MASSACHUSETTS INSTITUTE OF TECHNOLOGY
ARTIFICIAL INTELLIGENCE LABORATORY

AI Memo No. 519 February 9, 2022

EMACS
The Extensible, Customizable, Self-Documenting Display Editor

by
Richard M. Stallman

Abstract:

EMACS is a display editor which is implemented in an interpreted higher level language. This allows users to make extensions that fit the editor better to their own diverse applications, to experiment with alternative command languages, and to share extensions which are generally useful. The programming system used for the implementation has several features which serve directly to make extensions simpler and easier to write, and to facilitate sharing them. The user extensions, and the results of their experiments, have been used to improve the basic EMACS system, which has thereby become itself more sophisticated and powerful than a nonextensible editor can easily be made. This paper will advocate organizing editors in the EMACS fashion, by first summarizing the EMACS system as it presents itself to the user, and how extensibility helped it to be written, and second discussing the programming system features which make EMACS easy to extend.

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

This report describes research 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 N00014-75-C-0643.

Preparation: Real-Time Display Editors

A growing number of interactive computer systems are now moving on from printing terminal oriented line-replacement editors to display editors which show the text being edited at all times and allow the user to move the cursor and make changes at the cursor with single-character commands. 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 is because editing on a printing terminal requires a mental skill like that of blindfolded 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" (though there is still room for the expert user to plan ahead mentally).

A good display editor cannot be just a printing terminal editor modified to update the display after each line or unit of user input. This fails to take full advantage of the display. For example, a printing terminal editor usually has an insert command which takes many characters and inserts them all at once. To update the display only after the entire insert command is not maximally helpful, since the display terminal makes it possible to show each character inserted in its proper place in the text as soon as it is typed by the user. Giving the user feedback more frequently makes editing easier. The lesson is that a display editor should have (primarily!) short, simple commands that are visible in the display as soon as they are typed. We call this "real-time" editing.

The simplest real-time editors are the local editing modes of so-called intelligent terminals: arrow keys are provided to move the terminal's cursor to any point, and text can be inserted where the cursor is just by typing it. There will also be keys that can erase various portions of the screen. The EMACS user can position the cursor to the place to be changed moving vertically or horizontally, by searching for a string, or by moving across words, sentences, paragraphs, expressions, pages, etc. Then the user can type text to be inserted or use nonprinting character commands to delete or alter text. Text (printing characters and Carriage Returns) is inserted just by typing it; other commands are nonprinting characters, or begin with nonprinting characters. Only many-character commands echo; simple commands are acknowledged by displaying their effects. This helps the user feel the editor as an extension of his own fingers, and feel that he is touching the text directly rather than through an intermediary.

However, while real-time editing is an important factor in the popularity of EMACS, it is not original with EMACS. The way in which EMACS advances beyond other real-time display editors is that it contains an interactive program development system which made it easier to write in the first place, and enables

any user to add to and modify the editor as he pleases.

Why Extensibility?

The display editor is the best sort of editor known, but no one display editor is ideal for all purposes. Each programming language, each format of text being edited, offers scope for special editing commands that are particularly useful for the syntax of that language or that format of text. Nor is the ideal display editor for one user the same as the ideal for another user. Each user has his own preferred style, which determines which commands and sequences he will want to use. Only by being extensible can an editor offer a user even the possibility of approaching his ideal. This is what EMACS does. EMACS users frequently customize their environments to their personal taste, and also frequently write new commands of general usefulness and share them with other users.

The simplest kind of customization, rearrangement of the commands, is by itself very useful. One person on first learning about EMACS considered it a great deficiency (relative to another display editor) that commands for moving up, down and sideways were on characters that could not be typed with one hand (these characters were chosen, instead, to be mnemonic). Such a problem is of no intellectual interest, but that does not reduce its practical importance to a person who wants to operate in that way, nor does it make the difficulty any less insurmountable in the absence of extensibility.

Another sort of customization is making old commands behave a little differently. For example, some users prefer to have a command to move to the beginning of the next line rather than a command to move vertically down.

There is no sharp boundary between this sort of customization and significant extension. Users who have unusual programming languages have written command sets just for editing them. A command can be written for almost any sequence of actions which is frequently performed. One can also reorganize the entire system of commands. Some users like a two-dimensional command organization wherein one character selects the syntactic unit and another selects the operation to be performed on it. Implementing this is not difficult.

But user extensions do not help only the users who write them. They have also played an important role in the development of the core system as provided, before extensions, to users. The most straightforward way in which they contribute is that some important parts of the core system are actually well-publicized user extensions. EMACS contains a library system designed to help users make their extensions available to others. 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. This testing ground enables development to proceed more quickly by allowing more people to contribute their effort. The extra input from users also helps development stay responsive to them.

In short, if an editor designer has the hubris to assume he can design an editor which is right for everyone and all applications, and wishes to do all the work himself, he should omit extensibility.

The Organization of the EMACS System

It has been a long time since programmers first realized that they wanted the power of a programming language while editing. Editors such as TECO attempted to provide this power by including programming constructs such as conditionals, iteration, and arithmetic in the editing language. This was a natural idea, because it allows the editing operations in a program to be requested with the same syntax that would be used during ordinary interactive editing.

However, this technique always leads to a language which, as a programming language, is ugly, hard to read, and grossly inefficient—bad by every criterion used to judge programming languages. TECO is a good example of this. A good interactive editing language is composed primarily of single-character commands which act immediately when typed, with a few commands that escape to longer sequences for less frequently used commands. If the editor is to be customizable, the user must be able to redefine each character. This in a programming language would be intolerable!

EMACS rejects that approach and consists of two separate languages, an editing language, optimized for interactive use, with which editing is normally done, and a programming language, oriented toward writing the definitions of the commands of the editing language. These two languages are completely distinct so that each one can serve its own purpose better.

The interpreter underlying EMACS is specifically slanted toward editing, with primitive operations for a text-buffer data type (insertion, deletion, searching, etc). However, it also provides a full range of the usual features of higher level languages, including some error handling and symbolic variable features that are new and interesting. The interesting semantic aspects will be discussed below.

In designing the underlying interpreter it is most important to start with one which is a good programming language generally, since it is easier to add the special features for the editing application than it is to make a bad programming language good. This has been learned by harsh experience: EMACS did it the

hard way, and its successors are now doing it the easy way.

Aside from the interpreter and the editing commands written in it, EMACS includes a display processor whose responsibility is to maintain on the screen a window into the selected text buffer, a command dispatcher which reads the user's editing commands and invokes their definitions (also maintaining state, such as numeric arguments, from one command to the next), and a library system which allows interpreted functions to be grouped, together with their names and documentation, into library files that can be quickly loaded and shared between users.

How EMACS Extensibility Benefits the Naive Non-Programmer User

Some extensible systems are presented to the user as collections of building-blocks, requiring every user to build his own user interface before he can use the system at all. This is fine for the sophisticated user but bad for the novice. However, this approach is not a necessary concomitant of extensibility. EMACS uses the ease of development which comes with extensibility to provide a richer and more usable initial core to the naive user than most other editors do, and with less effort on the part of the implementors. To make this concrete, here are some of the features, available to all users without further extension, that extensibility made it easier to provide.

The most basic EMACS editing commands are similar to the commands of other display editors. They do things like moving forward and backward by characters, up and down by lines, to the beginning or end of line, and deleting characters forward and backward. Another command, that is not usually thought of as one, is "self-insert". This is the usual definition of all graphic characters, so that text can be inserted at the cursor just by typing it (existing text following the cursor is shoved over to the right, not destroyed). More powerful operations which other editors are also capable of include killing and moving stretches of text, scrolling by screenfuls or by lines, searching for strings, and global replacements.

Beyond this point the power of EMACS's extensibility begins to be felt. EMACS can be programmed to understand the syntax of the language being edited and provide operations particular to it. A set of "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 variables, 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, `-*-Text-*-` anywhere in the first nonblank line of a file says that the file should be edited in Text mode.

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, delete or move font-change commands.

Many commands are controlled by the values of variables 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 one interactive one for the user to use. The paragraph commands can be told which strings, appearing at the beginning of a line, constitute the beginning of a paragraph. All of these variables 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.

Redefining Ordinary Characters

A very powerful extension facility is the ability to make ordinary graphic and formatting characters, which normally when typed just insert themselves into the text being edited, instead run an arbitrary function. The function usually will 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", which was the first editing command ever written using the interpreter. It is a program, intended to be used as the definition of the space character, which inserts a space and then breaks the line into two if it has become too long. With this redefinition of the space character, the user can ignore the right margin and never needs to type the Return key. 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.

What one user did write was an abbreviation facility, which allows the user to type an abbreviation which will automatically expand into another string. For example, if "cd" were defined as an abbreviation for "command", you could type "i/o-cd" and would see "i/o-command" appear in the buffer. "Cd" would turn into "Command". This facility works by redefining all punctuation characters (the list of which can be altered by the user) to run a program which thinks about expanding the preceding string. (When this mode is enabled together with auto fill, space both fills and expands.) Abbreviations can be defined so as to be in effect only for a certain major mode. Besides the function which implements

expansion, the abbreviation library contains commands to add and undo abbreviation definitions, list them all, save them in a file and reload them, unexpand an abbreviation if the user didn't really want it to expand, and many others. All these commands are packaged in one sharable library file which any EMACS user can load at any time. The user-author was able to implement this with no modifications to the standard installed EMACS system.

Editing Programs

Interpreter power is even more useful for editing programs than for editing English text, because programming languages have more simple syntactic regularities that are easy to exploit. Clearly there can be commands to move over expressions, move to the beginning or end of a function definition, insert and align comments, etc. But the most important operation, and the first one to be implemented for any new language, is automatic indentation.

Automatic indentation or "pretty printing" has been in use among Lisp programmers since the earliest times, and has even been used for PL/I on Multics (probably because of the proximity of the Multics developers to a community of Lisp users). Traditionally, this is done with a program that reads in a source file, reformats it completely, and writes it out again. Every line is reindented by the pretty-printer in its own favorite style; it probably also moves code from one line to another at its own whim. Every vestige of the previous formatting is lost. Such pretty-printers are very slow, because they have the last word on all formatting, and must therefore have extremely clever aesthetic heuristics if they are to be useful. But none is so good that it avoids generating formats that look ugly to some users.

In EMACS, indentation is conventionally performed by the Tab character as a command. Tab's meaning is redefined by each major mode (there is one for each programming language) to be an indenter suitable for that language. Linefeed is also an indenting command; it is defined to insert a line-separator and then do Tab on the new line. The user can type in correctly indented code simply by using Linefeed instead of the Return key. Linefeed can also be used to break an existing line. When he makes corrections to code, he can reindent lines as necessary using Tab.

The advantages of moving indentation into the editor are twofold. First, the programmer can see the code properly indented while he is typing it in. With a conventional pretty-printer, he would have the choice of typing it in unindented before the first pretty-printing, or of generating the initial indentation himself. Second, because the user rather than the indent command has the final say, it is not necessary for the automatic indenter to be a paragon of aesthetic perfection. The user can make the indentation facility his habitual first recourse, and

manually fix any lines for which he does not consider it optimal. When the user fixes manually a line at one level of nesting, following lines at the same level or deeper levels will be indented (if Tab is invoked on them) with respect to the indentation as set by the user.

The reason why an explicit convention—that indentation is done by Tab—is needed is that each programming language needs its own indentation function. When the user enters the major mode for PL/I files by invoking the function PL1 Mode, this function obeys the convention by installing the indentation function for PL/I on the character Tab. The other half of the same convention is that any function which needs to indent one or more lines does so using the current definition of Tab. It is important that conventions of this sort be established and made public. For example, the function PL1 Mode was written as an extension by a user, who had to know of the convention.

When the programming language being edited is Lisp, with its ultimate simplicity of syntax, understanding it becomes even easier. Commands to move up and down in the list structure, and to move over and delete expressions, make it possible to edit code just by thinking of its nesting. The most difficult thing about writing correct Lisp code—the balancing of parentheses—is made easy by an alternate definition for the ")" character which moves the cursor to the matching "(" momentarily and then back again. The expression motion commands also help, and asking the editor to reindent the function is a good way to see how it balances. This way, EMACS offers most of the benefits that usually come from editors which operate directly on list structure, but in conjunction with the benefits of display editing; and the implementor need not write, nor need the user learn to use, a totally separate editor for his Lisp programs.

Editing Large Programs

Large programs (such as EMACS) 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.

To use TAGS, a separate program is run, given a list of files, to produce a file called a tag table which contains a list of all the functions in all of the files, sorted by file, each with its location in its file. The language of each file is also supplied so that it can be parsed properly. Once this is done, the tag table is loaded into EMACS. A special command provided by the TAGS package can find any function quickly by finding it in the tag table, determining which file it is in, selecting that file, going to the recorded location of that function's definition, and then searching the vicinity for the definition. First small intervals

and then larger intervals are searched, so that the function has to be found if it is still in the file; but if things have not changed too much since the tag table was made, it will be found much more quickly than by searching from the top of the file.

Editing Other Things

The RMAIL extension package is for editing computer mail. RMAIL loads the whole file of incoming mail but displays only one message at a time. Commands are provided for moving from one message to another and deleting a message. Sending a reply automatically initializes it with recipients and a subject extracted from the message being replied to. The reply is edited with the same commands used for all other editing.

DIRED enables a user to edit his file directory. He is shown a listing of his directory, in which he can move from one file to another with the usual cursor-motion commands, but other commands are provided to move, examine, compare and delete files. Commands are also available to find files which appear to be deletable (for example, old versions of programs, and temporary files) and mark them, tentatively, to be deleted later. DIRED has been implemented mostly by users.

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 finite, 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 editor's own cursor in the file 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 or causing the text where the cursor is to move to a specific line.

Unfortunately, to obtain reasonable performance, the EMACS display processor has to be coded in assembly language. This is unfortunate because extensions to the display processor could be very valuable. In the descendants of EMACS, the display processor is written in the same high-level language as the editing commands, and can be extended. However, the EMACS display processor embodies an unusual principle which makes for much faster responsiveness 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 the user's further commands, already buffered by the operating system.

To understand the problem, imagine that the terminal being used does not have the ability to insert or delete lines on the screen. If a Carriage Return is typed and a new line created, all the lines below that point need to be displayed again. If, while that is still going on, the user types an additional Carriage Return to create another new line, the rest of that 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. While this particular sequence of events poses no problem on terminals which can insert and delete lines, other sequences of events continue to do so. The EMACS display processor is able to avoid this waste.

The EMACS display processor is best understood as being a separate, lower priority process that runs in parallel with the editing process (this is not how it is implemented). The editing process reads keyboard input and makes changes in the editing buffer. The display process is always trying to change the screen to match the editing buffer; 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, with a speed limited by the terminal baud rate, a target which the editing process can move arbitrarily. Actually, since process-switching takes place at only at a few well defined places, it is easy to simulate multiprocessing by polling for input when it is safe to suspend display.

One consequence of EMACS's input-before-output philosophy is that EMACS uses less computer resources on a heavily loaded system. 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.

The Library System and the Command Dispatcher

Sharing of user extensions is made possible by the library system. 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.

For the sake of uniformity, the standard EMACS functions also reside in a library, which is always the first one loaded. When a function name is looked up, all the loaded libraries are searched, most recently loaded first. 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 a vector of definitions to find the function to be called (the definition-object, not the name). Functions residing in the command vector can be invoked either by the character command or by name. Since users often wish to connect command characters to functions which were not necessarily intended to be invoked other than by name, the 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 read the argument from the terminal instead (escaping to an interpreted function which, in standard EMACS, is defined to provide editing of the argument).

Some libraries contain functions that are intended to be called with single-character commands. Such functions can be placed in their slots in the command dispatch vector when the library is loaded, if the library has a function named Setup. Such a function, if it exists, is called automatically when the library is loaded. 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 using 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.

One of the functions invokable from the command dispatcher is one which reads in the name of a function and calls it. This is how the user invokes functions which are not assigned to any command character.

Documentation Features

An editor which makes itself as easy to change and add to as EMACS must provide better than the usual in the way of integrated on-line documentation, or "help" features, and EMACS does. The EMACS help features can use the command dispatch vector and the loaded libraries as their data base, thus automatically reflecting the state of customization in effect when they are invoked.

The most obvious form of help feature is to be able to ask what a command does. Many systems provide such features. EMACS provides commands to describe the action of either a character command or a function specified by name.

A more important help feature is the ability to ask what is available. A mere list of all commands would usually be no use because it would be too long. The EMACS command `Apropos` performs a substring retrieval on all the function names in all the loaded libraries. For example, `Apropos` of "Paragraph" would tell about all the commands for manipulating paragraphs and how to invoke them. For each function containing the substring, one line is printed which contains the function name and a brief statement of what it does. The function definition is then obtained and the command dispatch vector searched to see if any character commands will invoke the function. If any are found, the user is told what they are.

Another important help feature is a record of the last 60 characters of keyboard input. This is useful when the user types something by mistake and sees surprising things happen to his buffer.

All of the help features are centered on a single character, called the Help character, which can be typed at any time to ask for assistance. It offers help which depends on what you are doing at the moment. For example, if Help is typed while the arguments to the `View File` command are being read in, it will print a description of the `View File` command, including the sort of arguments it needs. It does this by examining the function call stack, finding the name of each function on the stack, looking for one which appears to be what the user invoked (as opposed to subroutines used by it, which are seen first), and retrieving its documentation. If Help is typed when a command is expected, it offers the options described above for asking about commands.

When a system is customized by an individual it is a frequent problem to find that the documentation has not been updated. The organization of the EMACS help features automatically takes care of this. Because all of the help commands use as their data base the same tables used by execution, they all reflect the actual state of affairs, including any customizations performed by the user. If the user writes functions of his own, he is responsible for writing the documentation, but the format of source files for EMACS libraries encourages this. If he simply moves commands from one character to another, everything is automatic.

The Programming Language in EMACS

The programming language in which EMACS is written has several unusual features which are specifically important for convenience in writing editing commands and making them reliable. Actually, they are not specific to editing so much as to complex extensible interactive systems, which most programming languages are not designed for. They aid by making it easier to decompose the system into small, modular pieces so that a desired extension can be accomplished by replacing only a few of them. The MacLisp language and system have often influenced and been influenced by developments in EMACS programming constructs, Lisp being one of the few other languages to address the same issues.

The syntax and detailed definitions of these constructs are far from ideal, for historical reasons, so they will not be mentioned.

Free Variables

Most programming languages provide for names of variables to be known only at compile time. Compilation makes all references to the variable refer to the same location in memory, and from then on the name is superfluous (assuming the programmer is superhumanly perfect). A static scoping rule is imposed so that all possible legal references to a variable are guaranteed to be known during the one compilation.

While EMACS functions have use for such variables, they also require another kind: one which serves as a global switch which the user can set to control the behavior of one or more functions, which refer to the variable "freely" (as if it were defined in an enclosing block). For example, the commands for manipulating comments expect the variable `Comment Start` to have a value which is a string to be used to mark the start of a comment. The names of such variables must be remembered at run time so that the user can set them. A function called `Alter Options` exists which allows the user to edit a list of the

names and values of variables. When the user says he is finished, the values are set the way he edited them. This relies on the fact that all variables can be found in a single symbol table. In addition, EMACS provides for giving variables documentation strings as well.

Some variables come with EMACS when it is started. Others are created by libraries of extensions to hold their own data structures. Yet others are looked for by extensions; they are to be set by the user, as part of his customization, to be seen by those libraries in case he should decide to load them.

Even when a variable is intended to be set by one command and used by another, or by a later invocation of the same command (like OWN variables in ALGOL), and never seen by the user, it is still important that the name be kept at run time, because the user may wish to modify the command, or to define a new one which should, logically, use the same variable. For example, the variable Auto Save Mode is set by the functions that select a file and looked at by others to decide whether and how to save changes automatically on disk. The user should never deal with it explicitly himself. But if he would like to implement a different kind of auto-saving (as some users have) he needs to be able to make his functions refer to that variable. Otherwise, he'd have to replace the file selection commands as well. Even if he were satisfied with the usual ones, he would have to make copies of them to compile with his new auto-save commands. Then he would not have the benefit of bug fixes and new features in the standard file-selection commands.

The standard practice of keeping variable names only at compile time may be satisfactory when a program is to be compiled completely before any of it is run, but it is unsuitable for a system which can be extended while it is running.

Dynamic Binding

This use of variables creates a need for another language feature found normally only in APL and Lisp: dynamic binding. What this means is that a function can make a new, local binding for an existing variable, which will hide the old, outer binding until the function returns. In the meantime, if the function calls other functions which refer to the variable freely, they will see only the new, inner binding.

How is dynamic binding useful? Suppose a function wants to offer the user some text to be edited for a special purpose, and it is known that, while editing that text, it would be convenient for the "*" character to be considered part of a word. This function can dynamically bind the syntax table which controls word parsing to a new value which marks "*" as alphabetic, and then call the editor command/display loop. This loop does not itself refer to that syntax table, but

if the user invokes any of the word commands (or anything else, perhaps written by him) which parses words and looks in the conventional place for the syntax table, then "*" will be considered part of a word.

Dynamic binding is also important—even more important—for the elements of the command dispatch vector (or, alternatively, for the vector as a whole). It is extremely common to offer the user text to edit and provide him with a few special purpose commands for doing so. For example, when a user is editing a reply to a message, he is given temporarily a special command to insert a copy of the message he is replying to. This is done by dynamic binding.

Some theorists believe that dynamic binding is "dangerous" and likely to lead to incorrect programs. They advocate explicit passing as parameters of all information that is going to be needed by a subroutine (dynamic binding is a sort of implicit parameter passing). There are some cases in which an explicit parameter might be better, such as for the command dispatch vector. However, the word syntax table example can be used to see how an insistence on only explicit parameter passing is not workable in an extensible system. Consider the consequences: it would be necessary for the editor command/display loop to have the word syntax table as an explicit parameter, even though it otherwise has no need to know that such a syntax table exists (IT doesn't parse words). What's worse, in order for the command dispatcher to pass the syntax table on to the commands that really use it, it must pass it as an explicit parameter to every command (it can't tell which ones are interested). So every command must explicitly expect to receive a word syntax table as a parameter—along with five dozen other such variables to which the same arguments apply. But further, suppose the user defines functions which refer to a new free variable which is also the user's own invention, and then goes on to define a function which wants to present text to be edited with this new variable temporarily altered in value. That is, he creates a new variable and wants to do to it exactly what was done above to the word syntax table. According to the explicit parameter passing discipline, he would have to make the command/display loop take this variable as an explicit parameter, and make all calls to that loop provide the parameter. What should have been a local extension of the system now requires a global rewrite.

Dynamic binding makes it possible to change the value of a variable "temporarily", and be sure that the old value will be restored if control passes out of the scope of the binding. A natural extension of this is the ability to change anything temporarily, not just the value of a variable. This uses a construct called "unwind-protect", which allows the programmer to provide an arbitrary computation to be performed when control passes out of the unwind-protect in any fashion. Explicit exit from the unwind-protect may or may not also perform that computation, at the programmer's option.

Variables Local to a File

Suppose you want a file to be formatted with comments starting at column 50. Clearly your task will be easier if the variable Comment Column, which is used (by convention) to decide where to align comments, is always set to 50 whenever you are editing that file. EMACS provides this feature, 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:  */
```

(Here, "/* " and " */" are arbitrary strings that could be replaced uniformly by any two others, or by nothing. Their purpose is to disguise the local modes for the compiler that will eventually process the text. These particular strings would be used if the file were a PL/I program).

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.

Variables That "Project"

Usually a global variable is checked explicitly each time it matters. Sometimes, in the interest of efficiency or modularity, it is better not to check the value each time but rather to have other data structures change automatically when the value of the variable changes. For example, changing the value of Auto Fill Mode to turn auto-filling on or off automatically redefines the Space character's command definition. It happens automatically because the variable Auto Fill Mode has been provided with a function to be run whenever the value is set. The function is interpreted, but the mechanism that decides to call it is part of the interpreter itself.

Another thing that Auto Fill Mode's function does is signal that the place on the screen which says whether auto-filling is in effect may need to be rewritten. This makes it unnecessary to keep testing the variable's value to see whether it is necessary to update the screen.

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). Sometimes a user error is detected only because it results in a program error, but it might also be caught by an explicit check which does not result (unless it wishes to) in a program error.

There are several responses to a user error that might be desirable. One is that the command simply does nothing. For some commands, this is comfortable for the user. Another is to ring the bell on the terminal. This is desirable when it is likely that the user is a little confused about the situation, but will probably understand as soon as he sees that all is not as he expected—for example, trying to delete a character backwards when at the beginning of the buffer. Pending keyboard input is usually thrown away when the bell is rung, but this can be explicitly inhibited. If it is necessary to give the user more information, the best way is to cause a program error with an explicitly supplied error message. A primitive exists just for this. Of course, these responses are possible only if the user error is detected explicitly. If a type of user error is deemed to deserve one of these responses, a conditional is installed. More obscure errors that only result in program errors can be left alone, according to how much effort the programmer wishes to invest.

Program errors are handled in a uniform manner, which is normally to print an error message and abort the program, returning control to the command dispatcher. However, it is necessary for this to be alterable by programming. When the user can write his own command loop, he must have a way to say that it should get control back after an error. This done with an "errset", a construct which is placed around an expression and catches any errors that occur within the execution of the expression. When an error happens, control returns (as by a nonlocal goto) to the innermost errset. The command loop example could have an errset around the entire loop, itself contained in another loop. Any error would return from the errset, thus looping around and re-entering the errset and the inner loop. This way, there is no place where an error can escape from the command loop. The value returned by an errset is either 0 if there was no error, or the error message string. This allows the command loop to print the error message when the errset returns.

Another use for errset is in case errors happen while an asynchronous action is being performed. For example, an automatic save of the file being edited is asynchronous; it is not expected at any precise moment by the user. If an error occurs during auto-saving, the usual handling of an error (which involves discarding any keyboard input typed ahead) would be undesirable. So the auto-save function employs an errset to regain control after the error, print its own flavor of message, and return to the user's editing. Another similar application is in inner parts of the system which call a function supplied by the user, and do not want an error in the user's function to cause them to lose control.

Returning to the user-written command loop example, there probably needs to be a command which exits from the loop. How can it be done? There are two loops doing their best to keep control from getting out. It could be done with ad-hoc flags and conditionals, but this would be unnecessarily complicated. A better way is to use the "catch" construct, which provides a named context to which control can be explicitly returned. A catch, like an errset, surrounds an expression, and control can pass to the end of the catch from within the evaluation of the expression. Unlike an errset, a catch has a name, and it receives control not because of an error but because of an explicit request to "throw" to that name. If the expression does no throw, then when it is finished its value becomes the value of the catch. A throw takes an argument which is the value to return from the catch. A catch placed around the loops and errset of the user-written command interpreter would allow any commands to exit from the loop easily.

Unlike non-local gotos in algebraic languages, whose labels obey static scope rules, the name of a catch is dynamically scoped; that is, any function called within the catch can throw to that catch. This is vital because, using the same example, it would be a great disadvantage to have to include the code for the individual commands in the same function as the command loop. This would require a gross conditional and make the function very large, and also limit extensibility. Much better is for the command loop to read the command and use it to create the name of a function, which is then called. This technique allows the user to extend the command interpreter by defining his own functions with suitable names. For example, the mail-file reading subsystem RMAIL has its own command reading loop; if it reads the command N, it calls the function named # RMAIL N. The exit command is Q, which simply calls the function named # RMAIL Q. It is necessary for this function to be able to throw to the right tag to get out of RMAIL. A similar consideration applies to the normal command dispatcher, since its commands must all be distinct functions found in the dispatch vector. It would not do to have the command dispatcher know specially about the command to exit, because then the user would not be able to redefine this command, or move it to a different character, in the usual way. The effect of an errset is also dynamic in this sense, of course.

One more error handling feature is the user-supplied function which is called when an error occurs not within an errset. This, together with primitives that allow all the data in the subroutine call and variable binding stacks to be accessed by programs, takes the details of error handling out of the kernel and into the domain of extensibility. Standard EMACS supplies an interactive backtrace function as the error handler, but a user-written extension provides interactive single-stepping and breakpoints for debugging.

Historical Perspective

EMACS is now a mature program, no longer new though still improving (variables that project were added this year), and in use at a dozen places. EMACS drew many ideas from earlier systems at MIT, and a couple of new systems patterned after EMACS are now coming into use on other makes of computers. However, it is in EMACS that the genre first came to full flower, and since I was closely involved with the later and earlier systems as well, I feel entitled to offer EMACS as a canonical example.

The display processor was first implemented in 1974 (inspired by the editor E of the Stanford Artificial Intelligence Lab), with a hard-wired command set, but a few months later after it was debugged the ability to redefine commands was added. The interpreter used to hold the display processor was, of course, TECO, that being already provided with appropriate text-manipulating, control, and I/O primitives—everything an editor needs aside from a display processor. Several people at MIT began writing editors using this system, while I continued to add to the interpreter the features necessary for reasonable system programming. In 1976, several ideas had become ripe, and I began EMACS, at the same time implementing many of the interpreter features connected with long-named variables and the library system, to make EMACS possible. Since then, development has proceeded steadily, with new code mostly being interpreted, new constructs only being added to the interpreter to speed up particular operations or to allow access to previously internal data structures.

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 Tops-20 operating system.

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 main motivation for these implementations was to transfer the ideas of EMACS to other computer systems. Two of these projects, now entering use, are Multics EMACS, now becoming an experimental 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.

Implications for the Process of System Design

It is generally accepted that when a program is to be written, specifications should be designed in advance. If this is not done, the result will be inferior. Some people know better than this, but none dare to speak up.

The writing of EMACS was as far from along these lines as can be imagined. It is best thought of as a continuous deformation of TECO into something which, for users, has no resemblance to TECO.

And indeed, there are ways in which EMACS shows the results of not having been completely thought out in advance, if only in being based on TECO rather than Lisp. (Nevertheless, EMACS has shown itself to be reliable and suitable for widespread use). If EMACS had been specified in advance, it would resemble the post-EMACS editors described above. However, the post-EMACS editors were specified in advance by EMACS itself, and could not have been written if not preceded by EMACS (this is not to say that they have copied slavishly; they have continued the process of gradual development). And EMACS could never have been arrived at except in the way it actually was. The chain of necessary steps leading to EMACS, starting with the display processor, was simply too long for anyone to have imagined the final result before the first step had been taken. If we had insisted on moving only toward destinations visible at the beginning, we would never have got here from there!

This is true of all the details of the individual commands as well as of the structure of the system. Each command in EMACS behaves as it does as a result of experimentation by many different users customizing their editors in

different ways, in the years when the display processor existed but EMACS had not yet been begun. This experimentation was possible only because a programmable display editor existed. It would not have been possible to design the EMACS standard command set without it.

Nor can one maintain the position that it was right to create EMACS this way because it was research, but ordinary system development is a different matter. This is because the difference between the two is also a matter of hindsight. EMACS was not the result of an intentional "editor research project" (nor would such a project have arrived at EMACS, because research projects aim only at goals which are visible at the start). The display processor and command dispatcher were seen only as an improvement to TECO; no one could have known, when any step was taken, how much farther the path would lead. One cannot even identify a "first" step, because the development, until the writing of EMACS per se, blends smoothly back into the development of TECO.

But why isn't such program of exploration doomed to be sidetracked by a blind alley, which nobody will realize due to the lack of a planned goal? 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.

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 simply 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 what will be 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.

Appendix: 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 copy 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. Logically, it ought not to matter whether the pointers are the same, since the contents of the buffer depends only on the contents of the strings and is not affected by replacing one string by another with the same contents. However, comparing the pointers is very useful for heuristics about moving blocks of lines from one part of the screen to another. In fact, it often finds and moves blocks of text which are not completely unchanged, but almost unchanged. If some of the lines in the block are completely unchanged, they do not need to be redisplayed. Even if this is not the case, it is clearer to the user what his text is doing if they move.

An additional efficiency factor is the global clock, a counter incremented whenever anything is changed. Each line has not only a string but the counter from when that string's contents were last changed. In addition to saving a pointer to the last line displayed on a line of the screen, the line's clock value at that time is saved. Then the clock values can be compared instead of the lines' contents.

This new scheme relaxes the requirements on commands to say what they have changed, also. They say only whether any text might 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.

Appendix: 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 named objects using offsets from the beginning of the file, so that the file can be valid at any location in memory. The names and named objects are all examples of the EMACS string data type, in the internal EMACS 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~ which 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~.

From a named object in a library, the name can be found, because it is immediately before the object in memory. Since one can tell which library an object is in by comparing its address with the 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.

Appendix: Why Isn't Any Editor That Is a Computer Program an Extensible Editor?

It is a truism that any computer program can be modified into anything at all. Thus, any editor program which can be accessed to be modified is extensible in a theoretical sense.

However, this does not mean that the user can conveniently extend it in practice. The usual editor is compiled and then run. If a user wishes to modify it, he must make his own copy of the entire editor, change some of it, compile it and run that. In addition to the disadvantages of a less interactive mode of program development, he has the problem that he must decide before starting to edit which version he wants to use. If two users make independent extensions, it is impossible to use both sets of extensions together without merging the two programs, which is real programming work, and compiling yet another version. If maintenance is done on the standard version of the editor, the extended versions require maintenance even if the changes do not interact with the extensions. In practice, these problems are enough to discourage just about everybody from trying.

EMACS Availability

EMACS is available for distribution to sites running the Digital Equipment Corporation Tops-20 ("Twenex") 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.

Notes — EMACS-related Editors

Multics EMACS

Multics EMACS was begun in early 1978 by Bernard S. Greenberg of Honeywell's Cambridge Information Systems Lab, the maintainer of Multics MacLisp. It has been responsible for convincing the Multics community of the desirability of character-at-a-time interaction with programs. 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 soon improved Multics to make that unnecessary. Multics EMACS pioneered the use of a doubly linked list of lines to represent the text being edited. It is now just entering widespread use. See

Bernard S. Greenberg, Real-Time Editing on Multics. Multics Technical Bulletin 373, Honeywell Information Systems, Inc. April 1978, Honeywell Cambridge Information Systems Lab.

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 smallness of the machine, an Interdata at the MIT Architecture Machine Group, running their own Multics-based virtual memory operating system. SINE was written by Owen T. Anderson. 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 extended, 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 Artifical Intelligence Lab, based on PDP-1 TECO which was written by Murphy in 1962. This was transported to Digital, which for a time distributed it as a product. Versions of TECO, and editors similar in concept to TECO, exist on a

great many systems. Those unfamiliar with TECO can get an idea of what a typical TECO is like from the manual

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

It should be noted that the TECO used as the interpreter in EMACS contains considerable extensions, including most of the features of higher-level languages in general that are conspicuously absent from the typical TECO. But the basic bad taste is the same.

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. When I implemented these features I assumed I was copying them from TMACS. Later I found that I had simply assumed that TMACS did them in the obviously right way—which it did not. TMACS was the project of Dave Moon, Charles B. 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 led 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; see

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

Notes — 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 provides little 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. Unfortunately, this ruined any chance of making it customizable, though it is possible that a different implementation of a line editor, done with this in mind, would be possible. 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. E has other interesting features as well. See the on-line documentation file E.ALS[UP,DOC] of the Stanford Artificial Intelligence Laboratory.

TRIX

TRIX is a language designed at Lawrence Livermore Lab specifically for writing editors; it is the only other example of such a system known to me. It has been used to write commands that are specific to particular languages, and to write text formatters. Many of the same ideas developed in the EMACS community were developed independently for TRIX. Its only, and 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 Tops-20 and Tenex operating systems. These systems do not provide a line editor, so TVEDIT has its own facilities for changes within lines. However, it is not programmable. 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)

Notes — 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.