ARTIFICIAL INTELLIGENCE APPLICATIONS TO
COMPUTER-ASSISTED INSTRUCTION

Project Progress Report No. 5:

ON PROGRAM CONVERSION FROM MACLISP TO VAX LISP

Jesse M. Heines, Ed.D.

Systems Based Courseware
Educational Services Development & Publishing

George Poonen

Productive Information Management, Inc.

December 29, 1983

Educational Services
Burlington, MA 01803

1 ABSTRACT

This report documents our experiences in converting a program from MacLISP on TOPS-20 to VAX LISP on VAX/VMS. It includes:

- a list of changes required to get the MacLISP code running in VAX LISP, particularly with regard to data types,
- a number of stylistic programming guidelines for people writing MacLISP code that will eventually be converted to VAX LISP,
- general comments comparing the two LISPs on such issues as performance and facilities, and
- advice for new VAX LISP users on setting up an EMACS/LISP programming environment on VAX/VMS.

The code that we converted is the prototype IReGIS program described extensively in AI/CAI Project Progress Report No. 4. The complete IReGIS course was designed to be an intelligent, rule-based tutor on ReGIS. The IReGIS program that currently exists represents one aspect of the complete course, the ReGIS Laboratory described in Section 5.3 of AI/CAI Project Progress Report No. 3. This program consists of approximately 85 LISP functions each averaging about 25 lines of code.

The comments made in this report are based on our use of VAX LISP Version X0.2-8 running on a VAX-11/780 with 4 MB, 3 RP06's, and an average daytime load of 40 interactive users. Our two most significant findings from this experience are that:

1. VAX LISP is an extremely rich implementation of the base language, particularly with regard to data types and interfaces to the operating system. It appears to be particularly valuable for rapid prototyping.
2. VAX LISP is large and slow, and therefore is not a viable environment for delivering highly interactive CAI-type software in production mode on heavily-loaded systems such as ours.

2 TABLE OF CONTENTS

3 INTRODUCTION

The purpose of this report is to document our experiences in converting a program from MacLISP to a VAX LISP so that others may perform the task more quickly and with less effort. While the report does contain comments on the overall design and implementation of VAX LISP, it should not be construed as a critique of either VAX LISP itself or the VAX LISP Development Group. First, the version of VAX LISP we used was a preliminary, pre-field test version, and newer versions will have significantly different characteristics in some categories. Second, while we feel that we learned a number of valuable lessons that are worth sharing, we have converted only one program. We did not make exhaustive benchmark-type studies of the differences between VAX LISP and MacLISP, and we certainly did not make use of all the VAX LISP facilities. We focused on functionality rather than efficiency, and our purpose was merely to get the code up and running rather than making it particularly elegant.

We did, of course, find some bugs during the conversion process, and these have been reported directly to the development group rather than documenting them here. We gratefully acknowledge the help of Gary Brown and Jerry Boetje in analyzing our programming problems to determine if the bugs found were in our code or theirs. Indeed, the conversion would have been much more difficult without Jerry Boetje's help in using the CALL-OUT facility.

The code that we converted is the prototype IReGIS program described extensively in AI/CAI Project Progress Report No. 4. The complete IReGIS course was designed to be an intelligent, rule-based tutor on ReGIS. The IReGIS program that currently exists represents one aspect of the complete course, the ReGIS Laboratory described in Section 5.3 of AI/CAI Project Progress Report No. 3. This program consists of approximately 85 LISP functions each averaging about 25 lines of code. This code was converted from MacLISP running on a DECsystem-20 to VAX LISP Version X0.2-8 running on a VAX-11/780.

The report includes the following major sections:

- changes required to get MacLISP code running in VAX LISP, particularly with regard to data types,
- stylistic programming guidelines for MacLISP code that will eventually be converted to VAX LISP,
- general comments comparing the two LISPs on such issues as facilities and performance, and
- how to set up an EMACS/LISP programming environment for VAX/VMS similar to that available on TOPS-20.

4 CODE CHANGES FROM MACLISP TO VAX LISP

Code changes from MacLISP to VAX LISP that we ran into may be classified into three categories:

- changes required to get the code to run,
- changes not absolutely required, but highly beneficial for enhancing code clarity and/or system performance, and
- changes useful for other reasons.

The changes discussed in the sections that follow should be interpreted as representative of those needed for MacLISP to VAX LISP conversions rather than as a comprehensive list of such changes.

4.1 Required Changes

Required changes can be separated into four subcategories:

- the need to declare global variables,
- straight translations from a MacLISP function to a VAX LISP function,
- minor transformations on MacLISP functions or syntax, and
- major transformations on MacLISP functions or syntax.

4.1.1 Need to Declare Global Variables - The need for declarations was one of two instances we found in which the functionality of the VAX LISP interpreter differred from that of the VAX LISP compiler. (The other instance was a simple bug.)

NOTE

> We have been informed by the VAX LISP Development Group that beginning with Version X0.4-0, the VAX LISP interpreter and compiler will share the same front end, thus eliminating all differences between the interpreter and the compiler.

In the Version X0.2-8 interpreter, unbound variables were simply treated as global and can be referred to at any level. In the Version X0.2-8 compiler, unbound variables that are not declared using DEFVAR are flagged with warnings and sometimes cause fatal compilation errors. Declaring all global variables via DEFVAR eliminated this problem, and we therefore recommend that all globals be declared simply as a matter of programming practice.

4.1.2 Straight Translations - Some MacLISP functions and predicates can be translated directly into VAX LISP equivalents. In these cases, conversion is trivial. The translations we used most often are shown in Table 1.

Table 1

REPRESENTATIVE STRAIGHT TRANSLATIONS
FROM MACLISP TO VAX LISP

MacLISP		VAX LISP
#\ALTMODE	-->	#\ESCAPE
(ADD1 n)	-->	(1+ n)
(SUB1 n)	-->	(1- n)
(GREATERP x y)	-->	(> x y)
(LESSP x y)	-->	(< x y)
(ASCII n)	-->	(INT-CHAR n)
(READCH)	-->	(READ-CHAR)
(^G)	-->	Version X0.2-8: (THROW 'TOP-LEVEL-CATCHER T) Later Versions: (THROW-TO-COMMAND-LEVEL :TOP)

4.1.3 Minor Transformations - Some forms require minor transformations from MacLISP to VAX LISP. For example, the PROBEF function in MacLISP is PROBE-FILE in VAX LISP, and files specifications supplied as arguments to the MacLISP version are expressed as lists, while those supplied to the VAX LISP version are expressed as strings or atoms. In most cases, these minor transformations are documented in the Common LISP Reference Manual by Guy Steele.

4.1.4 Major Transformations - A small number of MacLISP forms require major transformations for use in VAX LISP. Most of these forms involve the MacLISP STATUS function, which does not exist in VAX LISP. For example, the MacLISP function:

```
(STATUS DATE)
```

must be expressed in VAX LISP as:

```
(DEFUN CURRENT-DAY ()
    (MULTIPLE-VALUE-BIND (A B C D E F) (LIST D E F)))
```

Likewise, the MacLISP function:

```
(STATUS DAYTIME)
```

must be expressed in VAX LISP as:

```
(DEFUN CURRENT-TIME ()
    (MULTIPLE-VALUE-BIND (A B C) (LIST C B A)))
```

A far more complex transformation is required for the MacLISP function:

```
(SSTATUS LINMODE NIL)
```

This function essentially puts the terminal into single character input mode so that the MacLISP READCH function returns a character as soon as one is typed, rather than waiting for the user to press RETURN. There is simply no equivalent function in VAX LISP. We use single character input for all user entries in our courseware so that the keypad keys, which are defined to perform special functions such as help, replot the screen, and exit, are active at all times. For example, pressing the PF2 key causes help to be displayed without waiting for the student to press RETURN. Implementation of single character input in VAX LISP requires putting the terminal into passall mode and this has several side effects. Section 6.3 of this report describes our single character input routine in detail.

4.2 Optional But Beneficial Changes

We made a number of changes to the IReGIS code that were not absolutely necessary, but that proved to be beneficial for clarifying the code or improving system performance. These changes are described below.

4.2.1 Data Types - One of the distinguishing and most pleasing features of VAX LISP is the richness and structure of its data types. Because of the paucity of data types in MacLISP, some objects had to be mapped unnaturally into lists. In VAX LISP, these objects could often be more easily be handled as strings, sets, or, more generically, as sequences. We also found it relatively easy to coerce one data type into another data type where appropriate. This permitted easier conversion where multiple data types were expected by the MacLISP functions already defined.

In the MacLISP version of our course, all student input is represented as lists of characters. In the VAX LISP version, it is represented as strings. This data type conversion has several advantages:

- Code Clarity: "Student Name" is considerably easier to work with than (S /t /u /d /e /n /t | | N /a /m /e).

- System Performance: A very large number of IMPLODEs and EXPLODEs were completely eliminated.

- Algorithm Simplification: String comparisons, editing, and case conversions could be done more easily with the built-in VAX LISP functions STRING=, STRING~, STRING-TRIM, STRING-EQUAL, etc. We also used a large number of the VAX LISP SEQUENCE functions such as SEARCH, POSITION, and REMOVE-IF to perform useful operations on strings that we had had to program ourselves in MacLISP.

Since the subject matter of our course is ReGIS, the use of strings also cleared up a number of instances in which we had to worry about macro characters such as comma. In MacLISP, a ReGIS string address had to be expressed as '[123/,456], while in VAX LISP it could be expressed as "[123,456]" with no special consideration for the comma.

4.2.2 &OPTIONAL Function Parameters - The &OPTIONAL qualifier for function parameters proved to be very useful because it allowed the number of arguments in a function call to be different from the number in that function's definition. This allowed us to change a function's definition slightly by, for example, adding a new parameter to improve performance, without having to change all of the calls to the function. Use of the &OPTIONAL qualifier thus saved a considerable amount of debugging.

4.2.3 &REST Function Parameters - The &REST qualifier paid similar dividends. A number of our MacLISP functions expected association lists as arguments and then parsed these with the ASSOC function to extract relevant values. The parsing process could be totally eliminated in VAX LISP by using the &KEY parameter qualifier, but this qualifier was not yet implemented in Version X0.2-8 for user-written functions [1]. We therefore retained our use of association lists, but we preceded the single function argument with &REST. This allowed us to pass values to the function as quoted pairs such as '(ANSWER-AT "[0,460]") '(CHAR-LIMIT 15) rather than having to put all of these pairs into a list.

---

[1] From a performance point of view, Gary Brown feels that any coding efficiency gained by the use of &KEY in this manner would probably be offset by the system overhead that it requires.

4.2.4 SETF vs. SETQ - Access and updates of variables are done in a much more uniform way in VAX LISP than they are in MacLISP. The VAX LISP function for updating variables is SETF, and essentially eliminates the need for for SETQ, SET and RPLACA. The form of simultaneous array access and update in VAX LISP is:

```
(SETF (AREF ARRAY SUBSCRIPTS) NEWVALUE)
```

4.2.5 Built-In Functions on Sets - As stated earlier, VAX LISP contains a much richer set of built-in functions than MacLISP, and we were able to replace some of the functions we wrote ourselves in MacLISP with built-in VAX LISP functions. The most notable area in this regard dealt with functions on sets such as SUBSETP. In MacLISP we wrote our own function, while in VAX LISP we used the built-in one.

VAX LISP also provides a rich set of built-in functions that operate on SEQUENCEs, i.e., lists and vectors (the string data type is a simple vector). As mentioned in Section 4.2.1 above, we used a number of these functions to manipulate strings.

4.2.6 Documentation Strings - Last but certainly not least, is the availability of VAX LISP documentation strings. These are simply strings added after a function's formal parameter section which can contain any desired text. This text can then be printed on the terminal using the DESCRIBE function in the LISP interpreter.

It is surely unnecessary for us to preach the values of documentation in this document, but one or two points are in order. First, this project involved two programmers -- Jesse and George -- converting code written solely by one of the two (Jesse). George did the first pass on most of the code, getting each program module at least to the point that it would load and run under VAX LISP. He tested each function as best he could in isolation from the overall course. After George converted a module, Jesse attempted to integrate it with modules that were already converted and tested.

Jesse was understandably much more familiar with what each function was supposed to do than George was, and George sometimes inadvertently changed a basic characteristic of a function during the conversion. The easiest way for Jesse to find these instances was via the documentation strings. He could test out the integrated functions in the interpreter, and if things didn't work as expected he checked what George had done by looking at the documentation string. Thus documentation strings provided a simple and effective method for recording what each programmer did that was accessible from the interpreter. (Note that GRINDEF does not print out comments.)

4.3 Other Possible Changes

Our conversion effort focused on functionality rather than efficiency. If we were to recode IReGIS in VAX LISP completely from scratch, we would of course have coded some things differently from the way in which they coded in MacLISP. The changes that follow were not made, but are worth noting for future projects.

4.3.1 STRUCTURE Data Type - Some of the objects in the IREGIS course could have been modelled as structures using DEFSTRUCT. This would have involved substantial changes to existing programs, and we chose not to change the underlying program architecture. We believe that STRUCTURE data types may provide significant enhancement for this type of program in future implementations.

4.3.2 Packages - The Common LISP Reference Manual describes a PACKAGE facility which was not implemented in Version X0.2-8 but which could further enhance modular decomposition of functions. We did not use this facility, but we believe that it may be worth using it for future development to enhance code readability and maintainability.

5 STYLISTIC GUIDELINES

George reports that this was his first attempt at converting code written by someone else to a different language/dialect. He found many of the programming and documentation standards used in the existing program to be valuable, particulary those discussed below.

5.1 Modular Decomposition of Functions

The IReGIS code resides in nine separate files, each containing about a dozen functions averaging about 25 lines of LISP code. Extensive use is made of recursive auxiliary functions -- functions that do the actual work after being called by higher level functions that set them up (see, for example, the program on page 164 of LISP by Winston & Horn). Small, one-task functions were much easier for us to convert than larger, multi-task functions.

As stated earlier, VAX LISP provides a package facility which could further enhance this aspect of program development. We did not use this facility, but hope to do so in the future.

5.2 Inclusion of Test Driver Functions

Many of the program modules contain test driver functions that can be used to test the functionality of other functions within the module. These functions are not called when the course is actually run, but are very useful during debugging. The inclusion of these test functions was particularly useful in our case because all of the program I/O is done in ReGIS mode. Once the terminal is put into this mode, error messages do not get displayed because the characters in the messages are interpreted by the firmware as ReGIS command strings. The test driver functions allowed individual functions to be tested in non-ReGIS mode so that their interactions and possible error messages could be monitored.

5.3 Documentation

From a program conversion point of view, comments associated with each LISP statement were less useful than overall function documentation, except when the code was particularly abstruse. Most comments were on the processing aspects of the code, but it would have been helpful to have more detailed descriptions of each function's input and output behavior. Examples were provided for some of the more complex functions, and these proved to be particularly useful. Inclusion of additional examples would have been useful.

5.4 Parentheses

As in all LISPs, parenthesis syntax is a annoyance, and it often takes nothing less than black magic to correct unbalanced parentheses errors. Using LISP mode in EMACS certainly helps, but can still be a problem when the DEFUN statement containing the first parenthesis scrolls off the screen or comment lines confuse automatic EMACS indentation. Some well thought out tools are needed to address this problem, and the VAX LISP Development Group is indeed working on an integrated VAX LISP Editor which promises to just that. There is little else we can say stylistically at the present time, except to note that unbalanced parentheses are the major cause of LISP LOAD errors when the reader informs you that it hit an unexpected end of file.

5.5 Error Checking with the Compiler

We found that the compiler does considerably more error checking than the interpreter. The more extensive error checking of the compiler proved to be extremely helpful in avoiding run-time bugs of a somewhat esoteric nature. We therefore recommend that all VAX LISP code be run through the compiler for a quick syntax check prior to attempting run-time debugging.

6 A GENERAL COMPARISON OF THE TWO LISPS

The discussion thus far has made many comments comparing MacLISP and VAX LISP details. The comments in this section are more general in nature, but they should not be construed as an exhaustive review of both LISP dialects. Our purpose in presenting this information is simply to provide information for other programmers who may undertake conversion tasks similar to ours.

6.1 Functionality

The functionality of VAX LISP is superior to that of DECsystem-20 MacLISP in almost every regard. The preceding sections have pointed out a number of instances which demonstrate increased functionality, and there is little more we can add here. Suffice it to say that if you can do it in MacLISP, you can surely do it in VAX LISP, and perhaps with greater simplicity. One notable exception is single character input, which is discussed in Section 6.3 of this report.

6.2 Performance

Performance is a different story. We are currently running Version X0.2-8 of VAX LISP on a VAX-11/780 with 4 MB memory, 3 RP06's, and an average daytime load of 40 interactive users (not counting system jobs). This is a relatively heavy load by any measure. Now, there is some question as to whether any LISP with the richness of VAX LISP could run on such a heavily loaded system with reasonable response time. Leaving that question aside, one must be prepared to pay a significant performance price to use VAX LISP, particularly on heavily loaded systems.

We have performed a number of timed tests to get a handle on VAX LISP's speed, but we have since found that most of the results of these tests could be altered dramatically by adjusting system and process parameters. For example, we wrote a function that analyzed a directed graph of 50 nodes to create two 50-element arrays specifying all of the predecessors and successors of each node (see AI/CAI Project Progress Report No. 3 for a discussion of this structure). Running this function with a working set limit of 500 and using the default LISP memory allocation of 10000 pages required over 29 minutes of CPU time (including about a dozen garbage collections), but changing the working set limit from 500 to 1500 and running LISP with a memory allocation of 13000 pages reduced this to less than 2 minutes of CPU time without any garbage collections.

Suffice it to say that VAX LISP performance is highly dependent on specific system configurations and parameters, particularly memory and load. The VAX LISP Development Group assures us that

pending documentation will include detailed information and suggested parameter settings to help users optimize LISP performance on their systems.

6.3 The Special Case of Single Character Input

We use single character input for all user entries in our software because we want the keypad keys to be active at all times. Our CAI courses define the keypad keys to perform special functions such as:

- display a HELP message for the CAI program in general (the PF2 key),
- EXIT the course (the PF3 key),
- display ADVICE on the current subject matter context (the keypad "0" key), and
- REPLOT the screen (the keypad "." key).

We want the program to respond immediately to these keys, so we can not use VAX LISP's standard READ-CHAR or READ-LINE functions, because these require the student to press RETURN before any input is actually processed.

Our approach to this problem in MacLISP was to put the terminal into single character input mode with the standard MacLISP function:

```
(SSTATUS LINMODE NIL)
```

Once this statement is executed, the MacLISP function READCH returns a character as soon as one is typed rather than waiting for the user to press RETURN. We also enveloped the READCH function call with DO-WITH-TTY-OFF, a standard MacLISP package that inhibits character echoing. We then wrote our input input handler to echo desired characters, process the DELETE key and CTRL/U correctly, and read escape sequences when one of the keypad keys was pressed. This was a sizable job, but the result was a very clean routine to get student input with the keypad keys active.

Since VAX LISP does not contain STATUS and SSTATUS commands, converting this routine required a large number of changes. Our basic approach was to put the terminal into PASSALL and NOECHO modes with the SET-TERMINAL-MODES statement:

```
(SET-TERMINAL-MODES TTY-STREAM :PASSALL T :ECHO NIL)
```

The VAX LISP READ-CHAR function then returned a character as soon as one was typed. The main problem with using this approach

occurred during debugging. UNWIND-PROTECT was not present in Version X0.2-8, so each time an error occurred the terminal remained in PASSALL and NOECHO modes. We finally wrote a RESET function to reset the terminal and typed this "blind" in response to the breakpoint so that we could interact with LISP normally. Once the program was fully debugged, the VAX LISP approach worked as well as the MacLISP one.

Gary Brown has informed us that UNWIND-PROTECT will certainly be present in the first released version of VAX LISP, but has also advised us that SET-TERMINAL-MODES should be wrapped in an error handler as well as UNWIND-PROTECT. With this approach, he suggests preceding a call to SET-TERMINAL-MODES with a call to GET-TERMINAL-MODES to save the current terminal settings. When an error occurs, the error handler should reset the terminal to the save settings and then call the debugger via BREAK. Using UNWIND-PROTECT and error handlers in conjunction in this manner would appear to provide maximum control in this situation.

There seem to be at least three better approaches to our problem than putting the terminal in PASSALL and NOECHO modes. The first of these is already in VAX LISP: the ability to define the ESCape key as an AST. Since all of the character sequences transmitted by the keypad (and arrow) keys begin with an ESCape character, we could have used regular input processing with either the READ-CHAR or READ-LINE commands and made the ESCape character cause an immediate jump (via an AST) to a special routine for handling these keys. We should investigate this possibility for future courseware.

The second possible approach would require the implementation of a new statement by the VAX LISP Development Group, something akin to READ-CHAR-HANG. VAX LISP already has a READ-CHAR-NO-HANG statement which is "exactly like READ-CHAR except that if it would be necessary to wait in order to get a character, NIL is immediately returned without waiting" (see the Common LISP Reference Manual). From a purely functional point of view, READ-CHAR-NO-HANG would give us the functionality we desire simply by creating a loop which called this function until a non-NIL value was returned. Given LISP's performance, however, we consider this an untenable approach.

The third possible approach would also require implementation of a new statement, something akin to BIND-TO-KEY function that is available in EMACS' customization language, MLISP. This function allows any sequence of characters to be bound to a function name, causing that function to be evaluated whenever the matching sequence of characters is detected in the input stream. For example, the four arrow keys can be made to perform functions analogous to those in EDT with the following statements:

```
(bind-to-key "previous-line" "\e[A")        ; up
(bind-to-key "next-line" "\e[B")            ; down
```

```
(bind-to-key "forward-character" "\e[C")   ; right
(bind-to-key "backward-character" "\e[D")  ; left
```

Gary Brown has informed us that we could implement this functionality ourselves using FUNCALL, so we intend to investigate the feasibility of this approach for our next AI/CAI software implementation.

6.4 The VAX LISP CALL-OUT Facility

The most important difference between MacLISP and VAX LISP from a functionality point of view is the ability to "call out" of VAX LISP to routines written in other VAX/VMS native mode languages. We used this capability to read an RMS indexed file that contained all of the ReGIS code for generating the IReGIS course screen displays. For those unfamiliar with this facility, the basic concept is to create a shareable library of executable code (.EXE format) that can be called from VAX LISP. We first tried to write the RMS interface in VAX BASIC, but it turns out that the VAX BASIC compiler contains an obscure bug that prohibits VAX LISP from accessing the shareable library correctly. We therefore rewrote the RMS interface in Fortran, and everything worked perfectly.

As described in AI/CAI Project Progress Report No. 3, the skill hierarchy for IReGIS contained 50 nodes. We expect the skill hierarchy for our next subject matter (GKS graphics on the VAXstation) to contain several hundred nodes. Given VAX LISP's performance, we are considering a variety of database techniques for dealing with this volume of data. Some of these techniques might best be programmed in PASCAL or other VAX native mode languages to gain speed, but we still have the problem of returning data to LISP. At present, only integers can be returned to LISP from a CALL-OUT. It would be nice to be able to call out to PROLOG, but that doesn't appear to be in the cards for a long time to come. (We recently obtained a copy of the LOGLISP functions from the University of Syracuse and need to evaluate their applicability to our work.) In any event, we are sure that the CALL-OUT facility will be crucial for making the performance our future courseware viable.

7 A TOPS-20-LIKE EMACS/LISP PROGRAMMING ENVIRONMENT FOR VAX/VMS

The implementation of VAX LISP as an incremental compiler makes it an extremely powerful tool for rapid program development. What was missing in Version X0.2-8 was an effective program editor. The VAX LISP Development Group is working on a tailored LISP editor to be integrated with LISP's interactive programming environment, but this new editor had not yet been implemented for Version X0.2-8. The environment described here will be obsolete when the integrated LISP editor is available, but a general discussion of its characteristics is still relevant with regards to effective programming environments.

To get around this shortcoming, we emulated the TOPS-20 EMACS/-LISP programming environment by running both LISP and EMACS in subprocesses and using a pair of interfacing programs written by Hal Shubin of DEC's AI Technology Center. This environment is shown schematically in Figure 1. Two subprocesses exist below the top level DCL process, one running EMACS and the other running VAX LISP, and we go back and forth between these as follows:

- From the top level DCL process, typing EMACS attaches to EMACS.

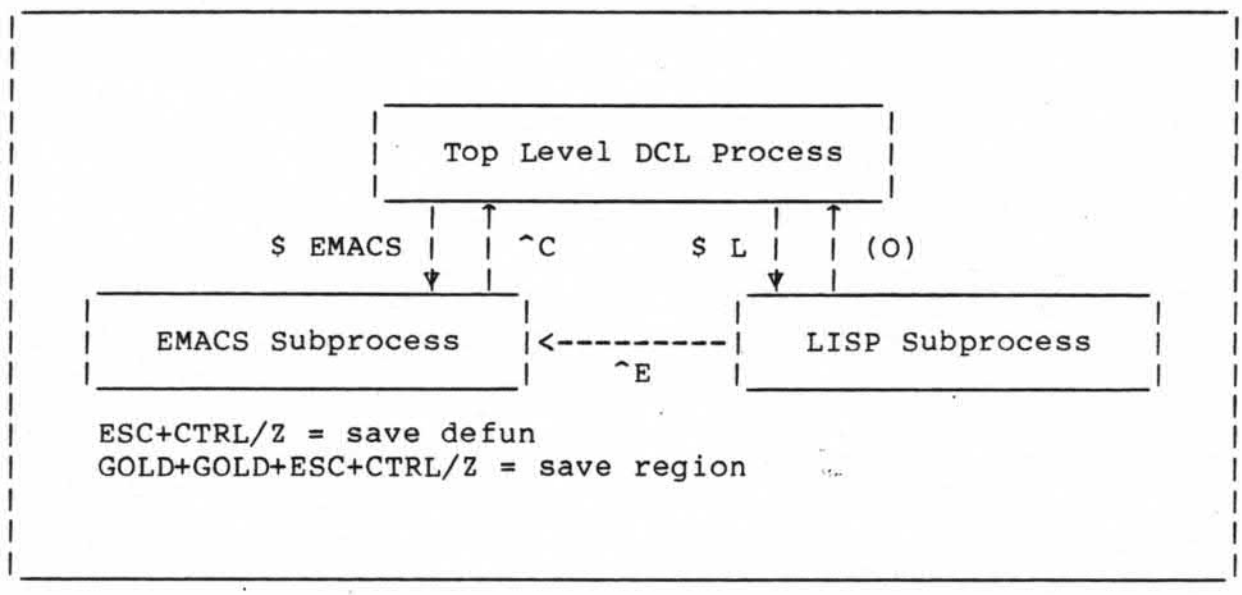

Figure 1

LISP/EMACS ENVIRONMENT AND INTERFACES

- From EMACS, typing CTRL/C executes a function (bound via the BIND-TO-KEY construction described earlier) that writes out all modified buffers, pauses EMACS, and returns to the top level DCL process.

- From the top level DCL process, typing L attaches to the subprocess running LISP.

- From LISP, typing (O) returns to the top level DCL process, or typing CTRL/E attaches directly to EMACS.

Hal Shubin has written two complementary programs that make this interface even more effective. The first is an EMACS MLISP program and makes it easy to save parts of a VAX LISP program that is being edited in a file called TOLISP.LSP in your SYS$LOGIN directory. Typing ESC+CTRL/Z saves the function indicated by your current cursor position. Prefixing this command with a non-zero argument saves the region defined by the "mark" and your current cursor position. (A default argument is typically indicated by typing CTRL/U, but this key combination is bound to the function EDT-ERASE-TO-BEGINNING-OF-LINE in our EDT simulator. We have therefore defined two presses of the GOLD (PF1) key to indicate a prefix argument.)

Shubin's second program is a VAX LISP program that defines CTRL/E as an AST that attaches to EMACS. (Before doing the actual attach, Shubin's program also does some nice cleaning up so that CTRL/G and CTRL/Y work correctly in the EMACS subprocess.) When the user returns to LISP after exiting via CTRL/E, this program automatically reads SYS$LOGIN:TOLISP.LSP if it is present.

Thus one can easily go from LISP into EMACS, save a function or region, pause EMACS, and automatically load the save code upon returning to LISP. (If EMACS had a ATTACH function, the need to return to the top level DCL process intermittently while returning to LISP could be eliminated.) We found this interface to be a godsend, as it allowed us to work much faster than would have been possible if we had to reload entire files each time a correction was made. This interface may become obsolete once the integrated VAX LISP editor is available, but we highly recommend it until it is.

Our general pattern for working with LISP and EMACS therefore looks as follows:

1. Spawn a new EMACS via KEPTEMACS.COM and load Shubin's TOLISP20.ML program.

2. Create or edit a LISP program.

3. Type CTRL/C to write out the modified buffers, pause EMACS, and return the top level DCL process.

4. Spawn a subprocess and run LISP in that subprocess.

5. Load Shubin's TOEMACS20.FAS file and the program just modified via EMACS.

6. Test the modified program.

7. Type CTRL/E to return directly to EMACS from LISP.

8. Edit the program.

9. Use ESC+CTRL/Z and/or GOLD+GOLD+ESC+CTRL/Z to save the modified function or region in SYS$LOGIN:TOLISP.LSP.

10. Type CTRL/C to write out the modified buffers, pause EMACS, and return the top level DCL process.

11. Type L to attach to the LISP subprocess. File SYS$LOGIN:TOLISP.LSP will be loaded automatically.

12. Loop back to Step 6.

8 FINAL REMARKS AND FUTURE PLANS

All-in-all, we were generally pleased with VAX LISP and the quality of the compiler. Using VAX LISP is clearly a very effective way of doing rapid prototyping. The speed with which we could go through the design, edit, and test cycle clearly makes for improved productivity. Whether particular products can be built using VAX LISP is an issue that still needs to be evaluated.

With the completion of the IReGIS conversion, we are now ready to move into designing a new AI/CAI course. This course will be designed to run on the VAXstation, and our current plan for subject matter is the newly emerging Graphics Kernal Standard, GKS. We are just beginning to look at this subject matter and hope to have a design spec similar to AI/CAI Project Progress Report No. 3 available by January. We welcome input from DEC's graphics as well as AI communities on the design and implementation of this course.

AI/CAI PROJECT PROGRESS REPORTS IN THIS SERIES

These reports are available from Jesse Heines, Educational Services Development and Publishing, Burlington FPO/A2, DTN 283-7634, Engineering Net CLOSUS::HEINES.

These AI/CAI reports are intended for Digital internal distribution only. Special versions of Reports 1 and 3 exist that have been approved for external distribution. If you intend to redistribute these reports to non-Digital personnel, please request the appropriate external versions.

1. Basic Concepts in Knowledge-Based Systems. April 20, 1982.

   External Version: Basic Concepts in Knowledge-Based Systems, published in Machine-Mediated Learning, 1(1):65-95, Spring 1983. Available as Educational Services Technical Report No. 13.

2. Where AI Can Fit in CAI. November 4, 1982.

3. The Design of a Prototype AI/CAI Course Employing a Rule-Based Tutorial Strategy. (Coauthored with Tim O'Shea of The Open University, Milton Keynes, England.) May 3, 1983.

   External Version: The Design of a Prototype AI/CAI Course Employing a Rule-Based Tutorial Strategy. (Coauthored with Tim O'Shea.) Available as Educational Services Technical Report No. 14.

4. An Initial Attempt at Building a Student Model Using Production Rules. June 2, 1983.

5. On Program Conversion from MacLISP to Common LISP. (Coauthored with George Poonen of Productive Information Management, Inc.) December 29, 1983.

Readers involved with AI languages may also be interested in "Logic and Recursion: The PROLOG Twist," coauthored with Jonathan Briggs and Richard Ennals of Imperial College, London (May, 1982). This paper was published in the November, 1983, issue of Creative Computing and is available as Educational Services Technical Report No. 15.