages into a separate repository, with the ability to manage and update the packages independently from the core releases, and installed individual maintainers for many of these packages. In 2007, Michael Sperber migrated the source code from CVS to Mercurial, hosted on the Bitbucket platform.

3 EARLY LANGUAGE DESIGN

The design of Emacs Lisp was based on the past experiences with the extension languages of TECO, ZWEI, Gosling Emacs, and Multics Emacs. Arguably, the strongest influence for the language itself was MacLisp while the design of the editing primitives was influenced by Gosling Emacs's MLisp.

3.1 Mock Lisp

Gosling Emacs, arguably the immediate predecessor of today's Emacs, featured an extension language called *MLisp* or *Mock Lisp*, which bears visual resemblance to Emacs Lisp. MLisp featured function definitions via `defun`, as well as many built-in functions (such as `eolp`, `forward-character`, `save-excursion`) whose names survive in Emacs Lisp. Emacs even contained some limited backward-compatibility support for MLisp until Emacs 22 in 2007.

MLisp was a quite limited language: It lacked cons cells and lists. MLisp did have dynamic scoping 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.

With the design of the primitives, Richard Stallman adopted MLisp's basic approach [Stallman 2018b], diverging from ZWEI. ZWEI primitives would always take explicit arguments to specify on what text to operate. Here is what a ZWEI function might have looked like that inserted a string surrounded by parentheses:

```
(DEFUN INSERT-PARENS (BP STRING)
  (LET* ((BP1 (INSERT BP "("))
         (BP2 (INSERT BP1 STRING))
         (BP3 (INSERT BP2 ")")))
    BP3))
```

The `BP` parameter is for the location at which the insertion is to take place. `INSERT-PARENS` calls the primitive `INSERT` function three times; `INSERT` returns a new location, after the insertion, which gets threaded through the subsequent `INSERT` calls and finally returned.

The Mock Lisp primitives differed from those of ZWEI in that they operated at the current *point* (the current "cursor position"), as was the case in TECO. Here is the `in-parens` function from the Gosling Emacs manual [Gosling 1981]:

```
(defun
      (in-parens
        (insert-string "(")
        (insert-string (arg 1 "String to insert? "))
        (insert-string ")")
      ))
```

Stallman, who had worked on ZWEI, found ZWEI's approach clumsy and adopted MLisp's approach for Emacs.

3.2 Maclisp

MacLisp was a Lisp dialect developed at MIT's Project MAC in the early 1970s, which was named for its conceptual relationship with "Multiple Access Computers" and "Machine-Aided Cognition" [Pitman 1983]. It was developed for use in artificial intelligence research and related fields. It descended from Lisp 1.5, albeit with many changes [Moon 1974]. MacLisp ran on the DEC PDP-10 series machines under various operating systems, and on the Honeywell 6180/6880 under Multics. Maclisp had all basic properties associated with Lisp today [Moon 1974]:

- Structure is defined through parentheses and prefix notation.
- Code forms have an internal representation as *S-expressions*, which print the same as the code, a property called homoiconicity.
- There is an `eval` function that evaluates an expression represented as an S-expression.
- The basic data structure is *cons*. A cons pairs two objects, with the first component named *car*, the second *cdr*. Conses are used to implement singly-linked *lists*, where a special *nil* value indicates the end of the list. Conses are mutable.
- A special *symbol* data type represents identifiers in S-expressions, but is also often used to represent enumerations, status values etc.
- It is possible to define new compound data types via the `defstruct` form.
- The language has first-class functions via the `lambda` form.

- MacLisp also has an `apply` function that will apply a function to a list of arguments, converting between argument lists and regular lists.
- Local variables are bound *dynamically* rather than lexically: A local variable occurence refers to the last active binding at run time, rather than the lexically enclosing binding.
- The language has *macros* that introduce new syntactic forms; macros are essentially defined as functions on the S-expressions representing code.

Here is a simple example of a MacLisp function definition from the original manual [Moon 1974]:

```
(defun assoc (x y)
  (cond
    ((null y) nil)
    ((equal x (caar y)) (car y))
    ((assoc x (cdr y)))))
```

This looks for an entry `x` in a list of conses `y`, an association list. The `cond` is a branching construct with a list of branches. The first one has condition `(null y)`, which checks whether the `y` list is empty. If it is, `assoc` returns `nil`. The second branch has condition `(equal x (caar y))`, which checks whether `x` is equal to the car of the first element of `y`. (The `caar` function takes the "car of the car.") In that case, it returns the first cons, whose car is equal to `x`. Finally, if none of the cases matched, `assoc` recurs on the cdr of `y`.

This function definition works in Emacs Lisp unchanged to this day.

4 BASE LANGUAGE DESIGN

When designing Emacs Lisp, Richard Stallman was not really interested in the design of a new language, but had more pragmatic concerns: he wanted to have a real Lisp system but was constrained by the need for the implementation to be simple and lightweight enough to fit the limited resources of the machines of the time. So the base language of Emacs Lisp is a straightforward subset of MacLisp [Moon 1974; Pitman 1983] and Lisp Machine Lisp [Weinreb and Moon 1981].

Contrary to its siblings Common Lisp and Scheme, which both strive for a kind of completeness and consistency (although in different ways), Richard Stallman has had no such desire for Emacs Lisp, even occasionally reminding contributors not to "propose new Emacs features for mere completeness' sake." [Stallman 2005].

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

Emacs Lisp also supports arrays, using a subset (specifically only uni-dimensional arrays) of the primitives offered in Lisp Machine Lisp, which happens to also be a subset of the Common Lisp primitives, rather than Maclisp's approach of accessing arrays as if they were functions.

Like MacLisp, Emacs Lisp offered only dynamic scoping of variables. Also like MacLisp, Emacs Lisp was (and still is) a Lisp-2 language [Steele and Gabriel 1993], which means that 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.

Unlike MacLisp, the original Emacs Lisp featured no way to define new data types and some specialized control structures were missing in the original Emacs Lisp, among them the `do` loop construct, and the accompanying `return` and `go` forms for non-local control transfer. This reduced Emacs's memory footprint to make it work on Unix systems with one megabyte of memory [Stallman 2018b].

This section discusses the notable choices in language design such as the choice of dynamic scoping, notable differences from and additions to MacLisp, and language features designed to satisfy requirements of the Emacs editor.

4.1 Symbols and Dynamic Scoping

Like any Lisp, Emacs Lisp has always had a symbol data type: The expression `'emacs` yields a symbol named `emacs`. One way to look at symbols is that they are immutable strings with a fast equality check and usually fast hashing. This makes them suitable for values representing enumerations or options. Another is that symbols can represent names in an Emacs Lisp program. Symbols are thus a key feature to make Emacs Lisp *homoiconic*, meaning that each Emacs Lisp form can be represented by a data structure called an *S-expression* that prints out the same as the form.

Emacs Lisp took from MacLisp the choice of dynamic scoping implemented using *shallow binding*, where the latest binding of a variable is simply stored in the *value slot* of the heap object that represents the symbol, resulting in simple and efficient variable lookups and let-bindings. Symbols can be used to refer to variables. In particular, the `set` function, which performs assignment, accepts a symbol naming the variable as its argument. A symbol, again borrowing from MacLisp, also carries a *property list*, essentially a key-value mapping stored in a list.

Richard Stallman chose dynamic scoping despite the fact that the two major emerging Lisps at the time—Common Lisp and Scheme—were moving to lexical scoping [Clinger 1985; Steele 1984]. Richard Stallman chose dynamic scoping to meet extensibility requirements in Emacs [Stallman 1981]. The Emacs code base uses variables to hold configuration options. These configuration options often need to be modified temporarily, restoring the original value after a piece of code has run. Here is an example:

```
(defun dired-smart-shell-command (command &optional output-buffer error-buffer)
  "Like function `shell-command', but in the current Virtual Dired directory."
  (interactive ...)
  (let ((default-directory (or (and (eq major-mode 'dired-mode)
                                    (dired-current-directory))
                               default-directory)))
    (shell-command command output-buffer error-buffer)))
```

This function invokes `shell-command` to execute an external command. The working directory of that command is determined by the `default-directory` variable: In Dired mode (a built-in file manager) that directory needs to be the directory the cursor is at. So in this case, the `let` is used to temporarily bind `default-directory` to the return value of `(dired-current-directory)`.

It would in principle be possible to structure the code base to pass configuration options explicitly, but this would make the code verbose, and require changing many function signatures once a new configuration option gets added to the system.

Richard Stallman considered the possibility of having an alternate scope rule early, but deemed the availability of dynamic scoping necessary [Stallman 1981]. More recently, lexical scoping has been added as an option to Emacs (Section 8.1).

4.2 Backquote

Quasiquotation is a classic feature of Lisp readers, including MacLisp's [Bawden 1999]: It allows creating nested list structures—particularly those representing code—via templates.

Quasiquotation is a more general mechanism than `quote` for creating nested list structure without using constructors explicitly: `quote` is usually introduced via `'` where the *reader* translates `'`*SEXP* to `(quote` *SEXP*`)`, and this expression evaluates to the value *SEXP*. For example:

```
'(a (b c d) c)
```

evalutes to a list whose first element is the symbol `a`, the second a list consisting of elements `b`, `c`, and `d`, and whose third element is `c`. It does so in constant time, by always returning the same value already present in the source code returned by the *reader*.

Quasiquotation is usually introduced via `` ` ``, where the reader translates `` `*exp* `` to `(quasiquote` *exp*`)` (nevertheless, the mechanism is often referred to as *backquote*). Quasiquotation operates like `'`, but a `quasiquote` form may contain `,` and `,@` subforms, which insert or splice the value of expressions into the result.

Here is a `quasiquote` example:

```
`(do ((i 0 (+ 1 i)))
     ((>= i ,array-size))
  (aset ,array-name i ,init-val))
```

This might expand into the following `quasiquote`-less code:

```
(list 'do '((i 0 (+ 1 i)))
      (list (list '>= 'i array-size))
      (list 'aset array-name 'i init-val))
```

Emacs Lisp did not until 1994 include the reader syntax for quasiquotation. Instead, the special forms that came with quasiquotation—backquote (`` ` ``), unquote (`,`) and unquote-splicing (`,@`) were simply the names of special forms. Thus, what is usually written as:

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

was written in early Emacs Lisp 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 former. The extended reader had to rely on a heuristic for the translation, however, because `` (`a) `` is valid in both syntaxes but means different things: It could either denote an old-style backquote expression (i.e., `` (` a) `` in the old notation) or a single-element list containing the new-style backquoted symbol `a` (i.e., `` ((` a)) `` in the old notation).

So the old format, though obsolete, was still supported, and the new format was recognized only in some cases; more specifically the new unquote was recognized only within a new backquote, and the new backquote was recognized only 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 by themselves, 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 recognized only if it follows a parenthesis and is followed by a space. The main motivation for this change was the introduction of the `pcase` macro (Section 8.3) 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.

Support for the old-style syntax has been preserved for many years because it was used very widely and removing it would have caused regressions for too many users relying on outdated or unmaintained third-party packages. But over the years, users have slowly updated or stopped using those packages, so this syntax is finally removed in Emacs 27.

4.3 Lambda

In Lisp dialects, anonymous functions are usually the result of `lambda` expressions. Here is an anomymous function that adds 1 to a number:

```
(lambda (x) (+ x 1))
```

Interestingly, `lambda` was not originally a keyword of the Emacs Lisp language (unlike in MacLisp). But anonymous function *values* could be lists of the following form:

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

The use of dynamic scope made it unnecessary to create closures, so it was possible to write these anonymous functions in source code using the pre-existing quoted Lisp literal mechanism:

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

This *expression* evaluates to a nested list structure whose first element happens to be the `lambda` symbol. When a program calls `funcall` passing such a value, `funcall` would recognize the list structure as representing a `lambda` expression and invoke the interpreter on it.

The extra quote character was a small price to pay for a slightly simpler Lisp implementation. But the byte-code compiler is not allowed to compile the content of such a quoted list except in those rare cases where it can determine that it will be used only as a function, so Emacs Lisp acquired in version 1.4 the `function` special form (also imported from MacLisp) for the alternative notation:

```
(function (lambda (..ARGS..) ..BODY..))
```

In 1992, `lambda` was added as a macro, early during the development of Emacs 19. The macro simply returns the list representation wrapped inside the `function` special form:

```
(defmacro lambda (&rest args)
  (list 'function (cons 'lambda args)))
```

It is not completely clear why it took so long to introduce this simple macro, but it was likely due to the fact that anonymous functions were not used very often and that they were mostly used in places that were not very sensitive to performance.

While the `lambda` macro has made `(lambda ..)` superior to `'(lambda ..)` for almost 30 years now, instances of this practice can still be found, thanks to the power of copy&paste, in many code snippets posted on the web, users' configuration files, and third party Emacs Lisp packages, even though it prevents byte-code compilation of the function's 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.

4.4 Macros

An important feature of Emacs Lisp is the ability to define new syntactic forms via `defmacro`, as shown in the previous example. This is important because it means not only Emacs but also Emacs Lisp itself can be extended to adapt it to the user's needs. This ability has had a profound impact on the evolution of the language, letting it develop independently from the core constructs.

Emacs Lisp macros were taken directly from MacLisp and also correspond very closely to the `defmacro` of Common Lisp. This form of macro definition is well known to suffer from a lack of hygiene, a problem that has been addressed in more recent macro systems like that of Scheme.

While the most obvious consequences of the lack of hygiene are traditionally circumvented in Emacs Lisp using `gensym`, there has never been any attempt or even discussion about addressing this problem of hygiene in a more general way. Maybe this is in part due to the heavy reliance on dynamic scoping which already forces the programmer to be careful with the choice of identifiers.

4.5 Structures

MacLisp offers the `defstruct` form for defining new data types called *structures*. Here is an example which defines a `KONS` data type, which has two fields called `KAR` and `KDR`:

```
(DEFSTRUCT KONS KAR KDR)
```

This form would define four functions, `MAKE-KONS`, `KAR`, `KDR`, and `ALTER-KONS`. A program could create a `KONS` structure with `MAKE-KONS`:

```
(MAKE-KONS KDR 3 KAR 4)
```

`KAR` and `KDR` can then be used to access the respective field contents of a `KONS` structure, and `ALTER-KONS` to mutate these contents.

Emacs Lisp never had `defstruct` as a primitive form. This reflects a more general design philosophy, which uses the existing data types of pairs and vectors for even complicated data structures such as keymaps (Section 6.1).

Instead, a Common-Lisp-compatible `defstruct` has been part of the `cl.el` package (Section 8.4). Use of `defstruct` in Emacs Lisp code began to be commonplace only around 2010. Originally `defstruct` used plain vectors under the hood to represent the structures, making it impossible to reliably distinguish them from vectors, but in 2013 Lars Brinkhoff added the *record* data type to make structures distinct from vectors (Section 8.7). Making `defstruct` use this new data type introduced some backward incompatibilities, so the change lived in a separate branch for four years during which various mitigation strategies were developed, until it was finally included in Emacs 26.1.

4.6 Non-Local Exits

Very early on, Emacs Lisp featured primitives to handle non-local exits. It inherited the `catch`, `throw`, and `unwind-protect` primitives from MacLisp. Here is an example:

```
(defun outer ()
  (catch 'marker
    (inner)))

(defun inner ()
  ...
  (if x
      (throw 'marker 'catch-value))
  ...)
```

When `outer` is called, it calls `inner`. If that call to `inner` evaluates to the `throw` form, it immediately aborts the current computation and jumps to the `catch` form (the last active `catch` form with the same `marker` label), which returns the `catch-value` value passed to it.

The `unwind-protect` form allows attaching a "cleanup form" to an expression, that will run even if the expression evaluates to a non-local exit through `catch`.

MacLisp's error-handling system was quite primitive, and had poor separation of signaling and handling [Pitman 2001]. As a result, Emacs Lisp features a *condition system* inspired by the Lisp Machine (it does not cover all of the features of the Lisp Machine exception 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. Note that it does not dispatch on the error symbol. For the `'wrong-type-argument` error symbol, `(get 'wrong-type-argument 'error-conditions)` returns the condition names `wrong-type-argument` and `error`. This allows `condition-case` dispatch on both `wrong-type-argument` and `error` to work.

Here is an example for `condition-case`:

```
(condition-case err
    (key-binding (this-command-keys))
  (wrong-type-argument
   (message "Incorrect type error: %S" err)))
```

This evaluates `(key-binding ...)`. If a `wrong-type-argument` is signalled during evaluation, then `(message ...)` is evaluated and will print the content of `err` which will be 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 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 a variable bound to a list of zero-argument functions. The `add-hook` function adds a function to such a list by reassigning the variable. For example, the following form causes auto-fill mode to be turned on whenever TEX mode is entered:[1]

```
(add-hook 'TeX-mode-hook #'auto-fill-mode)
```

The `run-hooks` function calls all functions in that list—there is a `(run-hooks 'TeX-mode-hook)` in the code setting up TEX mode.

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

The old TECO version of Emacs also allowed attaching hooks to variable changes [Stallman 1981], but this feature was not provided in Emacs Lisp because Richard Stallman considered it a misfeature, which could make it difficult to debug the code. Yet this very feature was finally added to Emacs Lisp in Emacs 26 in the form of *variable watchers*, ironically meant to be used as debugging aids.

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. This mechanism, similar to aspect-oriented programming [Kiczales et al. 1997] as well as CLOS's method combination [DeMichiel and Gabriel 1987], allows attaching code to a function even if it does not run hooks. This mechanism makes Emacs even more malleable, hence fitting the design goals of Emacs very well, which is why Richard Stallman was happy to include it, but he has always made it clear that its use should be avoided:

> "It is bad practice to make a Lisp program put advice on another Lisp program's function, because that is confusing. When you see that the program calls `mumble`, it might take you hours before you think of checking whether it has advice." [Stallman 2018b]

[1] `add-hook` makes use of the ability to refer to a variable like `TeX-mode-hook` by its name symbol (Section 4.1).

Nevertheless, the advice functionality is very popular, used in many packages, including within Emacs's own Emacs Lisp code. For some packages, it is merely used to address corner-case interoperability with another package, but for others it is the core mechanism on which they are built (such as the *Emacspeak* package, which advises hundreds of Emacs Lisp functions to make Emacs better suited for visually impaired users).

Here is an example `defadvice` form:

```
(defadvice eval-region (around cl-read activate)
  "Use the reader::read instead of the original read if cl-read-active."
  (with-elisp-eval-region (not cl-read-active)
    ad-do-it))
```

This specifies a piece of code that is run on each invocation of the `eval-region` function. The `around` means that the body of the advice explicitly delegates to the advised function via `ad-do-it`. (Also possible are `before` and `after`, where the delegation is done implicitly after or before evaluating the advice's body). The `activate` flag means that the advice becomes active immediately. The `cl-read` is the name that was chosen to identify this advice.

Despite its popularity, most of the non-core features of `defadvice` were very rarely used, and even more rarely used properly because the original design did not match actual usage patterns. Furthermore, the way the `defadvice` macro gave access to function arguments did not work with lexical scoping. This was solved in late 2012: as a result of a discussion with a user asking how to "put several functions into a single variable", Stefan Monnier developed a new package `nadvice.el` that can not only combine several functions into a single variable, but uses that to provide the same core features as `defadvice` but with a much simpler design. This simplification is mostly derived from better reuse of existing language features. For example, the old advice system had special features to control whether a piece of advice should be compiled whereas in the new system there is no need for that because a piece of advice is simply a normal function. Similarly, the old advice system has special primitives to get access to the function's arguments, whereas in the new system, the function's original arguments are passed as normal arguments to the piece of advice and can hence be accessed without any special construct. The `nadvice.el` package was released as part of Emacs 24.4, and has proved popular in the sense that it has largely replaced the old `defadvice` in new code, but rather few packages that used `defadvice` have been converted to the new system.

4.8 Docstrings

An important feature of Emacs from the start was the idea of *self-documentation* [Stallman 1981], which originated with the first, TECO-based, Emacs [Stallman 2018b]. To that end, every definition in Emacs Lisp 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 adapted from the original TECO-based Emacs. Many languages have since adopted docstrings, notably Common Lisp [Pitman 2005] and Clojure [Clojure Clojure]. More distant descendants generate documentation from the source code, such as Javadoc comments in Java.

4.9 Interactive Functions

The most direct method to make a function implemented in Emacs Lisp 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 a function by typing `M-x` *function-name*. This makes sense only 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 `C-u` *number* 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—it may lead to the region being activated.

4.10 Buffer-Local Variables

Buffer-local variables are the prominent feature marking Emacs Lisp as the extension language of a text editor. In Emacs, *buffers* are objects that store the contents of a file while it is 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 affects only dereferences with the same current buffer in effect. Any variable can be made local to any buffer, so it 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;[2] the bug was that this code ended up "restoring" the value of `buffer-file-name` into the wrong buffer when exiting the `let`.

4.11 Strings

Emacs Lisp 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* (Appendix B). Each char of a string can be annotated with a set of properties, mapping 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 without merging them. Richard Stallman felt that 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 [Stallman 2018a].

Along with buffer-local variables (Section 4.10), this is one of the cases where the core Emacs Lisp 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, and 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 the representation of 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, but it does impose a significant constraint: There is an indirection between the string object and the chars it holds because of the `aset` primitive. `(aset string index char)` replaces one of `string`'s characters at index `index` with `char` by side-effect. Because of the variable number of bytes per character, this will sometimes change the string's length in bytes, which can require relocating the string's bytes elsewhere.

Originally, an `aset` call that would have required changing the byte length was not allowed and signaled an error. But when a user reported this limitation as a bug, the implementation was adjusted to relocate the string as needed: the necessary indirection was already present for the

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

purpose of compacting the heap after a garbage collection, and the cost of an $O(n)$ operation to implement the `aset` operation which is usually expected to take constant time was not even discussed among the developers. In practice this extra cost is negligible, because `aset` is very rarely used on strings.

4.12 I/O

One of the ways Emacs Lisp distinguishes itself from most other programming languages is in its treatment of input/output. Rather than follow the usual design based on file or stream objects and primitives like `open`/`read`/`write`/`close`, Emacs Lisp offers only coarser access to files via two primitive functions `insert-file-contents` and `write-region` that 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. Primitive functions 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 Emacs Lisp language was defined by its sole implementation. Various aspects of the design had impact on the users of the language, and this section discusses some of them.

5.1 Byte-Code Interpreter

Emacs has two execution engines for Emacs Lisp, the original versions of which were both already in Emacs 1.1 (the earliest version mentioned in the release notes), and both written by Richard Stallman [Stallman 2019]. One is a very simple interpreter written in C operating directly on the S-expression representation of the source code. The other is a byte-code engine, implemented as a primitive `byte-code` function that interprets its string argument as a sequence of stack-based byte codes to execute. A compiler, written in Emacs Lisp, translates Emacs Lisp code to that byte-code language.

While Emacs Lisp is basically a "run of the mill" programming language, 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 Emacs Lisp 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 Emacs Lisp 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 better support lexical scoping (Section 8.1).

5.2 Tail-Call Optimization

Emacs Lisp does not implement proper tail calls [Clinger 1998]: Each function call consumes stack space.

With Scheme being familiar to many Emacs Lisp 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 were two patches developed independently and submitted to Emacs maintainers in late 2012 (by Troels Nielsen first and Chris Gray 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-or-egg 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 Emacs Lisp. 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 affected only 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 Emacs Lisp code, especially during development or debugging. So if Emacs Lisp code started to rely on proper tail calls, it would tend to cause problems when interpreted. One alternative is to always byte-compile the code and get rid of the Emacs Lisp interpreter, but that's a change that would have much further consequences, such as a more complex bootstrap.
- The available implementations of TCO affect the behavior not only in terms of reducing the stack allocation and increasing performance, but they also eliminate some activation frames from stack backtraces, which would be detrimental to debugging. This is particularly true for tail calls to other functions, such as non-recursive tail calls.

5.3 Bootstrap

Since the Emacs Lisp compiler is itself written in Emacs Lisp, 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 [Tichy 1985], which did not support any form of remote access, so changes were installed by logging remotely into a machine owned by the Free Software Foundation where the RCS files were kept, and performing the commits there. This machine also kept the (last) compiled form of the Emacs Lisp files and so naturally provided the needed compiled files for bootstrap. When development moved to CVS [Berliner 1990] to ease collaboration between contributors, another bootstrapping solution was needed since storing those generated files in the CVS repository would have lead to never ending merge conflicts.

Of course, since Emacs also comes with a simple direct Emacs Lisp 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 Emacs Lisp 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. Consequently, a pre-built copy of that file is checked into revision control.

5.4 Data Representation

The representation of Lisp objects has only slowly evolved over the years in response to changes in the usage patterns. The main changes have been in the representation of an immediate *boxed* Lisp value, which can be either a number or a reference to a heap object, and in heap objects used to represent user-defined objects such as closures and records.

Emacs started with a data representation of its boxed data based on 32-bit words. Those used a 7-bit tag located in the most significant bits, an extra "mark" bit (see below), and 24-bits of immediate data: either an integer or a pointer. Not all the possible 128 tags were actually used, but

the 24 remaining bits were more than 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 [McCarthy 1960], and allocated objects by blocks of 4 KB, with each block dedicated to a particular kind of object: cons cells, floats, symbols, markers, and strings. All other objects were allocated directly with `malloc`. To avoid fragmentation in the block of strings, those were compacted during each GC. The "mark" bit in each 32-bit box belonged to the object that contained this 32-bit box rather than to the object referenced from this box. More specifically, it was used so that cons cells could occupy only two 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 (that is, of the `car`).

Over time, this tagging scheme became problematic, since it limited the Lisp heap to 16 MB. Moreover, the limit on file size is fundamentally linked to the maximum representable integer since that is how buffer positions are represented. This means that 8 MB was the size limit for a file that could be edited.

So in 1995, with the release of Emacs 19.29, the scheme was tweaked and the tag was reduced to 3 bits, pushing the maximum file size to a more comfortable 128 MB and the maximum heap size to 256 MB. To reduce the tag to 3 bits, the types that received their own tag were ints, strings, symbols, cons cells, buffers, and floats. 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 occupy the same space as Lisp markers (6 words), and hence be allocated from the same 4KB block (apart from markers, these were mostly various types of forwarding pointers). 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 be allocated directly by `malloc`.

Richard Mlynarik implemented a similar change in 1993 for Lucid Emacs 19.8, giving Emacs Lisp 28-bit integers and a 28-bit address space. For the same release, Mlynarik added an alternative C-level representation for the `Lisp_Object` type for object descriptors that used the C `union` feature and bit fields rather than "manual" manipulation of bits in the machine word.

Mlynarik's change reflected a general difference in attitude towards using custom data type definitions for representing Emacs Lisp entities (Section 4.5), which went back to discussions between Richard Mlynarik and Richard Stallman years earlier at MIT [Mlynarik 2019]. Lucid Emacs was open to Mlynarik's changes, while Emacs was not. The union representation has been maintained in XEmacs since, but needed to be disabled on many platforms because of bugs in the implementation of unions and bit fields in C compilers.

For the release of XEmacs 19.13 in 1995, Ben Wing merged the data representation from Emacs 19.30. Of course, a limit of 256 MB was still not actually comfortable on machines becoming common at the time.

Kyle Jones and Martin Buchholz added a "minimal-tagbits" configure option to XEmacs with the release of XEmacs 21.0 in 1998, yielding 31-bit integers and a 1 GB maximum file size. The mark bits moved to the object headers, which were also added to cons cells, making them three-word objects. The mark bit was removed from the boxes, and the number of type tags was reduced to four: heap-allocated objects, characters, even and odd fixnums, leaving 30-bit for pointers to heap-allocated objects which matches the usual constraint that objects must be aligned on multiple-of-4-bytes boundaries. This became default with XEmacs 21.2 in 2002.

During the development of Emacs 22 in 2007, Stefan Monnier reworked the tagging scheme of Emacs with the similar goal to be able to use all of the address space and larger files: The mark bit was removed, and the 3 tag bits were moved to the least significant bits, allowing the Lisp heap to grow as large as the full address space allowed, in exchange for the constraint that all objects need

to be aligned on a multiple-of-8-bytes boundary. Moreover, the markbit of cons cells was moved to a separate bitmap stored alongside each block of cons cells, which required allocating those blocks on 4 KB 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 also used for floats, thus reducing the typical memory footprint of floats from 96 bits to 64 bits. Reducing the memory footprint of floats to 64 bits 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.

In Emacs 23.2 the tagging scheme was tweaked to use two tags for integers, pushing the maximum file size to 512 MB, similar to what XEmacs had done for XEmacs 21.0.

The design of Emacs made it basically impossible anyway to view files larger than 4 GB or edit files larger than 2 GB on a 32-bit system, no matter the tagging scheme, so there was not much room for improvement. Nevertheless repeated complaints from users about the inability to edit large files motivated Paul Eggert to cover the remaining space between 512 MB and 2 GB by adding a new compilation option `-with-wide-int` in Emacs 24.1 to make the boxed data use 64 bits on 32-bit systems. This imposes a significant extra cost in terms of space and time but makes it possible to edit files up to about 2 GB. When this compilation option is used, tag bits are placed in the most significant bits again, so that the 32-bit of pointers can be extracted at no cost at all.

5.5 Vector-Bloc Allocation

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 object, Emacs allocates them from "vector blocs". The motivation was not that `malloc` was too slow, but that the implementation of conservative stack scanning (Section 5.6) keeps track of every part of the Lisp heap allocated with `malloc` in a balanced tree, so every such `malloc` costs 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 Emacs Lisp code. There were two factors that changed this situation: first, over time the style of Emacs Lisp coding evolved and the use of `cl.el`'s `defstruct`, which internally represents those objects as vectors (Section 4.5), 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 was improved further so as to reduce their header from 2 words down to a single word. This was not motivated by any concrete performance problem, but by the general desire to reduce the cost of those objects since it was expected that their use would keep increasing as more code is converted to use lexical binding and `defstruct`.

Finally, for Emacs 27, the object representation was changed again, by Paul Eggert: 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. This time the motivation was just to simplify the code, though it came with a very slight performance improvement according to the limited benchmarking performed at the time.

5.6 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 the fields were 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 contain 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 aid, 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.

5.7 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 (then Michael Sperber's students) implemented a scanning algorithm that used the memory-layout descriptors that had been added to support the portable dumper (Section 5.9). 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 [Crestani 2005].

In the process of these GC improvements, Crestani and Michael Sperber also added *ephemerons* [Hayes 1997] to Emacs Lisp: (`make-ephemeron` *KEY VALUE FINALIZER*) returns an ephemeron, a reference to *VALUE*. *VALUE* is reachable through the ephemeron only as long as *KEY* is reachable. When *KEY* becomes unreachable, while the ephemeron itself is still reachable, *VALUE* is queued for finalization.

5.8 New GC Algorithms

During the early 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, possibly because it is not apparent to the user when a pause is actually due to GC. Most of the remaining complaints 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 integrate his new redisplay engine, Gerd Möllmann 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 occur only during Emacs Lisp 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 do not match the ones offered by Boehm's GC (for example, hash-tables can be weak in their keys, or their values, or both, or either).

In XEmacs, GC pauses continued to be a perceived problem (and to former XEmacs users, Emacs's GC pauses are quite recognizable and annoying). In 2005, Marcus Crestani developed an incremental collector for XEmacs, again using a VM-based barrier but a (cheaper) *write* barrier this time. It was released with XEmacs 21.5.21 in 2005 [Crestani 2005] 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.

5.9 Image Dumping

An important feature of Emacs Lisp 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. To take just a single example, in 2016 the Glibc maintainers decided to make internal changes to their `malloc` implementation which broke this functionality [Edge 2016].

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`ped. Galibert added explicit memory-layout descriptors for all Lisp types to the system, which would later also benefit the new incremental garbage collector (Section 5.8). 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 Emacs Lisp 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, which has been integrated into the code for Emacs 27.

5.10 Debugging

Debugging support was added very early to Emacs Lisp: Emacs 16.56 (which appeared in 1985, a mere four months after the initial Emacs release) already included the *backtrace debugger*, which

suspends execution at the time a condition is signaled, showing the current stack backtrace and letting the user examine the state of the application and place breakpoints before pursuing execution.

For many developers, this is still the main debugger for Emacs Lisp. 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 invoked only 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 1988, Daniel LaLiberte developed another Emacs Lisp debugger, called Edebug. It was included into Emacs a few years later during the early development of Emacs 19. Edebug works without any special support in the interpreter; instead it instruments the Emacs Lisp source code the user wants to debug, such that running this code lets the user step through this code and displays the various values returned by the evaluation of each step. We do not know from where LaLiberte took this idea, but Edebug was probably the inspiration for the Portable Scheme Debugger [Kellomäki 1993], and the same basic technique was used (and significantly improved) in SML/NJ [Tolmach and Appel 1990].

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 Emacs Lisp 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. For example, `dolist` is defined as:

```
(defmacro dolist (spec &rest body)
  (declare (debug ((symbolp form &optional form) body)))
  ...)
```

where `(symbolp form &optional form)` explains that the first argument is expected to be a list whose first element is a mere symbol rather than an expression and hence should not be instrumented, whereas the subsequent (second and optional third) elements are expected to be `form`s which means normal expressions that can be instrumented.

5.11 Profiling

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

In 1994, Barry A. Warsaw implemented the `elp.el` package, which took a similar approach but without modifying functions's bodies. Instead it replaced the instrumented functions with wrappers that 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, as part of a more general effort to improve performance, added an Emacs Lisp 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.

While `elp.el` is sufficient for many use cases, it requires prior instrumentation of a specific set of functions, which makes it useless when the potential culprits are completely unknown, as is typically the case when a user simply notices that "Emacs is slow". So in early 2011, Tomohiro Matsuyama started implementing in Emacs's C code a sampling-based profiler for Emacs Lisp, which relies on the same information already used for the backtrace debugger. 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 an actual call tree.

5.12 JIT Compilation

The existing implementations of Emacs Lisp are based on fairly naive interpretation techniques and hence not very efficient. Over the years, a number of efforts tried to remedy the poor performance with just-in-time native-code compilation.

First attempt. In 2004, Matthew Mundell developed the first JIT compiler for Emacs Lisp. This took byte-compiled Emacs Lisp code and used the GNU Lightning library [Lightning 2020] 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: It removed only the immediate interpretation overhead, but did not speed up function calls nor was it able to remove redundant type checks. Consequently, the extra maintenance burden was not considered justified, and this JIT compiler was not included into Emacs.

Second attempt. Around 2012, Burton Samograd developed a second JIT compiler for Emacs Lisp. This took a similar path, but using GNU LibJIT [LibJIT 2020] instead. It was very simplistic, turning each byte-code into a call to a C function. This compiler yielded a disappointing 25% speedup on a `raytracer.el` test.

Third attempt. In 2016, Nickolas Lloyd developed the third JIT compiler for Emacs Lisp, 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 it did get to the point of being stable enough to handle all Emacs code by default.

Fourth attempt. In 2018, Tom Tromey took another stab at it, again using GNU Libjit. Compared to Lloyd's JIT compiler, Tromey's focuses exclusively on code using lexical scoping, 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 reached a speed up factor of 3.5. It can be used on all Emacs code but is still very naive. Thus, it requires further work to avoid pathological behavior in some situations where the JIT compilation is performed too eagerly, leading to significant slowdowns.

During the discussions of whether this JIT deserves further development to include it into Emacs, one of the hurdles that appeared is that the main purpose of a faster Emacs Lisp implementation should be the ability to write new code—for example code that previously would have been written in C. However, this can only work if the faster Emacs Lisp is available for all supported platforms, whereas GNU Libjit does not currently support all the architectures that Emacs supports.

6 XEMACS PERIOD

The focus of Lucid Emacs was on providing a proper graphical user interface. As a result, most of the changes to Emacs Lisp in Lucid Emacs / XEmacs were made to support the move from a TTY-based purely textual model to a graphical model. Lucid's original requirements described in Section 2.3 led to the creation of additional library features and data types, such as events, keymaps, characters, and extents, but put no immediate pressure on the core language or its implementation. Hence this section, which describes those changes, is rather short. The numerous changes to the implementation in Lucid Emacs / XEmacs happened as a matter of more general efforts to improve both the editor and the programming experience and are described in Section 5.

6.1 Event and Keymap Representations

One significant departure from Emacs in Lucid Emacs was the representation of keymaps. A keymap maps a sequence of keystrokes to a function to invoke. Many Emacs key assignments consist of several keystrokes in sequence—for examples, Control-X Control-S saves the current buffer. Keymaps represent this by *nesting* the keymap. For example, the binding for Control-X is not a function, but a nested keymap, and Control-S in the nested keymap is bound to the `save-buffer` function. Moreover, a keymap can inherit the bindings of another keymap.

Emacs, to this day, uses a transparent S-expression representation for keymaps. In early Emacs, 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 keymap representation in Emacs is richer but follows largely the same design. Here is an example [Lewis et al. 2018]:

```
(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)))
```

Reading from the top, 3 is ASCII for Control-C and 26 is ASCII for Control-Z. The nesting in the keymap means that the sequence Control-C Control-Z runs the `run-lisp` function. Next, 27 is ASCII for Escape, which, in Emacs, doubles as the "Meta" modifier, Thus, both Escape Control-X and Meta-Control-X trigger `lisp-send-defun`. Similarly, 127 is ASCII for Delete, which triggers `backward-delete-char-untabify`. Finally, Escape Control-Q and Meta-Control-Q both trigger `indent-sexp`.

This transparent representation 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 of Lucid Emacs (specifically Richard Mlynarik) foresaw these problems, and instead made keymaps into an opaque datatype, forbidding manipulation via the S-expression primitives and giving themselves significantly more implementation freedom [Mlynarik 2019]. This allowed Lucid Emacs to evolve more rapidly the representation of keymaps to cater to a richer set of input events, including mouse events, for instance.

This step reflected a general difference in philosophy (Sections 4.5, 5.4, and 5.5). Lucid Emacs also used custom 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 8^{th} bit represented "meta," basically restricting Emacs to ASCII. Character values could have more modifiers in higher bits than the characters stored in a string.

This situation was no longer tenable when multi-language support came to XEmacs. The work on MULE [Ohmaki 2002] 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 was based on ISO-2022 and encoded a character as an integer divided in two parts: One represented the national character set, the other the associated codepoint within that set.

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 (i.e., `make-char` and `char-int`). Generally, Emacs Lisp 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 Emacs Lisp runtime was written in C, it was always possible to add new Emacs Lisp functions written in C to the system. Those C functions could also call Emacs Lisp functions.

However, functions written in C originally lacked the dynamic nature of Emacs Lisp, as they had to be linked statically into the Emacs executable. Starting in 1998, J. Kean Johnston added facilities to XEmacs (released with version 21.2 in 1999), which allowed Emacs Lisp code to build and dynamically load shared libraries (called *modules*) written in C into the 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, and PostgreSQL.

Even with modules in place, developers still had to create wrappers to make existing C libraries accessible in Emacs Lisp. In 2005, Zajcev Evgeny wrote an FFI for SXEmacs [SXEmacs], a fork of XEmacs.[3] This FFI allows loading and calling existing C libraries directly, without intervening wrappers, by declaring the type signatures of C functions in Emacs Lisp. For example, the FFI allows using Curl like this:

[3]SXEmacs was forked XEmacs 21.4.16 by Steve Youngs at the end of 2004, due to perceived problems with the code quality and stability of XEmacs, and to allow more significant changes than XEmacs could accomodate at the time.

```
(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 features like 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 its licensing terms [Stallman 2003].

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 accept to load only libraries that come 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 the ability to load external libraries, somewhat like XEmacs's FFI. This functionality was highly anticipated, but is still not used very widely, maybe in part because packages that use it, unlike Emacs Lisp packages, cannot be installed directly from source without the end user having a C compiler on his machine, which can be a significant barrier to distribution.

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 Emacs Lisp code written for previous versions, 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.

In took more than ten years until these additions were merged into Emacs 22.1 in 2007.

7 EMACS/XEMACS CO-EVOLUTION

Some aspects of Emacs Lisp 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-code compiler for Lucid Emacs, which, after some initial resistance, was merged into the Emacs codebase by 1992.

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

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 Emacs and XEmacs. As part of this rewrite, a new object type for byte-compiled Emacs Lisp functions was introduced. Prior to that, a byte-compiled function had looked like this:

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

Then in Emacs 19.29 in an attempt to speed up loading of Emacs Lisp 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 Emacs Lisp 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.

In 1989, Martin Buchholz added a just-in-time optimization pass to the XEmacs byte-code interpreter. 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 cycle 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 the XEmacs developers consciously avoided making changes to the byte-code format, this was not a goal with Emacs, and 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 [FSF 2020b], 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%. Still, he pushed stronger for its inclusion, which happened only 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 Custom Library

One of the tenets of Emacs's original design was that customization would happen via Emacs Lisp. For example, whether Emacs allows selecting a region via shifted motion keys is controlled by an Emacs Lisp variable called `shift-select-mode`. If a user wanted to change a default behavior controlled through a variable, they could put appropriate Emacs Lisp code into an "init file", a file loaded by default on startup, like this:

```
(setq shift-select-mode nil)
```

Expecting the user to use Emacs Lisp for customization creates a high barrier. As a reaction, Per Abrahamsen in 1996 contributed two libraries called `Custom` and `Widget`, which enabled users to change the values of customization variables via a UI rather than through Emacs Lisp. Custom first shipped with Emacs 20.1, XEmacs 20.1 and XEmacs 19.15 (which was released after XEmacs 20.0). Figure 1 shows the UI for `shift-select-mode`.

Custom was originally created to customize the Gnus news reader. With the integration into both Emacs and XEmacs, Custom also gained an Emacs Lisp programming interface. The original declaration of `shift-select-mode` via the `defvar` primitive would have looked like this:

```
(defvar shift-select-mode t
  "When non-nil, shifted motion keys activate the mark momentarily.
   ...")
```

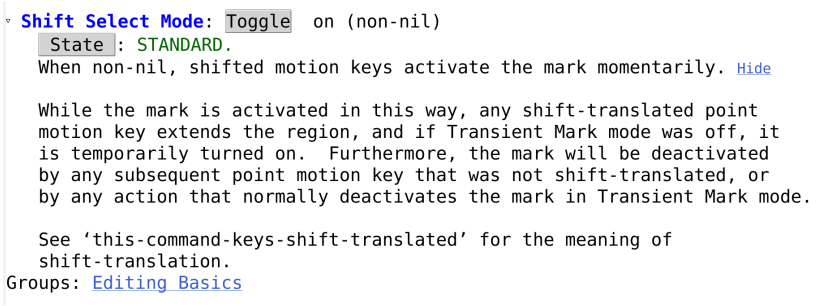

Fig. 1. Custom user interface

The Custom library provides the `defcustom` form, which enables this declaration:

```
(defcustom shift-select-mode t
  "When non-nil, shifted motion keys activate the mark momentarily.
   ..."
  :type 'boolean
  :group 'editing-basics)
```

This declaration instantly enables the UI for customizing `shift-select-mode`. The `:type` declaration as a boolean makes Custom display a `Toggle` button. Emacs Lisp programmers can further gather multiple `defcustom` variables into groups, creating a hierarchy that Custom also turns into a navigation API. The `:type` declaration can concisely describe elaborate structures, as in this example:

```
(defcustom cc-other-file-alist
  '(("\\.cc\\'"  (".hh" ".h")) ...)
  "Alist of extensions to find given the current file's extension.
..."
  :type '(repeat (list regexp (choice (repeat string) function))))
```

From this `:type`, Custom will generate an appropriate UI to manipulate the value knowing that it is a list of elements, each of which is a pair of a regular expression and either a list of strings or a function. This `:type` can also play the role of documentation. Custom is now used pervasively in Emacs Lisp code, and closely ties the programming language to the UI.

7.3 Unicode

As Unicode [The Unicode Consortium 2011] became universally adopted, Emacs and XEmacs both supported the standard. Emacs 21.1 added support for the `utf-8` coding-system, by adding it as another "national" character set. This meant that an "é" coming from a `utf-8` file was considered a different character from an "é" coming from a `latin-9` file. This sort of distinction already existed before. For example between `latin-1` and `latin-9` (and many other character sets), but during the transition to `utf-8` many more users were using a mix of two coding systems and were hence exposed to this problem. For this reason, in 2001, Emacs 22.1 introduced a limited form of unification between character sets. XEmacs did the same in 2001 with release 21.4.

As Unicode became 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 now its Unicode scalar value.

7.4 Bignums

Somewhat surprisingly for Lisp, Emacs Lisp 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 Emacs Lisp (Section 5.4). As a result, various functions dealing with numbers beyond the fixnum range had to implement workarounds. Notable are `file-attributes` and `current-time`. The former 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. The latter 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 files that can be edited.

Moreover, Emacs Lisp 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 [GMP 2020]. In Emacs, Gerd Möllmann started work on adding support for bignums via GMP around October 2001, but never finished it. It was 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. Thus, while it is fairly complete, XEmacs's Emacs Lisp 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 as to simplify the system for programmers: 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.

Interestingly, bignums in Emacs were always perceived as desirable but never important enough to overcome the hassle of requiring GMP or some other multiprecision library. The deciding factor which changed the tradeoff was probably the fact that by Emacs 25, most builds of Emacs linked the GNUtls library, which included GMP to perform cryptography (HTTPS support requires GNUtls).

The introduction of bignums raises some design issues in Emacs Lisp, as previously integers were always unboxed. This meant that the fast `eq` behaved differently from `eql` only on floating point numbers. As a result, some Emacs Lisp code assumes that if two integers represent the same number, `eq` will 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. After a long discussion that did not reach a consensus, Emacs's maintainers followed XEmacs's design, because the cost of making `eq` behave like `eql` was considered to be too high, and also because it's a decision that can easily be changed later without any significant risks of introducing bugs, whereas the reverse is not true.

7.5 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 analogy by adding the notion of *terminal-local* variables, which are variables that take different values depending on the X11 server (the *terminal*) the current frame belongs to. 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 Emacs Lisp programs to create others.

XEmacs chose a different route: Starting in 1995, Ben Wing (working from a prototype by Chuck Thompson) implemented *specifiers*, objects that manage properties that depend on a generalized notion of *display context* [Wing et al. 1998]. The first prototype implementation was released with XEmacs 19.12. A specifier's value (its *instance*) depends on its *locale*, which can be a buffer, a window, a frame, a device, or a MULE character set, or certain properties of these.

Emacs 20 in 1998 added the ability to set a 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, 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 it was then 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. Buffer-local variables, however, are used extensively in Emacs Lisp and replacing them with explicit accesses to fields or properties of buffer objects would make Emacs Lisp code heavier. In contrast, 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 PERIOD

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 Emacs Lisp evolve, whereas XEmacs started to lose momentum starting about 2010.

This section discusses some notable evolution of the design of Emacs Lisp after 2010 which have not found their way into XEmacs so far.

8.1 Lexical Scoping

When Richard Stallman started working on Emacs Lisp, lexical scoping was becoming the established standard in the Lisp family in both Common Lisp and Scheme. So of course, the question of adding lexical scoping to Emacs Lisp 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` Dave Gillespie introduced in 1993. The `lexical-let` macro 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. Moreover, 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 used only 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-code compiler. Emacs 23.1 introduced the curry operator `apply-partially` to cover similar use cases without those drawbacks. In all cases, the programmer had to manually specify which variables to capture from the environment, with little help from the tools to detect errors in this respect.
- *The global visibility of variable names,* requiring more care with the choice of local names. The convention to name all global variables with a package-specific prefix works well to avoid name conflicts: it not only avoids conflicts between global variables, but also between local variables and global variables since local variables do not have such a prefix. The only remaining possible conflicts occur between local variables of different functions. In Emacs Lisp these tend to happen only in the presence of higher-order functions. For example in code such as:

  ```
  (let ((lst (some-list)))
    (cl-every (lambda (x) (memq x lst)) elements))
  ```

  where a variable capture occurs if `cl-every` happens to bind a local variable `lst` before calling its first argument. Name conflicts were also problematic in one other special case, the byte-code compiler: in order to emit warnings about the use of undeclared variables, the byte-code 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-code compiler itself. So some code in the byte-code compiler 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 Emacs Lisp was that it should be the scoping used by default by all binding constructs, with an easy way to request dynamic scoping for some variables, as is the case in Common Lisp. But at the same time, there was a non-negotiable need to preserve compatibility with existing Emacs Lisp code, although some limited breakage for rare situations could be tolerated.

The vast majority of existing Emacs Lisp code was (and still is) agnostic to the kind of scoping used, in the sense that either dynamic or lexical scoping of local variables gives the same result in almost all circumstances. This was true of early Emacs Lisp code and has become even more true over time as the byte-code compiler started to warn about references to undeclared variables. Warnings about unused variables would have probably pushed even more Emacs Lisp code to be agnostic. But in any case, it seemed clear that despite the above, the majority of Emacs Lisp packages relied somewhere on dynamic scoping. So while there was hope to be able to switch

Emacs Lisp 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 [Neubauer and Sperber 2001]. This tool could have been used to mechanically convert Emacs Lisp packages to a lexically scoped version of Emacs Lisp, while preserving the semantics. The plan with this approach was to facilitate moving Emacs Lisp code to Scheme eventually, but the overall project was too large to ever be realized.

In late 2001, Miles Bader started working on a branch of Emacs he called *lexbind* with support for lexical scoping, which was finally included in Emacs 24.1. The solution he adopted was to have two languages: an Emacs Lisp with dynamic scope for backward compatibility, and another with scoping rules like those of Common Lisp: lexical scope by default except for those variables that have been declared as dynamically scoped (in practice, these are all the global variables). Each file is tagged to indicate which language is to be used, defaulting to the backward compatible mode. In turn, each function value is tagged with the language it is using, so functions using dynamic scope can seamlessly call functions with lexical scope and vice versa. This way, old code keeps working exactly as before and any new code that wants to benefit from lexical scoping simply has to add the corresponding "`-*- lexical-binding:t -*-`" annotation at the beginning of the file.

The two languages were sufficiently similar that the new lexically scoped variant required only minor changes to the existing interpreter. But the changes needed to support this new language in the byte-code 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 modifications to the byte-code compiler were never completed.

Finally in 2010 Igor Kuzmin worked on a summer project under the direction of Stefan Monnier, in which he tried to add lexical scoping to the byte-code compiler differently: instead of directly adding support for lexical scoping and closures to the single-pass byte-code compiler code (which required significant changes to the code and was made more complex by the need to fit into a single pass), he implemented a separate pass (itself split into two passes) to perform a traditional closure conversion as a pre-processing step. This freed the closure conversion from the constraints imposed by the design of the single-pass byte-code compiler, making it much easier to implement, and it also significantly reduced the amount of changes needed in the byte-code 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 Emacs Lisp packages;
- make sure performance of existing code was not measurably 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 used its stack only 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 Emacs Lisp 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.

The introduction of `lexical-binding` in Emacs 24.1 went very smoothly, causing only few backward incompatible changes for users, in part because very few of Emacs's own files made use of it. Converting existing code to use the new language is usually easy (consisting mainly in adding a few variable declarations, with the help of warnings from the byte-code compiler) but is not automated and can sometimes require a non-trivial effort for packages which make heavy or creative use of dynamic scoping. Nowadays, most new packages choose to use the new lexically scoped language, and about half the well-maintained packages have converted to it. However, there is still a large amount of code using the old language, either because they still want to support Emacsen older than Emacs 24.1, or because of the effort needed to convert. For example, as of Emacs 26, only a third of Emacs's own Lisp code has been converted to use lexical scoping.

8.2 Eager Macro-Expansion

The exact time at which a macro is expanded has never been clearly specified in Emacs Lisp. Until Emacs 24, macro-expansion mostly took place as late as possible for interpreted code, whereas for byte-compiled code, macro-expansion always took place during byte-code 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-code compilation, using a new `macroexpand-all` function. This caused some visible differences in corner cases where some macro invocations were expanded which earlier had been eliminated by optimizations before getting to the point of macro-expansion. 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. This means 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 [Pitman 2005], so as to improve the handling of macros that expand to both definitions and uses of those definitions.

8.3 Pattern Matching

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 Emacs Lisp, or the set of accepted patterns was too limited and not easily extensible.

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

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 Monnier was made aware of Racket's `match` construct [Flatt and PLT 2018], which had 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 Emacs Lisp because it relies too much on Racket's efficient handling of locally defined functions and tail calls, 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.

8.4 CL-Lib

While the core of Emacs Lisp has evolved very slowly over the years, the evolution of other Lisps (mostly Scheme and Common Lisp) progressed, putting pressure on Emacs Lisp to try and add various extensions to the language. As it turns out, Emacs Lisp, to a first approximation, can be seen as a subset of Common Lisp, so by 1986 Cesar Quiroz had already written a `cl.el` package that provided various Common Lisp facilities implemented as macros. Emacs 18.51 was the first Emacs release to ship with `cl.el`, in 1988. It was superceded 5 years later for Emacs 19.18 by a new version contributed by Dave Gillespie with a more detailed emulation of Common Lisp as well as a few extensions.

Richard Stallman never wanted Emacs Lisp to morph into Common Lisp, but he saw the value of offering such facilities, so `cl.el` was included with Emacs fairly early, and has been one of the most popular packages, used by a large proportion of Emacs Lisp packages. Yet, Stallman did not want to impose `cl.el` onto any Emacs user. Consequently, he imposed a policy to restrict the use of `cl.el` *within* Emacs itself: Emacs Lisp packages bundled with Emacs could only use `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 within Emacs were macros and inlined functions.

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

(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 Emacs Lisp into a reasonably complete implementation of Common Lisp.
(2) Some aspects of Common Lisp's design can incur significant overhead, and Emacs already carried the stigma of being too big (`vi` advocates used the derogatory epithet *eight megabytes and constantly swapping* in the days when eight megabytes were a significant amount of memory). Consequently, there were good reasons to avoid making Emacs Lisp's efficiency any worse.

(3) Many aspects of Common Lisp design were decided by majority not consensus, as evidenced by the divide between Common Lisp and Scheme. Stallman disliked several aspects of Common Lisp's design, such as the use of keyword arguments, especially in low-level primitives like `mapcar` [Stallman 2012a].
(4) Keeping Emacs Lisp 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, Stallman would often point out the cost in terms of the need for more, and more complex, documentation [Stallman 2012b].
(5) The implementation of `cl.el` was fairly invasive, redefining some core Emacs Lisp functions.
(6) Finally, turning Emacs Lisp 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 fourth 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 clean and popular to justify moving them into Emacs Lisp proper:

1997 The release of Emacs 20.1 saw the move of the macros `when` and `unless` as well as the functions `caar`, `cadr`, `cdar`, and `cddr`.
2001 Then maintainer Gerd Möllmann had a more favorable opinion of Common Lisp. As a result, Emacs 21.1 included the hash-table functions, reimplemented in C and extended. It also adopted the Common Lisp concept of *keyword symbols*, to ease the work of `cl.el` as well as for the benefit of other packages making use of them. A keyword symbol is a symbol starting with a colon; its literals are "self-quoting"—the expression `:foo` evaluates to `:foo`. This makes keyword symbols notationally convenient for specifying keyword arguments. Additionally, the macros `dolist`, `dotimes`, `push`, and `pop` were also added to Emacs Lisp, although they introduced some difficulties: in `cl.el` those macros included extra functionality, which relied on parts of `cl.el` which were not needed in core Emacs Lisp, specifically `block`/`return` and generalized variables. For that reason the macros added to Emacs Lisp did not actually replace those of `cl.el`; instead when `cl.el` was loaded, it overrode the original macros with its own version.
2007 Emacs 22.1 added `delete-dups`, which provided a subset of `cl.el`'s `delete-duplicates`.
2012 Emacs 24.1 added `macroexpand-all` and lexical scoping, which obsoleted the `lexical-let` form from `cl.el`.
2013 Emacs 24.3 added compiler macros, `setf`, and generalized variables.
2018 To the `cXXr` functions incorporated in Emacs 20.1, Emacs 26.1 added the remaining `cXXXr` functions. The resistance against those was mostly one of style, since they tend to lead to code that is difficult to read.

The details here are not important, but rather the fact that there has been a regular trickle of features seeping from `cl.el` into Emacs Lisp, and that in many if not most cases this took place not by just moving code from `cl.el` but by reimplementing it, often with slightly different semantics.

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. Richard Stallman still opposed it, but this time, a compromise was found: replace the `cl.el` package with a new package `cl-lib.el` that provides the same facilities, but

with names that all use the `cl-` prefix. This way, the `cl-lib.el` package does not turn Emacs Lisp into Common Lisp, but instead provides Common Lisp facilities under its own namespace, leaving Emacs Lisp free to evolve in its own way [Stallman 2012c].

The implementation of `cl-lib` mostly just took `cl.el` and added the `cl-` prefix to its definitions while at the same time a new `cl.el` was implemented which is just a very thin wrapper re-exporting the `cl-lib` definitions under their old names. This thin wrapper is sufficiently simple that it does not incur much cost either in terms of maintenance or performance. Some details of `cl.el`'s implementation were also reworked to be less invasive. Prominently, `cl.el` had redefined Emacs's macro-expansion wholesale with its own implementation, which incorporated support for compiler macros, `lexical-let`, `flet`, and `symbol-macrolet`. The standard macro-expansion code was reworked so that `cl-lib.el` could provide those features more cleanly.

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`.

Nowadays, both `cl-lib.el` and `cl.el` are bundled with Emacs and supported, but `cl.el` is not used by Emacs's own Lisp code any more, and it is expected to be declared obsolete in Emacs 27. But it has been very popular for many years, so it will take a long time before we can really retire it.

8.5 Generalized Variables

To facilitate the move to `cl-lib.el`, some frequently used functionality from `cl.el` was moved directly to Emacs Lisp. 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 both as an expression and as an updateable reference. The concept comes from Common Lisp, and the Emacs implementation was originally a part of `cl.el`. In both Common Lisp and Emacs, a number of special forms take generalized variables as operands. For example, `(setf` *PLACE VALUE*`)` treats a generalized variable as a reference and sets its value.

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

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

VARS and *VALS* together form a list of bindings that perform the computation needed to reach the place and should be done only once (for reasons of performance or side-effects); *ACCESS-FORM* reads the current value of the place; and *STORE-VAR* and *STORE-FORM* specify how to *set* the place (by binding *STORE-VAR* to the desired value and then executing *STORE-FORM*).

For example, the macro `(push` *EXP PLACE*`)` which adds *EXP* to the head of the list stored in *PLACE*, could be defined to expand to:

```
(let ((v EXP))
  (setf PLACE (cons v PLACE)))
```

But that would duplicate *PLACE* which could perform a costly operation and have side-effects. So the macro instead uses `get-setf-expansion` in order to expand to code of the following form:

```
(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 Emacs Lisp, a fresh new implementation of the concept was used. The reasons for this new implementation were:

- The implementor of `cl-lib` (Stefan Monnier) found the existing code hard to follow, arguably a case of "not-invented-here syndrome" on Stefan's part.
- The previous code made use of internal helper functions from `cl.el`, which the maintainers did not want to move to core Emacs Lisp, so some significant massaging was needed anyway.
- Stefan Monnier considered this part of Common Lisp's design ugly.

So the reimplementation uses a different design: instead of a five element list, the new function `gv-get-place-function` turns a *place* into a single higher-order function. This higher-order function takes as its sole argument a function of two arguments (the *ACCESS-FORM* and the *STORE-FUNCTION*) which should return the code that we want to perform on the place. For example, the `push` macro could be 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))))))
```

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

A Common-Lisp-style representation of a place can easily be turned into its 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` provided compatibility with Common Lisp's `defstruct` early on (Section 4.5), 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 was chosen, because of the comic reference to the nursery rhyme. EIEIO started as an experiment to try to use an object system in Emacs, first following a model like that of C++, but very soon switching to a CLOS-inspired model.

8.6.1 CLOS. EIEIO is an implementation of a subset of CLOS, the Common Lisp Object System [DeMichiel and Gabriel 1987]. CLOS is a somewhat unusual object system where methods are not attached to objects or classes; instead, it provides the notion of *generic function* which are functions implemented by a set of methods where each method indicates when it is applicable by annotating its arguments with a *specializer* which is usually a type indicating that this method is

applicable only if this argument belongs to this type. This type-based dispatch is not limited to the first argument but can be applied to any number of arguments, a feature called *multiple-dispatch*. Furthermore, when several methods can be applied, the programmer can control how the methods are *combined*, for example by adding *qualifiers* (like `:after`, or `:before`). Beside generic functions, CLOS also provides the `defclass` macro to define new types by listing their parents, fields, and various other properties.

Here is an example of CLOS code which defines a new `point3d` subclass of a pre-existing `point2d` class, and adds a new method to the generic function `point-distance`, but only when both arguments are subtypes of `point3d`:

```
(defclass point3d (point2d)
  (z :documentation "Third dimension"))

(defmethod point-distance ((p1 point3d) (p2 point3d))
  (let ((distance2d (call-next-method))
        (distancez (- (slot-value p1 'z) (slot-value p2 'z))))
    (sqrt (+ (* distance2d distance2d)
             (* distancez distancez)))))
```

8.6.2 EIEIO. Just like Emacs Lisp, the development of EIEIO 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, a package providing IDE-like features [Ludlam 2018]. EIEIO included support for most of CLOS's `defclass`, as well as support for a subset of `defmethod`, but was limited to single-dispatch methods, dispatching on the first argument. Moreover, it could dispatch only based on types of `defclass` objects. It also had incomplete support for method combinations, allowing only `: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 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".

8.6.3 CL-Generic. 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 mostly to add a "`cl-`" prefix as was done for `cl-lib` (rather than an "`eieio-`" prefix, perceived as too verbose to be popular), as well as improve `defmethod` with support for `:around` methods and dispatch on other types than those defined with `defclass`. But 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 global variables to preserve state in a way that was not only brittle and somewhat inefficient but made it hard to extend with `:around` methods.

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 has not been excessively optimized, but is 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 only globally by adding appropriate methods to `cl-generic-`

`combine-methods`. This was trivial to implement but is basically unusable, as evidenced by the fact that there is no known user of this feature at this time, not even internally.

(2) The set of supported specializers is not hard-coded. In CLOS, a specializer can be either a type (indicating that this method applies when the argument is of this type) or of the form `(eql` *VAL*`)`, indicating that the method applies only when the argument is equal to *VAL*. Emacs Lisp extends this so new specializers can be defined in a modular way via the notion of *generalizer* inspired by a paper by Rhodes et al. [Rhodes et al. 2014]. 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.

The second difference was made necessary because methods need to dispatch on EIEIO objects even though `cl-generic.el` could not depend on EIEIO. There were additional motivations for it, though: not only was it 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.

Some existing Emacs Lisp functions appeared like good candidates to use that machinery to split them into independent methods but required dispatching on contextual information (that is, 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 means that this method is applicable when *EXP* evaluates to a value that satisfies the *SPECIALIZER* constraint. This is used for methods which are applicable only in specific contexts, such as in specific major modes or in frames using a particular kind of GUI.

8.6.4 Overall Support for Classes. 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 Emacs Lisp 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 browse it. This started as an attempt to adapt to `cl-generic.el` the EIEIO facilities to interactively explore EIEIO objects and methods, but is more modular and better integrated with the rest of Emacs's on-line help system.

As of Emacs 26, object support in Emacs Lisp is hence split into four parts: the old `cl-defstruct` provided by `cl-lib.el` which allows defining new object types and supports single inheritance; `defclass` provided by EIEIO which offers similar functionality plus multiple inheritance and a few other benefits but at the cost of slower object creation and field accesses; `cl-defmethod` provided by `cl-generic.el` which allows defining methods and supports `cl-defstruct` objects and `defclass` objects equally; and finally `defmethod`, which has been reimplemented as a wrapper on top of the new `cl-defmethod` (this backward compatibility library is deprecated and a bit less efficient than using `cl-defmethod` directly but still more efficient than its earlier implementation, and it is fully implemented using documented features of `cl-defmethod`, so it does not impose any performance or maintenance issue). Emacs will likely live with both `cl-defstruct` and `defclass` for the foreseeable future.

8.7 Actual Objects

While Emacs 25's `cl-generic.el` introduced object-oriented programming facilities into Emacs Lisp, 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 records via the `make-record` primitive and a corresponding new object type (Section 4.5). 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 Emacs Lisp was lacking in abstraction, so in 2015, Daniel Colascione developed `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 on the use of lexical scoping, to work around the fact that Emacs Lisp does not directly provide something like `call/cc` to access underlying continuations. It handles only a (relatively large) subset of Emacs Lisp, because CPS conversion of forms like `unwind-protect` [Haynes and Friedman 1987] cannot be defined in general in Emacs Lisp.

8.9 Concurrency

Emacs Lisp is a fundamentally sequential language, and relies 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. Concurrency appeared very early on: since Emacs 16.56, Emacs has included support for asynchronous processes, that is, the execution of separate programs whose output was processed by so-called *process filters* whenever the Emacs Lisp execution engine is idly waiting for the next user input.

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 (timers had earlier been 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 Emacs Lisp is problematic because of the pervasive reliance on shared state in existing Emacs Lisp code. In some cases, shared state is not a problem and concurrency can be mimicked via asynchronous programming: When a program waits for an operation (such as an external program), instead of blocking, it registers a continuation callback and returns control to the main event loop. Yet many Emacs Lisp packages instead block, because slicing the execution via callbacks means effectively writing code in continuation-passing style which is poorly supported in Emacs Lisp, interacts badly with dynamic scoping, and requires significant surgery to retro-fit to existing code.

So, shared-memory concurrency was largely considered as inapplicable to Emacs Lisp. Nevertheless, in November 2008, Giuseppe Scrivano posted a first naive attempt at adding threads to Emacs Lisp. This effort did not go much further, but it inspired Tom Tromey to try his own luck.

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 most approaches were abandoned simply because it was too difficult to keep them up-to-date with the evolving Emacs codebase.

A working approach was finally released in 2018, as part of Emacs 26.1. Context switches still take place only at a few known points where Emacs Lisp 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 would have required more extensive and delicate changes to existing code. Therefore, while this approach may be reconsidered in the future, the current implementation favored a simpler and safer approach.

The inclusion of such a form of shared-state concurrency was hotly debated between the maintainers. They all agreed that Emacs Lisp needs to develop concurrency and parallelism in order to take advantage of the increasing number of CPU cores available, especially since single-core performance is not increasing significantly any more; but there was also a consensus that shared memory is a very bad fit to the current Emacs Lisp world. Tom Tromey's patch was finally accepted only because it was non-invasive, and because there was a feeling that it was important to do *something*.

This is still a fairly experimental feature, and two years after its appearance, its use appears to still be limited to experimental patches to a handful of packages such as the Gnus MUA. Arguably the main outcome so far has been to expose some latent bugs in some packages's asynchronous processing.

Over the years, other approaches to concurrency and parallelism have been developed as Emacs Lisp packages, most notably the `async.el` package [Wiegley 2019] developed in 2012 that runs Emacs Lisp code in parallel in a separate Emacs subprocess. Its applicability is limited by the fact that the buffers's contents need to be explicitly sent as needed between the two processes, forcing a very coarse grain of parallelism. Moreover, there is no guarantee that the configuration of the subprocess is consistent with that of the main process—the subprocess may even be another version of Emacs in some cases. Still, several third party packages make limited use of `async.el`.

8.10 Inline Functions

Function calls are fairly expensive in Emacs Lisp, 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 to tell the byte-code compiler it should inline 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 way of inlining in the form of the `defsubst*` macro, which from the outside is almost identical to `defsubst`, except for including support for Common Lisp extensions like keyword arguments, but its implementation tries to generate more efficient code by substitution with the actual arguments (without quite preserving the usual Emacs Lisp semantics of function calls).

```
(define-inline cl-typep (val type)
  (inline-letevals (val)
    (pcase (inline-const-val type)
      (`(not ,ty)         (inline-quote (not (cl-typep ,val ',ty))))
      (`(eql ,v)          (inline-quote (eql ,val ',v)))
      (`(satisfies ,pred) (inline-quote (funcall #',pred ,val)))
      ((and (pred symbolp) ty (guard (get ty 'cl-deftype-satisfies)))
       (inline-quote (funcall #',(get ty 'cl-deftype-satisfies) ,val)))
      ...
      (ty (error "Bad type spec: %s" ty)))))
```

Fig. 2. Example use of `define-inline`

A third way to implement an "inlinable function" in Emacs Lisp is by defining it as a macro, although this comes with the drawback that it does not define a proper function.

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 known only at run-time. The compiler macro enables optimizations—for example, it turns `(cl-typep x 'integer)` into `(integerp x)`. The type specification can also take the form `(not` *TYPE*`)` to specify "any type other than *TYPE*", `(eql` *VAL*`)` to specify the type containing a specific value, or `(satisfies` *PRED*`)` to specify the type of values for which a given predicate returns true.

Consequently, Monnier developed the new macro `define-inline`, which was included in Emacs 25.1. It lets the programmer define a normal function and a corresponding optimizing compiler macro with a single form. Figure 2 shows the skeleton of the current version of `cl-typep`, using `define-inline`: it interprets the body as both a function definition by removing the `inline-` prefixes from the various forms in the body, and at the same time defines the compiler macro. The former performs the `pcase` dispatch at run time, the latter at compile time. This inlines the corresponding branch if `type` is statically known and, if not, `inline-const-val` will detect the problem and give up on inlining altogether. Nowadays a bit short of 50 functions in the Emacs distribution are defined using `define-inline`.

8.11 Module System

While Emacs Lisp 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 Emacs Lisp side of Emacs is now a rather large system, so in order to avoid name conflicts, Emacs Lisp relies on a poor man's namespace system, as mentioned in Section 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 `fakespaces` package that allows defining 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 that 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 that 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 that instead tries to provide 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 more thorough job to make it interact correctly with code manipulation tools like the generator of autoload declarations or to make it possible for the source-level debugger to instrument a single declaration rather than the whole namespace.
- In 2015, the same Artur Malabarba developed the `Nameless` package which takes a completely different approach: it does not provide any new Emacs Lisp 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, was around 2013, following a blog post by Nic Ferrier [Ferrier 2013]. No consensus emerged, but other than inertia one of the main arguments in favor of the status quo was that Emacs Lisp's poor man's namespacing is only a mild annoyance and in return it makes life for the uninitiated and for cross-referencing easier [Guerry and Ingebrigtsen 2013]: 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 CONCLUSION

Emacs Lisp started out primarily driven by the demands of the application at hand: Richard Stallman was interested in developing a programmable editor, not primarily in language design. His choice of Lisp, however, was not only motivated by immediate requirements, but rather framed by Stallman's environment and experience at the time. As a result, the original Emacs Lisp was the simplest it could be while still avoiding the pitfalls of TECO and getting most of the benefits that Multics Emacs enjoyed from Maclisp.

Being a Lisp has meant that the core language did not have to change significantly since its original inception. What would have to be a language addition in many other languages is often just a library in Emacs Lisp. For many years, the things that were added were some conveniences or driven by new editor features (notably the graphical UI). Some differences that evolved between the Emacs and XEmacs dialects were driven by different convictions about programming-language design, most notably the use of opaque datatypes.

Over time, starting around 2000 with Gerd Möllmann's maintainership and then even more so when Stefan Monnier and Chong Yidong took over maintainership in 2008, language design received more attention as a goal in itself. The language designer's perspective has driven changes such as the introduction of lexical scope, features from Common Lisp, and `pcase`.

The core language of Emacs Lisp is still present, essentially unchanged from its original form, as is its implementation: Any substantial change would have invalidated a substantial code base. Moreover, the implementation has generally been good enough to fulfill the requirements of the Emacs editor for over 30 years. Consequently, Emacs Lisp is showing no signs of going away any time soon: we are looking forward to reporting about its next 30 years of evolution.

ACKNOWLEDGMENTS

Emacs and Emacs Lisp are the result of the contributions of an impressive number of individuals. We thank them all for their contributions. 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. Moreover, we thank Joseph Arceneaux for the interview in Appendix B, as well as Richard Gabriel, Richard Stallman, and Jamie Zawinski for patiently answering our questions.

This work was partially supported by the Natural Sciences and Engineering Research Council of Canada grant N^o 298311/2012 and RGPIN-2018-06225. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author and do not necessarily reflect the views of the NSERC.

A ALTERNATIVE IMPLEMENTATIONS

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

A.1 Edwin

Edwin is the editor that ships with MIT Scheme [MIT 2014]. 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 Emacs Lisp interpreter in Scheme that was able to run substantial Emacs Lisp packages [Birkholz 1993] at the time, among them the Gnus news reader.

A.2 Librep

In 1993, John Harper started working on an embeddable implementation of Emacs Lisp 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 Emacs Lisp, it has since significantly diverged, including a module system, lexical scoping, tail-call elimination, and first-class continuations.

A.3 Elisp in Common Lisp

In 1999, Sam Steingold also implemented the Emacs Lisp language as a Common Lisp package [Steingold 1999]. He was motivated by the hope of moving Emacs to work with a Common Lisp language instead of Emacs Lisp, as well as the desire to reuse some Emacs Lisp code elsewhere, most notably from Emacs's Calendar package. 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" Emacs Lisp 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.

A.4 JEmacs

JEmacs [Bothner 2002] is an editor that ships with Kawa Scheme [Bothner 2020]. JEmacs comes with support for running some Emacs Lisp code. Its implementation (written partly in Java and partly in Scheme) works by translating Emacs Lisp code to Scheme, and running the result.

A.5 Guile

Guile Scheme [FSF 2020a] was conceived as the universal extension language of the GNU project, with the specific intention of replacing Emacs Lisp in Emacs at one point [Stallman 1994]. This has not happened (yet), but Guile does ship with a fairly complete implementation of Emacs Lisp that translates Emacs Lisp 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 Emacs Lisp engine is provided by Guile.

A.6 Emacs-Ejit

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

B INTERVIEW WITH JOSEPH ARCENEAUX

Joseph Arceneaux is an American software architect. In the early 1990, Arcenaux was the maintainer of GNU Emacs, working first for the Free Software Foundation, then for Lucid, Inc. Lucid Inc. wanted to base its Energize development environment for C++ on Emacs (Section 2.3). During Arcenaux's tenure, the commercial interests of Lucid Inc. and those of Emacs's original author, Richard Stallman, clashed, which finally resulted in Lucid creating its own fork of Emacs, Lucid Emacs.

Arcenaux graciously agreed to be interviewed by Michael Sperber on May 30, 2019. The purpose of the interview was to provide background material for the present article, specifically what role Emacs Lisp played in the rift.

> *Could you tell me how you got into software, tracing a line to how you got involved with Emacs.*

When I was 15 or 16, I was planning to become a veterinarian. I got out of high school early because I took a test, and then I took a course in computer science and got really absorbed. I went to what is now the University of Louisiana, then the University of Southwest Louisiana, and I had a Finnish professor who used Emacs and who convinced me to use it. By the time I got out of graduate school, I was not just using Emacs but contributing code to the project, and I became interested philosophically in the Free Software movement.

After I graduated I went to work doing research for the INRIA in France. Since I was a researcher, I had a fair amount of spare time doing Emacs development, and I got to know Richard Stallman pretty well. I had him come over and give a couple of talks at INRIA. This was the first time Stallman rode a motorcycle when I took him to give a talk. He was scared to death.

We formed this professional relationship, and I continued to contribute code to Emacs, and then my research group started a startup company. It was an X terminal on steroids. This was about the time X Windows was really becoming popular. We had a our own hardware design, and our main

windows guy ported X Windows to it. I was the systems guy, so I wrote the BIOS, and we adopted a new microkernel operating system, similar to Mach, called Chorus. At some point, when our company grew, it became more and more political, and at the same time, Stallman, whom we all called RMS, because that was his login name, invited me to work for the Free Software Foundation, and I thought this would be a great career move, plus I was having a lot of fun writing code for Emacs. Emacs at that point had been mostly written by Stallman and contributors like myself, with the main structure written by Stallman. I found that the structure of the code was such that it was mostly easy to understand but especially easy to make extensions to. Occasionally I would submit something and RMS would say no, but most of time, he accepted my stuff. There were some clever things he did that required pretty deep insight into how computers worked, but apart from two or three of those examples, it was really easy to write new code.

At the FSF, I spent a couple of years having a really good time at the MIT AI lab. RMS and I would have occasional arguments about how to do something in Emacs. And RMS is a really smart guy. He was a MacArthur genius award grantee. Weird things would happen, like I'd get this phone call, and it would be somebody from Norway inquiring about chaos maths. It turned out that RMS was this expert in chaos maths. So when we had arguments, he won a lot of arguments. But I'd say that only part of this was because he was a really smart guy. Part of this was because he was very passionate. I remember going down the stairwell, with him behind me screaming, and then me going back up chasing him up the stairwell with him covering his ears, so he couldn't hear my argument. So he's kind of an interesting guy. I did win some arguments, though.

One of the big ones that came up was the window system. So as I said X Windows was becoming really popular. The X Windows development group was on the 2nd floor of our building. It seemed natural to me that we needed Emacs to become a full-blown X-Windows-integrated system. RMS was adamantly opposed to this. Not for a philosophical reason, but he thought there were more important things to do. A version of Emacs that had been integrated with X10 worked pretty well, but once the move to X11 was made, that didn't work anymore, and nobody had bothered to port it (and, as I recall, the two systems were rather incompatible).

There were some interesting technical questions, such as: At the time, it wasn't computationally advisable to have characteristics associated with every character in the buffer. So a guy named Dan LaLiberte came up with this system that he called "intervals". Basically the idea there was that you would have this shadow structure to the contents of a buffer. And it was essentially: From here to there the face that we're using is this, and from there to over there is this other face. A "face" was a structure that encompassed color, font, background, and all those characteristics. So I took that stuff that he'd written and prototyped, and I fleshed some stuff out and I added some other features like caching, where we were in the buffer, etc. I was doing this in my spare time while I was maintainer of Emacs, and integrating contributions. I started to push this as a major feature development for Emacs. RMS continued to be adamantly opposed to this. So I wasn't sure what to do, and I had a lot of discussions with the board members of the FSF. At some conference I met a French guy from Lucid, Matthieu. And partially because I spoke French, we got along. Eventually Lucid offered me a contract to do this integration with X11 because they had a project to do an IDE for C and C++, and wanted to use Emacs as the primary interface. RMS and I had a big fight about it because he didn't want me to do it. My argument was "Hey, these guys are going to basically pay my salary for a while, and I get this done, and it won't cost the FSF anything." And the board members supported me on this, but RMS didn't want to do it. So I went on to do that, and RMS didn't talk to me for a year or so. Also, I moved to California after I got the contract at Lucid.

My contract with Lucid was in three parts. The first one was the most challenging, because it had to conform to certain specs, performance-wise. So if we had all of this visual annotation information, and you did a search for some string, the requirement was that that search should

not be any slower than on any version of Emacs without visual annotation. So I passed that part. I forget what the other two parts were, but after the first phase of my contract, Lucid changed the direction they wanted to go on, and tried to hire me as a full-time employee.

Do you remember why Lucid changed direction, or why their requirements changed?

I don't remember exactly. I feel that some of that was political. But I don't remember the details. I just remember that we had arguments about which way to go. By then I was back in contact with RMS and I started checking in my code to the main repository of Emacs. And RMS was not always happy with this. So there was a conflict with this between Lucid and RMS about where we should go, and I was trying to mediate that difference.

But you were working mainly on a fork of Emacs at Lucid, or were you always checking things into the Emacs mainline?

No, it was a fork. And that was probably a big mistake. If Git had been around back then, it wouldn't have been that big a deal. The extent of changes was such that there would still have been conflicts. So we had a fork that was going to be version 19 of Emacs. Had I just continually incrementally checked in my changes, I think that would have avoided a lot of political issues that came down the road.

I'm not sure that was an official position, but were you still been maintainer of Emacs, or had you been and then went back to being that?

No. Eventually, the FSF hired a guy named Jim Blandy as the official maintainer. But again, this bifurcation became a big political issue, and I was struggling to mediate between these two somewhat contradictory sources of requests. If I had just continued in my role as maintainer and incrementally checked stuff into the Emacs 19 fork, I think it would have avoided all of those issues.

You said that for a long time RMS was opposed to the X11 windowing stuff that ended up in Emacs 19. But eventually Emacs 19 came out with it. So was that something that was forced onto the project or did RMS come around and agree that it was a good idea?

As I recall, I think it was mostly that I convinced Jim Blandy that this was really what we should do. He and I worked on integrating all of my changes into the Emacs 19 fork. I think together we overcame RMS's opposition. This was 1992, I went to a conference to the Isle of Jersey with RMS. He basically had done an extensive review of my code. We talked about that, and when I saw that he had spent so much time reviewing my code, I knew that he had been sold on the idea.

So the various actors involved didn't just pull you in different political directions but also technical directions. But eventually things came either to a head or an end, right?

Yes. Lucid, when I declined their offer to hire me, essentially took the position that they were going to create their own version of Emacs. They assigned a guy to be in charge of that.

Probably Jamie Zawinski …

I think so. So he went off maintaining that, and they publicized it, and it became known as Lucid Emacs, and later XEmacs. And meanwhile, there was some guy from Carnegie Mellon. He was also working on an X11 integration. So for a while there were at least three versions of Emacs that were integrated with X …

There was a windowed version of Emacs called Epoch at some point.

Yeah, that's it.

After you declined Lucid's offer to hire you, did you continue working for them on that contract basis, or was it done by then?

I continued working on that for a little while, but at some point, especially after I went to that conference and talked to RMS, I went back to mostly maintaining Emacs, and doing some of the

other things. I was also maintainer of GNU Indent. My transition from Lucid was gradual. They didn't tell me at first that they were forking their own version. I was working out of my house, rather than the offices of Lucid.

> *I'm still trying to understand. I got that part where you worked on the interval implementation, and how that performed to the speed that Lucid needed. Do you remember anything about the technical change of direction that happened at Lucid?*

As best as I can remember, it was about integrating with the compiler to build a complete IDE. So that had specific requirements about how that should look like, how that should be. As I recall, some of those crossed a boundary between the vision I and RMS had for Emacs, and the more commercial direction that Lucid wanted to go into.

> *One thing that, looking back, surprised me was: You mentioned that what Lucid was doing required quite significant performance. They wanted to stick annotations on every single letter of a C program or a C++ program as you were editing it.*

Not just C, but any text.

> *Lucid ended up working on a development environment for C and C++, and a compiler that would actually generate the data that would go into these annotations. What I'm getting at is: You got the performance out of the redisplay and the buffer management. I'm also interested in Emacs Lisp, which was always a slow Lisp, it was never very fast. Was that never seen as a potential issue with Lucid that Emacs Lisp would not be fast enough?*

I remember doing an improvement to the string garbage collection. But I would not say that the Lisp environment really entered the picture due to performance because all of the annotation code was done in C. But over the years, there were several improvements. RMS and I came up with this algorithm for block-based storage management. It basically replaced the Unix `sbrk()` system calls. That greatly improved the performance of Emacs in general. So that was the C world, hence I can't say what the impact was on Lisp, although the C world was the base of the Lisp world, so I'm sure that had some impact.

With respect to Emacs Lisp, I should add, one of the great pleasures was that I got to work with some Lisp machines. The Lisp machines were really great because you could just do Meta-Dot (as in Emacs) and you could see the function definition of the thing that you'd just invoked, in the very source code itself.

Richard Greenblatt, who invented the Lisp machine, had been absent from the AI lab for years (see the book *Hackers*, by Steven Levy for this story), but—just after Steve Job's formation of NeXT, Inc—someone donated one of their machines to us, and this attracted Greenblatt (aka 'rg') to come visit us. We started hanging out, and (apart from performing some amazing technical feats to help me out) he gave me much insight into how Lisp came about. Perhaps due to those conversations I changed my view on the interaction between the C and Lisp world, and started doing more Lisp-y things, like adding infinite minibuffer history, and some of that was all in Lisp.

> *Excellent. Thank you very much!*

REFERENCES

Alan Bawden. 1999. Quasiquotation in Lisp. In *Proceedings of the ACM SIGPLAN Workshop on PEPM Partial Evaluation and Semantics-Based Program Manipulation PEPM '99*, Olivier Danvy (Ed.). San Antonio, Texas, USA (Jan.), 4–12. NON-ARCHIVAL http://people.csail.mit.edu/alan/ftp/quasiquote-v59.ps.gz (retrieved 4 March 2020) https://web.archive.org/web/20170515044951/http://people.csail.mit.edu/alan/ftp/quasiquote-v59.ps.gz

Brian Berliner. 1990. CVS II: Parallelizing software development. In *USENIX Winter 1990 Technical Conference*, Vol. 341. 352. NON-ARCHIVAL https://docs.freebsd.org/44doc/psd/28.cvs/paper.pdf (retrieved 3 March 2020) https://web.archive.org/web/20190128051609/https://docs.freebsd.org/44doc/psd/28.cvs/paper.pdf

Matthew Birkholz. 1993. *Emacs Lisp in Edwin Scheme.* Technical Report A.I. Memo No. TR-1451. Massachusetts Institute of Technology (Sept.). NON-ARCHIVAL ftp://publications.ai.mit.edu/ai-publications/1000-1499/AITR-1451.ps.Z (retrieved 3 March 2020) https://web.archive.org/web/2017*/ftp://publications.ai.mit.edu/ai-publications/1000-1499/AITR-1451.ps.Z

Per Bothner. 2002. JEmacs - The Java/Scheme-based Emacs. *Free Software Magazine* (March). NON-ARCHIVAL http://jemacs.sourceforge.net/JEmacs-FSM.html (retrieved 4 March 2020) https://web.archive.org/web/20190924233555/http://jemacs.sourceforge.net/JEmacs-FSM.html

Per Bothner. 2020. The Kawa Scheme language. NON-ARCHIVAL https://www.gnu.org/software/kawa/index.html (retrieved 4 March 2020) https://web.archive.org/web/20200108175853/https://www.gnu.org/software/kawa/news.html

Robert J. Chassell. 2018. *An Introduction to Programming in Emacs Lisp* (Emacs version 26.1 ed.). Free Software Foundation, Boston, Massachusetts. NON-ARCHIVAL https://www.gnu.org/software/emacs/manual/html_mono/eintr.html https://web.archive.org/web/20181214081848/https://www.gnu.org/software/emacs/manual/html_mono/eintr.html 2018-12-14.

William Clinger. 1985. *The Revised Revised Report on Scheme.* Technical Report AI Memo No. 848. MIT (Aug.). NON-ARCHIVAL https://dspace.mit.edu/handle/1721.1/5600 (retrieved 4 March 2020) https://web.archive.org/web/20150919023621/https://dspace.mit.edu/handle/1721.1/5600

William D. Clinger. 1998. Proper Tail Recursion and Space Efficiency. In *Proceedings of the ACM SIGPLAN 1998 Conference on Programming Language Design and Implementation* (Montreal, Quebec, Canada) (*PLDI '98*). ACM, New York, NY, USA, 174–185. https://doi.org/10.1145/277650.277719

Clojure [n.d.]. The Clojure Programming Language. Web site. NON-ARCHIVAL https://clojure.org/ (retrieved 6 March 2020) https://web.archive.org/web/20200218082557/https://clojure.org/

Marcus Crestani. 2005. *A New Garbage Collector for XEmacs.* Master's thesis. Universität Tübingen. NON-ARCHIVAL http://crestani.de/xemacs/pdf/thesis-newgc.pdf (retrieved 4 March 2020) https://web.archive.org/web/20190921060916/http://crestani.de/xemacs/pdf/thesis-newgc.pdf

Linda G. DeMichiel and Richard P. Gabriel. 1987. The Common Lisp Object System: An Overview. In *European Conference on Object-oriented Programming on ECOOP '87* (Paris, France). Springer-Verlag, Berlin, Heidelberg, 151–170. https://doi.org/10.1007/3-540-47891-4_15 NON-ARCHIVAL https://www.dreamsongs.com/Files/ECOOP.pdf (retrieved 4 March 2020) https://link.springer.com/chapter/10.1007/3-540-47891-4_15

Jake Edge. 2016. Removing support for Emacs unexec from Glibc. Web article. Jan. 2016. NON-ARCHIVAL https://lwn.net/Articles/673724/ (retrieved 6 March 2020) https://web.archive.org/web/20191101083505/https://lwn.net/Articles/673724/ LWN.net.

Nic Ferrier. 2013. Hatching a plan to add namespaces to EmacsLisp. Blog post. June 2013. NON-ARCHIVAL http://nic.ferrier.me.uk/blog/2013_06/adding-namespaces-to-elisp (retrieved 12 March 2020) https://web.archive.org/web/20190921040559/http://nic.ferrier.me.uk/blog/2013_06/adding-namespaces-to-elisp

Matthew Flatt and PLT. 2018. *The Racket Reference* (v.7.0 ed.). PLT (Aug.). NON-ARCHIVAL https://docs.racket-lang.org/reference/index.html (retrieved 3 Sept. 2018) https://web.archive.org/web/20180930065049/http://docs.racket-lang.org/reference/index.html

FSF. 2020a. *GNU Guile 3.0.0 Reference Manual.* Free Software Foundation (Jan.). NON-ARCHIVAL https://www.gnu.org/software/guile/manual/ (retrieved 15 Jan. 2020) https://web.archive.org/web/20200304160710/https://www.gnu.org/software/guile/manual/

FSF. 2020b. *Using the GNU Compiler Collection (GCC).* Free Software Foundation, Chapter Labels as Values. NON-ARCHIVAL https://gcc.gnu.org/onlinedocs/gcc/Labels-as-Values.html (retrieved 6 March 2020) https://web.archive.org/web/20191021235120/https://gcc.gnu.org/onlinedocs/gcc/Labels-as-Values.html

Richard P. Gabriel. 1999. Letter to Chris DiBona and Tim O'Reilly. Web page. NON-ARCHIVAL https://www.dreamsongs.com/DiBona-OReillyLetter.html (retrieved 18 Aug. 2018) https://web.archive.org/web/20190104032051/https://www.dreamsongs.com/DiBona-OReillyLetter.html

Richard P. Gabriel. 2019. Personal communication. Email. May 2019.
Richard Gabriel responded to a series of questions asked via email by Michael Sperber about the role of Emacs in the design of Lucid's Energize IDE. In particular, Gabriel wrote that Emacs was chosen in response to a series of interviews with potential users, who wanted to use their favorite interactive tools. Gabriel also confirmed that Emacs Lisp was not a big factor in the implementation of Energize, which meant that its performance did not matter much for the technical success of the Energize implementation.

Richard P. Gabriel, Nickieben Bourbaki, Matthieu Devin, Patrick Dussud, David N. Gray, and Harlan B. Sexton. 1990. Foundation for a C++ Programming Environment. In *Proceedings of C++ at Work.* (Sept.).
In this paper, the authors describe the architecture of the nascent C++ development environment Lucid had been working on. The architecture was based on a set of lightly coupled separate tools integrated by a kernel. These tools produce annotations that can be displayed and analyzed by a UI.

GMP 2020. The GNU Multiple Precision Arithmetic Library. NON-ARCHIVAL https://gmplib.org/ (retrieved 3 March 2020) https://web.archive.org/web/20200225023116/https://gmplib.org/

James Gosling. 1981. *Unix Emacs.* Carnegie-Mellon University. https://archive.org/details/bitsavers_cmuGosling_4195808

Bernard S. Greenberg. 1996. Multics Emacs: The History, Design and Implementation. April 1996. NON-ARCHIVAL https://multicians.org/mepap.html (retrieved 6 March 2020) https://web.archive.org/web/20200102215120/https://www.multicians.org/mepap.html

Philip Greenspun. 2003. 10th rule of programming. Sept. 2003. NON-ARCHIVAL http://philip.greenspun.com/bboard/q-and-a-fetch-msg?msg_id=000tgU (retrieved 6 March 2020) https://web.archive.org/web/20170919102254/http://philip.greenspun.com/bboard/q-and-a-fetch-msg?msg_id=000tgU

Bastien Guerry and Lars Magne Ingebrigtsen. 2013. Re: lexicons. Aug. 2013. NON-ARCHIVAL https://lists.gnu.org/archive/html/emacs-devel/2013-08/msg00078.html (retrieved 30 Aug. 2019) https://web.archive.org/web/20190918090610/https://lists.gnu.org/archive/html/emacs-devel/2013-08/msg00078.html

Barry Hayes. 1997. Ephemerons: A New Finalization Mechanism. In *Proceedings of the 12th ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications* (Atlanta, Georgia, USA) (*OOPSLA '97*). ACM, New York, NY, USA, 176–183. https://doi.org/10.1145/263700.263733

Christopher T. Haynes and Daniel P. Friedman. 1987. Embedding Continuations in Procedural Objects. *ACM Trans. Program. Lang. Syst.* 9, 4 (Oct.), 582–598. https://doi.org/10.1145/29873.30392 NON-ARCHIVAL ftp://ftp.extreme.indiana.edu/pub/techreports/TR213.pdf

Pertti Kellomäki. 1993. PSD - a Portable Scheme Debugger. *SIGPLAN Lisp Pointers* VI, 1 (Jan.), 15–23. https://doi.org/10.1145/173770.173772

Gregor Kiczales, John Lamping, Anurag Mendhekar, Chris Maeda, Cristina Lopes, Jean-Marc Loingtier, and John Irwin. 1997. Aspect-oriented programming. In *European Conference on Object-Oriented Programming*. 220–242. https://doi.org/10.1145/503209.503260 NON-ARCHIVAL https://www.cs.ubc.ca/~gregor/papers/kiczales-ECOOP1997-AOP.pdf Also NON-ARCHIVAL https://link.springer.com/chapter/10.1007/BFb0053381 https://web.archive.org/web/20200209063553/https://www.cs.ubc.ca/~gregor/papers/kiczales-ECOOP1997-AOP.pdf

Bil Lewis, Dan LaLiberte, Richard Stallman, and GNU Manual Group. 2018. *GNU Emacs Lisp Reference Manual* (3.1 ed.). Free Software Foundation, 51 Franklin St, Fifth Floor, Boston, MA 02110-1301. NON-ARCHIVAL https://www.gnu.org/software/emacs/manual/elisp.html (retrieved 3 March 2020) https://web.archive.org/web/20200122091452/https://www.gnu.org/software/emacs/manual/elisp.html

LibJIT 2020. GNU LibJIT. Project web page. NON-ARCHIVAL https://www.gnu.org/software/libjit/ (retrieved 6 March 2020) https://web.archive.org/web/20191222233941/https://www.gnu.org/software/libjit/

Lightning 2020. GNU Lightning. Project web page. NON-ARCHIVAL https://www.gnu.org/software/lightning/ (retrieved 6 March 2020) https://web.archive.org/web/20191222201436/https://www.gnu.org/software/lightning/

Eric Ludlam. 2018. Personal Communication. Email. 13 Aug. 2018 19:44:23–04:00.

In response to an email message from Stefan Monnier about the early history of EIEIO, he wrote:

> Back around 1995 or 6, I was working at Raytheon and taking a class on using Object request brokers, and the concept looked like something that would be fun to hack in Emacs. To support that however I needed an object system, so I wrote something that looked like c++ because I didn't know much of anything about CLOS. There was also a widget toolkit I built on it as a demo.
>
> Anyway, I got tons of feedback about that, and why not CLOS. I looked into CLOS and it seemed fine so I converted it over to that style of syntax. Small bits of the old C++ variant remained however, such as the array based storage and slot protection, and the :custom slot feature which I had originally intended to be based on the widget set I built on EIEIO.
>
> As for the name, finding a name for an object system is tricky. There was no "eio" predecessor. I think it was always eieio once I had hit upon the name idea. I never did decide what the first "ei" should stand for, I just knew it needed all the Es and Is. If I'd known about CLOS originally, I probably would have named it ECLOS or something boring like that. I still prefer the full EIEIO name though.

John McCarthy. 1960. Recursive Functions of Symbolic Expressions and Their Computation by Machine, Part I. *Commun. ACM* 3, 4 (April), 184–195. https://doi.org/10.1145/367177.367199

MIT. 2014. *MIT/GNU Scheme.* Massachusetts Institute of Technology. NON-ARCHIVAL https://www.gnu.org/software/mit-scheme/documentation/mit-scheme-user/index.html (retrieved 24 Dec. 2014) https://web.archive.org/web/20141224003823/https://www.gnu.org/software/mit-scheme/documentation/mit-scheme-user/index.html Version 9.4.

Richard Mlynarik. 2019. Personal communication. Email. Aug. 2019.

Mlynarik responded to an email from Michael Sperber about the authorship of the union implementation. Mlynarik wrote about the philosophical differences he had with Richard Stallman over the use of type-safe abstractions in Emacs Lisp. He wrote:

> Actually having a real C structure representing the base Lisp object implementation was MASSIVELY helpful in debugging, since GDB started with at least a clue about what was in them.

David Moon et al. 1978. Emacs Lore. Mailing list excerpt. July 1978. NON-ARCHIVAL https://ban.ai/multics/doc/emacs.lore.txt (retrieved 6 March 2020) https://web.archive.org/web/20200306154734/https://ban.ai/multics/doc/emacs.lore.txt

David A. Moon. 1974. *MACLISP Reference Manual.* Project MAC — M.I.T., Cambridge, Massachusetts (April). NON-ARCHIVAL http://www.softwarepreservation.org/projects/LISP/MIT/Moon-MACLISP_Reference_Manual-Apr_08_1974.pdf (retrieved 3 March 2020) https://web.archive.org/web/20200102200158/http://www.softwarepreservation.org/projects/LISP/MIT/Moon-MACLISP_Reference_Manual-Apr_08_1974.pdf Revision 0.

Dan Murphy. 2009. The Beginnings of TECO. *IEEE Annals of the History of Computing* 31, 4 (Oct), 110–115. https://doi.org/10.1109/MAHC.2009.127

Matthias Neubauer and Michael Sperber. 2001. Down with Emacs Lisp: Dynamic Scope Analysis. In *International Conference on Functional Programming*. Florence, Italy (Sept.), 38–49. https://doi.org/10.1145/507635.507642 Also NON-ARCHIVAL http://www.deinprogramm.de/sperber/papers/dynamic-scope-analysis.pdf

Kazuhito Ohmaki. 2002. Open source software research activities in AIST towards secure open systems. In *7th IEEE International Symposium on High Assurance Systems Engineering*. (Oct), 37–41. https://doi.org/10.1109/HASE.2002.1173098

Kent M. Pitman. 1983. *The Revised Maclisp Manual.* Technical Report Technical Report 295. Laboratory for Computer Science, MIT, Cambridge, Massachusetts. NON-ARCHIVAL http://www.maclisp.info/pitmanual/ (retrieved 3 March 2020) https://web.archive.org/web/20181108211218/www.maclisp.info/pitmanual/ Saturday Morning Edition.

Kent M. Pitman. 2001. Condition Handling in the Lisp Language Family. In *Advances in Exception Handling Techniques*, A. Romanovsky, C. Dony, J.L. Knudsen, and A. Tripathi (Eds.). Lecture Notes in Computer Science, Vol. 2022. Springer. NON-ARCHIVAL http://www.nhplace.com/kent/Papers/Condition-Handling-2001.html (retrieved 4 March 2020) https://link.springer.com/chapter/10.1007%2F3-540-45407-1_3

Kent M. Pitman. 2005. Common Lisp HyperSpec. NON-ARCHIVAL http://www.lispworks.com/documentation/HyperSpec/Front/index.htm (retrieved 4 March 2020) https://web.archive.org/web/20200114180058/http://www.lispworks.com/documentation/HyperSpec/Front/index.htm

Christopher Rhodes, Jan Moringen, and David Lichteblau. 2014. Generalizers: New Metaobjects for Generalized Dispatch. In *European Lisp Symposium*. 20–27. NON-ARCHIVAL https://research.gold.ac.uk/9924/ (retrieved 3 March 2020) https://arxiv.org/abs/1403.2765

Peter Seibel. 2009. *Coders at Work: Reflections on the Craft of Programming.* Apress (Sept.).

Richard Stallman. 1994. Why you should not use Tcl. Sept. 1994. NON-ARCHIVAL https://vanderburg.org/old_pages/Tcl/war/0000.html (retrieved 3 March 2020) https://web.archive.org/web/20200108104230/https://vanderburg.org/old_pages/Tcl/war/0000.html Newsgroup posting.

Richard Stallman. 2002. My Lisp Experiences and the Development of GNU Emacs. Speech transcript. Oct. 2002. NON-ARCHIVAL https://www.gnu.org/gnu/rms-lisp.en.html (retrieved 3 March 2020) https://web.archive.org/web/20200217150413/https://www.gnu.org/gnu/rms-lisp.en.html International Lisp Conference.

Richard Stallman. 2003. Re: A plea for dynamically loadable extension modules. July 2003. NON-ARCHIVAL https://lists.gnu.org/archive/html/emacs-devel/2003-07/msg00425.html (retrieved 26 May 2019) https://web.archive.org/web/20190925063202/https://lists.gnu.org/archive/html/emacs-devel/2003-07/msg00425.html

Richard Stallman. 2005. Re: Real constants. July 2005. NON-ARCHIVAL https://lists.gnu.org/archive/html/emacs-devel/2005-07/msg00882.html (retrieved 26 July 2019) https://web.archive.org/web/20190918080041/https://lists.gnu.org/archive/html/emacs-devel/2005-07/msg00882.html

Richard Stallman. 2012a. Re: CL package serious deficiencies. Feb. 2012. NON-ARCHIVAL https://lists.gnu.org/archive/html/emacs-devel/2012-02/msg00350.html (retrieved 26 July 2019) https://web.archive.org/web/20190918064247/https://lists.gnu.org/archive/html/emacs-devel/2012-02/msg00350.html

Richard Stallman. 2012b. Re: CL package serious deficiencies. Feb. 2012. NON-ARCHIVAL https://lists.gnu.org/archive/html/emacs-devel/2012-02/msg00272.html (retrieved 26 July 2019) https://web.archive.org/web/20190918064231/https://lists.gnu.org/archive/html/emacs-devel/2012-02/msg00272.html

Richard Stallman. 2012c. Re: CL package serious deficiencies. Feb. 2012. NON-ARCHIVAL https://lists.gnu.org/archive/html/emacs-devel/2012-02/msg00283.html (retrieved 26 July 2019) https://web.archive.org/web/20190918064231/https://lists.gnu.org/archive/html/emacs-devel/2012-02/msg00283.html

Richard Stallman. 2018a. *GNU Emacs Lisp Reference Manual.* Free Software Foundation, Chapter Why Text Properties are not Intervals. NON-ARCHIVAL https://www.gnu.org/software/emacs/manual/html_node/elisp/Not-Intervals.html (retrieved 3 March 2020) https://web.archive.org/web/20191224141917/https://www.gnu.org/software/emacs/manual/html_node/elisp/Not-Intervals.html

Richard Stallman. 2018b. Personal communication. Email. Nov. 2018.

This email exchange originated from a review Richard Stallman conducted on an early version of this paper. In this exchange, he provided detailed comments on selected sections. In particular, he mentioned his work on ZWEI and the fact that docstrings were already present in the first TECO-based Emacs. Here is the relevant section from that exchange on ZWEI, written by Richard Stallman:

I did some work on ZWEI, the second iteration of EINE. In ZWEI (Zwei Was Eine Initially), buffer-modifying primitives took arguments to specify what text to operate on. I found that clumsy, so in Emacs Lisp I made them operate at point.

Stallman, referring to the docstrings feature, also wrote:

These came from the original TECO-based Emacs.

Richard Stallman. 2019. Personal communication. Email. 26 Aug. 2019 19:19:01−04:00.

In response to an email message by Stefan Monnier asking details about the history of the byte-compiler, he wrote:

> 3- I see in etc/NEWS.1-17 that byte-recompile-directory was introduced in
> Emacs-1.1, which implies the byte-compiler already existed pretty much
> from the very beginning of Emacs. I assume you're the author of the
> original 'byte-code' primitive and associated byte-compiler, right?

I don't remember whether anyone else contributed code for it, but I certainly did this early on.

> Do you remember why you immediately wrote a byte-code implementation of Elisp

For speed.

Richard M. Stallman. 1981. EMACS: The Extensible, Customizable Self-documenting Display Editor. In *Proceedings of the ACM SIGPLAN SIGOA Symposium on Text Manipulation* (Portland, Oregon, USA). ACM, New York, NY, USA, 147–156. https://doi.org/10.1145/872730.806466 NON-ARCHIVAL https://www.gnu.org/software/emacs/emacs-paper.html (retrieved 4 March 2020) https://dl.acm.org/doi/10.1145/872730.806466

Guy L. Steele. 1984. *Common Lisp the Language* (1st ed.). Digital Press, 313 Washington Street Newton, MA, United States.

Guy L. Steele, Jr. and Richard P. Gabriel. 1993. The Evolution of Lisp. In *The Second ACM SIGPLAN Conference on History of Programming Languages* (Cambridge, Massachusetts, USA) (*HOPL-II*). ACM, New York, NY, USA, 231–270. https://doi.org/10.1145/155360.155373

Sam Steingold. 1999. Load Emacs-Lisp files into Common Lisp. NON-ARCHIVAL https://sourceforge.net/p/clocc/hg/ci/default/tree/src/cllib/elisp.lisp (retrieved 4 March 2020) https://web.archive.org/web/20190925063736/https://sourceforge.net/p/clocc/hg/ci/default/tree/src/cllib/elisp.lisp

SXEmacs [n.d.]. SXEmacs – Redefining Emacs. Web site. http://www.sxemacs.org/

Li-Cheng Tai. 2001. The History of the GPL. Web page. July 2001. NON-ARCHIVAL http://www.free-soft.org/gpl_history/ (retrieved 36 May 2019) https://web.archive.org/web/20191015034917/http://www.free-soft.org/gpl_history/

The Unicode Consortium. 2011. *The Unicode Standard.* Technical Report Version 6.0.0. Unicode Consortium, Mountain View, CA. NON-ARCHIVAL http://www.unicode.org/versions/Unicode6.0.0/ (retrieved 3 March 2020) https://web.archive.org/web/20180801000000*/http://www.unicode.org/versions/Unicode6.0.0/

Walter F. Tichy. 1985. RCS - A System for Version Control. *Software Practice&Experience* 15, 7, 637–654. https://doi.org/10.1002/spe.4380150703 NON-ARCHIVAL https://www.gnu.org/software/rcs/tichy-paper.pdf (retrieved 3 March 2020) https://web.archive.org/web/20191224233920/https://www.gnu.org/software/rcs/tichy-paper.pdf Purdue University.

Andrew Tolmach and Andrew W. Appel. 1990. Debugging Standard ML without reverse engineering. In *Conference on Lisp and Functional Programming*. 1–12. https://doi.org/10.1145/91556.91564

Daniel Weinreb and David Moon. 1981. *Lisp Machine Manual* (third ed.). Massachusetts Institute of Technology, Cambridge, Massachusetts (March). NON-ARCHIVAL http://www.bitsavers.org/pdf/mit/cadr/chinual_3rdEd_Mar81.pdf (retrieved 6 March 2020) https://web.archive.org/web/20200103040539/http://www.bitsavers.org/pdf/mit/cadr/chinual_3rdEd_Mar81.pdf

Daniel L. Weinreb. 1979. *A Real-Time Display-oriented Editor for the LISP Machine.* Technical Report. MIT EECS Department (Jan.).

In his Undergraduate thesis, Daniel Weinreb describes the ZWEI user interface, its interaction with the Lisp machine, its implementation strategies and engineering trade-offs, and the future of the system.

John Wiegley. 2019. Simple library for asynchronous processing in Emacs. Github project. NON-ARCHIVAL https://github.com/jwiegley/emacs-async (retrieved 6 March 2020) https://web.archive.org/web/20190925224033/https://github.com/jwiegley/emacs-async

Ben Wing, Bil Lewis, Dan LaLiberte, and Richard Stallman. 1998. *XEmacs Lisp Reference Manual* (version 3.3 ed.) (April). NON-ARCHIVAL https://www.xemacs.org/Documentation/21.5/html/lispref.html (retrieved 4 March 2020) https://web.archive.org/web/20160504192740/https://www.xemacs.org/Documentation/21.5/html/lispref.html For XEmacs Version 21.0.

Jamie Zawinski. 2007. Emacs Timeline. Oct. 2007. NON-ARCHIVAL https://www.jwz.org/doc/emacs-timeline.html (retrieved 6 March 2020) https://web.archive.org/web/20200212221812/https://www.jwz.org/doc/emacs-timeline.html