Date: 20 March 1980

From: Bernard S. Greenberg

To: MTB Distribution

Subject: Paper being submitted to Stanford Lisp Conference

Attached is a draft (as submitted) of a technical paper which has been submitted to the Stanford Lisp Conference, which is being held in August, in Stanford, California. This paper describes Multics Emacs from a Lisp-oriented person's point of view: why Lisp was chosen, what things had to be done because Lisp was chosen, and what ensued. It has not yet been accepted.

Although some material has been excerpted from my previous paper (MTB 439), which was submitted (and accepted) to the Fourth Honeywell Software Conference, this paper is basically a different paper.

Prose and CONS
(Multics Emacs: a commercial text-processing system in Lisp)

Bernard S. Greenberg
Cambridge Information Systems Laboratory
Multics System Development/LISD
Honeywell Information Systems, Inc.
575 Technology Sq. (Mail Sta. MA22)
Cambridge, MA, 02139

Prose and CONS
(Multics Emacs: a commercial text-processing system in Lisp)

Overview

Multics Emacs is an video oriented text processing system released as part of Honeywell's Multics System. Multics Emacs is written in Lisp. This program is now in use by several hundred people, who view it both as their standard editor, and a process and display management system. It is the first Honeywell software written in Lisp ever to be released.

This paper addresses the choice of Lisp as the implementation language, and its consequences, including some of the implementation issues. The detailed history of Multics Emacs, its system-level design considerations, and its impact on Multics and its user community are discussed in [Greenberg]. One of the immediate and profound consequences of this choice has been to assert Lisp's adequacy, indeed, superiority, as a full-fledged systems and applications programming language. Multics Emacs has established an awareness of Lisp in quarters where the term had never been heard.

Perhaps even more significantly, persons who have never encountered Lisp, or who have encountered it in college and subsequently neglected it, have learned Lisp in order to write extensions (user-supplied code running in the Multics Emacs environment). In some cases, the persons involved were not even programmers and knew no programming language. The ability of Lisp to be shaped into highly specialized languages, via the macro facility of MacLisp (the Lisp dialect in use on Multics) [Moon], has given rise to a coding formalism of transparency and simplicity; an unprecedented (in the history of Multics) number of unsolicited contributions coded in this formalism have been offered.

The Choice of Lisp

Multics Emacs grew out of a need for Multics text processing to evolve into the world of display editing. In early 1978, the author became familiar with the EMACS [Stallman] editor on the ITS time sharing system at the MIT Artificial Intelligence Laboratory, which provided a well-debugged user interface design. EMACS at the AI Lab was coded in TECO, itself a display editor (coded in PDP-10 machine language). TECO tries to bridge the gap between being a user editing interface, and a programming language which facilitates the construction of complex layered subsystems. Stallman [Stallman] points out how TECO falls short of both of these goals by attempting this compromise. Nevertheless, a significant feature of the TECO programming environment is the modularization of programs into functions (called macros in TECO), which live in a global environment, and call each other and themselves in a by-value fashion, very much like Lisp.

TECO environments, including EMACS on ITS, encourage extension. Users build functions and load them into the global environment, utilizing functions and data already there. This extensibility encourages the development of large optional packages, such as mail editing systems and specialized editing modes knowledgeable about specific programming languages. The abundance of such packages is an earmark of EMACS.

The ability of users to extend the implementation was the most important factor in the choice of a programming language for Multics Emacs. The two choices at the time appeared to be PL/I, the traditional Multics system programming language, and TECO. The outcome was the decision to implement it in Lisp.

Historically, nearly all Multics programs have been written in PL/I. Multics PL/I [AG94] is one of the most complete implementations of the ANSI PL/I standard in existence, and has matured over the years as the sole system support language implementation for Multics. PL/I presented itself as the natural choice. Viewing the ITS experience in perspective, it seemed as though marked efficiency could be gained by implementing an EMACS-like editor directly in PL/I, as opposed to as a system of interpreted code in some other language, such as TECO.

Various scenarios for extensibility in a PL/I-based implementation were evaluated. The PL/I-based Multics process environment is one of the classic models of extensibility in the literature. The ability to extend and customize one's Multics process environment via PL/I subroutine call and definition has provided the model for many operating systems since. Yet, several features of PL/I pointed away from its choice as the Multics Emacs implementation language. Given that any reasonable implementation of an EMACS-like modularity would associate editor primitives (e.g., "move the virtual pointer forward a character", "delete the current character", etc.) with PL/I subroutines, extension code would degenerate into a sequence of subroutine calls. Calls between separately compiled modules are expensive. The by-reference semantics of the PL/I call, as well as considerations of the PL/I signalling mechanism, contribute this expense. Calls to internal subroutines are less expensive, but by definition, such subroutines are not accessible to other modules. Thus, if externally accessible procedures were to be had, they would have to be of the (expensive) external kind. This would add substantial overhead to even the smallest editor primitive. PL/I is also notorious for requiring declaration of the smallest artifacts of every module; all variables used, all external names, etc., must be explicitly declared.

These considerations led to the choice of Lisp as an implementation language. Lisp inter-function calls all have the same overhead, and are traditionally very cheap. Lisp programs are commonly written with many small functions, which therefore use inter-function call very heavily. Thus, function calling has been highly optimized in most Lisp implementations. Lisp's

by-value call is innately more efficient than PL/I's by-reference call. What is more, every Lisp function (and global variable) in a given environment may be accessed by any other function, unless special measures are taken. Of course, this can be a mixed blessing, in terms of both programming style and the pitfalls of a global namespace.

Choosing Lisp for reasons of efficiency is a notable departure from the inefficiency arguments usually levelled against Lisp! The existence of the compiler ends all efficiency arguments about Lisp being "an interpreted language". The need to allocate storage and garbage collect is often raised as well; sagacious storage management policies, which ought be used in any program in any language, put this "problem" well within limits. Even though traditional programming style in textbook presentations of Lisp often consumes storage in a wasteful fashion, it is possible with minimal added difficulty to code without wasting storage. Part of the problem here can be traced to what the author considers gross philosophical flaws in the classical presentation of Lisp. The basis of those arguments is rooted in the object/pointer/value distinction outlined below; almost no presentation of Lisp correctly (in the author's opinion) portrays the concept of "object". (For a presentation of the alternative view, see [LispNotes]).

Lisp's notion of data abstraction is uniquely suited to subsystem construction. The Lisp programmer can define a "data type" by program convention only, without "informing the language" in any way. For instance, Multics Emacs defines editor buffer pointers (called "marks", which are conceptually inter-character pointers to text, dynamically updated as text is added and deleted) built of Lisp list cells. Lisp programs in Emacs manipulate and store marks, without knowledge of the internal structure of marks. Here, Lisp fosters isolation of levels of implementation, both within the internal levels of Emacs itself and within extension code.

The case can validly be made that passing pointers in PL/I or some other non-Lisp language could achieve the same effect: indeed, internally, Lisp does just that. However, Lisp encourages conceptualization of a value as "a list, a record, a mark, or a buffer" rather than a pointer thereto; all values in Lisp (other than numbers) are "pointers" to some object. Correctly explained, Lisp elegantly drops the phrase "a pointer to" as conveying no meaning. This is, in the author's view, perhaps the most fundamental conceptual advantage of Lisp. Lisp programs pass each other records, buffers, sentences, houses, students, and toy blocks, rather than pointers, without need for knowledge or declaration of their internal structure. Such knowledge need exist only in those modules that actually construct or decompose them. In the view of the author, this is the source of Lisp's power and elegance for modelling "real-world" systems, and facilitates and encourages the construction of programs that behave like "real-world" systems.

Another very powerful feature of Lisp, specifically of MacLisp, is the macro feature of the language, by which the syntax of the language itself can be extended. The "macro language" of Lisp is Lisp itself. Lisp allows programmers to specify code to run at compile time to implement a macro-defined language. The Multics Emacs extension language is one such. Please see the Appendix for an overview and samples of the extension language.

Multics MacLisp has a fully mature debugging system, I/O facilities, and the ability to interface to other facilities in Multics. Multics MacLisp also has a powerful compiler, and all "production" programs are compiled (although the existence of the Lisp interpreter is invaluable during debugging). Many other Lisp systems lack these features, and are thus ill suited to development of production software.

The Lisp Machine project at the MIT Artificial Intelligence Lab [Chineual] greatly influenced the concurrent development of Multics Emacs. The Lisp machine provided many models of Lisp-coded, full fledged, interactive user environments. There was substantial design crosscurrent between Multics Emacs and the Lisp Machine's editor, ZWEI [Weinreb]. Both editors are coded in Lisp, and implement an EMACS-like interface.

Implementation Considerations

The first implementation question raised was that of text representation. Traditional editors represent text as contiguous vectors of characters, sometimes divided in two ("gap editors") at the "point of editing". Lisp does not provide a natural representation of such an object. Lists or arrays of characters could be used, but potent hardware string-manipulation instructions would then be ruled out. Lists of characters utilize storage inefficiently as well. When editing of several multi-hundred-line programs simultaneously is a design goal, this inefficiency cannot be tolerated. Arrays of characters are storage-efficient, but the language is not designed to deal with them as character strings, and hence provides no appropriate primitives.

Many Lisp implementations, MacLisp included, provide a "character string" data type, in addition to the normal "atoms" (which are called symbols in MacLisp). Character strings have printable representations, and that is all (i.e., they do not have property lists or bindings, and are not catalogued in an oblist (obarray in MacLisp)). This type of object facilitates use of hardware string primitives.

Text buffers in Emacs are represented as doubly-linked lists representing lines of text. Each line is represented by a triplet of a string, the previous line, and the next line in that buffer. The string contains the character content of the

line. This representation is a compromise between the flexibility of list structure and the hardware and storage efficiency of the character vector.

The decision to represent text lines by Lisp objects was also designed to facilitate display management. Most editing operations consist of changes within a line, or insertion or deletion of lines. Current video terminals support clearing, inserting, and deleting lines, and updating of data within lines. By representing buffers as lists of lines, a continuous mapping of lines, from the user's text file, through the buffer structure, through the display manager, to the terminal, is maintained. The Emacs display manager maintains a screen image, which is an array of objects representing the text on each screen line. Each such object comprises a string representing the contents of the screen line, and the editor line triplet for the buffer line being displayed on this screen line. At screen update time, the display manager computes a new screen image, and searches for matching lines. An old line and a new line are considered to "match" if they are the same object, that is, "EQ". Whether or not the contents of the line has changed, the identity of the line is the display manager's basic cue to direct insertion, deletion, and updating of lines.

The normal behavior of Lisp is to return objects as function results. String operations such as concatenation and substring extraction generate new strings, allocating storage every time such an operation is performed. This behavior was felt to be unacceptable for a text editor: if this strategy were utilized unmodified, the number and frequency of operations of an editing system (e.g., entry or deletion a single character) would allocate storage proportional to the square of the length of text being entered.

A new type of Lisp object, the "rplacable string", was devised to solve this problem. A "rplacable string" is one whose "contents" (i.e., the set of characters in its printable representation) may be changed. Its "length" may be changed as well. Rplacable strings are similar to the treatment of strings on the Lisp Machine, where strings are a special case of arrays. They constitute a new fundamental data type in Multics MacLisp, and were implemented not by in-language extension, but by augmenting the MacLisp implementation by out-of-the-language techniques.

In Emacs, the "current line" being edited is represented by such a string. The first time a line is modified, the string representing that line's contents is copied into the "current line" rplacable string (only the current line can be modified). Insertion or deletion of text from the current line is accomplished by changing the contents of this string. Changing the contents of this string, in turn, is accomplished by powerful hardware primitives which can efficiently move character strings left or right in place. When the editor moves off the modified line, the rplacable string is copied back into

a normal string (rplacable strings operate outside the normal Lisp storage management scheme, and thus the number of them is kept to a minimum).

Emacs takes advantage of the storage management and garbage collection policy of Lisp in a novel way in another phase of the display manager. When Emacs computes a screen image, it saves not only the triplet representing a line, but its "contents of line" element (the string representing the actual characters). An array of these "contents of lines" is saved between screen updates. When display update (redisplay) time arrives, and the display manager finds a buffer line (i.e., a triplet) in common between the old and new screen image, it checks to see if the "contents of that line" (the string in the triplet) is the same object as what was saved in the array of "contents of lines" for that line, at the time of the last screen update. If this is indeed the case, the display manager knows for certain that the user-visible contents of the line, and thus its representation on the screen, have not changed, and need not be recomputed. This simple comparison (for identity of object, a pointer comparison in hardware terms) avoids the need for keeping or comparing arrays of characters from the buffer at display update time and saves vast amounts of computation. This technique is based upon the fact that Lisp objects retain their identity as long as they are known in the environment, and no two objects share identity.

Subsequent Developments

In the two years since its inception, Multics Emacs has grown from an experimental Lisp program to a twenty-thousand line plus subsystem that is in use across the country and sold as a product. It is currently used in the preparation of almost all Multics documentation. Its growth and subsequent development were a direct result of the decision to implement it in Lisp. User experimentation with modifications and alternative interfaces were possible only through the extensible nature of the Lisp environment. Significant comprehensibility of code (cf. the well-known opaqueness of TECO) was achieved by the use of Lisp and Lisp macros. This comprehensibility was a prerequisite for user extension.

Multics Emacs has acquired, as hoped for, a proliferation of optional packages, including a mail system, an interactive message system, various menu-driven interfaces, packages for editing languages as diverse as FORTRAN and Lisp. Recent developments include a word-processing system similar to turnkey text processing systems, and "modes" for managing dialogues with remote computers connected via communications lines or the ARPANET.

The Lisp editing mode, notably, by providing automatic parenthesis balancing, among other syntactic aids, renders Emacs an invaluable aid in Lisp program construction. Emacs would be

of tremendous value in Lisp program preparation even if it were not written in Lisp. The existence of such aids removes one of the major obstacles to many people's use of Lisp.

One of the unplanned benefits of Lisp which has proven to be of inestimable value is the ability to develop Emacs extensions from within Emacs. Persons developing extension code enter Lisp functions into an Emacs buffer set up in Lisp Mode. As the extension coder enters and edits functions, he or she can ask Emacs to evaluate the function definitions, thus adding these functions to the Emacs environment. Functions so defined can then be invoked explicitly by Emacs command, and their results and effects observed. In this fashion, programs can be developed function by function, coding higher level functions as lower level ones are debugged. The effect of editor extension code being developed can be observed as it is written; by means of screen-splitting (windows) the extension developer can view extension code under development and observe its effect when run simultaneously. Facilities to trace and set breakpoints in functions being debugged are provided. In Emacs Lisp Debug mode, the programmer can divide the screen into three regions: one displaying code being edited and debugged, one containing the sample text buffer upon which the code is operating, and one an interactive dialogue between the programmer and the Lisp interpreter. The editing features of Emacs are available for every interaction.

A most exciting frontier of Multics Emacs is "Multics Mode", in which control of all user input and output is managed by Emacs. It is an instance of an "editor top level", wherein Emacs screen management and editing features apply to all user interaction. Emacs editing becomes applicable to all input, and earlier input and output can be edited (or searched through) using standard Emacs commands. Here, the distinction between editor and system vanishes, and the Multics user interface takes on entirely new dimensions. Emacs, particularly with Multics mode, is, like the Lisp Machine, an experiment in Lisp-coded process environments. Multics, the "PL/I machine" becomes, in fact, a Lisp machine.

The diversity, scope, and power of Emacs is directly attributable to the Emacs extension environment, all of whose power derives from Lisp. The straightforward and simple extension language was possible only through the power of MacLisp macros and compile time facilities, and the simplicity of the Lisp function call. The resultant clarity of the extension language has allowed almost all extension coders to acquire proficiency through imitation of examples. Incremental debugging of code, as in the Emacs extension development, is only possible in languages with powerful interpreters; only when a full compilation facility is available as well, as in MacLisp, can production code be developed in this manner.

The growth and development of Multics Emacs can be summarized in one word: Lisp.

\014

References

Multics Emacs is defined and documented by two published Honeywell Manuals, [CH27], which describes the total user interface, and [CJ52], which details the extension writing language and facilities.

[AG94] Multics PL/I Language Specification, Order #AG94, Honeywell Information Systems, Inc.

[CH27] Emacs Text Editor User's Guide, Order #CH27, Honeywell Information Systems, Inc., December 1979.

[Chineual]
Weinreb, D. & Moon D., "The Lisp Machine Reference Manual", MIT Artificial Intelligence Laboratory, 1979.

[CJ52] Emacs Extension Writer's Guide, Order #CJ52, Honeywell Information Systems, Inc., January, 1980.

[Greenberg]
"Multics Emacs: an Experiment in Computer Interaction", in Proceedings, Fourth Annual Honeywell Software Conference, Honeywell Information Systems, March 1980.

[LispNotes]
Greenberg, Bernard S., "Notes on the Programming Language Lisp", MIT Student Information Processing Board, 1976, 1978

[Moon] Moon, David A., "The MacLisp Reference Manual", MIT Project MAC, 1974.

[Stallman]
Stallman, Richard M., "Emacs, the Extensible, Customizable Self-Documenting Display Editor", AI Memo 519, MIT A.I. Lab, June 22, 1979

[Weinreb]
Weinreb, Daniel L., "A Real-time Display-Oriented Editor for the Lisp Machine", S.B. Thesis, MIT Dept. of EE & CS, January, 1979.

Appendix: The Extension Language

Multics Emacs extensions, whether supplied or user written, are written in Lisp, augmented by a set of macros provided as a lexically includable program fragment. Extensions are written in an environment consisting of the native MacLisp functions (other than I/O), functions in the editor and standard extensions, and occasionally the display manager. The editor functions provide the ability to manipulate the current point of editing and the buffers, and inspect and change the contents of lines and buffers. Lisp macros are provided for syntactic sugaring of commonly used syntactic cliches, such as "create a temporary variable, assign a mark at the current point to it, perform some code, and free the mark", as well as to augment the expressive power of MacLisp.

The extension writer creates Lisp functions using the Emacs command definition facility, which associates with the defined function name a set of properties facilitating argument checking and prompting, as well as automatic documentation. In addition to invoking supplied functions in the extension environment, functions defined via the Emacs command definition facility may invoke each other (as may any Lisp functions), or be "connected" to keys, so that they will be invoked automatically by Emacs when selected keys are struck.

Extensions use strings, integers, buffer names, and marks (see above). The basic Lisp data types are only occasionally used. In fact, reasonably expert extension writing has been accomplished by persons completely ignorant of fundamental Lisp data object types. The extension writer has no knowledge of or dealings with the internal representations of any data structure of the editor.

The most useful class of function used in extensions are those which are already capable of being invoked on behalf of user keystrokes by Emacs. For instance, "forward-word" is very commonly used in editing to position the cursor past the current word, which is how the Emacs user conceptualizes "what Escape-F does" (Escape-F being the two key sequence standardly used to invoke this common command). The extension writer, on the other hand, conceptualizes the "forward-word" function as moving the current buffer point to beyond the current word. Using these functions in extension functions is a valuable technique: the extension programmer can always experiment with the function to be used by invoking it in the normal interactive (i.e., by keystroke) way to determine details of its behavior.

Here is a simple example of an extension function based upon commands normally available through the keyboard. Its name is "bracket-word", and it places the word at which the cursor points in angle brackets:

```
(define-command bracket-word
       &documentation "Puts angle brackets around the word
       at which the cursor points."
       (forward-word)
       (insert-string ">")
       (backward-word)
       (insert-string "<"))
```

The function "insert-string" has the same effect as the interactive user typing a sequence of self inserting (trivial, printing) characters. The invocations of forward-word and backward-word position the current point prior to the insertions of the character strings. The end result of running this function would be the same as if the user had typed Escape-F (which invokes forward-word), a right-angle-bracket, Escape-B (which invokes backward-word), and a left angle bracket. The net result on the buffer (and the screen) is the same. However, the intermediate states which would be visible to the user typing the above sequence will not be visible on the screen when this extension is run, only the final state will be. This is because the command interpreter invokes the display manager after each command character is typed, but this function (as is visible by inspection) does not invoke the display manager at all: it invokes only what it is seen to invoke.

The most common extension environment macro is "save-excursion", which is used to remember the location of the current point, and restore it after the execution of the included code within the macro. For example, the following extension function places a star at the beginning of the current line, but leaves the cursor at the same place in the current line: (Bear in mind that the position is represented as a mark, which is relocated automatically as the buffer text changes)

```
(define-command put-star-at-beginning-of-line
       (save-excursion
          (go-to-beginning-of-line)
          (insert-string "*")))
```

The "save-excursion" macro encompassing the invocations of go-to-beginning-of-line and insert-string ensure that the current point will restored after these functions run. Another similar macro, save-excursion-buffer, is used to restore the selection of buffer during its dynamic scope. As switching out of a buffer saves the location of the current point within that buffer, save-excursion-buffer subsumes the task of saving the point within that buffer. A powerful MacLisp facility (unwind-protect) ensures that the point will be restored even if put-star-at-beginning-of-line terminates abnormally and its execution is aborted.

Another set of very common macros in extension writing are those dealing with marks, providing for the creation thereof, and freeing at the end of the contained code. The macro "with-mark" names a variable to which a mark is assigned at execution time: that mark will denote the point in the buffer

which is current at the time the code contained in the macro begins execution. The following extension function deletes two words forward from the current point:

```
(define-command delete-two-words-forward
       (with-mark here
          (forward-word)
          (forward-word)
          (wipe-point-mark here)))
```

When delete-two-words-forward is invoked, a mark designating the current point in the buffer is created, and assigned to the local variable named "here". The generation of the mark and the local variable are all artifacts of the "with-mark" macro. The two calls to forward-word are then executed, presumably moving the buffer point (but not the saved mark) two words forward in the buffer, and then the function wipe-point-mark is invoked, passing that mark as an argument. The function wipe-point-mark deletes all text between the current buffer point and the point designated by the mark (saving it, incidentally, for possible user recovery). At the end of execution of delete-two-words-forward, the mark created by the macro is freed.

Another class of Emacs extension environment macros are those used to supplement (or reimplement) features in MacLisp thought to be inadequate, either for learning purposes, or ill adapted to the extension environment. For example, the extension documentation teaches the use of the "if" macro as opposed to the native MacLisp "cond" as the fundamental conditional construct. "if" is much simpler and straightforward, suffices for almost all cases, and is similar to the conditional construct in almost all languages other than Lisp. "cond" is more general and powerful, but this power is not often needed, and seems to present a stumbling block to those learning Lisp. Another macro of this class is "do-forever", and its exit form, "stop-doing". The native MacLisp "do" has two forms, one like the FORTRAN "do", and the other a powerful multi-variable generalization of this. Most often, the extension writer wants to iterate not over integer variables, but over buffer lines or characters: the iteration variable is thus the global editor state, and the need to specify or deal with variables which are almost never needed is undesirable. "if" and "do-forever" are illustrated by the following extension function, which either finds the first blank line of the buffer or complains if there are none:

```
(define-command find-first-blank-line
       &documentation "Moves cursor to the first blank line of
       the buffer."
       (go-to-beginning-of-buffer)
       (do-forever
          (if (line-is-blank)
              (stop-doing))
          (if (lastlinep)(display-error "No blank lines!"))
          (next-line)))
```

The form "(stop-doing)", if executed, causes control to exit the "do-forever" form. The predicate "lastlinep" tests for the current point being on the last line of the buffer. The function "display-error" causes an error message to be printed at the bottom of the screen, and a non-local transfer of control out of find-first-blank-line, aborting its execution. This non-local control transfer provides the reason that a "stop-doing" is not needed after the call to display-error.

Experience with the extension language has shown that its meaning is so transparent that the underlying Lisp is all but invisible: the emphasis of Lisp shifts from its data world to its being a formalism for organizing function invocation. People begin to write Lisp programs naturally, without realization that they are doing so, and the universe of Lisp grows with each such keystroke.