MASSACHUSETTS INSTITUTE OF TECHNOLOGY
ARTIFICIAL INTELLIGENCE LABORATORY

AI Memo 519a

26 March 1981

EMACS
The Extensible, Customizable Self-Documenting Display Editor

by

Richard M. Stallman

Abstract: EMACS is a display editor which is implemented in an interpreted high level language. This allows users to extend the editor by replacing parts of it, to experiment with alternative command languages, and to share extensions which are generally useful. The ease of extension has contributed to the growth of a large set of useful features. This paper describes the organization of the EMACS system, emphasizing the way in which extensibility is achieved and used.

Keywords: Display, Editor, Extensible, Interactive, Self-documenting

This report describes work done at the Artificial Intelligence Laboratory of the Massachusetts Institute of Technology. Support for the laboratory's research is provided in part by the Advanced Research Projects Agency of the Department of Defense under Office of Naval Research contract N00014-80-C-0505.

A truncated version of this paper appeared in the proceedings of the ACM SIGPLAN/SIGOA Symposium on Text Manipulation, Portland Oregon, June 1981.

© MASSACHUSETTS INSTITUTE OF TECHNOLOGY 1981

1. Introduction

EMACS[1] is a real-time display editor which can be extended by the user while it is running.

Extensibility means that the user can add new editing commands or change old ones to fit his editing needs, while he is editing. EMACS is written in a modular fashion, composed of many separate and independent functions. The user extends EMACS by adding or replacing functions, writing their definitions in the same language that was used to write the original EMACS system. We will explain below why this is the only method of extension which is practical to use: others are theoretically equally good but discourage use, or discourage nontrivial use.

Extensibility makes EMACS more flexible than any other editor. Users are not limited by the decisions made by the EMACS implementors. What we decide is not worth while to add, the user can provide for himself. He can just as easily provide his own alternative to a feature if he does not like the way it works in the standard system.

A coherent set of new and redefined functions can be bound into a *library* so that the user can load them together conveniently. Libraries enable users to publish and share their extensions, which then become effectively part of the basic system. By this route, many people can contribute to the development of the system, for the most part without interfering with each other. This has led the EMACS system to become more powerful than any previous editor.

User customization helps in another, subtler way, by making the whole user community into a breeding and testing ground for new ideas. Users think of small changes, try them, and give them to other users. If an idea becomes popular, it can be incorporated into the core system. When we poll users on suggested changes, they can respond on the basis of actual experience rather than thought experiments.

To help the user make effective use of the copious supply of features, EMACS provides powerful and complete interactive self-documentation facilities with which the user can find out what is available.

A sign of the success of the EMACS design is that EMACS has been requested by over a hundred sites and imitated at least ten times.

1.1. Background: Real-Time Display Editors

By a *display editor*, we mean an editor in which the text being edited is normally visible on the screen and is updated automatically as the user types his commands. No explicit commands to "print" text are needed.

As compared with printing terminal editors, display editor users have much less need for paper listings, and can compose code quickly on line without writing it on paper first. Display editors are also easier to learn than printing terminal editors. This

---

[1] EMACS stood for Editing Macros, before we realized that EMACS is composed of functions written in a programming language rather than macros in the editor TECO.

is because editing on a printing terminal requires a mental skill like that of blindfold chess; the user must keep a mental image of the text he is editing, which he cannot easily see, and calculate how each of his editing command "moves" changes it. A display editor makes this unnecessary by allowing the user to see the "board".

Among display editors, a *real-time* editor is one which updates the display very frequently, usually after each one or two character command the user types. This is a matter of the input command language. Most printing terminal editors read a string of commands and process it all at once; a useful feature on a printing terminal. For example, there is usually an "insert" command which inserts a string of characters. When such editors are adapted to display terminals, they often update the display at the end of a command string; thus, the insertion would be shown all at once when it was over. It is more helpful to display each inserted character in its position in the text as soon as it has been typed.

A real-time display editor has (primarily!) short, simple commands which show their effects in the display as soon as they are typed. In EMACS, text (printing characters and formatting characters) is inserted just by typing it; there is no "insert" command. In other words, each printing character is a command to insert that character. The commands for modifying text are nonprinting characters, or begin with nonprinting characters. Many-character commands echo if typed slowly; if there is a sufficiently long pause, the command so far is echoed, and then the rest of the command is echoed as it is typed. Aside from this, EMACS acknowledges commands by displaying their effects.

EMACS is not the first real-time display editor, but it derives much appeal from being one. It is not necessary to know how to program, or how to extend EMACS, to use it successfully.

2. Applications of Extensibility

To illustrate and demonstrate the flexibility which EMACS derives from extensibility, here is a summary of many of the features, available to EMACS users without the need to program, to which extensibility has contributed. Many of them were written by users; some were written by the author, but could just as well have been written by users.

2.1. Customization

Many minor extensions can be done without any programming. These are called customizations, and are very useful even by themselves. For example, for editing a program in which comments start with <** and end with **>, the user can tell the EMACS comment manipulation commands to recognize and insert those strings. This is done by setting parameters which the comment commands refer to. It is not necessary to redefine the commands themselves. Another sort of customization is rearrangement of the command set. For example, some users prefer the four basic cursor motion commands (up, down, left and right) on keys in a diamond pattern on the keyboard. It is easy to reassign the commands to these positions. It is also possible to rearrange the entire command set according to a different philosophy.

2.2. Operating on Meaningful Units of Text

EMACS can be programmed to understand the syntax of the language being edited and provide operations particular to it. Many *major modes* are defined, one for each language which is understood. Each major mode has the ability to redefine any of the commands, and reset any parameters, so as to customize EMACS for that language. Files can contain special text strings that tell EMACS which major mode to use in editing them. For example, `-*-Lisp-*-` anywhere in the first nonblank line of a file says that the file should be edited in Lisp mode. The string would normally be enclosed in a comment.

For editing English text, commands have been written to move the cursor by words, sentences and paragraphs, and to delete them; to fill and justify paragraphs; and to move blocks of text to the left or to the right. Other commands convert single words or whole regions to upper or lower case. There are also commands which manipulate the command strings for text justifier programs: some insert or delete underlining commands, and others insert or delete font-change commands.

Many commands are controlled by parameters which can be used to further adapt them to particular styles of formatting. For example, the word moving and deletion commands have a syntax table that says which characters are parts of words. There are two commands to edit this table, one convenient for programs to use and an interactive one for the user. The paragraph commands can be told which strings, appearing at the beginning of a line, constitute the beginning of a paragraph. Such parameters can be set by the user, or by a specification in the file being edited. But normally they are set automatically by the major mode (that is, by telling EMACS what language the file is written in) and do not require attention from the user.

2.3. Redefining Self-Inserting Characters

A very powerful extension facility is the ability to redefine the graphic and formatting characters as commands. These characters, which include letters, digits and punctuation, are normally all defined as commands to insert themselves into the text. Useful alternate definitions for these characters usually insert the character as usual, and then do additional processing which is in some way meaningfully associated with the insertion of that character.

The single most useful command for editing text is the "auto-fill space". It is a program intended to be used as the definition of the space character. In addition to inserting a space, it breaks the line into two lines if it has become too long. With the space character redefined in this way, the user can type endlessly ignoring the right margin, and the text is divided into lines of a reasonable length. Of course, this feature is not always desirable. It is turned on or off by redefining the space command. If the auto-fill space did not exist, any user could write it and also the command to turn it on and off.

A bolder use of redefinition of self-inserting characters is the abbreviation facility, part of the standard EMACS system but still implemented as an extension maintained by the user who wrote it. The abbreviation facility allows the user to define abbreviations for words, and then type the abbreviations in order to insert the words. For example, if "cd" were defined as an abbreviation for "command", typing "i/o-cd" would insert "i/o-command" into the text. Abbreviation expansion preserves case, so "Cd" would expand into "Command". Abbreviation works by redefining all punctuation characters (the list of which can be altered by customization) to run a program which looks at the preceding word and, if it is a defined abbreviation, replaces it with its expansion.

Yet another application of redefining printing characters is automatic parenthesis-matching. When this feature is in use, every time the user inserts a close-parenthesis, the cursor moves briefly to the matching open-parenthesis, then back again. Automatic matching is especially useful in editing Lisp code, but it is helpful with most other programming languages also. It is implemented by redefining the close-parenthesis character.

2.4. Editing Programs

Extensibility is especially useful for editing programs. One might conceivably design in advance all the editing commands needed for editing English text, but each programming language has its own set of useful syntactic operations, which suggest useful editing commands. Because languages differ so much, simple customization is not in general enough to implement familiar operations for a new language. A new extension package is required.

EMACS commands have been written, for many languages, to move over or kill balanced expressions, to move to the beginning or end of a function definition, and to insert or align comments. But the most useful editing operation for programs, and the first one to be implemented for any programming language, is automatic indentation.

The structure of a program can be made clear at a glance by adjusting the indentation of each line according to its level of nesting. Most programming communities attempt to indent code properly but do it manually. Automatic indentation is used mostly by Lisp programmers.

Automatic indentation was traditionally done by a program which would read in an entire source file, rearrange the indentation, and write out a corrected source file. Such a tool has several disadvantages. For one thing, processing the entire file is likely to take a while. For another, the tool insists on imposing its own idea of proper formatting, which the user cannot override. Even after a lot of effort is put into heuristics for good indentation, users are still dissatisfied.

Automatic indentation in EMACS is done incrementally. The Tab character is redefined, as a command, to update the indentation of the current line only, based on the existing indentation of the preceding lines. The Tab command is used on lines whose nesting has changed. With it, the user can indent code properly as it is first typed in. If he does not agree with the Tab command's choice of indentation, he can override it.

Because the indentation function must understand the syntax of the programming language being edited, each language requires a separate indentation function. It is the job of the major mode for each programming language to redefine the Tab character to run an appropriate indenter. Users can always use the same command to indent, no matter what sort of program they are editing. In addition, another editing command can do indentation by calling the current definition of Tab as a subroutine. (One such function is the one which indents several consecutive lines.)

Conventions such as this are vital, in an extensible system, for enabling unrelated extensions to avoid interacting wrong; one user can write an indentation function for a new language, while another user writes new language-independent operations for requesting indentation, and the two automatically work properly together.

Languages which have support for indentation include Lisp, Pascal, PL/I, Bliss, BCPL, Muddle and TECO.

Comprehension of the user's program reaches its greatest heights for Lisp programs, because the simplicity of Lisp syntax makes intelligent editing operations easier to implement, while the complexity of other languages discourages their users from implementing similar operations for them. In fact, EMACS offers most of the same facilities as editors such as the Interlisp editor which operate on list structure, but combined with display editing. The simple syntax of Lisp, together with the powerful editing features made possible by that simple syntax, add up to a more convenient programming system than is practical with other languages. Lisp and extensible editors are made for each other, in this way. We will see below that this is not the only way.

2.5. Editing Large Programs

Large programs are composed of many functions divided among many files. It is often hard to remember which file a given function is in. An EMACS extension called the TAGS package knows how to keep track of this.

The TAGS package makes use of a file called a tag table, which records each function in the program, stating what file it is defined in and at what position in the file. The tag table is made by running a special program named TAGS, which is not part of EMACS. Once the tag table is loaded into EMACS, the command Meta-Period[2] finds the definition of any function, using the information in the tag table to select the proper file and find the function in it.

The positions within the source file, remembered in the tag table, are used to find the function in the file instantly. Changing the file makes the remembered positions inaccurate. If this has happened, Meta-Period searches in both directions away from the remembered position until it finds the definition. So small inaccuracies cause only slight delays.

When many new functions have been added, or moved from one file to another, the TAGS program can reprocess the tag table into an updated one. To make this more automatic, the tag table also remembers which language each source file is written in. This information is needed for recognizing the function definitions in the file.

2.6. Editing Other Things

Interactiveness is useful in many activities aside from editing text. For example, reading and replying to mail from other users ought to be interactive. Many of these activities occasionally involve text editing: for example, editing the text of a reply. If a special editor is implemented for the purpose, it can easily be much more work to write than all the rest of the system. It is easier to write the other interactive system within the framework of an extensible editor.

EMACS has two extensions, RMAIL and BABYL, for reading mail. Commands in RMAIL and BABYL are not like EMACS commands; typical commands include "D" for "delete this message", and "R" for "reply to this message". Editing the text of the reply is done with ordinary EMACS commands.

DIRED is used for editing a file directory. The normal editing commands, as extended, can be used to move the cursor through the directory listing. Other special commands defined only in DIRED delete, move, compare or examine the file whose name is under the cursor.

The INFO extension is designed for reading tree-structured documentation files. These files are divided textually into nodes, which contain text representing pointers to other nodes. INFO displays one node at a time, and INFO commands move from one node to another by following the pointers.

---

[2] "Meta" is the name of a shift key on the ideal EMACS terminal. On terminals which do not have this key, the ASCII character Escape is used as a prefix instead.

3. The Organization of the EMACS System

The primary components of the EMACS system are the text manipulation and I/O primitives, the interpreter, the command dispatcher, the library system, and the display processor.

The text and I/O primitives are used to operate on the text under the command of the program. The interpreter executes programs, using the primitives when called for. The command dispatcher remembers which program corresponds to each possible input character; it reads a character from the terminal and calls the associated function. The library system associates functions with their names and documentation, and allows groups of related functions to be loaded quickly together. The display processor updates the screen to match the text as changed by the text primitives; it is run whenever there is nothing else to do.

3.1. Editing Language vs. Programming Language

An EMACS system actually implements two different languages, the editing language and the programming language. The editing language contains the commands users use for changing text. These commands are implemented by programs written in the programming language. When we speak of the *interpreter*, we mean the one which implements the programming language. The editing language is implemented by the *command dispatcher*.

Previous attempts at programmable editors have usually attempted to mix programming constructs and editing in one language. TECO is the primary example of this sort of design. It has the advantage that once the user knows how to edit with the system, he need only learn the programming constructs to begin programming as well.

However, there are considerable disadvantages, because what is good in an editor command language is ugly, hard to read, and grossly inefficient as a programming language. A good interactive editing language is composed primarily of single-character commands, with a few commands that introduce longer names for less frequently used operations. As a programming language, it is unreadable. If the editor is to be customizable, the user must be able to redefine each character. This in a programming language would be intolerable!

When the programming language is the editing language, the built-in editing commands and the primitive operations they use have to be written in another language. Then the user cannot change part of the standard system slightly by making a small change to its definition; it has to be reimplemented from scratch as a macro. Since the primitives available are only the commands he uses for editing, this will often be impossible because the necessary primitives will be internal routines that the user cannot call. The primitives that an extension would like to use are not always the same as the editing operations the user wants.

The implementor of a macro processor is encouraged to ignore such deficiencies because he himself does not use the language in implementing the rest of the system. Since it is traditional, in designing a macro language, to ignore the standards of

readability, power and robustness typically applied to the design of programming languages, these deficiencies are usually considerable. The original TECO is a good example of this sort of problem.

In EMACS, each language is designed for its purpose. The editing language has single-character redefinable commands. The programming language is TECO, modified and extended to be more suitable for writing well-structured and robust programs, and to provide the primitives needed by editing programs as opposed to editor users. It remains hard to read, so the descendents of EMACS generally use Lisp instead. TECO was used only for reasons of historical convenience.

More information on the requirements extensibility imposes on the system's programming language is in the next chapter.

3.2. The Library System and the Command Dispatcher

An important part of any practical extensible system is the ability to use more than one extension at one time, and begin using an additional extension at any time. Extensions should be able to override or replace parts of the standard system, or previous extensions. In EMACS the library system is responsible for accomplishing this.

An EMACS library is a collection of function names, definitions and documentation that can be loaded into an EMACS in mid-session. Libraries are read-only and position-independent, so that they can be loaded just by incorporating them into the virtual memory of the EMACS. This allows all EMACS's using a library to share the physical memory. Each library contains its own symbol table which connects function names with definitions, and also with their documentation strings. Libraries are generated from source files in which each function definition is accompanied by its documentation; this encourages all functions to be documented.

When a function name is looked up, all the loaded libraries are searched, most recently loaded first. For the sake of uniformity, the standard EMACS functions also reside in a library, which is always the first one loaded. Therefore, any library can override or replace the definition of a standard EMACS function with a new definition, which will be used everywhere in place of the old. This, together with the fact that EMACS is constructed with explicit function calls to named subroutines at many points, makes it easy for the user to change parts of the system in a modular fashion without replacing it all.

Subroutines are normally called by their full names. The user can also call any command by name, and many commands are primarily intended to be used in that way. However, the most common editing operations need to be more easily accessible. This is the purpose of the command dispatcher, which reads one character and looks it up in the *dispatch table*, a vector of definitions to find the function to be called (the definition-object, not the name).

Functions residing in the dispatch table can be invoked either by the character command or by name. A function which does not appear in the dispatch table can be called only by name. The user calls functions by name by means of a single-character command (Meta-X) whose definition is to read the name of a function and call that function.

Each user has his own patterns of use. Many functions in EMACS are accessible only by name because we expect most users to use them infrequently. If a particular user uses one such command often, he can place the definition in the dispatch table using the function Set Key. The function calling conventions are designed so that almost any function definition will behave reasonably if called by the command dispatcher. If a function tries to read a string argument from its caller, then when called by the command dispatcher it will automatically prompt and read the argument from the terminal instead.[3]

Some libraries contain functions that are intended to be called with single-character commands. The library can arrange to place those functions' definitions in the dispatch table by defining a function called Setup. This will be called automatically when the library is loaded, and it can redefine character commands as needed. However, because EMACS is intended to be customized, no library can reasonably make the assumption that a function belongs on a particular character without allowing the user who loads the library to override that assumption. For example, a library might wish to redefine Control-S on the assumption that it invokes the search function, but a user might prefer to keep his search on Control-T instead, and he might prefer that same library to alter the definition of Control-T when loaded by him. The author of the library cannot anticipate the details of such idiosyncrasies, but he can provide for them all by following a convention: in the Setup function of the library (TAGS, say), he checks for a variable called **TAGS Setup Hook**, and if it exists, its value is called as a function instead of the usual setting up.

3.3. The Display Processor

The display processor is the part of EMACS which maintains on the display screen an up-to-date image of the text inside the editor. Since the size of the screen is limited, only a portion or "window" can be shown. The display processor prefers to continue to start its display at the same point in the file, so as to minimize the amount of changes necessary to the screen. However, the text where the editor's own cursor is located must appear on the screen so that the terminal's cursor can show where it is. This sometimes forces a new window position to be computed. The user can also command changes in the window position, moving the text up or down on the screen.

The EMACS display processor embodies an unusual principle which makes for much faster responses to the user: display updating has lower priority than cogitation.

Most display editors change the display after each user command. This is the simplest strategy to implement, since each command knows precisely how it has changed the text. But it is very inefficient, not just of the computer's time, but of the user's time, because it makes the user wait for the completion of display updates that have already been made obsolete by further commands waiting to be executed.

Here is an example of the problem. If the user types Carriage Return to create a new line, all the lines below that point need to be redisplayed in their new positions.

---

[3] The process of reading the argument from the terminal is implemented by a function which the user can replace.

While this is still going on, if he types an additional Carriage Return to create another new line, the rest of the display update is obsolete; there is no use displaying the rest of the lines in their second positions, only to display them again in their third positions.[4]

The EMACS display processor is best understood as being a separate, lower priority process that runs in parallel with the editing process. The editing process reads keyboard input and makes changes in the text. The display process is always trying to change the screen to match the text; it keeps a record of what is on the screen, and in each cycle of operation finds one discrepancy between the editing buffer and the screen record and corrects it. After each cycle, the display process can be pre-empted by the editing process, which has higher priority. The display process can be thought of as chasing an arbitrarily moving target, the edited text, with a speed limited by the terminal baud rate.

Multiple processes are not actually used in the implementation. Instead, after each line of display output, the display processor updates its data base and polls for input.

An additional benefit of this input-before-output philosophy is that it uses less computer resources when the system is heavily loaded. When not enough computer power is available, EMACS gets behind in processing the user's input. When the first command is completed, more input is available, so no effort is put into display updating yet. By saving computer time this way, EMACS eventually catches up with the user and does its display updating all at once.

Since display updating is not necessarily done at the same time as the editing operation which necessitates it, display updating cannot be the responsibility of the editing command itself. Instead, the display update must be done by somehow comparing the new text with the previous displayed text, or information about it. In EMACS, each editing command returns information on the range of text it has changed, but aside from that the display processor operates independently. This is good for extensibility as well: it is easier to write or change an editing command if it does not have to contain algorithms for updating the screen.

Because the TECO language is not very efficient, the display processor had to be written in assembler language to get adequate performance. This is unfortunate because extensions to the display processor could be very valuable. In later implementations of EMACS, the display processor is written in Lisp along with the editing commands, and can be extended.

---

[4] This particular sequence of events poses no problem on terminals which can move text up and down on the screen. But the same problem can still result from other events.

4. Extensibility and Interpreters

Despite its syntactic obscurity, TECO is actually one of the best languages to use for implementing an extensible editor. This is because most traditional programming languages simply cannot do the job! Implementing an extensible system of any sort requires features that they intrinsically lack. Specifically, it requires a language with an interpreter and the ability for programs to access the interpreter's data structures (such as function definitions).

Adherents of non-Lisp programming languages often conceive of implementing an EMACS for their own computer system using PASCAL, PL/I, C, etc. In fact, it is simply impossible to implement an extensible system in such languages. This is because their designs and implementations are *batch-oriented*; a program must be compiled and then linked before it can be run. An on-line extensible system must be able to accept and then execute new code while it is running. This eliminates most popular programming languages except Lisp, APL and Snobol. At the same time, Lisp's interpreter and its ability to treat functions as data are exactly what we need.[5]

A system written in PL/I or PASCAL can be modified and recompiled, but such an extension becomes a separate version of the entire program. The user must choose, before invoking the program, which version he wants. Combining two independent extensions requires comparing and merging the source files. These obstacles usually suffice to discourage all extension.

The only way to implement an extensible system using an unsuitable language, is to write an interpreter for a suitable language and then use that one. Prime is now implementing an EMACS using a simple Lisp written in PL/I. This technique works because an editor does not require a very efficient interpreter; even the most straightforward Lisp interpreter is more efficient than the TECO interpreter which is empirically observed to be good enough. I would not regard this as implementation "in" the original language, however.

A PASCAL or PL/I implementation which uses an interpreter, and allows the user program to access the interpreter data structures sufficiently, could be used just as a Lisp implementation would be used. However, such implementations are very rare, because these languages are not designed for them. If the implementor appreciates the importance of the interpreter, and of treating functions as data, he will usually choose to implement Lisp.

It is also possible to use dynamic linking—the ability to load additional modules of compiled code during execution, and refer to subroutines therein by name—in place of an interpreter. However, dynamic linking operating systems are rarer than good Lisps, harder to implement, and not as convenient for the job. One of the few such operating systems, Multics, has an EMACS written in Lisp. SINE, the EMACS implementation on Interdata computers, uses dynamic linking to load files compiled from a language which resembles Lisp.

---

[5] It is o.k. to use a Lisp compiler, if there is one. What counts is not using the interpreter all the time, but having it available all the time.

5. Language Features for Extensibility

When a language is used for implementing extensible systems, certain control structure and data structure features become vital.

5.1. Global Variables

One difference between Lisp (and TECO) and most other programming languages, which is very important in writing extensible systems, is that variable names are retained at run time; they are not lost in compilation.

In typical compiled languages, variable names are meaningful only at compile time. In the compiled code, uses of one variable name become references to one location in memory, but the name itself has been discarded.

By contrast, Lisp remembers the connection between variable names and their values, so that new programs can be defined.

Global variables are essential for parameters used for customization. EMACS has a variable named **Comment Start** which controls the string recognized as starting a comment in the text being edited. Its value is supposed to be that string. This variable is used by the comment indenting command to recognize an existing comment. The fact that the variable name is known at run time enables the user to

- ask to see the value of the string.

- change the string.

- define or redefine major modes, for various programming languages, which change the string.

- define or redefine comment-manipulation commands, which refer to the variable so that they will work on text in various languages.

5.2. Dynamic Binding

Most batch languages use a lexical scope rule for variable names. Each variable can be referred to legally only within the syntactic construct which defines the variable.

Lisp and TECO use a dynamic scope rule, which means that each binding of a variable is visible in all subroutine calls to all levels, unless other bindings override. For example, after

```
(defun foo1 (x) (foo2))
(defun foo2 () (+ x 5))
```

then **(foo1 2)** returns 7, because **foo2** when called within **foo1** uses **foo1**'s value of **x**. If **foo2** is called directly, however, it refers to the caller's value of **x**, or the global value. We say that **foo1** *binds* the variable **x**. All subroutines called by **foo1** see the

binding made by **foo1**, instead of the global binding, which we say is *shadowed* temporarily until **foo1** returns.

In PASCAL the analogous program would be erroneous, because **foo2** has no lexically visible definition of **x**.

Dynamic scope is useful. Consider the function **Edit Picture**, which is used to change certain editing commands slightly, temporarily, so that they are more convenient for editing text which is arranged into two-dimensional pictures. For example, printing characters are changed to replace existing text instead of shoving it over to the right. **Edit Picture** works by binding the values of parameter variables dynamically, and then calling the editor as a subroutine. The editor "exit" command causes a return to the **Edit Picture** subroutine, which returns immediately to the outer invocation of the editor. In the process, the dynamic variable bindings are unmade.

Dynamic binding is especially useful for elements of the command dispatch table. For example, the RMAIL command for composing a reply to a message temporarily defines the character Control-Meta-Y to insert the text of the original message into the reply. The function which implements this command is always defined, but Control-Meta-Y does not call that function except while a reply is being edited. The reply command does this by dynamically binding the dispatch table entry for Control-Meta-Y and then calling the editor as a subroutine. When the recursive invocation of the editor returns, the text as edited by the user is sent as a reply.

It is not necessary for dynamic scope to be the *only* scope rule provided, just useful for it to be available.

5.3. Formal Parameters Cannot Replace Dynamic Scope

Some language designers believe that dynamic binding should be avoided, and explicit argument passing should be used instead. Imagine that function A binds the variable FOO, and calls the function B, which calls the function C, and C uses the value of FOO. Supposedly A should pass the value as an argument to B, which should pass it as an argument to C.

This cannot be done in an extensible system, however, because the author of the system cannot know what all the parameters will be. Imagine that the functions A and C are part of a user extension, while B is part of the standard system. The variable FOO does not exist in the standard system; it is part of the extension. To use explicit argument passing would require adding a new argument to B, which means rewriting B and everything that calls B. In the most common case, B is the editor command dispatcher loop, which is called from an awful number of places.

What's worse, C must also be passed an additional argument. B doesn't refer to C by name (C did not exist when B was written). It probably finds a pointer to C in the command dispatch table. This means that the same call which sometimes calls C might equally well call any editor command definition. So *all* the editing commands must be rewritten to accept and ignore the additional argument. By now, none of the original system is left!

5.4. Variables Local to a File

Suppose one file is formatted with comments starting at column 50. Editing this file is easier if the variable **Comment Column**, which is used (by convention) to decide where to align comments, is always set to 50 whenever this file is being editing. EMACS provides a way to request this; but since it also provides the feature of visiting several files at once, it must take special care to keep each file's variables straight. Suppose one file wants **Comment Column** to be 50 while another is formatted with 40?

This is solved by allowing each file to have its own local values for any set of variables. Specially formatted text at the end of the file specifies them:

```
Local Modes:
Comment Column:50
End:
```

When a file is brought into EMACS, this local modes list is parsed and the variables and values remembered in a local symbol table. While the file is not selected, its local symbol table contains the local values of the variables. While a file is selected, its local symbol table contains the global values, and the real symbol table contains the file's local values instead.

5.5. Hooks

When an extensible system allows the user to provide a function to be called on certain well-defined occasions, we call it a *hook*. For example, we have already mentioned the hook which is executed whenever a certain library is loaded; for the TAGS library, the hook is named **TAGS Setup Hook**.

Another important class of hooks is executed when a major mode is entered. Each major mode has its own hook. For example, Text mode's hook is named **Text Mode Hook**. This hook can be used to request arbitrary actions in advance for each time text mode is entered. Many users always define this hook to turn on Auto Fill mode, so that Auto Fill mode is always on when Text mode is.

Hooks can be associated with variables as well. Then, each time the value of the variable changes, its hook is run. Usually these hooks are used to change other data structures so that they always correspond to the value of the variable. This is often more efficient and more modular than checking the variable itself whenever its value is relevant. For example, changing the value of **Auto Fill Mode** to turn auto-filling on or off calls a function which automatically redefines the Space character's command definition.

Some hooks are attached to specific points within the interpreter or display processor. For example, there is a hook which is called whenever it is time to read a character of input from the terminal. The hook program can supply the character itself. These hooks can be thought of as compensating for the fact that some parts of the system are written in assembler language and cannot simply be redefined by the user.

5.6. Errors and Control Structure

A system for programming editor commands needs more sophisticated facilities for handling errors and other exceptional conditions than most programming systems provide. Let us consider what an error is, and what ought to happen when there is an error.

First of all, what exactly is an error? Sometimes the user asks to do something that cannot be done (a user error). Sometimes a program asks to do something which cannot be done (a program error). Program errors often accompany user errors, but either one can happen without the other.

Program errors can be defined objectively: any event which executes a certain part of the interpreter is a program error. User errors cannot be defined objectively in this way because they are a matter of attitude toward events rather than events themselves. If a command has done nothing, we can regard this either as the response to an error or as normal functioning. And this choice of attitude has no necessary connection with whether the command definition required special code to make it do nothing in the circumstances in question.

When a program error happens, EMACS prints the error message and then gives the user the chance to invoke the error handler to debug it. If he does not do this, control returns to the innermost error return point. Programs can create error return points with a special construct. (We use a Lisp-style syntax in these examples for clarity).

```
(error-return
  (arbitrary-code-here))
```

The end of the error-return construct becomes an error return point which is in effect while the code inside the construct is being executed. Error returns are usually used by loops which read and execute commands of some sort, including the built-in one which reads and displays editing commands.

```
(do-forever
  (error-return
    (read-and-execute-one-command)))
```

Sometimes interpreted functions are called asynchronously or unpredictably. An example is the one which optionally saves the text every so often to reduce the amount lost if the system crashes. If this function gets a program error, it should notify the user, but should not interfere in any way with the user's explicit commands. This requires a construct known in Lisp as *errset*, which prevents *all* normal processing of errors that occur within it. An error occurring within an errset does nothing but return control immediately to the end of the errset.

The programming system does not provide any such uniform handling for user errors because the concept of a user error is not defined at that level. Instead, the designer of each editing command must decide what conditions ought to be considered errors, and what to do in each case. Sometimes the command simply does nothing. Sometimes it rings the terminal's bell and perhaps throws away type ahead. This can be best if we expect that, once the user is told that there is something wrong, it will be obvious what it is. When the cause of the error is less

obvious, causing a program error deliberately with a specially chosen error message is a good way of informing him. A special primitive is used to cause a program error with an arbitrary specified error message so that the error-return processing can be invoked.

Sometimes the user error leads naturally to an error in the program, which may be all the handling it needs. This can be so if the program error's error message is an adequate explanation for the user, or if the situation is not deemed likely enough to deserve the effort required to make anything else happen.

The error handler for debugging program errors is an interpreted program itself. This is possible because primitives are provided for examining the function call stack and all other data structures which the programmer would want to examine while debugging. Users have actually written extensions and complete replacements for the standard error handler program.

5.7. Non-local Control Transfers

Returning to the example of the user-written command loop, there has to be a command to exit the loop. How can it be done?

```
(do-forever
  (error-return
    (read-and-execute-one-command)))
```

We do it by means of a non-local control transfer. We create the transfer point by means of a *catch* construct around the loop. The catch creates a named transfer point at the end of the loop, which is accessible only within the loop.

```
(catch
  (do-forever
    (error-return
      (read-and-execute-one-command)))
  exit-my-loop)
```

At any time during the loop, execution of **(throw exit-my-loop)** transfers control immediately to the end of the catch, thus exiting the loop. The catch and throw constructs were copied from Maclisp.

Like variable names, catch names have dynamic scope: the program can throw to a catch from any of the subroutines called while inside the catch. This is important because ease of extension dictates that each command which the command-reading loop understands be implemented by a separate function, so that the user can redefine one command without replacing the framework of the loop.[6]

---

[6]Normally the command reading loop uses the name of the command to compute the name of the function to call. For example, if RMAIL reads the letter N as a command, it calls the function **# RMAIL N**. This way the user can easily define new commands.

6. Self-Documentation and Extensibility

A complex program is much easier to learn if it can answer questions about how to use it. When the program is customizable, it is important for the answers to reflect any customization that has been done. The easiest way to do this is for questions to be answered based on the same tables and data structures that control the functioning of the system. In EMACS, these include the command dispatch table and the loaded libraries.

The most basic kind of question that a user might want to ask is, "What does this command do?" He can inquire about either a function name or a command character. A library contains a documentation string for each function in it, and this is used to answer the question. When the question is about a command character, the dispatch table is used to find the function object which is currently the definition of that character. Then the library system is used to find the name of the function, and then, from that, the documentation string.

The ability to ask what a certain command does, only helps users who know what commands to ask about. Other users need to ask, "What commands might help me now?" EMACS attempts to answer this by listing all the functions whose names contain a given substring. Since the function names tend to summarize what the functions do (such as "Forward Word" or "Indent for Comment") and follow systematic conventions, this is usually enough. The list also contains the first line of each function's own documentation, and how to invoke the function with one or two characters, if that is possible.

The documentation for a function is usually just a string of text, but it can also contain programs to be executed to print the documentation, interspersed with text to be printed literally. This comes in handy when the description of one function refers to another function which is usually accessed as a one or two character command. It is better to tell the user the short command, which he would actually use, than the name of the function which defines it. But exactly which command—if any—runs the function in question depends on the user's customization. What we do is to use a program, in the middle of the documentation string, which searches the dispatch table and prints the command which would invoke the desired function. Another application of this facility is for functions which simply load a library and call a function in it. The documentation string for such functions is a program to load the library and print the documentation of the function which would be called.

To help users remember how to ask these questions, we make it simple and standard. A special character, called the Help character, is used. This character is only used for asking for help, and is always available. Help is normally followed by another character which specifies the type of inquiry. If the user does not remember these characters, he can type Help again to see a list of them. To close the remaining loophole of confusion, EMACS prints a message about the Help character each time it starts up.

Help is also available in the middle of typing a command. For example, if you start to type the Replace String command and forget what arguments are required, type Help. The documentation of the Replace String function will be printed to tell you what to do next.

Because questions are answered based on the data structures as they are at the moment, many changes in EMACS require no extra effort to update the documentation. It is only necessary to update the documentation of each function whose definition is changed. The format for EMACS library source files encourages this by requiring a documentation string for every function, between the function name and its definition.

7. History

I began the development of EMACS in 1974 with an improvement to TECO: the implementation of the display processor and a command dispatcher with a small fixed set of commands. These were inspired by the editor E of the Stanford Artificial Intelligence Lab. They were not considered a new editor, but rather one new feature in TECO to join many existing features. The user would give the TECO command Control-R to enter display editing mode, whose commands were suitable only for making local changes to the file. He would exit display editing mode to do anything else.

But once display editing was implemented, it was fairly easy to allow commands to be redefined to call functions written in TECO. TECO already contained considerable facilities for text manipulation, I/O, and programming, so almost immediately many users began to implement large collections of editing commands, powerful enough to do every part of editing. One of the most popular of these systems was TECMAC. Others included MACROS, RMODE, TMACS, Russ-mode and DOC. The need to exit from display editing mode to use TECO directly became less and less frequent until new users no longer learned how.

But TECO was still missing many of the important control and programming constructs which allow programs to be readable and maintainable (for example, named functions and variables!). So the early TECO-based display editors were very hard to maintain. In 1976 the TMACS system experimented with adding named functions and variables, with good results limited by the inefficiency of implementing them with TECO programs. This inspired me to implement EMACS itself.

Writing EMACS involved simultaneously adding to TECO the features which make up the library system and self-documentation, which permitted a new readable programming style, and writing a new set of display editing commands using this style. The design for the commands themselves was based on examining the command sets of the many TECO-based editors for inspiration, and choosing commands so that the most common operations would take few keystrokes. The first operational EMACS system existed in late 1976.

Since then, development has proceeded steadily, most new code being written in TECO. New features are added to TECO itself only to speed up loops such as table searching and s-expression parsing, or to make possible new kinds of I/O or interface operations.

EMACS was developed on the Digital Equipment Corporation PDP-10 computer using MIT's own Incompatible Timesharing System. By 1977, outside interest in

EMACS was sufficient to motivate Mike McMahon of SRI International to adapt it to Digital's Twenex ("Tops-20") operating system. EMACS is now in use at about a hundred sites.

7.1. Successors of EMACS

Several post-EMACS editor implementations have copied from EMACS both the specific command set and user interface and the fundamental principle of being based on a programmable interpreter. The motivation for these projects was to transfer the ideas of EMACS to other computer systems. Two of them, now in use, are Multics EMACS, a Honeywell product, and ZWEI, the editor for the MIT Artificial Intelligence Lab Lisp machine.

Because EMACS supplied the implementors with a clear idea of what was to be implemented, their focus was on making the foundations clean. The essential improvement was the substitution of an excellent programming language, Lisp, for the makeshift extended TECO used in EMACS. Lisp provides the necessary language features in a framework much cleaner than TECO. Also, it is more efficient. A Lisp interpreter is intrinsically more efficient than a string-scanning interpreter such as TECO's, and Lisp compilers are also available. This efficiency is important not just for saving a few microseconds, but because it reduces the amount of the system which must be written in assembler language in order to obtain reasonable performance. This opens more of the system to user extensions. Another improvement has been in the data structure used to represent the editing buffer: Multics EMACS developed the technique of using a doubly-linked list of lines, each being a string. This technique is used in ZWEI as well.

Many other editors imitate the EMACS command set and display updating philosophy without providing extensibility. Despite that deficiency, and despite the greatly reduced set of features that results from it, these can be useful editors, though not as useful as an extensible one. For a computer with a small address space or lacking virtual memory, this is probably the best that can be done.[7]

The proliferation of such superficial facsimiles of EMACS has an unfortunate confusing effect: their users, knowing that they are using an imitation of EMACS, and never having seen EMACS itself, are led to believe that they are enjoying all the advantages of EMACS. Since any real-time display editor is a tremendous improvement over what they probably had before, they believe this readily. To prevent such confusion, we urge everyone to refer to a nonextensible imitation of EMACS as an "Ersatz EMACS".

---

[7] The standard EMACS system is bigger than the entire 64k-byte address space of the PDP-11, despite constant strenuous efforts to reduce its size. And TECO is equally large. The post-EMACS editors are even larger.

8. Conclusions

8.1. Research Through Development of Installed Tools

The conventional wisdom has it that when a program intended for multiple users is to be written, specifications should be designed in advance. If this is not done, the result will be inferior. The place to try anything new is in a research project which users will not see.

Some people know better than this, but they have been silenced.

The development of EMACS followed a path that most authorities would say is a direct route to disaster. It was the continuous deformation of TECO into something which is totally unlike TECO, from the typical user's point of view. And during the whole process, TECO and programs containing TECO were the only text editors we had on ITS.[8] Indeed, there are ways in which EMACS shows the results of not having been completely thought out in advance: such as, in being based on TECO rather than Lisp. But it is still reliable enough to be widely used and imitated. The disaster which would have been forecast has not occurred. Instead, a new and powerful way of constructing editors has been explored and shown to be good.

I believe that this is no accident. EMACS could not have been reached by a process of careful design, because such processes arrive only at goals which are visible at the outset, and whose desirability is established on the bottom line at the outset. Neither I nor anyone else visualized an extensible editor until I had made one, nor appreciated its value until he had experienced it. EMACS exists because I felt free to make individually useful small improvements on a path whose end was not in sight.

While there was no overall goal, each small change had a specific purpose in terms of improving the text editor in general use, and each step had to be individually well designed and reliable. This helped to keep things on the right track. Research projects with no users tend to improve the state of the art of writing research projects, rather than the state of the art of writing usable system tools.

The individual commands of EMACS benefited from a stage of unregulated experimentation also. When the display processor and the capability for extension were created, many users began to write extensions, which developed into the complete editing environments of which EMACS is the most recent. Each command in EMACS benefits from the experimentation by many different users customizing their editors in different ways since that time. This experimentation was possible only because a programmable display editor existed.

New implementations of EMACS can now be carefully designed, because they have the advantage of hindsight based on the original EMACS. However, the implementor must carefully restrict his careful design to the parts of the editor that are already well understood. To go beyond the original EMACS, he must experiment.

But why isn't such a program of exploration doomed to be sidetracked by a blind

---

[8] The Incompatible Timesharing System.

alley, which will be unrecognized until too late? It is the extensibility, and a flexibility of mind, which solves this problem: many alleys will be tried at once, and blind alleys can be backed out of with minimal real loss.

8.2. Lisp is Loose!

The traditional attitude towards Lisp holds that it is useful only for esoteric amusements and Artificial Intelligence. The appearance of Multics EMACS as a Honeywell product is the death knell of this view. Now, a mainframe manufacturer is offering a system utility program written in Lisp; a program intended for heavy use by the general user community. The special properties of Lisp, which make extensibility possible, are a key feature, even though many of the users will not be programmers. Lisp has escaped from the ivory tower forever, and is a force to be reckoned with as a system programming language.

8.3. Blue Sky

The programmable editor is an outstanding opportunity to learn to program! A beginner can see the effect of his simple program on the text he is editing; this feedback is fast and in an easily understood form. Educators have found display programming to be very suited for children experimenting with programming, for just this reason (see LOGO).

Programming editor commands has the additional advantage that a program need not be very large to be tangibly useful in editing. A first project can be very simple. One can thus slide very smoothly from using the editor to edit into learning to program with it.

When large numbers of nontechnical workers are using a programmable editor, they will be tempted constantly to begin programming in the course of their day-to-day lives. This should contribute greatly to computer literacy, especially because many of the people thus exposed will be secretaries taught by society that they are incapable of doing mathematics, and unable to imagine for a moment that they can learn to program. But that won't stop them from learning it if they don't know that it is programming that they are learning! According to Bernard Greenberg, this is already happening with Multics EMACS.

9. Appendices

9.1. Display Processing

The way EMACS records what remains on the screen, and compares it with what is now in the text being edited, is determined by the representation used for that text. The post-EMACS editors use better text representations that make for easier display updating algorithms.

The representation used in EMACS is a straightforward linear string of characters. A movable gap which can grow and shrink makes it unnecessary for insertion and deletion within a small region of the file to move half of the file up and down. The gap was essential in making it practical to insert characters one at a time, instead of en masse in an "insert" command, but aside from that it is made invisible at all but the lowest levels of software, so essentially the representation is just a linear string. It is the task of the display processor's auxiliary data to make sense out of the amorphous mass of text.

The lowest level of avoiding wasteful output is a checksum of the characters displayed on each line of the screen. If a screen line is about to be rewritten, the new and old checksums are compared. If they match, the rewriting is skipped. Once in every 2^{36} times this will leave old incorrect text on the screen.

Higher levels of display optimization work by preserving information which is a byproduct of writing the display—namely, where in the text string the beginning of each screen line comes—and combining it with information which localizes the regions of the text string in which alteration has taken place. This allows it to restrict display update processing to a horizontal band of screen which contains all the necessary changes (often just one line). While processing the other lines on the screen would do no actual output, because of the checksums, even the time to compute the checksums is noticeable to the user as a delay. The same information can be used to decide when some lines on the screen should be moved up or down. When lines are inserted in the middle of the screen, it is much better to scroll the following lines downward (if the terminal can do this) than to rewrite them all in their new positions.

The record of where in the text string changes have taken place is maintained by requiring every command to return values saying what part of the string it has changed. It can identify a subinterval of the string which contains all the changes made, it can say that no change was made (though the cursor may have been moved), or it can say nothing, which requires the display processor to make no assumptions.

A better way, developed by Bernard Greenberg in Multics EMACS and used in ZWEI, is to represent the buffer as a doubly-linked list containing pointers to strings, one for each line. Newline characters are not actually present, but implicitly appear after each line except the last. This requires the lowest level insert, delete and search subroutines to be more complicated (for example, inserting a string cannot treat Newline characters like other characters), but this is just a finite amount of complexity; and it greatly simplifies efficient display computations. The state of the screen can be remembered in an array of pointers to the string that was displayed on each screen

line. When the display is updated, one can compare the strings in the buffer with the strings in the display, both to see whether they are the same objects (the pointers are equal; EQ, in Lisp), and to see whether their contents are the same.

Multics EMACS never changes the contents of a string in the buffer. It creates new strings to replace the old ones when the text changes. Thus, the string pointers in the screen state continue to record the screen as it was.

ZWEI does change the contents of existing strings. To make sure that it does not fail to notice that the text no longer matches the screen, ZWEI maintains a "clock" which increments each time a change is made in the text. Each line records the clock tick of the last modification. Each screen line records the clock tick as of the time it was displayed. If the line in the text matches the line in the screen record, but the tick counts do not match, then the contents of the line have been changed.

Line list representations also eliminate the requirements on commands to say what they have changed. Reducing the need for the programmer to worry about how display will be done is very desirable. Another advantage is that it becomes feasible to have pointers to characters in the text which relocate when insertions or deletions are done, so that they continue to point to the same place in the text.

9.2. Libraries

An EMACS sharable library contains, first of all, a symbol table which can be binary searched for the name of an object to find the object named. The symbol table points at both the names and the definitions using offsets from the beginning of the file, so that the file can be valid at any location in memory. The names and definitions are all examples of the TECO string data type, in the internal TECO format, so that the library does not need to be translated or parsed in any way when it is loaded.

The symbol table points to the documentation of functions in the library as well as their definitions. The documentation for the function **Visit File** is an object entered in the symbol table with the name **~Doc~ Visit File**. There is also a string named **~Directory~** whose definition contains a list of the names of all the objects in the file which the library wishes to advertise. This is used for documentation purposes, not for looking up names, and it does not contain names of auxiliary objects such as **~Doc~ Visit File** or **~Directory~**.

It is possible to search the symbol table in reverse, to take a definition and find its name. Since one can tell which library an object is in by comparing its address with the range of memory occupied by the library, this makes it possible to find the name of any object which has one. The ability to do this is important, because when the user asks what the character Control-K does, it is desirable to be able to tell him that it runs the function **Kill Line**. The names themselves are not kept in the dispatch table because looking up a name in the loaded libraries is slow. For other implementations, that is a reasonable strategy.

10. Notes

10.1. EMACS Distribution

EMACS is available for distribution to sites running the Digital Equipment Corporation Twenex ("Tops-20") operating system. It is distributed on a basis of communal sharing, which means that all improvements must be given back to me to be incorporated and distributed. Those who are interested should contact me. Further information about how EMACS works is available in the same way.

10.2. Further Information

A complete manual for use (but not extension) of EMACS is

Richard M. Stallman, EMACS Manual for ITS Users, Artificial Intelligence Lab memo 554, 1980.

Richard M. Stallman, EMACS Manual for TWENEX Users, Artificial Intelligence Lab memo 555, 1980.

Various lower level implementation strategies for parts of an EMACS-like editor are treated in

Craig A. Finseth, Theory and Practice of Text Editors, or, A Cookbook for an Emacs, L.C.S. Technical Memo TM-165, B.S. Thesis, May 1980.

10.3. EMACS-related Editors

These include the true extensible descendents of EMACS, and the editors which preceded EMACS and supplied some of the ideas for it. The many ersatz EMACS editors are not included.

Multics EMACS

Multics EMACS was written in MacLisp by Bernard S. Greenberg of Honeywell's Cambridge Information Systems Lab, starting in 1978. When first implemented, it could be used only by its author, because he alone had the necessary privileges to patch the Multics operating system so that a program could read one character from the keyboard instead of waiting for a complete line. After seeing the new editor in operation, the other Honeywell people were convinced to make the feature generally available. Because it is written in Lisp, Multics EMACS is even more extensible than the original EMACS, and as a result it has accumulated even more powerful features.

Bernard S. Greenberg, Multics Emacs: an Experiment in Computer Interaction, in proceedings, Fourth Honeywell International Software Conference, Bloomington, Minn., April, 1979

Bernard S. Greenberg, Prose and CONS (Multics Emacs: a commercial text processing system in Lisp), in proceedings, 1980 Lisp Conference, Stanford University, Stanford, California, August 1980.

Bernard S. Greenberg, and Katie Kissel, Multics Emacs Text Editor User's Guide, Publication # CH27, Honeywell Information Systems, Waltham, Mass., 1979

Bernard S. Greenberg, Multics Emacs Extension Writers' Guide, Publication # CJ52, Honeywell Information Systems, Waltham, Mass., 1980

SINE

SINE ("SINE Is Not EMACS") is based on compiling Lisp code to run in a non-Lisp editor environment, in which, unfortunately, no interpreter is present. However, the user can load his own compiled files into a running editor. This design was chosen because of the small address space of the machine, an Interdata at the MIT Architecture Machine Group. See

Owen T. Anderson, The Design and Implementation of a Display-Oriented Editor Writing System, Undergraduate Thesis, MIT Physics Department, January 1979.

TECMAC

TECMAC was the first editor implemented in TECO to work with the display processor. It developed many of the ideas used in the EMACS user interface. It was retired because, written when TECO was less suited to system programming, it was unable to attain either readability or efficiency. TECMAC was maintained from 1974 to 1976 by John L. Kulp and Richard L. Bryan.

TECO

PDP-10 TECO was originally written by Richard Greenblatt, Stew Nelson and Jack Holloway at the MIT Artificial Intelligence Lab, based on PDP-1 TECO which was written by Murphy in 1962. The TECO in which EMACS is implemented is its direct descendant. The PDP-10 TECO from Digital, a typical example of TECO, is also a descendant of an early version from MIT. It is documented in

Digital Equipment Corporation, Decsystem-10 TECO Programmer's Reference Manual, DEC-10-ETEE-D (revised from time to time).

Ordinary TECO lacks many important programming constructs. In MIT TECO, the constructs may be syntactically ugly, but they exist. So programs can be well organized, and clean except in the lowest level of detail.

TMACS

TMACS was an editor implemented in TECO which began to develop the idea of the sharable library with commands that could be assigned to keys by the user. TMACS was the project of Dave Moon, Charles Frankston, Earl A. Killian, and Eugene C. Ciccarelli. Interestingly, it had no standard command set. The implementors were unable to agree on one, which is what motivated them to work on making customization easier.

ZWEI

ZWEI ("ZWEI Was EINE Initially") is the editor for the Lisp machine. EINE ("EINE Is Not EMACS"), the former editor for the Lisp machine, was also based on EMACS; it was operational for late 1977 and 1978, and was redone to make it

cleaner. Both EINE and ZWEI are primarily the work of Daniel Weinreb and Mike McMahon; see

> Daniel L. Weinreb, A Real-Time Display-Oriented Editor for the LISP Machine, Undergraduate Thesis, MIT EECS Department, January 1979.

10.4. Other Interesting Editors

Augment

Augment (formerly known as NLS) is a display editor whose interesting feature is its ability to structure files into trees. Making the tree structure useful required the concept of the viewspec, which specifies that only certain levels in the tree structure will be visible. This is the sort of feature which cannot be added by a user to EMACS, because it involves modification of the display processor; but it could be added by a user to Multics EMACS or ZWEI. Augment popularized the graphical input device known as the "mouse", which is a small box with wheels or balls on the bottom and buttons on the top, which the user moves on the table with his hand. This device has been copied widely because of its simplicity and low cost. Augment was designed at SRI International but is now supplied by Tymshare. See

> Douglas C. Engelbart and William K. English, A Research Center for Augmenting Human Intellect, AFIPS Conference Proceedings, Vol. 33, Fall Joint Computer Conference, San Francisco, December 1968, pp. 395-410.
>
> Patricia B. Seybold, TYMSHARE'S AUGMENT—Heralding a New Era, The Seybold Report on Word Processing, Vol. 1, No. 9, October 1978, 16 pp. (ISSN: 0160-9572), Seybold Publications, Inc., Box 644, Media, Pa 19063.

Bravo

Bravo comes from the Xerox Palo Alto Research Center. Its orientation is toward text formatting, and it can display multiple fonts, underlining, etc. It makes heavy use of a graphical pointing device, the "mouse" (see Augment). It is not programmable and offers no special help for editing programs as opposed to text. For more information, see your local industrial espionage agent.

E

The editor used at the Stanford Artificial Intelligence Lab, E interfaces with a "line editor" (used to edit within a line, on a display terminal) which can also be employed to edit the input to any other program. The line editor does not allow commands to be redefined; since it is part of the timesharing system, that is not trivial (though possible in principle). E allows macros to be written using the same language used for editing. These are as powerful as a Turing machine, and as easy to program with. See the on-line documentation file E.ALS[UP,DOC] of the Stanford Artificial Intelligence Laboratory.

TRIX

TRIX is a language similar to TRAC designed at Lawrence Livermore Lab specifically for writing editors. It has been used to write commands that are

specific to particular languages, and to write text formatters. Its fatal flaw is that it was designed for printing terminals. See

Cecil, Moll and Rinde, TRIX AC: A Set of General Purpose Text Editing Commands, Lawrence Livermore Lab UCID 30040, March 1977.

TVEDIT

TVEDIT is a distant relative of E (above) which is used at Stanford on the Twenex and Tenex operating systems. These systems do not provide a line editor, so TVEDIT has its own facilities for changes within lines. TVEDIT is a good example of a generally reasonable but nonprogrammable display editor. See

Pentti Kanerva, TVGUID: A User's Guide to TEC/DATAMEDIA TV-Edit, Stanford University, Institute for Mathematical Studies in the Social Sciences, 1973. (Online document)

10.5. Other Related Systems

The Lisp Machine

The MIT Artificial Intelligence Laboratory has built a machine specifically for the purpose of running large Lisp programs more cheaply than ever before. One of its goals is to make the entire software system interactively extensible by writing it in Lisp and allowing the user to redefine the functions composing the innards of the system. Part of the system is an EMACS-like editor (ZWEI; see above) written entirely in Lisp, which shares in this extensibility. See

Daniel Weinreb and Dave Moon, The Lisp Machine Manual, MIT Artificial Intelligence Laboratory.

LOGO

LOGO is a language used for teaching children how to think clearly. Unlike conventional computer-aided instruction, which automates a method of teaching which offers little to motivate the student, LOGO invites students to write programs to produce interesting pictures and learn while doing something fun. See

Seymour Papert, Teaching Children to be Mathematicians vs. Teaching About Mathematics, MIT Artificial Intelligence Laboratory Memo 249, 1971.

MacLisp

The MacLisp language is very suitable for writing extensible interactive programs, and has been used for the implementation of Multics EMACS. See

Dave Moon, MacLisp Reference Manual, MIT Laboratory for Computer Science, 1974.

Smalltalk

The Smalltalk language and system is oriented toward writing extensible programs.

Dan H. H. Ingalls, The Smalltalk-76 Programming System Design and

Implementation, in proceedings, Fifth Annual ACM Symposium on Principles of Programming Languages.