Hannes Goldynia
St. Martiner Str. 45
9500 Villach

muLISP/muSTAR-80tm
Artificial Intelligence
Development System
Reference Manual

August 1980

8701-200-02

muLISP/muSTAR-80 is distributed exclusively by
Microsoft
10800 N.E. Eighth, Suite 819,
Bellevue, WA 98004

A Brief History of muLISP

muLISP-80 represents the culmination of 4 years of effort into designing a general purpose LISP system. It was developed almost entirely on microcomputers for microcomputers. The original interpreter was completed in 1977 by Albert D. Rich using LISP 1.5 as a guide. It was intended solely for use in mechanical theorem proving, specifically for the propositional and predicate calculus of formal logic. This system, now called muLISP-77, worked quite well for this purpose, although somewhat slowly.

Through the foresight of David R. Stoutemyer, it became apparent that the potential uses for a microcomputer LISP were tremendous. In particular, the possiblity of implementing for the first time a symbolic mathematics system on a microcomputer was set as a goal. Influenced by the invaluable suggestions of Martin Griss and Peter Deutsch, joint work by Rich and Stoutemyer yielded a greatly enhanced and robust LISP system. It included infinite precision arithmetic, streamed disk file I/O, and a powerful function body evaluation mechanism. This product was released in June of 1979 as muLISP-79 by The Soft Warehouse, a partnership set up by Stoutemyer and Rich to facilitate the widespread distribution of the software.

The success of muLISP-79 and its companion product the muSIMP/muMATH-79™ Symbolic Math System encouraged further work on improving the systems. The need for greater code density and faster loading capability resulted in the addition of a pseudo-code compiler and interpreter for the muLISP-80 system. A major effort went into making the documention for muLISP-80 of the same high caliber as the software. The clarity and completeness of this manual is due in large part to the persistence of Joan H. Rich.

Currently work is being done to make muLISP available for other popular micro and minicomputers. The advent of the 16 bit microprocessors will make possible the greatly improved performance and data space sizes required to satisfy the needs of ever larger AI research efforts. We intended to fulfill those needs.

August 2, 1980

LISP / STAR

Preface

The LISP computer language is based on a paper by John McCarthy [1960] entitled "Recursive Functions of Symbolic Expressions and Their Computation by Machine". Since its first implementation at the Massachusetts Institute of Technology in the early 1960's, LISP has remained the "machine language" of the AI (Artificial Intelligence) community. The language and its many derivatives continue to monopolize all serious work in such diverse fields as robotics, natural language translation, theorem proving, medical diagnosis, game playing, and program verification. LISP is the language of choice for such attempts at mechanical intelligence for the following reasons:

1. LISP is an applicative, recursive language which makes it an ideal formalism for describing complex mathematical concepts.

2. The principal data structures in LISP are binary trees. Such abstract objects can be made isomorphic to (i.e. a one-to-one model of) the actual data in most AI problems. Once this is accomplished, the properties of the original problem can be investigated by performing transformations on the LISP data structures.

3. When a computer is programmed to simulate intelligence, it must be able to respond to queries of arbitrary difficulty. The static storage allocation schemes of conventional programming languages makes it very difficult to deal with such open ended problems. These difficulties are alleviated in LISP by dynamic allocation and recycling of data storage resources by means of automatic garbage collection.

4. A highly interactive environment is essential for intelligent human-machine communication. The ease with which LISP function definitions can be regarded as data encourages incremental system development and makes LISP an ideal interactive language.

The development and distribution of muLISP-79 by The SOFT WAREHOUSE helped to make LISP available to the rapidly growing community of microcomputer users. This was the first efficient, "production" version of LISP for such computers and is being used in a wide variety of applications.

The challenge to meet the ever increasing memory and speed requirements of most AI software systems inspired the development of muLISP-80. An increase in code density by a factor of three and a 20% increase in execution speed has been achieved by means of a pseudo-code compiler and interpreter. Since compilation and de-compilation occur automatically, the process is invisible to the user. Thus, the interactive nature of muLISP, so essential for most AI applications, is not sacrificed in the interest of efficiency. Finally, the addition of the muSTAR-80 AI Development System to muLISP rounded out the package with a resident display-oriented editor and debugging facilities.

Table of Contents

Section I: An Introduction to muLISP-80

Congratulations on your purchase of the muLISP-80 Artificial Intelligence Development System (muLISP-80 AIDS). This system is a revolutionary and sophisticated software package for microcomputers. It has been designed to be capable of supporting a wide range of serious AI research efforts. Some degree of study and patience is required to properly use muLISP-80 as a development tool for the large software systems required for such applications.

This section of the muLISP-80 AIDS Reference Manual provides the minimum information necessary to load and use the system. The remainder of the manual provides a detailed explanation of muLISP data structures, memory management, and primitively defined functions. Every attempt has been made to make the manual as clear and precise as possible; however, it is not a tutorial on the LISP programming language. The best way to learn muLISP is by exploratory use of the system in parallel with study of this manual. If this reveals an insufficient knowledge of LISP on the part of the user, several good references are given in the bibliography at the end of this manual.

A. Major Features

1. A total of 83 LISP functions are defined in machine language for maximum efficiency. These functions provide an array of data structure primitives including a full complement of selectors, constructors, modifiers, recognizers and comparators. (See section V-A,B,C,D,E)

2. Infinite precision integer arithmetic, expressed in any desired radix base from 2 through 36, is supported by a complete set of numerical primitives. (See section V-L)

3. A two-pass compacting garbage collector performs automatic, dynamic memory management on all data spaces. A garbage collection typically requires less than half a second to complete. (See section III-B)

4. Dynamic reallocation of data space boundaries occurs automatically to most efficiently use all available memory resources. (See section III-C)

5. Program control constructs include an extended COND, a multiple exit LOOP, and a powerful function body evaluation mechanism. These features permit programs to be written in an elegant, pure LISP style while still allowing the efficiency of iteration when it is applicable. (See section V-O)

6. LAMBDA defined functions can be made either call-by-value (CBV) or call-by-name (CBN). In addition functions can be defined as being either spread or no-spread. (See section IV-B)

7. In addition to muLISP's interactive environment, program debugging is facilitated by a resident display oriented editor and a trace package. (See section VII)

8. muLISP is fully integrated into Digital Research's CP/M™ Disk Operating System and such upward compatible successors as Cromemco's CDOS™, and IMSAI's IMDOS™.

9. muLISP requires only 9K bytes of machine code storage, leaving the remainder of the computer's memory available for data structures. A minimum system will run in as little as 20K bytes of computer memory.

10. Extremely fast program execution speeds have been achieved through the use of such techniques as shallow variable binding, address typed data structures, and a closed pointer universe. (See section II)

11. Function definitions are automatically compiled into distilled code or D-code when they are defined. The inverse process of de-compiling occurs automatically when a definition is retrieved. This compilation results in a threefold increase in code density over storage as a linked list and about a 20% improvement in execution speed. (See section V-J)

12. Numerous I/O control variables have been included to handle such issues as upper/lower case conversion, console edit mode, and the printing of quoted strings. (See section V-M,N)

13. A means is provided to conveniently link to user-defined machine language subroutines. (See section V-J)

B. The Master Diskette

The muLISP-80 System is distributed for microcomputers as a set of disk files on CP/M formatted floppy diskettes. The executable command file MULISP.COM is an object code version of the muLISP interpreter and compiler. Also included on the diskette are the following muLISP SYS and library files:

MUSTAR.SYS	The muSTAR AI Development System
UTILITY.LIB	An assortment of utility functions
TRACE.LIB	A function-trace debugging package
METAMIND.LIB	A sample program for the MasterMind game
ANIMAL.LIB	A sample program for the Animal game
DOCTOR.LIB	An implementation of the Doctor program

As soon as possible after receipt of your muLISP-80 diskette, make a copy of the master on a blank diskette for use as a working copy. An appendix to this manual provides more information on how this can be accomplished. Once copied, the master diskette should be kept in a safe, cool place to be used only in emergencies.

C. The Basic Interaction Cycle

Once the master diskette is safely backed up, it is a simple matter to initiate execution of muLISP. First bring up the computer's disk operating system in the normal manner. Next, if necessary, switch to the drive with the copy of the muLISP-80 diskette. Then enter the following operating system command, terminated by a carriage return:

```
A>MULISP
```

After a few seconds of load time, the system should respond with a logon message of the following form:

```
muLISP-80  (8080 Version mm/dd/yy)
Copyright (c) 1980 The SOFT WAREHOUSE
$
```

where appropriate numbers appear for the version month, day, and year. This version date should be included in all inquiries concerning the system. Naturally, the Z80 version of muLISP-80 will have "Z80" instead of "8080" in the logon message.

muLISP prompts the user with a dollar sign to indicate readiness to accept character input from the console. After a complete expression followed by a carriage return is typed by the user, muLISP evaluates the expression, and prints the resulting value beginning on a new line. This interaction cycle is repeated indefinitely until a CTRL-C is typed, returning control to the disk operating system.

Since muLISP uses the line editing routines of the host computer's operating system, all the applicable features of that system are inherited by muLISP. Backspacing is usually accomplished by typing a CTRL-H, a RUBout, or a DELete. Some systems echo the deleted character; others erase the character from the screen and backspace the cursor. Entire lines can be deleted or flushed by typing a CTRL-U or a CTRL-X. By typing a CTRL-P, all subsequent muLISP console output will also be sent to the system's printer.

In general only the current line can be edited. There is no way to modify a line of input once a carriage return has been typed. However a runaway program can be interrupted as described in the next section.

D. The Executive Driver Loop

The default muLISP driver loop is an eval-LISP executive. First the prompt string, "$ ", is displayed, indicating the system is waiting for console input. The user can then enter an expression terminated by a carriage return. Multi-line expressions can be entered since an expression is not considered complete until all parentheses are balanced. Once entered, an expression is read using the function READ, evaluated using the function EVAL, and then the result is printed using the function PRINT. The following is a sample muLISP dialogue demonstrating the basic driver loop:


```
$ DOG
DOG

$ (PLUS 5 -2)
3

$ (EQUAL DOG CAT)
NIL

$ (MEMBER DOG (QUOTE (CAT COW DOG PIG)))
T
```

Often it is advantageous to define an executive driver loop especially suited for a particular application. This can be accomplished simply by redefining the function DRIVER. For example, an eval-quote-LISP driver is given in the library file UTILITY.LIB. If an error or interrupt occurs, program control will return to the user defined driver rather than the perhaps inappropriate default driver loop.

E. muLISP Programming

The following muLISP dialogue illustrates how a LAMBDA defined function can be defined and then used:

```
$ (PUTD (QUOTE FACTORIAL) (QUOTE (LAMBDA (N)
    (COND
      ((ZEROP N) 1)
      (T  (TIMES N (FACTORIAL (DIFFERENCE N 1))))) ) )))
FACTORIAL

$ (FACTORIAL 5)
120
```

This definition of the factorial function is defined in the style of the original LISP as described in McCarthy's LISP 1.5 Programmer's Manual [1962]. Although it does not fully utilize the capabilities of muLISP, it is a perfectly acceptable definition.

Study of this manual and the muLISP library files will reveal that muLISP-80 incorporates numerous upward compatible extensions of LISP 1.5. For the most part these extensions consist of defining useful, well-defined results for which the original LISP is noncommittal. They significantly increase readability and execution speed, while substantially decreasing the storage requirements for function definitions. The following are a few of the significant extensions:

1. As a name is read or generated by muLISP, its value is automatically set to itself. This self-referencing of new names is called auto-quoting. It reduces the need for using the QUOTE function when the EVAL-LISP executive driver loop is being used for input.

2. The COND function has been generalized so all the expressions following the predicate are evaluated in turn. The value returned by this extended COND is the value of the last expression.

3. The evaluation algorithm for a function body or lambda expression includes an implied COND. This obviates the need for explicit use of COND within function definitions. For example, compare the following definition of FACTORIAL with that given above:

```
(PUTD FACTORIAL (QUOTE (LAMBDA (N)
    ((ZEROP N)  1)
    (TIMES  N  (FACTORIAL (DIFFERENCE N 1))) )))
```

With the principal exception of the PROG program control construct, the original LISP was an exemplary applicative and structured language long before these adjectives became popular. muLISP has two features which dispose of any need for such unstructured control features:

1. The powerful multiple-exit LOOP function permits the programming of non-recursive loops. This is achieved without the use of the totally unstructured GO feature.

2. Arguments in a function's formal argument list that are in excess of the actual number of arguments used in a call to the function are simply bound to NIL. These excess arguments are available for use as local variables within the function; hence PROG is not needed to establish such local variables.

For these reasons muLISP-80 has no primitively defined PROG, GO, or RETURN constructs. Instead of having such explicit control constructs, program control within a muLISP function body is guided primarily by the structure of the linked list representing the function's definition. Implicit program control results in function definitions whose appearances are uncluttered and whose meanings are more transparent. See section V-O for a detailed explanation of the muLISP evaluation mechanism.

As an example of how to avoid the use of these constructs and thereby write more structured LISP programs, consider the following definitions of the FACTORIAL function:

```
(PUTD FACTORIAL (QUOTE (LAMBDA (N)
    (PROG (M)
        (SETQ M 1)
A       (COND
            ((ZEROP N) (RETURN M)) )
        (SETQ M (TIMES M N))
        (SETQ N (DIFFERENCE N 1))
        (GO A) ) )))
```

```
(PUTD FACTORIAL (QUOTE (LAMBDA (N M)
    (SETQ M 1)
    (LOOP
        ((ZEROP N) M)
        (SETQ M (TIMES M N))
        (SETQ N (DIFFERENCE N 1)) ) )))
```

The first is a conventional non-recursive definition. The second is the equivalent muLISP-80 definition, also non-recursive. The reader can decide for him/her self which definition is more elegant and structured. It is interesting to note that the muLISP definition requires only 29 nodes; whereas, the conventional version requires 38. This is not an insignificant ratio for a LISP system running in a very limited address space.

F. Interrupting Program Execution

At any time during the execution of a program, a user initiated software interrupt will halt program execution. This may be necessary to stop a "runaway" or non-terminating program and return control to the console. An interrupt is initiated by depressing either the ESCape key or ALTmode key. The following options available message will then be displayed on the console:

```
*** INTERRUPT:  To Continue Type: RET;
Executive: ESC, ALT;  Restart: RUB, DEL;  System: CTRL-C?
```

The user may then choose one of the following options by typing the appropriate option character:

1. The Continue option causes program execution to continue from the point of the interrupt. This option is selected by depressing the RETurn key.

2. The Executive option returns control to the current executive driver loop. All variable bindings, function definitions, or property values are preserved. This option is selected by depressing either the ESCape or ALTmode key. For terminals with neither of these keys, typing a CTRL-[(i.e. typing a left bracket while holding down the control key) is equivalent to ESCape.

3. The Restart option restarts muLISP from scratch and destroys all variable bindings, non-primitive functions, and property values. This option is selected by depressing either the DELete or RUBout key. Typing a CTRL-H is equivalent to depressing the DELete key.

4. The System option terminates muLISP and returns control to the disk operating system. This option is selected by typing CTRL-C.

muLISP output to the console can be interrupted and then restarted using CTRL-S as a toggle. This output pause is useful to permit reading of the text prior to its being scrolled off the screen.

G. Error Diagnostics

There is only one situation in muLISP-80 for which there is no satisfactory recovery other than program termination and an error trap. The exhaustion of all available storage in the four data spaces will result in an error trap. See section III-D for a discussion of the trap and the options then available to the user.

A less serious problem will cause a warning message to be displayed on the console. The primitive function in which the error occurs will return a value of NIL. It is then the responsibility of the user program to recognize the error and take the appropriate action. The following are the three possible warning messages. Their causes are fully described in the indicated sections of this manual:

ZERO Divide Error	Section V-L
End-Of-File Read	Section V-M
No Disk Space	Section V-N

H. Environment SYS Files

The muLISP function SAVE is used to save the current environment for retrieval at a later time. The environment consists of all the currently active muLISP data structures, including atom values, property values, and function definitions. The environment is saved as a disk file of type SYS. For instance, the following command will generate a SYS file named WHALE.SYS on the current drive:

```
$ (SAVE (QUOTE WHALE))
```

This environment can be restored at any later time by using the LOAD function. If the file WHALE.SYS is now on drive B, the following muLISP command will load the SYS file saved above:

```
$ (LOAD (QUOTE WHALE) (QUOTE B))
```

Alternatively, the following operating system command can be used to load the SYS file WHALE.SYS from drive B after loading muLISP:

```
A>MULISP B:WHALE.SYS
```

Either of the above methods for loading a SYS file will restore the environment exactly as it was at the time of the save. When a SYS file is loaded, the various data spaces are re-allocated according to the computer's current memory size. This means that the current memory size does not necessarily have to be the same as when the SYS file was created.

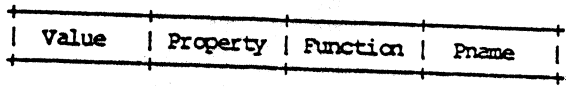
A SYS file can be useful in a variety of ways. For instance, an interactive session can be continued at a later time, intermediate results of a session can be backed up as insurance against computer failure, or a set of function definitions developed interactively can be preserved. Finally, program development is usually done by creating muLISP source files and reading them in using the RDS command. However, once the program has been perfected, reading it in each time can be tedious especially for an end-user. If a SYS file has been generated containing the source, the application program can be loaded both quickly and conveniently by the end-user.

Section II: Primitive Data Structures

muLISP has three distinct types of primitive building blocks, collectively called data objects. A data object is either a muLISP name, number, or node. Using the recognizer functions, the type of any data object can be determined. All the data objects of a given type consist of a fixed number of pointer cells. A pointer cell can point or refer to other data objects or to special purpose data structures. Thus the set of data objects can be envisioned as an interconnected network of pointers called a pointer universe.

It is important to note that all three types of data objects have a car cell and a cdr cell. Moreover, in muLISP the car and cdr cells are restricted to point only to other objects within the pointer universe. Thus by following only car and cdr cell pointers, there is no danger of wandering outside this closed universe. The advantage of this closed pointer universe is a simpler and more logical muLISP evaluation mechanism. For instance, it eliminates the time required for constant type checking on the part of the primitive LISP functions.

A. Names



A name is a recognizable data object consisting of four pointer cells. Names are uniquely stored in the sense that no two names in the system can have identical print-names. The use made of each of the four cells is as follows:

1. The car or value cell contains a pointer to the current value of the name as recognized by the evaluation functions. When a name is created, its value cell is initialized to point back to the name itself. This automatic self-referencing of a name, called auto-quoting, often eliminates the need for explicitly using the QUOTE function in function definitions. The assignment functions are used to change the value cell of a name. The value cells of a function's formal arguments are temporarily reassigned when the function is called, and then restored to their original value when the function is exited.

2. A name's cdr or property list cell contains a pointer to the property list of the name. This list is used and modified primarily by the property list and flag functions. Flags on a property list can be distinguished by their being atomic elements of the list (i.e. names or numbers). In contrast, properties are non-atomic elements of the list (i.e. nodes). The car of a property element points to the property's indicator and the cdr points to the property's value. When a name is created, its property list is set to NIL, indicating that no flags or properties are present.

3. A name's function cell contains a pointer to the current definition of the name. This function definition can either be a machine coded routine or the D-code representation of a LAMBDA expression. When a function application is to be made, the function cell is used by the muLISP evaluation mechanism to locate the name's definition. Access to the cell is limited to the function definition primitives, which are used to retrieve and modify function definitions. When a name is created, its function cell points to an undefined function trap.

4. A name's print name cell contains a pointer to the string of ASCII characters used to print the name. Access to this cell is restricted to the I/O and sub-atomic functions. When a print name string is read in or generated by the system, a check is first made to see if a name already exists with that print name. If so, the existing name is used. If not, another name is created using the new print name. Once created, a name's print name string cannot be modified.

B. Numbers

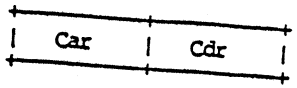
A number is a recognizable data object consisting of three pointer cells. Numbers are <u>not</u> uniquely stored in the sense that equivalent number vectors can occur in the system. The use made of each of the three cells is as follows:

1. The car or value cell of a number contains a pointer to the number itself. Thus numbers do not have to be quoted, since they evaluate to themselves. Naturally the contents of this cell can be changed by using an assignment function; however, there is no reason to do so and it is not a recommended thing to do.

2. The cdr or sign cell of a number points to NIL if the number is non-negative; otherwise, it points to TRUE. The value of this cell is established when a number is created.

3. The number vector cell contains a pointer to the binary number vector which establishes the number's numerical value. The number vector itself consists of a single-byte byte counter followed by the requisite number of bytes required to express the number in binary. The size of the byte counter limits the magnitude of numbers to 256^254-1 or approximately 10^611.

C. Nodes



Binary trees are the primary data objects in muLISP. Internally a binary tree is implemented as a network of cell pairs called nodes. Each node consist of a car cell and a cdr cell. As mentioned earlier, the node's cells may only point to other bonafide muLISP data objects: a name, a number, or a node.

Traditionally nodes have been called dotted pairs in deference to the dot notation used to represent them in print. The expression (X . Y) represents a node whose car cell points to the object X, and whose cdr cell points to the object Y. The dot notation is perfectly capable of expressing any LISP data structure.

It is often more convenient to think of data as a linked list of elements, rather than as a deeply nested binary tree structure. The structure represented by the list (X1 X2 X3 ... Xn) can be expressed using dot notation as

(X1 . (X2 . (X3 . (... . (Xn . NIL)...)))).

Thus a list consists of a chain of nodes linked through their cdr cells. The name NIL is used in the last cdr cell to terminate the list. NIL is also used to denote the null or empty list. Thus the expression () will be read as NIL by the function READ and printed as NIL by the functions PRIN1 and PRINT.

The function READ will accept both the dot notation and the list notation for expressing data structures. However, the functions PRIN1 and PRINT will use the list notation to the maximum extent possible when printing structures.

Section III: Memory Management

Dynamic, invisible memory management gives LISP much of its inherent power. This frees the programmer from concerns about allocating sufficient memory for a given problem. Such allocation before run-time is difficult, if not impossible, for most AI applications since the problems are generally not of a predetermined size. Memory management is accomplished in three phases by muLISP-80.

A. Initial Data Space Partition

During the initialization phase of muLISP, the amount of memory available to the system is computed. The memory is then partitioned into four distinct data spaces based on the ratios given in the following table:

Ratio	Space	Contents
4:32	Atom space	Name and number pointer cells
3:32	Vector space	Print-name strings and number vectors
23:32	Pointer space	D-code and nodes
2:32	Stack space	System control/value stack

To create data objects required for a running program, space is taken from one of the above spaces. Space for a new name's or a new number's pointer cells is taken from the atom space. At the same time space for the associated print-name string or number vector is taken from the vector space. The pointer space, which is by far the largest, provides storage for both D-code and nodes. The combined control stack and value stack is located in the stack space. The above ratios approximate the relative use made of the spaces by most application programs.

B. Garbage Collection

The nature of stack operations makes management of the stack space automatic and continuous. However, management of the remaining three spaces requires explicit recycling of data objects which are no longer needed. New data objects are constantly created during the execution of a LISP program, while others are implicitly discarded when they are no longer referenced by active structures. When the creation process uses up all available resources within a space, a garbage collection is performed. The storage space vacated by discarded data structures is then reclaimed, so the process may continue.

In muLISP-80 the exhaustion of resources in either the atom, vector, or node spaces will cause a collection to occur. The first pass of the collection consists of marking those data structures accessible from the value and property list cells of all system names and from the variable stack. During the second pass, the marked or active data objects are compacted into one end of their respective data spaces. This leaves the remainder of the spaces available for new objects.

Although garbage collection is automatic, it is not entirely invisible to the user since it periodically causes a pause in the execution of a program. Less than half a second is required for the collection process in a 48K byte muLISP-80 system using a 4MHz clock. The time varies linearly with the computer's memory size and clock speed. Normally this is of no concern to the application level programmer; however, it should be considered in the design of real time systems.

C. Reallocation of Data Space Boundaries

If after a garbage collection there is insufficient free storage within a data space to continue a process, the partitions of all the spaces are reallocated to give more memory to the exhausted space. Thus muLISP can respond to changing demands placed on the various data spaces by differing application programs.

D. Insufficient Memory Trap

Normally, automatically invoked garbage collections and dynamic reallocation of data spaces will provide sufficient storage in each space to continuously satisfy the demands of user programs. However, if the memory requirements for storing data objects finally exhaust all available resources, an insufficient memory trap will occur. Since every other problem that arises in muLISP has a satisfactory recovery, this is the only situation that causes an error trap. The trap displays the following message on the console:

```
ALL Spaces Exhausted
Executive: ESC, ALT; Restart: ROB, DEL; System: Ctrl-C?
```

The user may then choose the desired option by typing the respective option character. The Executive option is the least drastic since it merely causes control to return to the LISP executive driver loop, without changing function definitions, property values, etc. This is the most common response to the error trap. The Restart option destroys all non-primitive muLISP functions, property values, etc. and restarts muLISP from scratch. Finally, the System option terminates muLISP and returns control to the parent operating system.

A phenomenon known as thrashing occurs when the system is forced to spend an inordinate amount of time garbage collecting and reallocating data spaces for a very small net return. The symptom of thrashing is greatly increased execution time for a given task. This can only be resolved by increasing the computer's memory size or decreasing the amount of program and data storage requirements.

IV. The muLISP Meta-language

The applicative nature of the language makes LISP an ideal formalism or meta-language for the precise specification of both natural and computer languages. In fact LISP can be used as its own meta-language! The use of LISP in this way dates back to John McCarthy's original investigations into the language, and the practice has continued to the present. When LISP is used as a meta-language, it is customarily written in a more natural, high-level syntax called meta-LISP. While borrowing heavily from the concepts discussed in Allen's Anatomy of LISP [1978], the muLISP meta-language has been syntactically enhanced to more clearly reflect the extended evaluation capabilities of the underlying muLISP.

A. Meta-syntax

The following table defines the syntax of the muLISP meta-language using Backus-Naur Form (BNF) first described in the ALGOL 60 Report [1963] and summarized in an appendix to this manual.

```
<definition>   ::= <ftype> <identifier> [<var-list>] := <body>;
<ftype>        ::= CBV | CBN
<var-list>     ::= <variable>, ..., <variable>
<body>         ::= <clause>, ..., <clause>
<clause>       ::= <form> | <conditional>
<conditional>  ::= <form> --> <body>;
<form>         ::= <constant> | <variable> | <application> |
                   <assignment>
<assignment>   ::= <variable> <-- <form>
<application>  ::= <identifier> [<form>, ..., <form>]
<variable>     ::= <identifier>
<constant>     ::= NIL | TRUE | LAMBDA | NLAMBDA | <number>
<identifier>   is any muLISP name  (see Section II.A.)
<number>       is any muLISP number  (see Section II.B.)
```

This syntax will be used in Section V in the description of the primitive muLISP functions. However, the formal syntax rules will occasionally be supplemented in the interest of improving readability. For instance, the logical operators "AND" and "OR", and the numerical operators "+", "*", etc. will be written in their conventional infix forms.

B. Meta-semantics

1. The meta-LISP constant NIL denotes the corresponding muLISP name signifying both the empty list, ( ), and the truth value false. The constant TRUE denotes the muLISP name T signifying the truth value true.

2. The function type, <ftype>, indicates the argument evaluation scheme used in calls on the function. If a function is typed as call by value (CBV), then the arguments

in the function call are evaluated prior to passing them to the function. On the other hand, call by name (CBN) functions receive their arguments from the call without evaluation.

3. Form evaluation is predicated on the particular type of form involved. The value of a constant is that constant. The value of a variable is the contents of the value cell of the variable's name. The form associated with the right hand side of an assignment is first evaluated and then the value of the variable is set to the result. Function applications proceed as follows:

a. If the function is CBV, the arguments are successively evaluated from left to right.

b. The value of the function's formal arguments are replaced by the actual arguments, evaluated or unevaluated as applicable, and the old values are saved. Extra formal arguments which have no corresponding actual arguments are set to NIL.

c. The function is then applied as described below, and the result saved.

d. The values of formal arguments are restored to their original value.

e. The result of the application is returned.

4. The body of a function definition or of a conditional statement consists of zero or more clauses separated by commas and terminated by a semi-colon. In the application of a function, its clauses are successively evaluated as follows:

a. If a clause is a form it is evaluated as described above.

b. If a clause is a conditional, the predicate form is evaluated. If it evaluates to NIL, evaluation proceeds as normal to the next clause in the current clause list. Otherwise, the body within the conditional replaces the original body as the current clause list and the successive clause evaluation continues.

This process continues until the end of a clause list is reached, at which point the value of the last clause is returned as the value of the application.

Section V: Primitively Defined Functions

This section describes in detail all of the functions which have been implemented as machine language subroutines. In addition to the functions actually accessible to the user, several auxiliary functions are also defined. These "helper" functions are introduced solely to simplify and clarify the definitions of the functions directly accessible to the user. Only the accessible functions are numbered in the definitions below and indexed in the appendix.

The muLISP meta-language described in Section IV is used as much as possible to define the effects and value of each function. Digressions to English language text are made only when an irreducible, primitive concept must be introduced. The interpretation which follows many of the definitions is an attempt to give an informal description of what the function does and how it is typically used.

A. Selector Functions

Selector functions are used to select a desired sub-tree from a given binary tree. This gives a method for extracting information from the primary LISP data structure. Using the functions of this group, any desired sub-tree or terminal node of a given tree can be reached. The functions CAR and CDR return the car and cdr branch of a tree respectively. Successive applications of these two functions is sufficient to traverse any tree. As indicated below, the remaining functions are merely compositions of CAR and CDR. They are defined in machine language primarily for efficiency and convenience.

1. CBV CAR [X] :=
the structure pointed to by the car cell of X;

Interpretation: The correct interpretation of the CAR of an expression depends on whether that expression is an atom or not, and if not whether it is thought of as a list or a binary tree. If X is an atom, then CAR [X] returns the current value of X. If X is a list, then CAR [X] returns the first element of that list. Finally if X is a binary tree, then CAR [X] returns the left or car branch of the tree.

2. CBV CDR [X] :=
the structure pointed to by the cdr cell of X;

Interpretation: Remarks similar to those above apply to the interpretation of the CDR of an expression. Thus if X is an atom, then CDR [X] returns the property list of X. If X is a list, then CDR [X] returns the tail or everything but the first element of that list. If X is a binary tree, then CDR [X] returns the right or cdr branch of the tree.

3. CBV CAAR [X] :=
   CAR [CAR [X]];

4. CBV CADR [X] :=
   CAR [CDR [X]];

Interpretation: This function returns the second element of a list.

5. CBV CDAR [X] :=
   CDR [CAR [X]];

6. CBV CDDR [X] :=
   CDR [CDR [X]];

7. CBV CAAAR [X] :=
   CAR [CAR [CAR [X]]];

8. CBV CAADR [X] :=
   CAR [CAR [CDR [X]]];

9. CBV CADAR [X] :=
   CAR [CDR [CAR [X]]];

10. CBV CADDR [X] :=
    CAR [CDR [CDR [X]]];

Interpretation: This function returns the third element of a list.

11. CBV CDAAR [X] :=
    CDR [CAR [CAR [X]]];

12. CBV CDADR [X] :=
    CDR [CAR [CDR [X]]];

13. CBV CDDAR [X] :=
    CDR [CDR [CAR [X]]];

14. CBV CDDDR [X] :=
    CDR [CDR [CDR [X]]];

B. Constructor Functions

Constructor functions are used to generate the data structures required for the solution of a particular problem. In LISP such structures are realized as a tree or linked list. These can be designed to closely reflect or model the data structure of virtually any problem. The principal member of this group, the CONS function, creates a new node. The storage required for this node is taken from the area of memory called the Pointer Space. If previous consing has exhausted the Pointer Space, a garbage collection is automatically performed to reclaim the space used by data structures which are no longer required.

1. CBV CONS [X, Y] :=
   creates a node or cell-pair whose car cell points to X and whose cdr cell points to Y;

Interpretations: The correct interpretation of CONS depends on how the data structure being built is conceived. When the structure is thought of as a list, CONS [X, Y] returns the list whose first element is X and whose tail is Y. If the structure is a binary tree it returns the tree whose left or car branch is X, and whose right or cdr branch is Y. Note that this in no way alters the structures X or Y.

2. CBN LIST [X1, X2, ..., Xn] :=
   if n=0 —> NIL;,
   CONS [EVAL [X1], LIST [X2, X3, ..., Xn]];

Interpretation: This call-by-name function takes an arbitrary number of arguments and returns a list of the evaluated results.

3. CBV REVERSE [X, Y] :=
   ATOM [X] —> Y;,
   REVERSE [CDR [X], CONS [CAR [X], Y]];

Interpretation: When given a list X, this function returns the elements of X in reverse order. Normally REVERSE is called with only the one argument. However, if a second argument Y is also given the reversed list is appended to the list Y.

4. CBV OBLIST [] :=
   a list of the currently active names in the system;

Interpretation: The object list, or more properly for muLISP, name list is a list of all the names currently in the system. The names are listed according to the order in which they were read in or generated: the most recent names are first and the primitive names are last.

C. Modifier Functions

Modifier functions actually redirect pointers in LISP data structures. Thus modifier functions are used primarily for their effect rather than their returned value. They can be used very effectively to modify already existing structures, thereby eliminating the need for the costly consing together of a whole new structure. Since they can easily produce unwanted side-effects such as circular lists, these functions should only be used by the experienced LISP programmer. For a good example of their use, see the source listing for the muSTAR editor.

1. CBV RPLACA [X, Y] :=
   car cell of X <— Y,
   X;

   Interpretations:
   a. Replace the first element of a list X by Y,
   b. Replace the left element of a dotted-pair X by Y,
   c. Replace the value of an atom X by Y.

2. CBV RPLACD [X, Y] :=
   cdr cell of X <— Y,
   X;

   Interpretations:
   a. Replace the tail of a list X by Y,
   b. Replace the right element of a dotted-pair X by Y,
   c. Replace the property list of an atom X by Y.

3. CBV NCONC [X, Y] :=
   ATOM [X] —> Y;,
   ATOM [CDR [X]] —> RPLACD [X, Y];,
   NCONC [CDR [X], Y],
   X;

   Interpretation: Concatenate, without consing, the list Y onto the end of the list X. The resulting list is the same as would have been produced by APPEND. However, NCONC actually modifies the first list by redirecting the final cdr cell of that list to Point to the second list. Thus, if X and Y point to the same list, a circular list will result. If an attempt is then made to print this list, the printout will continue indefinitely.

D. Recognizer Functions

Recognizer functions are used to identify data structures. They all take exactly one argument and return a value of either T or NIL.

1. CBV NAME [X] :=
   if X is a name —> TRUE;,
   NIL;

   Interpretation: This function recognizes names.

2. CBV NUMBERP [X] :=
   if X is an integer —> TRUE;,
   NIL;

   Interpretation: This function recognizes numbers.

3. CBV ATOM [X] :=
   NAME [X] OR NUMBERP [X];

   Interpretation: This function recognizes atoms, that is, non-nodes.

4. CBV NULL [X] :=
   EQ [X, NIL];

   Interpretation: This function recognizes the null list.

5. CBV PLUSP [X] :=
   GREATERP [X, 0];

   Interpretation: This function recognizes positive numbers.

6. CBV MINUSP [X] :=
   LESSP [X, 0];

   Interpretation: This function recognize negative numbers.

7. CBV ZEROP [X] :=
   EQ [X, 0];

   Interpretation: This function recognizes the zero.

8. CBV EVEN [X] :=
   ZEROP [REMAINDER [X, 2]];

   Interpretation: This function recognizes even numbers.

E. Comparator Functions

Comparator functions are used to compare data structures. They all require two arguments and return a value of either T or NIL.

1. CBV EQ [X, Y] :=
   NUMBERP [X] AND NUMBERP [Y] —> X=Y;,
   if X and Y point to the same object —> TRUE;,
   NIL;

   Interpretation: Normally the EQ test is used for the equality comparison of atoms, i.e. names and numbers. For objects other than numbers, EQ tests to see if X and Y point to the same location in memory. As described in Section II, names are uniquely stored in muLISP. Thus the EQ comparison is an efficient test to determine the equality of names. However, since numbers are not stored uniquely, EQ actually compares the number vectors of numerical arguments.

2. CBV EQUAL [X, Y] :=
   ATOM [X] —> EQ [X, Y];,
   ATOM [Y] —> NIL;,
   EQUAL [CAR [X], CAR [Y]] —>
      EQUAL [CDR [X], CDR [Y]];,
   NIL;

   Interpretation: The function EQUAL is used for the equality comparison of two objects, as distinct from the identity comparison provided by the EQ test. The structures X and Y are considered equal if they have isomorphic tree structures with identical atomic terminal nodes. Or put more crudely, X and Y are equal if their print-outs are identical.

3. CBV MEMBER [X, Y] :=
   ATOM [Y] —> NIL;,
   EQUAL [X, CAR [Y]] —> TRUE;,
   MEMBER [X, CDR [Y]];

   Interpretation: MEMBER [X, Y] will return T if the expression X is EQUAL to any member of the list Y, NIL otherwise.

4. CBV GREATERP [X, Y] :=
   NUMBERP [X] AND NUMBERP [Y] —>
      X > Y;,
   NIL;

   Interpretation: A simple greater than comparision for the numbers X and Y. Note that NIL is returned if either of the arguments is not an integer.

5. CBV LESSP [X, Y] :=
   NUMBERP [X] AND NUMBERP [Y] —>
      X < Y;,
   NIL;

   Interpretation: A simple less than comparison for the numbers X and Y. Note that NIL is returned if either of the arguments is not an integer.

6. CBV ORDERP [X, Y] :=
   NUMBERP [X] AND NUMBERP [Y] —> LESSP [X, Y];,
   if the address of the object X is less than the
      address of the object Y —> TRUE;,
   NIL;

   Interpretation: This function provides a generic ordering for system names based on their order of introduction. Thus if the name X were introduced before the name Y (i.e. X occurs to the right of Y on the object list) then ORDERP [X, Y] will return TRUE; otherwise it returns NIL. When used with non-atomic arguments, the result of this function is essentially meaningless.

F. Logical Functions

The logical functions permit Boolean combinations of truth values like those returned by the recognizer and comparator functions. Here as elsewhere in muLISP, any non-NIL value is considered to be logically true.

1. CBV NOT [X] :=
   EQ [X, NIL];

   Interpretation: The logical NOT function returns T if and only if its argument is NIL. Thus it is entirely equivalent to the function NULL.

2. CBN AND [X1, X2, ..., Xn] :=
   if n=0 —> TRUE;,
   NOT [EVAL [X1]] —> NIL;,
   AND [X2, X3, ..., Xn];

   Interpretation: The logical AND function returns T if and only if each of its arguments evaluate to a non-NIL value. Note that AND is a CBN function and that its arguments are sequentially evaluated until one evaluates to NIL or until all have evaluated to a non-NIL value. Hence not necessarily all the arguments will be evaluated.

3. CBN OR [X1, X2, ..., Xn] :=
   if n=0 —> NIL;,
   EVAL [X1] —> TRUE;,
   OR [X2, X3, ..., Xn];

   Interpretation: The logical OR function returns T if any one of its arguments evaluates to a non-NIL value. The arguments are successively evaluated and if any evaluates to a non-NIL value, T is returned and none of the remaining arguments are evaluated.

G. Assignment Functions

Assignment functions are normally used to assign values to program variables. For instance, they permit the values of a function's formal arguments to be modified without the need for a recursive function call. Thus in some situations they can significantly improve a program's execution speed. If the variable being assigned is not a local variable (i.e. not a formal argument of the currently executing function), the assignment will remain in effect even after the function is exited. As with the modifier functions described above, this phenomenon is called a side-effect of the function.

Use of the function SETQ should be familar to programmers of more conventional languages such as BASIC and PASCAL where the assignment statement is used pervasively. Though powerful, indiscriminate use of the assignment functions will result in unstructured programs and detrimental side-effects which are exceedingly difficult to debug. To prevent developing bad habits, the novice LISP programmer should avoid use of these functions as they obscure the elegant, applicative nature of the language.

1. CBV SET [X, Y] :=
   RPLACA [X, Y],
   Y;

   Interpretation: Set the value of the name X to Y, and return Y. Note that the function is also defined when X is not a name; however, its use in that situation is strongly discouraged.

2. CBN SETQ [X, Y] :=
   SET [X, EVAL [Y]];

   Interpretation: Set the value of X itself to the value of Y, and return that value. This function is used more often than SET because usually it is desired to assign a value to a variable rather than to the value of a variable.

The distinction between the effect of SET and SETQ on their arguments is demonstrated by the following. If the value of DOG is BARK, then

```
SETQ [DOG, '(A B C)]
```

will change the value of DOG from BARK to (A B C). In contrast, if the value of DOG is BARK, then

```
SET [DOG, '(A B C)]
```

will change the value of BARK to (A B C), and the value of DOG will remain BARK.

3. CBN POP [X] :=
    POP1 [X, EVAL [X]];

   CBV POP1 [X, Y] :=
    SET [X, CDR [Y]];,
    CAR [Y];

   Interpretation: If X is the name of a list, then POP [X] returns the car of that list while setting X to the cdr of the list. This operation is the LISP analog of the familar pop stack operation widely used in machine languages.

4. CBN PUSH [X, Y] :=
    SET [Y, CONS [EVAL [X], EVAL [Y]]];

   Interpretation: If Y is the name of a list and X is an expression, then PUSH [X, Y] will cons X onto the list Y and update Y to point to this enlarged list. This operation is the LISP equivalent of pushing information onto a stack.

H. Property Functions

Property functions provide an excellent means of associating global properties with names. A name's property list is used to store these properties along with indicator tags. The property value associated with an indicator can be retrieved at any later time using the GET function. Used in conjunction with the flag functions described in the next section, extremely flexible and efficient data bases can be established in a very natural and convenient manner.

1. CBV ASSOC [X, Y] :=
    ATOM [Y] —> Y;,
    ATOM [CAR [Y]] —> ASSOC [X, CDR [Y]];,
    EQUAL [CAAR [Y], X] —> CAR [Y];,
    ASSOC [X, CDR [Y]];

   Interpretation: This function performs a linear search of the association list Y, looking for a non-atomic element whose car is EQUAL to X. If found, the entire element is returned; otherwise NIL is returned.

2. CBV GET [X, Y] :=
    X <— ASSOC [Y, CDR [X]],
    ATOM [X] —> NIL;,
    CDR [X];

   Interpretation: GET [X, Y] returns the property value associated with the name X under the indicator Y. If the indicator is not found, then NIL is returned.

3. CBV PUT [X, Y, Z] :=
    NULL [GET [X, Y]] —>
      RPLACD [X, CONS [CONS [Y, Z], CDR [X]]],
      Z;,
    RPLACD [ASSOC [Y, CDR [X]], Z],
    Z;

   Interpretation: PUT [X, Y, Z] places on the property list of the name X under the indicator Y the property value Z, destroying any previous value.

4. CBV REMPROP [X, Y] :=
    ATOM [CDR [X]] —> CDR [X];,
    EQUAL [CAADR [X], Y] —>
      Y <— CDADR [X],
      RPLACD [X, CDDR [X]], Y;,
    REMPROP [CDR [X], Y];

   Interpretation: REMPROP [X, Y] removes from the property list of the name X the property value associated with the indicator Y. It returns the property value.

I. Flag Functions

Like the property functions the flag functions also use a name's property list to store information. However, the name is only flagged as either having a particular attribute or not.

1. CBV FLAGP [X, Y] :=
   MEMBER [Y, CDR [X]];

   Interpretation: This predicate will return T if and only if the attribute Y is an element of the property list of X; and NIL otherwise. Note that if Y is an indicator put on the property list by using the PUT function, it will only be recognized by the function GET and not by FLAGP.

2. CBV FLAG [X, Y] :=
   FLAGP [X, Y] —> Y;
   RPLACD [X, CONS [Y, CDR [X]]],
   Y;

   Interpretation: FLAG [X, Y] will flag the name X with the attribute Y by making Y the first element of the property list of X.

3. CBV REMFLAG [X, Y] :=
   ATOM [CDR [X]] —> NIL;,
   EQUAL [Y, CADR [X]] —>
      RPLACD [X, CDDR [X]],
      Y;,
   REMFLAG [CDR [X], Y];

   Interpretation: This function removes the attribute Y from the property list of the name X, returning NIL if the attribute is not found.

J. Definition Functions

The function definition functions are the only means of access to a name's function definition cell. When a function is defined using PUTD, the definition is pseudo-compiled into an extremely dense form called D-Code or Distilled Code. This compilation results in about a 3 fold increase in code density and approximately a 20% improvement in execution speed over muLISP-79. The compilation of S-expressions containing cdr cells which point to non-NIL atoms results in the replacement of the atom by NIL. In general it is recommended that QUOTEd non-atomic constants NOT be included in function definitions directly. Instead, the constant can be assigned to a name which can be used in the definition in place of the constant. The inverse process of de-compiling D-code back into a linked list also occurs automatically when GETD is used to retrieve a definition. Thus the use of D-code is invisible to the user and the interactive nature of LISP is not compromised in the interest of efficiency and compactness.

1. CBV GETD [X] :=
   NOT [NAME [X]] OR UNDEFINED [X] —> NIL;,
   SUBR [X] OR FSUBR [X] —>
      memory address of machine language subroutine;,
   the S-expression equivalent of the D-code defining the function X;

   Interpretation: This function is used to get a function's definition for further processing. If the function is a machine language subroutine, the physical memory address of the function is returned. Otherwise the linked list equivalent of the D-code is returned. See section V-O for the definitions of SUBR and FSUBR.

2. CBV PUTD [X, Y] :=
   NOT [NAME [X]] —> NIL;,
   NUMBERP [Y] —>
      function cell of X <— address given by Y;,
   function cell of X <— D-code equivalent of Y,
   Y;

   Interpretation: If Y is a number, PUTD [X, Y] sets the function cell of X to the memory address equal to the number Y (modulo 64K). Otherwise the definition cell is set to the D-code equivalent of Y. The procedure for using PUTD to link to machine language subroutines is described in Appendix C of this manual.

3. CBV MOVD [X, Y] :=
   NOT [NAME [X]] OR NOT [NAME [Y]] —> NIL;,
   function cell of Y <— function cell of X,
   GETD [Y];

   Interpretation: Set the function cell of Y to point to the same memory location as that of the function cell of X. In cases where a MOVD is sufficient it should be used instead of a GETD and PUTD.

K. Sub-atomic Functions

The sub-atomic functions are so named because they provide access to a name's print name string or a number's number vector. This makes it possible to temporarily unpack an atom's print name, operate on the resulting list of characters, and ultimately repack the list to form a new name. In addition to its sub-atomic capability the LENGTH function determines the top-level length of an S-expression.

1. CBV PACK [X] :=
   ATOM [X] —> "";,
   NAME [CAR [X]] —>
      concatenate the print name of CAR [X] with PACK [CDR [X]] and return the resulting name;,
   NUMBERP [CAR [X]] —>
      concatenate the print name string of the number CAR [X] with PACK [CDR [X]] and return the resulting name;,
   PACK [CDR [X]];

   Interpretation: This function, called COMPRESS in some dialects of LISP, returns a name whose print name is a packed version of the print names of the atoms in the list X. The current RADIX base is used to determine the print name string of numbers. Note that PACK always returns a name, even if it only contains digits. As shown in the muLISP library file, PACK can be used to write a GENSYM (generate symbol) function.

2. CBV UNPACK [X] :=
   NAME [X] —>
      a list of names whose print names correspond to the characters in the print name of X;
   NUMBERP [X] —>
      a list of names whose print names are numerals corresponding to the digits of X expressed in the current radix base;,
   NIL;

   Interpretation: This function, called EXPLODE in some dialects of LISP, returns a list of names whose one-character print names correspond to the successive characters in the print name of X. The current radix is used for numbers and digits are converted to names with single numeral print names.

3. CBV LENGTH [X] :=
   NAME [X] —>
      the number of characters in the print name of X;,
   NUMBERP [X] —>
      the number of digits in the print name equivalent of X;,
   ATOM [CDR [X]] —> 1;,
   1 + LENGTH [CDR [X]];

   Interpretations: This function is effectively three functions in one. The value returned follows intuitively from the data type of its argument:

   a. If X is a name, the number of characters actually required to print X is returned. The current value of PRIN1 is taken into account while computing this length. The effect of PRIN1 is described under the control variable subsection under the Printer Functions.

   b. If X is a number, the number of characters actually required to print X is returned. The current radix base and if applicable the leading "-" sign and/or leading "0" are properly taken into account while computing this length.

   c. If X is a non-atomic, the number of top level nodes in the list is returned. As always in muLISP, a cdr which is atomic denotes the terminator of a list.

L. Numerical Functions

The numerical functions implement exact precision integer arithmetic for numbers of magnitude up to 2^2032-1, or about 611 decimal digits of accuracy. If non-numeric arguments are passed to any of the functions or if an overflow occurs, the function returns a value of NIL. Division by zero by any one of the functions QUOTIENT, REMAINDER, or DIVIDE causes the following warning message to be displayed on the console:

ZERO Divide Error

and the function returns a value of NIL.

```
1. CBV MINUS [X] :=
       NUMBERP [X] --> -X;
       NIL;

2. CBV PLUS [X, Y] :=
       NUMBERP [X] AND NUMBERP [Y] -->
           X + Y;
       NIL;

3. CBV DIFFERENCE [X, Y] :=
       NUMBERP [X] AND NUMBERP [Y] -->
           X - Y;
       NIL;

4. CBV TIMES [X, Y] :=
       NUMBERP [X] AND NUMBERP [Y] -->
           X * Y;
       NIL;

5. CBV QUOTIENT [X, Y] :=
       NUMBERP [X] AND NUMBERP [Y] -->
           X/Y (Truncated toward zero);
       NIL;
```

Interpretation: The truncated integer quotient.

```
6. CBV REMAINDER [X, Y] :=
       DIFFERENCE [X, TIMES [Y, QUOTIENT [X, Y]]];

7. CBV DIVIDE [X, Y] :=
       NUMBERP [X] AND NUMBERP [Y] -->
           CONS [QUOTIENT [X, Y], REMAINDER [X, Y]];
       NIL;
```


M. Reader Functions and Control Variables

The reader functions provide for character input to muLISP programs. The functions read characters from the current input source. This input source can be either the console or any text file on the disk or other secondary storage device. The current input source can be controlled through the use of the function RDS in conjunction with the control variable RDS.

```
1. CBV RDS [X, Y, Z] :=                    (ReaD Select)
       NULL [X] -->
           RDS <-- NIL;,
       NAME [X] AND NAME [Y] -->
           NULL [Z] -->
               a file named X.Y exists on the currently
                   logged in disk drive -->
                       open X.Y for input,
                       RDS <-- X;,
               RDS <-- NIL;,
           NAME [Z] -->
               a file named X.Y exists on drive Z -->
                   open X.Y on Z for input,
                   RDS <-- X;,
               RDS <-- NIL;,
           RDS <-- NIL;,
       RDS <-- NIL;
```

Interpretation: The read select function is used to select an input source file. If the selected file is found, the file is opened for input, and the value of the variable RDS is set to the name of the file. Thus this file becomes the new current input source. If RDS is called with no arguments, invalid arguments, or if the file is not found, the variable RDS is set to NIL making the console the current input source. The default DRIVER function sets RDS to NIL. This makes the console the current input source on initial system startup and after interrupts or error traps.

```
2. CBV RATOM [] :=                         (Read ATOM)
```

read one token from the current input source and return the corresponding name or number. A token is a string of characters delimited by either separator or break characters. See the notes at the end of this section for details on including comments and quoted strings within a token. Unlike separators, break characters are returned by RATOM as single character LISP names. The RATOM separator characters are: space, carriage return, line-feed, and tab (CTRL-I). The RATOM break characters are: ! $ & ' ( ) * + , - . / @ : ; < = > ? [\ ] ^ _ ` { | } ~

3. CBV READ [] := (READ Expression)

```
    READ0 [RATM []];

CBV READ0 [X] :=
    EQ [X, '( ] —>
        READLIST [RATM []];,
    EQ [X, '[ ] —>
        READBRACKET [READLIST [RATM []], RATM []];,
    EQ [X, ') ] OR EQ [X, '. ] OR EQ [X, '] ] —>
        READ [];,
    X;

CBV READBRACKET [X, Y] :=
    EQ [Y, ') ] OR EQ [Y, '] ] —> X;,
    CONS [X, READBRACKET [READLIST [Y], RATM []]];

CBV READLIST [X] :=
    TERMINATOR [X] —> NIL;,
    EQ [X, '. ] —>
        READDOT [READ [], RATM []];,
    CONS [READ0 [X], READLIST [RATM []]];

CBV READDOT [X, Y] :=
    TERMINATOR [Y] —> X;,
    CONS [X, READLIST [Y]];,

CBV TERMINATOR [X] :=
    EQ [X, ') ] —> TRUE;,
    EQ [X, '] ] —>
        unread "]" character back into input source for
        the next RATM read,
        TRUE;,
    NIL;

CBV RATM [] :=
    read one token from the current input source and
    return the corresponding name or number. This
    function is identical to RATOM except the RATM
    separator characters are: space, comma, carriage
    return, line-feed, and tab; and the RATM break
    characters are: ( ) . [ ]  See the notes below
    for details concerning comments and quoted strings.
```

Interpretation: The READ function reads one complete symbolic expression from the current input source while constructing the equivalent LISP data structure. Well formed expressions using either the list notation, dot notation, or a combination of both are acceptable means of input. Brackets can be used as super-parentheses. Thus the right bracket, "]", closes all left parentheses back to either the beginning of the expression or to a matching left bracket. Extra right parentheses, right brackets, and dots are ignored.

4. CBV READCH [] := (READ CHaracter)

    read one character from the current input source and return the corresponding LISP atom. A number is returned if the character is a decimal digit less than the current radix base.

Notes:

1. If a disk file is the current input source and an attempt is made to read past the end-of-file (EOF), the following warning message is displayed on the console:

    End-Of-File Read

In addition, the control variable RDS is set to NIL, making the console the current input source.

2. Comments can occur anywhere in the text of the input source so long as they are delimited by matching percent signs, "%". The text of comments is totally ignored by the functions RATOM and READ. However, the function READCH reads and returns percent signs just as any other character.

3. Special characters such as the percent sign, double quote sign, separator characters, and break characters can be read in as names or parts of names by the use of quoted strings. Such strings are delimited by double quote marks. The double quote can be included within the string by using two adjacent double quotes for each desired double quote.

4. As an added programming convenience, when a name is read into the system by READ, RATOM, or READCH, the value of the variable RATOM is set to the new name.

The reader control variables extend the flexibility of the reader functions by making several input options available to the user. They operate as flags or toggles which are either off or on. Except for ECHO, the reader control variables have the same names as the reader functions. This is done primarily to conserve precious memory space; however, each variable is usually used in conjunction with the function of the same name.

1. RDS: Normally control of the current input source is done through the use of the function RDS as described above. However, after a file has been opened and made current, control can be returned to the console without closing the input file, simply by setting the value of RDS to NIL. A subsequent non-NIL assignment to RDS will then return control to the previously opened disk file at the point at which reading was suspended.

2. READ: The variable READ controls the lower to upper case conversion of letters upon input. Normally READ is non-NIL and lower case letters are distinct from their upper case counterparts. However if it is NIL, all lower case letters are converted to upper case as they are read in. In any case, all lower case letters already in the system remain in lower case.

3. READCH: When the console is the current input source, the console input mode is controlled by the variable READCH. Normally the value of the variable READCH is non-NIL and console input is then in the line edit mode. In this mode, when all the characters have been read from the current line, the operating system's line edit routine is called for further input. Until a carriage return is typed, the system's normal editing procedures such as input echoing, backspacing, line deletion, printer output toggle using control-P, etc. are in force. If the variable READCH is NIL, all buffering and input echoing is eliminated. This raw input mode is useful for immediate response to one character commands as demonstrated in the LISP STAR editor.

4. ECHO: If a disk file is the current input source and the value of the control variable ECHO is non-NIL, the characters being read from the file are echoed to the current output sink, which is usually the console. Note that since comments are also echoed, English language text within a comment can conveniently be displayed without having to actually process the text. See the printer control variable section for other effects of ECHO.

The reader control variables extend the flexibility of the reader functions by making several input options available to the user. They operate as flags or toggles which are either off or on. Except for ECHO, the reader control variables have the same names as the reader functions. This is done primarily to conserve precious memory space; however, each variable is usually used in conjunction with the function of the same name.

1. RDS: Normally control of the current input source is done through the use of the function RDS as described above. However, after a file has been opened and made current, control can be returned to the console without closing the input file, simply by setting the value of RDS to NIL. A subsequent non-NIL assignment to RDS will then return control to the previously opened disk file at the point at which reading was suspended.

2. READ: The variable READ controls the lower to upper case conversion of letters upon input. Normally READ is non-NIL and lower case letters are distinct from their upper case counterparts. However if it is NIL, all lower case letters are converted to upper case as they are read in. In any case, all lower case letters already in the system remain in lower case.

3. READCH: When the console is the current input source, the console input mode is controlled by the variable READCH. Normally the value of the variable READCH is non-NIL and console input is then in the line edit mode. In this mode, when all the characters have been read from the current line, the operating system's line edit routine is called for further input. Until a carriage return is typed, the system's normal editing procedures such as input echoing, backspacing, line deletion, printer output toggle using control-P, etc. are in force. If the variable READCH is NIL, all buffering and input echoing is eliminated. This raw input mode is useful for immediate response to one character commands as demonstrated in the LISP STAR editor.

4. ECHO: If a disk file is the current input source and the value of the control variable ECHO is non-NIL, the characters being read from the file are echoed to the current output sink, which is usually the console. Note that since comments are also echoed, English language text within a comment can conveniently be displayed without having to actually process the text. See the printer control variable section for other effects of ECHO.

N. Printer Functions and Control Variables

The muLISP printer functions direct character output to the current output sink. As determined by the function and variable WRS, the sink can be either the console or a disk file.

1. CBV WRS [X, Y, Z] :=
   NOT NULL [WRS] —> (WRite Select)
   write out the final record of the currently open disk file and close the file,
   WRS <— NIL,
   WRS [X, Y, Z];,
   NULL [X] —>
   WRS <— NIL;,
   NAME [X] AND NAME [Y] —>
   NULL [Z] —>
   on the currently logged in disk drive, if a file named X.Y exists delete any existing file named X.BAK, then rename the file X.Y to X.BAK, and make a new directory entry for X.Y,
   WRS <— X;,
   NAME [Z] —>
   on drive Z, if a file named X.Y exists delete any existing file named X.BAK, then rename the file X.Y to X.BAK, and make a new directory entry for X.Y,
   WRS <— X;,
   WRS <— NIL;,
   WRS <— NIL;

   Interpretation: The write select function is used both to select and later to close output sink disk files. If the current output sink is a disk file and WRS is called, that file is closed. Next, if a file name, type, and a drive (optional) are supplied as arguments to WRS, an already existing file of that name is renamed to a .BAK file of the same name. This provides an automatic one level file backup feature. Finally a new file of the given name and type is created for output on the appropriate drive, and the variable WRS is set to the name of the file. Thus this file becomes the new current output sink. The default DRIVER function sets WRS to NIL making the console the current output sink on initial system startup and after interrupts or error traps.

2. CBV PRINT [X] :=
   PRIN1 [X],
   TERPRI [],
   X;

   Interpretation: Print the expression X, terminate the last line and return X.

3. CBV PRIN1 [X] :=
   NAME [X] —>
   output to the current output sink the print name representation of X;,
   NUMBERP [X] —>
   output to the current output sink the digits expressing X in the current radix base and preceeded by a "-" if MINUSP [X];,
   PRIN1 ["("],
   PRINLIST [X],
   X;

   CBV PRINLIST [X] :=
   PRIN1 [CAR [X]],
   NULL [CDR [X]] —> PRIN1 [")"];,
   PRIN1 [" "],
   ATOM [CDR [X]] —>
   PRIN1 [". "],
   PRIN1 [CDR [X]],
   PRIN1 [")"];,
   PRINLIST [CDR [X]];

   Interpretation: The function PRIN1 will output the standard list notation of the object X to the current output sink. X is the returned value.

4. CBV TERPRI [X] :=
   ZEROP [X] —> NIL;,
   output a carriage return and line feed to the current output sink,
   PLUSP [X] AND LESSP [X, 256] —>
   TERPRI [X-1];,
   NIL;

   Interpretation: If X is a non-negative number, TERPRI [X] outputs X number of new lines to the current output sink. Otherwise one new line is output to the sink.

5. CBV SPACES [X] :=
   PLUSP [X] AND LESSP [X, 256] —>
   PRIN1 [" "],
   SPACES [X-1];,
   the current cursor position;

   Interpretation: If X is a non-negative number, SPACES [X] outputs X number of spaces to the current output sink. Otherwise no spaces are output to the sink. In any case, the resulting cursor position is returned.

6. CBV LINELENGTH [X] :=
    GREATERP [X, 11] AND LESSP [X, 256] —>
        set maximum output linelength to X,
        return the previous linelength;,
    return the current linelength;

Interpretation: If X is a number between 11 and 256, LINELENGTH [X] sets the maximum number of characters output per line to X. The function returns the previous linelength. If X is not a number or outside the permissible range, the current linelength is returned. The default linelength is 72.

7. CBV RADIX [X] :=
    GREATERP [X, 1] AND LESSP [X, 37] —>
        set radix base to X,
        return the old radix base;,
    return the current radix base;

Interpretation: If X is a number between 1 and 37, RADIX [X] sets the radix base in which numbers are expressed for both input and output, and the function returns the previous base. If X is not a number or outside the permissible range, the current radix base is returned. The default radix base is decimal ten.

Notes:

1. If a disk file is the current output sink and there is insufficient disk space, the following warning message is displayed on the console:

No Disk Space

In addition, the control variable WRS is set to NIL, making the console the current output sink.

2. If the host computer's operating system supports a printer and muLISP is in the line edit mode (see note 3 under reader control variables, section V-M), typing a CTRL-P will direct all muLISP output displayed on the system console to the printer as well.

The printer control variables are used analogously to their reader control counterparts. They function as toggles to control the current output sink, case converion, printing of quoted strings, and output echoing of characters to the console.

1. WRS: Normally control of the current output sink is done through the use of the function WRS as described above. However, after a file has been opened using WRS, output can be directed to the console without closing the disk file by simply setting the value of WRS to NIL. A subsequent non-NIL assignment to WRS will then redirect output to the disk file and append data onto the end of the file.

2. PRINT: The variable PRINT controls the upper to lower case conversion of letters being output. Normally PRINT is non-NIL and all letters are printed as stored. However, if it is NIL, all upper case letters are converted to lower case as they are printed. This conversion in no way affects the internal storage of any name's character string.

3. PRIN1: If PRIN1 is NIL, names which contain separator or break characters will be printed using double quotes as necessary to permit the name to be subsequently read back in as the same name. Printing such names using quoted strings is essential for such applications as LISP editors. Normally, however, PRIN1 is non-NIL and a name's print name string will simply be output as is.

4. ECHO: If a disk file is the current output sink and the value of the control variable ECHO is non-NIL, the characters being output to the file are also echoed to the console.

0. Evaluation Functions

The evaluation functions are used for expression evaluation and program control. The algorithm for evaluating function bodies in muLISP has been enhanced to make program control implicit in the structure of the body itself. As discussed in section I-E, this makes function definitions cleaner, shorter, and easier to interpret.

A muLISP function definition is specified by a linked list representing the desired definition. The first element of the list determines the function's type. It should be either the name LAMBDA or NLAMBDA. LAMBDA indicates the function is a call by value (CBV) function. When a CBV function is called, arguments are first evaluated and only the resulting values are passed to the function. A function defined using NLAMBDA (i.e. a No-eval LAMBDA) is a call by name (CBN) function. A CBN function receives its arguments in unevaluated form, just as they were given in the call to the function. CBV functions are by far the most prevalent in LISP. Thus the novice programmer need not be concerned with learning to use the CBN functions until the other features of the language have been mastered.

The second element of a function's definition should be either a name or a list of names defining the function's formal arguments. If the formal argument is a name which is not NIL, the function is considered to be a no-spread function. A no-spread function receives its arguments as one list bound to the name. Thus no-spread functions can have an arbitrary number of arguments. However, if the second element is a list of atoms, the arguments will be passed to this spread function bound to each formal argument making up the list. Note that a function's being spread or no-spread is entirely independent of its being CBV or CBN.

The remaining elements of the list make up the definition's function body. The function body is a list of tasks which are to be successively performed when the function is called. The returned value of the function is the value of the last task performed. How a given task is to be performed depends on the structure of the task, as follows:

1) If the task is an atom, the value of the task is the value of the atom.

2) If the car of the task is an atom, the car is considered to be the name of a function which is to be applied to the list of arguments making up the cdr of the task. The arguments are evaluated before the application for CBV functions.

3) If the car of the car of the task is an atom, the car of the task is considered to be a predicate, which is then evaluated as described in 2) above. If the value of the predicate is NIL, the value of the task is NIL. However, if the predicate's value is non-NIL, the original function body is abandoned and evaluation proceeds using the cdr of the task as the new function body.

4) Otherwise the task is recursively evaluated as a function body itself before continuing with the evaluation of the top level function body. This permits conditional forks in function bodies to later recombine.

This evaluation scheme is very powerful but it does not have any provisions for non-recursive program control structures. Such iterative capability can be added to the algorithm above quite simply. A function body enclosed within the LOOP control construct will be evaluated as described above, except evaluation will start again at the beginning of the body instead of returning after the last task has been performed. This continues until a predicate as defined in case 3) above is non-NIL. The value of the loop construct is the value of the function body following the non-NIL predicate. Note that any number of predicates can occur within a loop at any desired location. This implementation of a LOOP preserves the basic evaluation method of muLISP while greatly improving the performance of the language.

The following auxiliary functions are recognizers used to define the evaluation and function definition functions.

```
CBV UNDEFINED [X] :=                    (Undefined function recognizer)
    NULL [GETD [X]];

CBV CBVP [X] :=                         (Call-by-value recognizer)
    SUBR [X] OR EXPR [X];

CBV CBNP [X] :=                         (Call-by-name recognizer)
    FSUBR [X] OR FEXPR [X];

CBV SUBR [X] :=                         (CBV subroutine recognizer)
    X is a CBV machine language routine --> TRUE;,
    NIL;

CBV FSUBR [X] :=                        (CBN subroutine recognizer)
    X is a CBN machine language routine --> TRUE;,
    NIL;

CBV EXPR [X] :=                         (CBV LAMBDA defined function
    EQ [CAR [X], 'LAMBDA];                  recognizer)

CBV FEXPR [X] :=                        (CBN LAMBDA defined function
    EQ [CAR [X], 'NLAMBDA];                 recognizer)
```

1. CBN QUOTE [X] :=
    X;

Interpretation: This function suppresses evaluation of its argument and returns the object X itself.

2. CBV EVAL [X] :=
    ATOM [X] —> CAR [X];,
    NAME [CAR [X]] —>
        UNDEFINED [CAR [X]] —>
            EQ [CAR [X], EVAL [CAR [X]]] —>
                EVLIS [X];,
            EVAL [CONS [EVAL [CAR [X]], CDR [X]]];,
        CBVP [GETD [CAR [X]]] —>
            APPLY [CAR [X], EVLIS [CDR [X]]];,
        CBNP [GETD [CAR [X]]] —>
            APPLY [CAR [X], CDR [X]];,
        EVLIS [X];,
    EXPR [CAR [X]] —>
        APPLY [CAR [X], EVLIS [CDR [X]]];,
    FEXPR [CAR [X]] —>
        APPLY [CAR [X], CDR [X]];,
    EVLIS [X];

Interpretation: If X is an atom, EVAL [X] returns the contents of the value cell of X. Otherwise, if the car of X is a CBV function, each element of the cdr of X is evaluated and the function is applied to the results. If the car of X is a CBN function, the function is applied to the cdr of X directly. Finally, if the car of X is not a function, each element of X is evaluated in turn, and a list of the results is returned.

CBV EVLIS [X] :=
    ATOM [X] —> NIL;,
    CONS [EVAL [CAR [X]], EVLIS [CDR [X]]];

Interpretation: This function evaluates each element of a list and returns a list of the results.

3. CBV APPLY [X, Y] :=
    NAME [X] —>
        UNDEFINED [X] —>
            EQ [X, EVAL [X]] —> NIL;,
            APPLY [EVAL [X], Y];,
        SUBR [GETD [X]] —>
            ATOM [Y] —>
                X [NIL, NIL, NIL];,
            ATOM [CDR [Y]] —>
                X [CAR [Y], NIL, NIL];,
            ATOM [CDDR [Y]] —>
                X [CAR [X], CADR [X], NIL];,
            X [CAR [Y], CADR [Y], CADDR [Y]];,
        FSUBR [GETD [X]] —>
            X [Y];,
        EXPR [GETD [X]] OR FEXPR [GETD [X]] —>
            BIND [CADR [GETD [X]], Y],
            Y <— EVALBODY [NIL, CDDR [GETD [X]]],
            UNBIND [CADR [GETD [X]]],
            Y;,
        NIL;,
    EXPR [X] OR FEXPR [X] —>
        BIND [CADR [X], Y],
        Y' <— EVALBODY [NIL, CDDR [X]],
        UNBIND [CADR [X]],
        Y;,
    NIL;

Interpretation: APPLY [X, Y] applies the function X to the list of arguments Y. If X is a machine language routine, control passes to the routine. If X is a LAMBDA defined function, the formal arguments of X are temporarily bound to the actual arguments, the function body is evaluated, the original values of the formal arguments are restored, and the value of the function body evaluation is returned.

CBV EVALBODY [X, Y] :=
    ATOM [Y] —> X;,
    ATOM [CAR [Y]] OR ATOM [CAAR [Y]] —>
        EVALBODY [EVAL [CAR [Y]], CDR [Y]];,
    ATOM [CAAAR [Y]] —>
        X <— EVAL [CAAR [Y]],
        NOT [X] —>
            EVALBODY [NIL, CDR [Y]];,
        EVALBODY [X, CDAR [Y]];,
    EVALBODY [EVALBODY [X, CAR [Y]], CDR [Y]];

Interpretation: This function evaluates a function body Y as described in the introduction to this section and returns the value of the last expression which was evaluated.

CBV BIND [X, Y] :=
    ATOM [Y] —>
        ATOM [X] —> NIL;,
        PUSH [EVAL [CAR [X]], ARGSTACK],
        SET [CAR [X], NIL],
        BIND [CDR [X], Y];,
    ATOM [X] —> NIL;,
    PUSH [EVAL [CAR [X]], ARGSTACK],
    SET [CAR [X], CAR [Y]],
    BIND [CDR [X], CDR [Y]];

Interpretation: This function saves the original values of the atoms making up the formal argument list X on the argument stack named ARGSTACK. Simultaneously the value of the formal arguments is set to the corresponding elements of the actual arguments given in the list Y.

```
CBV UNBIND [X] :=
    ATOM [X] —> NIL;,
    UNBIND [CDR [X]],
    SET [CAR [X], POP [ARGSTACK]];
```

Interpretation: This function restores the original values of the atoms on the formal argument list X stored on ARGSTACK.

4.
```
CBN COND [X1, X2, ..., Xn] :=
    EVALCOND [LIST [X1, X2, ..., Xn]];

CBV EVALCOND [X, Y] :=
    ATOM [X] —> NIL;,
    Y <— EVAL [CAAR [X]],
    NOT [Y] —> EVALCOND [CDR [X]];,
    EVALBODY [Y, CDAR [X]];
```

Interpretation: The COND function successively evaluates the car of X1, X2, ..., Xn until either a non-NIL value is encountered or all have evaluated to NIL. In the former case the cdr of that argument is evaluated as a function body (see the interpretation of APPLY for details). In the latter case NIL is returned by COND. This extended COND is a powerful upward compatible extension of the COND function described in original LISP 1.5.

5.
```
CBN LOOP [X1, X2, ..., Xn] :=
    EVALLOOP [LIST [X1, X2, ..., Xn],
        LIST [X1, X2, ..., Xn]];

CBV EVALLOOP [X, Y, Z] :=
    ATOM [Y] —>
        EVALLOOP [X, X];,
    ATOM [CAR [Y]] OR ATOM [CAAR [Y]] —>
        EVAL [CAR [Y]],
        EVALLOOP [X, CDR [Y]];,
    ATOM [CAAAR [Y]] —>
        Z <— EVAL [CAAR [Y]],
        NOT [Z] —> EVALLOOP [X, CDR [Y]];,
        EVALBODY [Z, CDAR [Y]];,
    EVALBODY [NIL, CAR [Y]],
    EVALLOOP [X, CDR [Y]];
```


Interpretation: The LOOP construct evaluates its argument in a manner identical to the evaluation of the clauses in a function body. However, if all the arguments are evaluated without a conditional having been satisfied, evaluation begins again with the first argument.

6.
```
CBN PROG1 [X1, X2, ..., Xn] :=
    EVALPROG1 [EVAL [X1], LIST [X2, X3, ..., Xn]];

CBV EVALPROG1 [X, Y] :=
    ATOM [Y] —> X;,
    EVAL [CAR [Y]],
    EVALPROG1 [X, CDR [Y]];
```

Interpretation: This function successively evaluates X1, X2, ..., Xn and return the result of the evaluation of X1.

7.
```
CBV DRIVER [] :=                          (Eval muLISP driver)
    RDS <— NIL,
    WRS <— NIL,
    ECHO <— NIL,
    READCH <— 'READCH,
    TERPRI [],
    PRIN1 ['"$ "],
    PRINT [EVAL [READ []]];
```

Interpretation: This is the default eval-LISP driver function as primitively defined in machine language. The function DRIVER is repeatedly evaluated by the muLISP executive driver loop. Therefore DRIVER can be redefined at will to suit the user's needs. For instance the following is a sample eval-quote-LISP driver:

```
CBV DRIVER [] :=                          (Eval-quote muLISP driver)
    RDS <— NIL,
    WRS <— NIL,
    ECHO <— NIL,
    READCH <— 'READCH,
    TERPRI [],
    PRIN1 ['"> "],
    PRINT [APPLY [READ [], READ []]];
```

P. Memory Management Functions

The memory management function is used to force a garbage collection and return the amount of currently available data space. Since memory management is fully automatic in muLISP, normally there is no need to explicitly use the function except for its value. See section III for a discussion of the memory management system.

1. CBV RECLAIM [] :=
   perform a garbage collection,
   the amount of free space expressed in bytes;

   Interpretation: This function forces a garbage collection to occur. The total amount of free space in the atom, vector, and pointer spaces is returned, expressed in bytes. Note that two bytes are required for each muLISP pointer cell. Thus nodes require 4 bytes, numbers require 6 bytes, and names require 8 bytes. In addition names and numbers require storage for their respective print name strings and number vectors.

Q. Environment Functions

The environment functions are used to save and load muLISP environments. Prior to saving a system, all the data spaces are automatically compacted into one area of memory so the resulting SYS file will be of minimum size. The SYS file can be loaded into a different size computer system than the one that produced the SYS file. See section I-H for a detailed discussion on the use of SYS files.

1. CBV SAVE [X, Y] :=
   NOT NULL [WRS] —>
   write out the final record of the current output sink and close the file,
   WRS <— NIL,
   SAVE [X, Y];,
   NAME [X] AND NAME [Y] —>
   NULL [Y] —>
   compact all current data structures,
   save a memory image of the current environment as a SYS file named X on the current drive,
   TRUE;,
   compact all current data structures,
   save a memory image of the current environment as a SYS file named X on drive Y,
   TRUE;
   NIL;

   Interpretation: This function saves the current muLISP environment on a disk file. Since the currently active data structures are compacted before the save, the size of the SYS file is proportional to the number of these structures in the system.

2. CBV LOAD [X, Y] :=
   NAME [X] AND NAME [Y] —>
   NULL [Y] —>
   load the memory image SYS file named X from the current disk,
   RDS <— NIL,
   return directly to the executive driver loop;,
   load the memory image SYS file named X from drive Y,
   RDS <— NIL,
   return directly to the executive driver loop;
   NIL;

   Interpretation: This function restores the muLISP environment present at the time of the SAVE. If the SYS file is successfully loaded, LOAD does not return a value but jumps directly to the new executive driver loop.

3. CBV SYSTEM [] :=
    compact all data structures into low memory,
    return to operating system;

Interpretation: When this function is executed, all data is compacted into low memory and then control is returned to the operating system. The compaction of data allows for a re-entry into muLISP with the same environment that was present at the time of the call to the function SYSTEM. The re-entry address for CP/M versions of muLISP is 100H (hexadecimal).

Section VI: The muSTAR AIDS

muLISP source files can be generated in one of two ways: either an external editor or a resident LISP editor can be used. The text editor provided with your computer's disk operating system is an example of an external editor. Program development using an external editor consists of iteratively editing, loading, and testing source files.

The principal advantages of using an external editor are that usually the user is familiar with it and that comments can easily be included as part of the text. Although an external editor is a perfectly satisfactory method of program development, it is somewhat slow and cumbersome, and not conducive to creative work. Repeated disk accesses are required to make even minor changes to function definitions.

These problems are conveniently resolved in muLISP-80 with the introduction of the muSTAR Artificial Intelligence Development System. muSTAR greatly reduces program development time and fully utilizes the interactive nature of LISP. Using the resident editor and tracing facilities provided by muSTAR, function definitions can be created, tested, and throughly debugged all within the system. This encourages the incremental approach to programming and avoids the cumbersome edit-debug-reedit cycle. Thus program development moves much faster and is definitely more enjoyable.

The remainder of this section will acquaint you with the use of muSTAR AIDS: how to get into it, how to fully utilize the various options available, and how to use the text editing commands. The text editing commands include cursor control, text display, and insertion/deletion of text. The description in section VI-C of how muSTAR works will be of value to those users who wish to extend or modify it to suit their needs.

The muSTAR editor will work for virtually any modern CRT terminal running at a high BAUD or data transfer rate. The only non-standard features used by muSTAR are the characters which move the cursor up a line and move the cursor to the home position, or upper left-hand corner of the screen. The distributed version is customized for the ADM-3A terminal. If some other terminal is being used, customized functions will have to be written to perform these functions, as described in section VI-C below.

A. Main Menu Commands

muSTAR is distributed as a muLISP SYS file. Thus it can be loaded by either method described in section I-H of this manual. Probably the easiest way is to use the following CP/M level command:

```
A>MULISP MUSTAR
```

This command will first load muLISP and display the logon prompt. Next the muSTAR system will be loaded and its executive driver will be called. The driver will display the muSTAR menu of options available to the user. This is the menu as displayed on the console:

```
OPTIONS: F  EDIT FUNCTION
         V  EDIT VARIABLE
         P  EDIT PROPERTY
         E  EVAL LISP
         Q  EVAL-QUOTE LISP
         T  TRACE FUNCTION
         U  UNTRACE FUNCTION
         R  READ FILE
         W  WRITE FILE
         D  SELECT DRIVE

ENTER CHOICE:
```

The facilities available in muSTAR fall into three catagories, as reflected in the list above. The first three options call the display-oriented editor. The next four are used for program debugging. The last three fall into the category of disk file storage.

When this menu is displayed, entering any of the valid option characters will initiate the corresponding facility as described below. The system will only respond to valid option characters. An unwanted choice can be aborted by simply typing a carriage return when the option prompts the user for input.

1. Editor Facilities

F EDIT FUNCTION — This option is used to create and/or edit a muLISP function's LAMBDA definition. When activated, this option displays the following prompt:

```
FUNCTION NAME(S):
```

As many function names as will fit on one line can then be typed. After a carriage return is entered, the system will display the definition(s) in pretty-printed form, using indentation to highlight the structure of the function definition(s). The cursor is positioned at the beginning of the text. The editor can then be used to modify the text as described in VI-B below. Once editing is completed and if the user desires, the function(s) are redefined using the modified definition(s).

V EDIT VARIABLE — This option is used to create and/or edit a program variable's value. When activated, this option displays the following prompt:

```
VARIABLE NAME(S):
```

As many variable names as will fit on one line can then be typed. After a carriage return is entered, the system will display the variable(s) and pretty-printed value(s) with the cursor at the beginning of the text. The editor can then be used to modify the text and the value(s) of the variable(s) can be reassigned.

P EDIT PROPERTY — This option is used to create and/or edit a name's property value. When activated, this option first displays the following prompt:

```
NAME:
```

Only one name is permitted for property edits; additional names will be ignored. After a carriage return is entered, the system will display the following prompt:

```
INDICATOR:
```

Again only one property indicator can be specified. After a carriage return is entered, the system will display the variable name, indicator, and current property value. After the edit, the modified expression can be reevaluated.

2. Debugging Facilities

E EVAL LISP — This option is used to initiate an eval-LISP driver loop as described in section I-D. The prompt string "* " distinguishes it from the default muLISP prompt string "$ ". This eval-LISP loop will continue until an expression evaluates to the name EXIT.

Q EVAL-QUOTE LISP — This option is used to initiate an eval-quote-LISP driver, also described in section I-D. This executive is preferred for debugging by some LISP users. The prompt string is "# ". This eval-quote-LISP loop will continue until the function EXIT () is executed.

T TRACE FUNCTION — This option is used to trace a LAMBDA or NLAMBDA defined function for debugging purposes. When activated, this option displays the following prompt:

```
FUNCTION NAME(S):
```

As many function names as will fit on one line can then be typed, followed by a carriage return. Whenever a traced function is called, the function name and actual arguments are displayed. When the function returns, both the function name and returned value are displayed.

Indenting is used to highlight the nesting of function calls.

U UNTRACE FUNCTION — This option is used to untrace or restore the definition of a traced function to the original definition. The same prompt as for the TRACE FUNCTION is displayed, calling for a sequence of functions to be entered by the user for untracing.

3. Disk I/O Facilities

R READ FILE — This option is used to read muLISP source files. When activated, this option displays the following prompt:

FILE NAME:

After a name and a carriage return is entered, an attempt is made to open the file with the given name of type LIB for input and read it into the system. If the file is not found the following message will be displayed and then you will be re-prompted for a file name:

FILE NOT FOUND

Files generated by either muSTAR or an external editor can be loaded in this way. For files made externally, the last statement of the file should be a call to the function RDS to return control to muSTAR. Also the function DRIVER should not be modified by the source file.

W WRITE FILE — This option is used to generate a muLISP source file. When activated, this option displays the following prompt:

FILE NAME:

After a name and a carriage return is entered, a new disk file with the given name of type LIB is created. A sequence of function definitions and/or variable values and properties is written to the file in pretty-printed form. The functions which are written are those on the property list of the file name under the indicator FUNCTIONS. The variables and/or property values which get written to the file are those of the names on the property list of the file name under the indicator VARIABLES. These property values themselves are also written to the file so they will automatically be read back for future muSTAR edits. Note that when the development phase is complete, files produced by muSTAR can be read by the muLISP system alone.

D SELECT DISK — This option is used to select the disk drive to be used in subsequent disk file accesses. Until this option is used to change the drive, the current default drive is used.

B. Text Editing

The muSTAR editor has been specifically designed to facilitate the editing of muLISP functions. It is a screen-oriented LISP editor, which continuously displays a "window" or "picture" of the text making up a function's definition. Thus the user can make changes and see the results instantly.

Because muSTAR is written entirely in muLISP, the user can extend the system to his/her own taste. Basically each ASCII control character is a function which can be redefined at will to perform whatever task is desired. This is more fully explained in section VI-C.

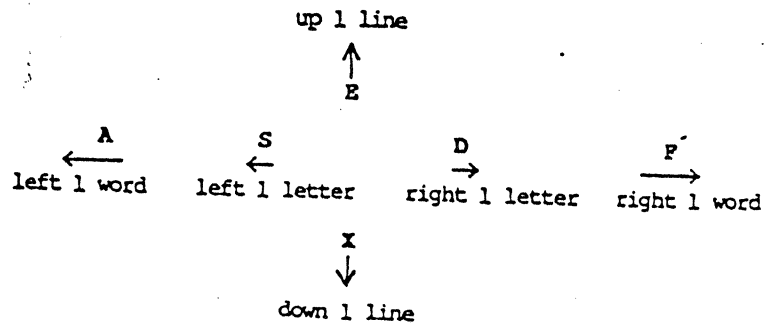
Function and variable names used in muSTAR which might conflict with user program names are terminated with a "$". In order to avoid such conflicts it is advisable to avoid using such names in user programs.

The general principle for using the muSTAR editor can be described in two steps: first, move the cursor to where you want to edit; second, insert, delete, or change the text at that point. The best way to learn your way around the system is to experiment with some simple function definitions.

1. Cursor Control

The cursor control commands are used to position the cursor at the point where the desired change is to be made in the text. The cursor (the block or line of light that shows where the next character will appear when typed) is manipulated mainly by directions entered with the left hand on the keyboard. On most terminals control characters are entered by depressing the control key, usually marked CTRL, and typing the desired letter while the control key is still depressed.

The basic cursor controls are oriented in the shape of a plus sign so that on standard keyboards the position of the key corresponds to the direction the cursor is to move. This is illustrated by the following diagram:



In the diagram above, the direction and length of the arrows indicate the direction and relative magnitude of cursor motion caused by the associated command. The best way to get the "feel" of the commands is to experiment with actual text.

CTRL-D moves the cursor one character to the right, CTRL-S moves it one character to the left. (Note that the latter command has the same effect as typing a CTRL-H or backspace). Amplifying this left-right movement, CTRL-F moves the cursor one word to the right, CTRL-A moves it one word to the left. Both of these commands will leave the cursor at the beginning of a word.

In muSTAR a word is defined to be a muLISP name in the text or a delimiter such as a parenthesis or period. Note that for these four commands, the text on the screen should be thought of as one long string of text. Thus the cursor is automatically advanced to the next line when the end of the current line is reacned. Alternatively, after the beginning of a line is reached when backspacing, the cursor will move to the end of the previous line.

Entering a CTRL-E will move the cursor up one line. A CTRL-X will move it down a line. After the cursor is moved to the new line by either of these commands, it is automatically advanced to the next word on the line. Typing a linefeed or CTRL-J will move the cursor to the beginning of the next line.

To move to the beginning of a line, the command is CTRL-Q. To move in the opposite direction, to the very end of a line (after the last character), type CTRL-B.

A carriage return (i.e. a CTRL-M) typed when in the non-INSERT mode (see section B-3 below) advances the cursor to the beginning of the next line. A new line is added if at the bottom of the current text. See section B-3 for the effect of carriage returns when in the INSERT mode.

Using only the cursor control commands, the cursor can not be moved to places on the screen that are not characters in the text. To move the cursor past the end of the current line or past the last line of text, use spaces and/or carriage returns to insert new text instead of using the cursor control characters.

2. Display Control

When long or multiple definitions are displayed on the CRT screen, only part of the text can be visible at any one time. The text can be thought of as being "written" on a long scroll. The CRT screen is then a "window" on the text, which shows only a portion of the scroll. Since our window, the CRT, can not move, muSTAR scrolls the text up (toward the end of the text) or down (toward the beginning of the text). Thus moving the text up or down is called "scrolling".

To see the next line of text, typing CTRL-Z moves the window DOWN a line or, relatively speaking, scrolls the text UP a line. To see the previous line of text, typing a CTRL-W moves the window UP one line or,

relatively speaking, scrolls the text DOWN one line. When scrolling, the cursor remains at the same position in the text, moving up or down with the text. However, if the cursor reaches the top or bottom of the screen, it remains there. The cursor never leaves the screen.

CTRL-C is used to move the window down a screenful (i.e. to display the next 18 lines on a 24 line screen). CTRL-R does the inverse operation of moving the window up a full screen.

The display control commands are deactivated if there is no more text in the direction of window motion.

3. Entering Text

There are two modes for entering text. CTRL-V is a toggle "switch" used to switch between modes. In either mode text is inserted by simply typing it in.

Initially, when the toggle switch is OFF, the editor is in the non-INSERT mode. Any existing text on a line will be overwritten by new text being entered. This is the easiest way to enter text initially. Mistakes can be corrected by simply typing over them with the corrections. The space bar will "erase" characters.

When a CTRL-V has been typed, muSTAR is in the INSERT mode. Characters and spaces that are typed are actually inserted in front of the remaining characters on the line, if any. The characters to the right of the inserted characters will be pushed over to the right. As a step-by-step example, the text "(CONS BETA)" can be modified to read "(CONS ALPHA BETA)" by inserting "ALPHA" before "BETA". In the INSERT mode, this insertion would appear on the console as follows, where the underscore indicates the cursor position:

```
(CONS BETA)
(CONS ABETA)
(CONS ALBETA)
(CONS ALPBETA)
(CONS ALPHBETA)
(CONS ALPHABETA)
(CONS ALPHA BETA)
```

When muSTAR is in the insert text mode, carriage returns (i.e. CTRL-M) cause new lines to be inserted in the text. The cursor will end up immediately following the new line. See section B-1 above for the effect of carriage returns entered in the Non-INSERT mode. Entering a CTRL-N will always insert a new blank line in the text; however, the cursor will end up on the new line.

CTRL-P is the escape control character used to make delimiter characters be used as names instead of delimiters. To use spaces, parentheses, periods, and brackets as names simply type a CTRL-P before typing the delimiter character.

4. Deleting Text

There are four ways to delete text:

1) In the Non-Insert mode, simply type spaces or new characters over the text to be deleted;

2) To delete the character that the cursor is on, type CTRL-G;

3) To delete the word to the right of the cursor, type CTRL-T.

4) The current line can be deleted by typing CTRL-Y.

A word of caution: since the letter Y is so easy to hit by mistake when reaching for a T, it is advisable to use extra care when deleting words, since a long reach can eliminate an entire line. It is also a good idea to use CTRL-Y with discretion since repeating the CTRL-T command will "walk" you down the line. This often allows the user to "recycle" words into a new line.

5. Finding Names

The ability to search for a muLISP name in text is the function of these commands. CTRL-O displays the following prompt on the top line of the screen:

FIND NAME?

The user can then enter a name, terminated by a carriage return. The cursor will then move to the next occurrence of the name in the text. Only the text actually on the screen is searched, and if the name is not found the search will begin again at the top of the screen. If the name is still not found, the cursor will end up advanced one word to the right of its original position.

To search for the next occurrence of the same name given the last time CTRL-O was used, type CTRL-L.

6. Exiting Editor

When the edit of a variable, function, or property is completed, typing a CTRL-K will cause the following menu to be displayed on the console:

```
OPTIONS: E  EVALUATE TEXT
         A  ABANDON TEXT
         C  CONTINUE EDIT

ENTER CHOICE:
```


Normally the "E" option is choosen since it evaluates the text which has been edited. The "A" option abandons the text and does not evaluate the text. In case CTRL-K was typed inadvertently, the "C" option can be used to return to the edit.

7. Command Summary Table

Control Character	Function
A	Move cursor left one word
B	Move cursor to end of current line
C	Scroll text up a screenful
D	Move cursor right one character
E	Move cursor up one line
F	Move cursor right one word
G	Delete current character
H	Move cursor left one character
I	UNUSED
J	Move cursor down one line following leading blanks (LINEFEED)
K	Exit editor command
L	Repeat last search command
M	Move cursor down one line to beginning of line (RETURN)
N	Insert a blank line
O	Find a name
P	Escape character used to enter delimiters as names
Q	Move to beginning of current line
R	Scroll text down a screenful
S	Move cursor left one character
T	Delete word to the right of cursor
U	UNUSED
V	Toggle insert mode switch
W	Scroll text down one line
X	Move cursor down one line
Y	Delete current line
Z	Scroll text up one line

C. Customizing muSTAR

1. Console Customization

All muSTAR users except those owning ADM-3A terminals will probably have to write a simple muLISP library file containing functions for moving the cursor up a line and for moving the cursor to the home position (the upper left corner of the screen). The muSTAR function UP-LIN$ is used to move the cursor to the beginning of the current line and up a given number of lines. If no argument is given, it still moves the cursor up one line. The function HOME$ moves the cursor to the upper left-hand corner of the screen without erasing the screen. It also includes a TERPRI to reset the muLISP cursor position counter. The following are the default definitions of these functions:

```
(DEFUN UP-LIN$ (LAMBDA (NUM)
  ((ZEROP NUM))
  (PRIN1 CR)
  (LOOP
    (PRIN1 UPLINE)
    (SETQ NUM (SUB1 NUM))
    ((NOT (PLUSP NUM))) ) ))

(DEFUN HOME$ (LAMBDA ()
  (PRIN1 HOME)  (TERPRI)  (PRIN1 HOME) ))

(SETQQ PAG-LEN$ 24)                    % Number of lines on CRT %
(SETQQ LIN-LEN$ 78)                    % Number of columns - 2 %

(SETQQ HOME "^^")                      % CTRL-^ = 1E hex %
(SETQQ UPLINE "^K")                    % CTRL-K = 0B hex %
(SETQQ CR "^M")                        % CTRL-M = 0D hex %
(RDS)
```

Note that the "^" notation used in the last lines of the above listing means that an actual control character is to be inserted in the text file between the two double quote signs. It also may be necessary to adjust for your console's page and line lengths.

To customize muSTAR for your terminal, first use an external editor to make a file of type LIB using the definitions above as a model. Note that the call to the function RDS at the end of the file must be included to make the console the current input source. Immediately after loading MUSTAR.SYS as described in section VI-A above, choose the "R" option and read in your customization file. The customized version of muSTAR should then be saved as a SYS file, using a new name to distinguish it from the uncustomized version. To accomplish this, select the eval-LISP "E" option and then make the following command:

```
(SAVE MYSTAR)
```

This will create a disk file named MYSTAR.SYS. Thereafter initiating muSTAR is simply a matter of loading MYSTAR instead of MUSTAR as described in the introduction to section VI-A.

2. The muSTAR Executive

The executive is a very simple loop which displays the main option menu and then calls a function to perform the requested option. On the property of each option character, under the indicator EXECUTIVE, is the LAMBDA body evaluated when the respective option is chosen. The function QUERY$ is used to display a prompt and input the response. It returns a list of expressions entered by the user.

If an editor option was chosen, this list is passed to the S-expression-to-text translator. Here the expression is converted to the internal text data structure as described in the next sub-section. The editor is then called to pretty-print and permit editing of the text. When the edit is complete and the user desires to evaluate the text, the text-to-S-expression translator is called to translate the text and evaluate the result. If parentheses are unbalanced or the continue option is chosen, the editor is called again so the edit can continue.

The trace and untrace options pass the list of names entered by the user to TRACE and UNTRACE functions respectively. A function is traced by redefining it to call the function EVTRACE. Thus when a traced function is called, it calls EVTRACE which prints its arguments, evaluates the original function, and then prints the resulting value.

The write file option opens a file for output on drive *DRIVE*. First it prints the basic functions needed to read-in the remainder of the file (i.e. DEFUN, SETQQ, PUTQQ, and FLAGQQ). Then a list of functions and variables in the file are printed. Finally the pretty-printed function definitions, name values, and property values are printed. If a function which is being saved has been traced, the original definition and not the traced version is saved.

3. Text Data Structure

It is a relatively simple matter for an experienced muLISP programmer to add features to the editor to suit his/her tastes. Changes which can be made include modifying the effect of control keys and taking advantage of special CRT features. The first step is to become thoroughly familar with how the text is stored and modified. Then the already existing building blocks in the system for manipulating the text can be combined to achieve the desired effect.

The text currently on the screen at any given time is stored as a list bound to the global variable *TEXT*. Except for the first element of *TEXT* which is always NIL, each element of this list represents a line of text. The global variable PAG-LEN$ is set to the maximum allowable number of lines on the CRT screen. Its default value is 24. Thus the length of the list *TEXT* can never be greater than PAG-LEN$.

Each line or row of text (i.e. element of *TEXT*) is a list specifying the text on that row. The first element of a row is always a non-negative integer which gives the number of leading blanks, or

initial spaces, on the line. The remaining elements of the list are called tokens. Tokens represent either the muLISP delimiters or names making up the text being expressed. The four possible delimiters are shown here enclosed in double quotes:

"(" ")" " " "."

These four delimiters are stored as names on the list. In contrast, tokens which represent names appear as sublists on the row list. Generally the sublist is a single-element list whose element is the name in question. Only if the cursor is "inside" a name (i.e. the cursor is positioned after the first character) will a name be unpacked or "exploded" into the characters making up the name using the function UNPACKS. When the cursor leaves the name, the characters are repacked or "compressed" into a new name using the function PACKS.

At any given time, the value of the global variable *ROW* is a sublist of *TEXT*. The second element (i.e. the CADR) of this sublist is the row of text which the cursor is currently on. This is the reason the first element of *TEXT* is the dummy row NIL. Among other things, being one row above, or ahead, of the current row is convenient for deleting the current row from *TEXT* using RPLACD.

The current position of the cursor in the current row is determined by the global variable *COL*. It is always a sublist of the current row of text. For the same sort of reasons as for *ROW*, the sublist begins one token to the left, or ahead, of the current token. Note that the cursor is never to the left of a line's leading blanks. Thus *COL* can always be ahead of the current token.

Lines of text which have been scrolled off the top or bottom of the screen are stored as lists under the variables *PRE-TEXT* and *POST-TEXT* respectively. These two lists are best thought of as stacks. The line of text which is immediately adjacent to the current "window" is the top element of the stack. Thus the rows making up *PRE-TEXT* are in reverse order. A row can be pushed onto *PRE-TEXT* or *POST-TEXT* only if there is insufficient room on the screen to store the row.

4. Text Primitives

When the text editor is running, console input is in the raw input mode as described in the reader control variable portion of section V-M. An input character can either be a delimiter, a normal printable character, or a control character. A delimiter or printable character is made part of the current row of text in accordance with the structure defined above for storing text. If the control variable *INSERT* is non-NIL, the character is added to the text instead of overwriting old text.

Each control character has as its value an alias name. For example, the value of control A is CTRL-A. The other control characters have analogous aliases. The definition of the respective alias is evaluated when a control character is entered. Thus the function CTRL-A moves the cursor and text pointers one token to the left. Each of these functions is documented by the description given in section VI-B above. Altering the definition of these functions is the easiest way to customize muSTAR. There is a very complete set of existing primitives in the muSTAR source. The purpose of each should be apparent from the function name.

Appendix A: Backus-Naur Form

Backus-Naur Form (BNF) equations provide a standard convention used to formally define the syntax of a language. Within a BNF equation a character string immediately preceded by a "<" and followed by a ">" denotes the class of objects named by the string. A character string not so delimited stands for the string itself. Note the contrast between this and the conventional practice of quoting strings which stand for themselves and not quoting strings which are names of objects.

A BNF equation defines the set of syntatic objects belonging to a particular category of the language's syntax. The symbol "::=" is used as an abbreviation for the phrase "is a" and the vertical bar "|" is short for "or". A BNF definition of numerals may be specified as follows:

```
<numeral> ::= <digit> | <digit> <numeral>
<digit>   ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9
```

As in this example, recursive definition is frequently used in BNF equations. Also note how juxtaposition of category names denotes the corresponding concatenation of the objects within those categories.

Backus-Naur Form is defined in the "ALGOL 60 Report, Revised Report on Algorithm Language ALGOL 60", Communications of the ACM, Volume 6, January 1963.

Appendix B: How to Copy the Master Diskette

As soon as possible after receipt of the master diskette supplied by The Soft Warehouse, make a working copy of the diskette. The following information is provided as a guide for those who are not yet familar enough with their computer's disk operating system to make a copy of a diskette. Since there are many different operating systems and computer configurations, this discussion can only be a general guide.

1. Study the documentation supplied with the computer's disk operating system. Most cases of accidental erasure of the master diskette or other irrecoverable errors are committed in the first few moments by overly anxious and inexperienced users.

2. Obtain an appropriate number of blank high-quality diskettes suitable for your drive.

3. Become thoroughly familiar with the terminal, computer, disk drives, and operating system.

4. Practice initializing a diskette, then transferring a disk operating system to the initialized diskette. Generate the largest version of the operating system that your computer is capable of supporting.

5. Practice transferring to the new diskette files from a spare, write-protected diskette.

6. For operating systems with a diskcopy utility, use this method to copy the master diskette since it is definitely the easiest and fastest way to do so. Make absolutely sure the drive with the master is the source and NOT the destination for the copy command!

7. For multiple drive systems, the following CP/M PIP (Peripheral Interchange Program) command will copy all files from drive A to drive B:

```
PIP B:=A:*.*
```

The equivalent command for Cromemco CDOS users is:

```
XFER B:=A:*.*
```

8. Copying diskettes is much more laborious on systems having only one drive. Generally, it involves repetitively reading a portion into RAM memory from the source disk, switching disks, then writing the portion onto the destination disk. This is repeated until the entire source disk has been copied. The process generally involves using the CP/M DDT utility program or a resident monitor in read-only memory.

9. If the above methods are unsuccessful because your disk drive is unable to read the master diskette and you have a multiple drive system, try reading the master on different drives. The master is recorded on a high quality diskette by a precisely adjusted drive. However, there are inevitably slight mechanical and electrical differences between various drives and diskettes.

10. If your system is still unable to read the master diskette, then:

a) Check to see whether the proper type of diskette was specified when you placed your order.

b) Get help from an experienced professional.

c) Make sure your drives are correctly aligned, and if not, have them professionally aligned.

d) Contact directly the supplier from whom the system was purchased.

Appendix C: Implementing Machine Language Subroutines

Some specialized muLISP applications may require the writing of machine language routines. For instance a user may wish to enhance muLISP with graphics capability, or perhaps it is necessary to hand compile a particularly critical function for efficiency reasons.

Address typing is used by muLISP to determine a function's type. This necessitates the beginning of all machine language subroutines in low memory. A dummy jump table, beginning at location 0103H (hexadecimal), has been set aside for jumps from low memory to user defined routines. There is sufficient room for four (4) JMP instructions (i.e. opcode C3H). These jump instructions can be altered to jump to the address of the user defined routine wherever they are located. Depending on your system, additional room for jumps might be found in unused areas in page zero of memory.

Since muLISP uses all available memory below the DOS (disk operating system), the best place to locate user defined routines is in the "protected" memory above the DOS. Of course this mandates generating a DOS slightly smaller than what your computer is normally capable of supporting. Another alternative is to change the JMP instruction at location 0005H to an address less than its current address thereby "fooling" muLISP into believing it is operating under a smaller DOS. Of course another JMP instruction will have to be placed at the new address to jump to the DOS. This will free some memory just below the DOS.

All user defined subroutines will be CBV, spread functions of at most three arguments (see section V-O). If more than three arguments are required, they can be passed in as a list. The addresses of arguments are passed to machine language routines in the register pairs. The first argument is in the HL register, the second in the DE register, and the third in the BC register.

To return control to muLISP simply terminate all routines with a RET instruction. The returned value of the function will be the data structured pointed to by the HL register pair. It must be the address of a bona fide muLISP data object (see section II). Even if no special value is desired to be returned, HL should still be set to some value such as NIL. The value of NIL can be determined by disassembling the machine language definition of the function NULL.

Linkage to machine language routines is done through the use of the muLISP function PUTD. For instance, evaluation of the following expression will define the function FOO to be a routine which begins at location 0103H:

```
(PUTD (QUOTE FOO) 259)
```

The function GETD can be used to find the starting address of primitively defined muLISP subroutines (see section V-J). Knowing these addresses, user defined subroutines can call the primitive subroutines directly.

Appendix D: LISP Bibliography

Allen, J. R., Anatomy of LISP,
McGraw-Hill Book Company, New York, NY, 1978.

Berkeley, E. C., and Bobrow, D. G., (eds),
The Programming Language LISP: Its Operation and Applications,
The M.I.T. Press, Cambridge, MA, 1964.

BYTE: The Small Systems Journal, LISP Issue, Vol 4, No 8,
BYTE Publications Inc., Peterborough, NH, August 1979.

Henderson, P., Function Programming: Application and Implementation,
Prentice-Hall, Englewood Cliffs, NJ, 1980.

Friedman, D. P., The Little LISPer,
Science Research Associates Inc.,
London, England, 1974.

McCarthy, J., Recursive Functions of Symbolic Expressions
and Their Computation by Machine,
Comm. ACM, pages 184-195, April 1960.

McCarthy, J., et al., LISP 1.5 Programmer's Manual,
The M.I.T. Press, Cambridge, MA, 1963.

Maurer, W. D., A Programmer's Introduction to LISP,
American Elsevier, New York, NY, 1973.

Siklossy, L., Let's Talk LISP,
Prentice-Hall, Englewood Cliffs, NJ, 1976.

Weissman, C., LISP 1.5 Primer,
Dickenson Publishing Co., Belmont, CA, 1968.

Winston, P. H., Artificial Intelligence,
Addison-Wesley Publishing Co., Reading, MA, 1977.

Appendix E: Function Index

Type	Name	Category	Group	Page
CBN	QUOTE	Evaluator	O-1	27
CBV	QUOTIENT	Numerical	L-5	16
CBV	RADIX	Printer/Reader	N-7	23
CBV	RATOM	Reader	M-2	17
CBV	RDS	Reader	M-1	17
CBV	READ	Reader	M-3	18
CBV	READCH	Reader	M-4	19
CBV	RECLAIM	Storage	P-1	31
CBV	REMAINDER	Numerical	L-6	16
CBV	REMFLAG	Flag	I-3	12
CBV	REMPROP	Property	H-4	11
CBV	REVERSE	Constructor	B-3	3
CBV	RPLACA	Modifier	C-1	4
CBV	RPLACD	Modifier	C-2	4
CBV	SAVE	System	Q-1	32
CBV	SET	Assignment	G-1	9
CBN	SETQ	Assignment	G-2	9
CBV	SPACES	Printer	N-5	22
CBV	SYSTEM	Environment	Q-3	33
CBV	TERPRI	Printer	N-4	22
CBV	TIMES	Numerical	L-4	16
CBV	UNPACK	Sub-atomic	K-2	14
CBV	WRS	Printer	N-1	21
CBV	ZEROP	Recognizer	D-7	5

Appendix F: Concept Index

```
(PROG1 ""
  (LOOP ((EQ (EVAL (READ)) STOP)) )
  (DRIVER) )

%************** U T I L I T Y  R O U T I N E S **************%

(PUTD DEFUN (QUOTE (NLAMBDA (FUNC EXP)
  ((EQUAL (GETD FUNC) EXP))
  ( ((NULL (GETD FUNC)))
    (PRIN1 "*** REDEFINING ")
    (PRINT FUNC) )
  (PUTD FUNC EXP)
  FUNC ))) )

(DEFUN SETQQ (NLAMBDA (NAME EXP)
  (SET NAME EXP)
  NAME ))

(DEFUN PUTQQ (NLAMBDA (NAME ATM EXP)
  (PUT NAME ATM EXP)
  NAME ))

(DEFUN FLAGQQ (NLAMBDA (NAME ATM)
  (FLAG NAME ATM)
  NAME ))


(DEFUN PACK$ (LAMBDA (TOKEN)
  ((NULL (CDR TOKEN))
    TOKEN )
  (CONS (PACK TOKEN)) ))

(DEFUN UNPACK$ (LAMBDA (TOKEN)
  ((OR (CDR TOKEN) (EQ (LENGTH (CAR TOKEN)) 1))
    TOKEN )
  (UNPACK (CAR TOKEN)) ))

(DEFUN RDC$ (LAMBDA (LST TAIL)
  (LOOP
    ((EQ (CDR LST) TAIL)
      LST )
    (POP LST) ) ))

(DEFUN CHOP$ (LAMBDA (LST)
  (LOOP
    ((NULL (CDDR LST))
      (PROG1 (CADR LST) (RPLACD LST NIL)) )
    (POP LST) ) ))

(DEFUN SPLIT$ (LAMBDA (LST NUM)
  (LOOP
    ((ATOM LST) NIL)
    (SETQ NUM (SUB1 NUM))
```

```
  ((ZEROP NUM)
    (PROG1 (CDR LST) (RPLACD LST NIL)) )
  (POP LST) ) ))


EFUN MENU$ (LAMBDA (LST
  READCH )
(SPACES  11)
(PRIN1 "OPTIONS: ")
(MAPC LST (QUOTE (LAMBDA (LINE)
  (PRIN1 (CAR LINE))  (SPACES 2)
  (MAPC (CDR LINE) (QUOTE (LAMBDA (WORD)
    (PRIN1 WORD)
    (SPACES 1) )))
  (TERPRI)
  (SPACES 20) )))
TERPRI)  (SPACES 6)
PRIN1 "ENTER CHOICE: ")
LOOP
  ((ASSOC (READCH) LST)
    (PRINT RATOM) ) ) ))

FUN QUERY$ (LAMBDA (TEXT)
TERPRI)
SPACES (DIFFERENCE 18 (LENGTH TEXT)))
PRIN1 TEXT) (PRIN1 :) (SPACES 1)
RD-LIN$) ))

FUN RD-LIN$ (LAMBDA (
 WORD LINE )
LOOP
 (SETQ WORD (RD-WRD$))
 ( ((NULL WORD))
    (PUSH WORD LINE) )
 ((EQ RATOM CR)
    (REVERSE LINE)) ) ))

FUN RD-WRD$ (LAMBDA (
 WORD )
LOOP
 ((OR (FLAGP (READCH) (QUOTE DEL-CHAR)) (EQ RATOM CR))
    ((NULL WORD) NIL)
    (PACK (REVERSE WORD)) )
 (PUSH RATOM WORD) ) ))

UN UP-LIN$ (LAMBDA (NUM)
(ZEROP NUM))
PRIN1 CR)
LOOP
 (PRIN1 UPLINE)
 (SETQ NUM (SUB1 NUM))
 ((NOT (PLUSP NUM))) ) ))

UN BACKUP$ (LAMBDA (NUM)
BCK-SPACE$ NUM)
```



```
  (SPACES NUM)
  (BCK-SPACE$ NUM) ))

(DEFUN BCK-SPACE$ (LAMBDA (NUM)
  (LOOP
    ((NOT (PLUSP NUM)))
    (PRIN1 BACK)
    (SETQ NUM (SUB1 NUM)) ) ))

(DEFUN HOME$ (LAMBDA ()
  (PRINT HOME)
  (PRIN1 HOME) ))

(DEFUN SPACE$ (LAMBDA (CHAR$)
  (EQ CHAR$ " ") ))


(DEFUN PRIN2 (LAMBDA (EXP PRIN1)
  (PRIN1 EXP) ))

(DEFUN APPEND (LAMBDA (LST TAIL)
  ((NULL LST) TAIL)
  (CONS (CAR LST) (APPEND (CDR LST) TAIL)) ))

(DEFUN ADD1 (LAMBDA (NUM)
  (PLUS NUM 1) ))

(DEFUN SUB1 (LAMBDA (NUM)
  (DIFFERENCE NUM 1) ))

(DEFUN MAPC (LAMBDA (LST FUNC)
  (LOOP
    ((NULL LST) NIL)
    (FUNC (POP LST)) ) ))
```

```
*************** E D I T O R   E X E C U T I V E ******************

(DEFUN DRIVER (LAMBDA ()
  (SETQ RDS)
  (SETQ WRS)
  (SETQ ECHO)
  (SETQ *DRIVE*)
  (SETQ PRIN1 (QUOTE PRIN1))
  (TERPRI 4)

  (SETQ READCH)
  ( ((EQ CTRL-Z (QUOTE CTRL-Z))
      (LOOP
        ((EQ (READCH) CR)) )
      (PRIN1 "TYPE CTRL-Z: ")
      (SETQ CTRL-Z (READCH))
      (SET CTRL-Z (QUOTE CTRL-Z)) ) )
  (SETQ READCH T)
  (SPACES 12)
  (PRIN1 "*** The muSTAR AIDS ***")
  (LOOP
    (LINELENGTH (ADD1 LIN-LEN$))
    (TERPRI 3)
    (SETQ CHAR$ (MENU$ MENU$))
    (APPLY (GET CHAR$ (QUOTE EXECUTIVE))) ) ))


(SETQQ LIN-LEN$ 78)
(SETQQ PAG-LEN$ 24)

(SETQQ MENU$ (
  (F EDIT FUNCTION)
  (V EDIT VARIABLE)
  (P EDIT PROPERTY)
  (E EVAL LISP)
  (Q EVAL-QUOTE LISP)
  (T TRACE FUNCTION)
  (U UNTRACE FUNCTION)
  (R READ FILE)
  (W WRITE FILE)
  (D SELECT DRIVE) ))


PUTQQ F EXECUTIVE (LAMBDA (
    LST )
  (SETQ LST (QUERY$ "FUNCTION NAME(S)"))
  ((NULL LST))
  (EDIT-TXT (DEF-TO-TXT LST)) ))

PUTQQ V EXECUTIVE (LAMBDA (
    LST )
  (SETQ LST (QUERY$ "VARIABLE NAME(S)"))
  ((NULL LST))
  (EDIT-TXT (SET-TO-TXT LST)) ))
```

```
(PUTQQ P EXECUTIVE (LAMBDA (
    NAME INDICATOR )
  (SETQ NAME (QUERY$ (QUOTE NAME)))
  ((NULL NAME))
  (SETQ INDICATOR (QUERY$ (QUOTE INDICATOR)))
  ((NULL INDICATOR))
  (EDIT-TXT (PUT-TO-TXT (CAR NAME) (CAR INDICATOR))) ))

(PUTQQ E EXECUTIVE (LAMBDA ()
  (LOOP
    (TERPRI)
    (PRIN1 "* ")
    ((EQ (PRINT (EVAL (READ))) EXIT)) ) ))

(PUTQQ Q EXECUTIVE (LAMBDA ()
  (LOOP
    (TERPRI)
    (PRIN1 "# ")
    ((EQ (PRINT (APPLY (READ) (READ))) EXIT)) ) ))

(DEFUN EXIT (LAMBDA ()
  EXIT ))

(PUTQQ T EXECUTIVE (LAMBDA ()
  (TRACE (QUERY$ "FUNCTION NAME(S)")) ))

(PUTQQ U EXECUTIVE (LAMBDA ()
  (UNTRACE (QUERY$ "FUNCTION NAME(S)")) ))


(PUTQQ R EXECUTIVE (LAMBDA (
    NAME ECHO )
  (LOOP
    (SETQ NAME (QUERY$ "FILE NAME"))
    ((NULL NAME))
    (SETQ NAME (CAR NAME))
    ((RDS NAME (QUOTE LIB) *DRIVE*))
    (TERPRI)
    (PRINT "FILE NOT FOUND") )
  ((NULL NAME))
  (LOOP
    (EVAL (READ))
    ((NULL RDS)) ) ))

(PUTQQ W EXECUTIVE (LAMBDA ()
  (W-EXEC) ))

(DEFUN W-EXEC (LAMBDA (
    NAME ECHO )
  (SETQ NAME (QUERY$ "FILE NAME"))
  ((NULL NAME))
  (SETQ NAME (CAR NAME))
  (WRS NAME (QUOTE LIB) *DRIVE*)
  (PRIN2 (LIST (QUOTE PUTD) (QUOTE DEFUN) (LIST
    (QUOTE QUOTE) (GETD DEFUN) )))
```

```
(TERPRI)
(PRIN2 (LIST (QUOTE DEFUN) (QUOTE SETQQ) (GETD SETQQ)))
(TERPRI)
(PRIN2 (LIST (QUOTE DEFUN) (QUOTE PUTQQ) (GETD PUTQQ)))
(TERPRI)
(PRIN2 (LIST (QUOTE DEFUN) (QUOTE FLAGQQ) (GETD FLAGQQ)))
(TERPRI 3)
(PRT-TXT (PUT-TO-TXT NAME (QUOTE FUNCTIONS)))
(TERPRI)
(PRT-TXT (PUT-TO-TXT NAME (QUOTE VARIABLES)))
(TERPRI)
(MAPC (GET NAME (QUOTE FUNCTIONS)) (QUOTE (LAMBDA (ATM)
  (TERPRI)
  ((EQ (CAR (CADDR (GETD ATM))) (QUOTE EVTRACE))
    (UNTRACE (LIST ATM))
    (PRT-TXT (DEF-TO-TXT (LIST ATM)))
    (TRACE (LIST ATM)) )
  (PRT-TXT (DEF-TO-TXT (LIST ATM))) )))
(TERPRI)
(MAPC (GET NAME (QUOTE VARIABLES)) (QUOTE (LAMBDA (ATM)
  (TERPRI)
  ( ((EQ ATM (EVAL ATM)))
    (PRT-TXT (SET-TO-TXT (LIST ATM))) )
  (MAPC (CDR ATM) (QUOTE (LAMBDA (EXP)
    (TERPRI)
    ((ATOM EXP)
      (PRIN2 (LIST (QUOTE FLAGQQ) ATM EXP))
      (TERPRI) )
    (PRT-TXT (PUT-TO-TXT ATM (CAR EXP))) ))) )))
(TERPRI)
(PRINT "(RDS)")
(WRS) ))

(SETQQ D EXECUTIVE (LAMBDA (
  CHAR$ )
(LOOP
  (SETQ CHAR$ (QUERY$ "DRIVE LETTER"))
  ((NULL CHAR$))
  (SETQ CHAR$ (CAR CHAR$))
  ((EQ (LENGTH CHAR$) 1)) )
(NULL CHAR$))
(SETQ *DRIVE* CHAR$) ))
```



```
;*************  TEXT EDITING FUNCTIONS  ********************;

(DEFUN EDIT-TXT (LAMBDA (*TEXT*
  *PRE-TEXT* *POST-TEXT* *ROW* *COL* *INSERT* *STRING* CHAR$ READCH )
(SETQ *PRE-TEXT*)
(SETQ *POST-TEXT* (SPLIT$ *TEXT* PAG-LEN$))
(LOOP
  (SETQ *ROW* *TEXT*)
  (SETQ *COL* (CADR *ROW*))
  (DISP-TXT *TEXT* *ROW* *COL*)
  (LOOP
    (SETQ CHAR$ (READCH))
    ((EQ (EVAL CHAR$) (QUOTE CTRL-K)))                ; EXIT EDIT CHAR
    ( ((FLAGP CHAR$ (QUOTE DEL-CHAR))
       (DEL-CHAR CHAR$) )
      ((OR (FLAGP CHAR$ (QUOTE PRT-CHAR)) (NUMBERP CHAR$))
       (PRT-CHAR CHAR$) )
      (APPLY (EVAL CHAR$) (LIST *INSERT*)) ) )
  (CTRL-F)
  (TERPRI (LENGTH *ROW*))
  (SETQ CHAR$ (MENU$ (QUOTE (
    (E EVALUATE TEXT)
    (A ABANDON TEXT)
    (C CONTINUE EDIT) ))))
  ((EQ CHAR$ (QUOTE A)))
  ((AND
      (EQ CHAR$ (QUOTE E))
      (EVAL-TEXT (CONS NIL (APPEND
        (REVERSE *PRE-TEXT*)
        (APPEND (CDR *TEXT*) *POST-TEXT*) ))) )) ) ))


(DEFUN PRT-CHAR (LAMBDA (CHAR$                        ; PRINTABLE CHARS ;
  TOKEN )
(PRIN1 (COND
  ((GET CHAR$ (QUOTE ALIAS)))
  (CHAR$) ))
((NULL (CDR *COL*))
  (INSERT-PRT CHAR$) )
((NOT *INSERT*)
  (SETQ TOKEN (DELETE-CHAR))
  (INSERT-PRT CHAR$) )
(INSERT-PRT CHAR$)
(PRT-REST-LINE *ROW* *COL* 0) ))

(MAPC (QUOTE (
A B C D E F G H I J K L M N O P Q R S T U V W X Y Z
a b c d e f g h i j k l m n o p q r s t u v w x y z
! # $ & ' * + - / : ; < = > ? @ \ ^ _ ` { | } ~ "%" **** ))
  (QUOTE (LAMBDA (CHAR$) (FLAG CHAR$ (QUOTE PRT-CHAR)))) )


(DEFUN DEL-CHAR (LAMBDA (CHAR$                        ; DELIMITER CHARS ;
  TOKEN )
(PRIN1 CHAR$)
```

```
((NULL (CDR *COL*))
  (INSERT-DEL CHAR$) )
 (NOT *INSERT*)
  (SETQ TOKEN (DELETE-CHAR))
  (INSERT-DEL CHAR$) )
INSERT-DEL CHAR$)
PRT-REST-LINE *ROW* *COL* 0) ))

PC (QUOTE ( " " "," "(" ")" "." "[" "]" ))
QUOTE (LAMBDA (CHAR$) (FLAG CHAR$ (QUOTE DEL-CHAR)))) )

FUN CTRL-P (LAMBDA (
 CHAR$ )                                    % ESCAPE CHAR %
SETQ CHAR$ (READCH))
PRT-CHAR CHAR$) ))

QQ "(" ALIAS {)
QQ ")" ALIAS })
QQ "_" ALIAS _)


UN CTRL-V (LAMBDA ()
ETQ *INSERT* (NOT *INSERT*)) ))             % TOGGLE *INSERT* %

UN CTRL-D (LAMBDA (
CHAR$ TOKEN )                               % ADVANCE CHAR %
ETQ TOKEN (NEXT-RIGHT-TOKEN))
NULL TOKEN)
 (NULL (CDDR *ROW*)))                       % IF AT END OF CURRENT LINE
 CTRL-M) )                                  %   AND NOT END OF TEXT,  %
ATOM TOKEN)                                 %   THEN NEW LINE.        %
 PRIN1 TOKEN)                               % IF TOKEN IS A DELIMITER  %
MOVE-RIGHT-TOKEN) )                         %   PRINT TOKEN AND       %
TQ TOKEN (UNPACK$ TOKEN))                   %   ADVANCE COL.          %
TQ CHAR$ (CONS (POP TOKEN)))
T-TOK CHAR$)
TOM (CAR *COL*))
(NULL TOKEN)
 (POP *COL*) )
RPLACD *COL* (CONS CHAR$ (CDR *COL*)))
OP *COL*)
RPLACA (CDR *COL*) TOKEN) )
LACA *COL* (APPEND (CAR *COL*) CHAR$))
LL TOKEN)
PLACD *COL* (CDDR *COL*)) )
ACA (CDR *COL*) TOKEN) ))


 CTRL-F (LAMBDA (
KEN )                                       % ADVANCE TOKEN %
Q TOKEN (MOVE-RIGHT-TOKEN))
LL TOKEN)
OOP
 (NULL (CDDR *ROW*)))
```



```
      (CTRL-J)
      ((CDR *COL*)) ) )
   (PRT-TOK TOKEN)
   (LOOP
     ((NOT (SPACE$ (NEXT-RIGHT-TOKEN))))
     (MOVE-RIGHT-TOKEN)
     (SPACES 1) ) ))

(DEFUN CTRL-B (LAMBDA (
    TOKEN )                                 % MOVE TO END LINE
  (LOOP
    (SETQ TOKEN (MOVE-RIGHT-TOKEN))
    ((NULL TOKEN))
    (PRT-TOK TOKEN) ) ))


(DEFUN CTRL-S (LAMBDA (
    CHAR$ TOKEN )                           % RETREAT CHAR %
  (SETQ TOKEN (CAR *COL*))
  ((ATOM TOKEN)
    ((MOVE-LEFT-TOKEN)
      (BCK-SPACE$ 1) )
    ((NULL (CAR *ROW*)))
    (CTRL-E)
    (CTRL-B) )
  (BCK-SPACE$ 1)
  (SETQ TOKEN (UNPACK$ TOKEN))
  ((NULL (CDR TOKEN))
    ((OR (NULL (CDR *COL*)) (ATOM (CADR *COL*)))
      (SETQ *COL* (RDC$ (CADR *ROW*) *COL*)) )
    (RPLACA (CDR *COL*) (PACK$ (APPEND TOKEN (CADR *COL*))))
    (SETQ *COL* (RDC$ (CADR *ROW*) *COL*))
    (RPLACD *COL* (CDDR *COL*)) )
  (SETQ CHAR$ (CONS (CHOP$ TOKEN)))
  (RPLACA *COL* TOKEN)
  ((OR (NULL (CDR *COL*)) (ATOM (CADR *COL*)))
    (RPLACD *COL* (CONS CHAR$ (CDR *COL*))) )
  (RPLACA (CDR *COL*) (NCONC CHAR$ (CADR *COL*))) ))

(MOVD CTRL-S CTRL-H)                        % RETREAT CHAR %


(DEFUN CTRL-A (LAMBDA (
    TOKEN )                                 % RETREAT TOKEN %
  (LOOP
    (SETQ TOKEN (MOVE-LEFT-TOKEN))
    ((NOT (SPACE$ TOKEN)))
    (PRIN1 BACK) )
  ((NULL TOKEN)
    (LOOP
      ((NULL (CAR *ROW*)))
      (CTRL-E)
      ((CDR *COL*)
        (CTRL-B) ) ) )
  (BCK-SPACE$ (TOK-PRT-LEN TOKEN)) ))
```

```
EFUN CTRL-Q (LAMBDA ()
(PRIN1 CR)
(SETQ *COL* (CADR *ROW*))
(SPACES (CAR *COL*)) ))

EFUN CTRL-E (LAMBDA (
 NUM )
(NULL (CAR *ROW*))
 ((NULL *PRE-TEXT*))
 (CTRL-W)
 (CTRL-E) )
SETQ NUM (SPACES))
UP-LIN$ 1)
SPLICE-TOKEN *COL*)
SETQ *COL* (CAR *ROW*))
SETQ *ROW* (RDC$ *TEXT* *ROW*))
SPACES (CAR *COL*))
LOOP
 ((NULL (CDR *COL*)))
 ((NOT (LESSP (SPACES) NUM)))
 (CTRL-F) ) ))

FUN CTRL-X (LAMBDA (
 NUM )
(NULL (CDDR *ROW*))
 ((NULL *POST-TEXT*))
 (CTRL-Z)
 (CTRL-X) )
SETQ NUM (SPACES))
TRL-J)
LOOP
 ((NULL (CDR *COL*)))
 ((NOT (LESSP (SPACES) NUM)))
 (CTRL-F) ) ))

UN CTRL-W (LAMBDA ()
NULL *PRE-TEXT*))
((NULL (CDDR *ROW*))
  (CTRL-E) ) )
PLACD *TEXT* (CONS (POP *PRE-TEXT*) (CDR *TEXT*)))
USH (CHOP$ *TEXT*) *POST-TEXT*)
((NULL (CAR *ROW*))
  (POP *ROW*) ) )
ISP-TXT *TEXT* *ROW* *COL*) ))

UN CTRL-R (LAMBDA (
NUM )
NULL *PRE-TEXT*))
ETQ NUM (DIFFERENCE PAG-LEN$ 6))
OOP
(PUSH (CHOP$ *TEXT*) *POST-TEXT*)
(RPLACD *TEXT* (CONS (POP *PRE-TEXT*) (CDR *TEXT*)))
```

% MOVE TO BEGIN LINE %

% MOVE UP LINE %

% MOVE DOWN LINE %

% SCROLL DOWN %

% SCROLL DOWN PAGE %


```
    ((NULL *PRE-TEXT*))
    (SETQ NUM (SUB1 NUM))
    ((ZEROP NUM)) )
  (SETQ *ROW* *TEXT*)
  (SETQ *COL* (CADR *ROW*))
  (DISP-TXT *TEXT* *ROW* *COL*) ))

(DEFUN CTRL-Z (LAMBDA ()
  ((NULL *POST-TEXT*))
  ( ((NULL (CAR *ROW*))
      (CTRL-X) ) )
  ( ((EQ *ROW* (CDR *TEXT*))
      (SETQ *ROW* *TEXT*) ) )
  (HOME$)
  (TERPRI (SUB1 PAG-LEN$))
  (PRT-ROW (CAR *POST-TEXT*))
  (TERPRI)
  (PUSH (CADR *TEXT*) *PRE-TEXT*)
  (RPLACD *TEXT* (CDDR *TEXT*))
  (NCONC *TEXT* (CONS (POP *POST-TEXT*)))
  (MOVE-CUR *TEXT* *ROW* *COL*) ))

(DEFUN CTRL-C (LAMBDA (
    NUM )
  ((NULL *POST-TEXT*))
  (SETQ NUM (DIFFERENCE PAG-LEN$ 6))
  (LOOP
    (PUSH (CADR *TEXT*) *PRE-TEXT*)
    (RPLACD *TEXT* (CDDR *TEXT*))
    (NCONC *TEXT* (CONS (POP *POST-TEXT*)))
    ((NULL *POST-TEXT*))
    (SETQ NUM (SUB1 NUM))
    ((ZEROP NUM)) )
  (SETQ *ROW* *TEXT*)
  (SETQ *COL* (CADR *ROW*))
  (DISP-TXT *TEXT* *ROW* *COL*) ))

(DEFUN CTRL-N (LAMBDA (
    NUM )
  ((NUMBERP (CAR *COL*))
    (SETQ *COL* (CONS 0))
    (RPLACD *ROW* (CONS *COL* (CDR *ROW*)))
    ( ((GREATERP (LENGTH *TEXT*) PAG-LEN$)
        (PUSH (CHOP$ *TEXT*) *POST-TEXT*)
        (SETQ NUM (ROW-PRT-LEN (CAR *POST-TEXT*))) )
      (SETQ NUM 0) )
    (ROLL-DWN-ROW (CDR *ROW*) NUM) )
  (RPLACD (CDR *ROW*) (CONS (CONS 0) (CDDR *ROW*)))
  (CTRL-J)
  ( ((GREATERP (LENGTH *TEXT*) PAG-LEN$)
      (PUSH (CHOP$ *TEXT*) *POST-TEXT*)
      (SETQ NUM (ROW-PRT-LEN (CAR *POST-TEXT*))) )
    (SETQ NUM 0) )
```

% SCROLL UP %

% SCROLL UP PAGE %

% INSERT NEW LINE %

```
   (ROLL-DWN-ROW (CDR *ROW*) NUM) ))

(DEFUN CTRL-J (LAMBDA ()                          % NEWLINE TAB %
  ((NULL (CDDR *ROW*)))
  (SPLICE-TOKEN *COL*)
  (SETQ *ROW* (CDR *ROW*))
  (TERPRI)
  (SETQ *COL* (CADR *ROW*))
  (SPACES (CAR *COL*)) ))

(DEFUN CTRL-M (LAMBDA (*INSERT*)                  % CARRIAGE RETURN %
  ((EVAL *INSERT*)
    (CTRL-N)
    (CTRL-F) )
  ((NULL (CDDR *ROW*))
    ((NULL *POST-TEXT*)
      (SPLICE-TOKEN *COL*)
      (SETQ *ROW* (CDR *ROW*))
      (TERPRI)
      (SETQ *COL* (CONS 0))
      (RPLACD *ROW* (CONS *COL*))
      ((GREATERP (LENGTH *TEXT*) PAG-LEN$)
        (PUSH (CADR *TEXT*) *PRE-TEXT*)
        (RPLACD *TEXT* (CDDR *TEXT*)) ) )
    (CTRL-Z)
    (CTRL-M) )
  (SPLICE-TOKEN *COL*)
  (TERPRI)
  (SETQ *ROW* (CDR *ROW*))
  (SETQ *COL* (CADR *ROW*))
  (LOOP
    ((ZEROP (CAR *COL*)))
    (RPLACA *COL* (SUB1 (CAR *COL*)))
    (RPLACD *COL* (CONS (QUOTE " ") (CDR *COL*))) ) ))

EFUN CTRL-G (LAMBDA (                             % DELETE CHAR %
    TOKEN )
  (SETQ TOKEN (DELETE-CHAR))
  ((NULL TOKEN))
  (PRT-REST-LINE *ROW* *COL* 1) ))

DEFUN CTRL-T (LAMBDA (                            % DELETE TOKEN %
    TOKEN NUM)
  (SETQ TOKEN (DELETE-TOKEN))
  ((NOT TOKEN))
  (SETQ NUM (TOK-PRT-LEN TOKEN))
  (LOOP
    ((NOT (SPACE$ (NEXT-RIGHT-TOKEN))))
    (SETQ NUM (ADD1 NUM))
    (DELETE-CHAR) )
  (PRT-REST-LINE *ROW* *COL* NUM) ))

EFUN CTRL-Y (LAMBDA ()                            % DELETE LINE %
  ((NULL (CDDR *ROW*))
```



```
    (PRIN1 CR)
    (SPACES (ROW-PRT-LEN (CADR *ROW*)))
    (PRIN1 CR)
    (SETQ *COL* (CONS 0))
    (RPLACA (CDR *ROW*) *COL*) )
  ((AND (NULL *POST-TEXT*) *PRE-TEXT*)
    (RPLACD *TEXT* (CONS (POP *PRE-TEXT*) (CDR *TEXT*)))
    (RPLACD *ROW* (CDDR *ROW*))
    (SETQ *COL* (CADR *ROW*))
    (DISP-TXT *TEXT* *ROW* *COL*) )
  ( ((NULL *POST-TEXT*))
    (NCONC *TEXT* (CONS (POP *POST-TEXT*))) )
  (ROLL-UP-ROW (CDR *ROW*))
  (RPLACD *ROW* (CDDR *ROW*))
  (SETQ *COL* (CADR *ROW*))
  (SPACES (CAR *COL*)) ))


(DEFUN CTRL-O (LAMBDA (                           % FIND TOKEN %
    READCH )
  (HOME$)
  (SPACES (ROW-PRT-LEN (CADR *TEXT*)))
  (PRIN1 CR)
  (PRIN1 "FIND NAME? ")
  (SETQ READCH T)
  (SETQ *STRING* (CAR (RD-LIN$)))
  (HOME$)
  (SPACES LIN-LEN$)
  (HOME$)
  (PRT-ROW (CADR *TEXT*))
  ((NULL *STRING*)
    (MOVE-CUR *TEXT* *ROW* *COL*) )
  (CTRL-L) ))

(DEFUN CTRL-L (LAMBDA ()                          % SEARCH AGAIN %
  ((NULL *STRING*))
  (MOVE-RIGHT-TOKEN)
  (SRCH-TXT *STRING* *ROW* *COL*)
  (MOVE-CUR *TEXT* *ROW* *COL*) ))

(DEFUN SRCH-TXT (LAMBDA (TOKEN ROW COL)
  (LOOP
    ((SETQ COL (SRCH-ROW COL))
      (SETQ *COL* COL)
      (SETQ *ROW* ROW) )
    (POP ROW)
    ((NULL (CDR ROW))
      (SETQ ROW *TEXT*)
      (LOOP
        (SETQ COL (CADR ROW))
        ((SETQ COL (SRCH-ROW COL))
          (SETQ *COL* COL)
          (SETQ *ROW* ROW) )
        (POP ROW) ) )
    (SETQ COL (CADR ROW)) ) ))
```

```
DEFUN SRCH-ROW (LAMBDA (COL)
 (LOOP
   ((NULL (CDR COL)) NIL)
   ((EQ TOKEN (CAADR COL)) COL)
   (POP COL)
   ((EQ COL *COL*) COL) ) ))
```



```
;*****************  CURSOR CONTROL PRIMITIVES  ********************

(DEFUN INSERT-PRT (LAMBDA (CHAR$)
  ((ATOM (CAR *COL*))
    (RPLACD *COL* (CONS (CONS CHAR$) (CDR *COL*)))
    (POP *COL*) )
  (RPLACA *COL* (NCONC (UNPACK$ (CAR *COL*)) (CONS CHAR$))) ))

(DEFUN INSERT-DEL (LAMBDA (CHAR$)
  ((AND (NUMBERP (CAR *COL*)) (SPACE$ CHAR$))
    (RPLACA *COL* (ADD1 (CAR *COL*))) )
  (RPLACD *COL* (CONS CHAR$ (CDR *COL*)))
  (SPLICE-TOKEN *COL*)
  (POP *COL*) ))


(DEFUN DELETE-CHAR (LAMBDA (
    CHAR$ TOKEN )
  (SETQ TOKEN (NEXT-RIGHT-TOKEN))
  ((NULL TOKEN) NIL)
  ((ATOM TOKEN)
    (RPLACD *COL* (CDDR *COL*))
    TOKEN )
  (SETQ TOKEN (UNPACK$ TOKEN))
  (SETQ CHAR$ (CONS (POP TOKEN)))
  ((NULL TOKEN)
    (RPLACD *COL* (CDDR *COL*))
    CHAR$ )
  (RPLACA (CDR *COL*) TOKEN)
  CHAR$ ))

(DEFUN DELETE-TOKEN (LAMBDA (
    TOKEN )
  (SETQ TOKEN (NEXT-RIGHT-TOKEN))
  ((NULL TOKEN) NIL)
  (RPLACD *COL* (CDDR *COL*))
  TOKEN ))

(DEFUN MOVE-RIGHT-TOKEN (LAMBDA (
    TOKEN )
  (SETQ TOKEN (NEXT-RIGHT-TOKEN))
  ((NULL TOKEN) NIL)
  ((AND (NUMBERP (CAR *COL*)) (SPACE$ TOKEN))
    (RPLACA *COL* (ADD1 (CAR *COL*)))
    (RPLACD *COL* (CDDR *COL*))
    TOKEN )
  ((SPLICE-TOKEN *COL*)
    (POP *COL*)
    TOKEN )
  TOKEN ))

(DEFUN NEXT-RIGHT-TOKEN (LAMBDA ()
  ((NULL (CDR *COL*)) NIL)
  (CADR *COL*) ))
```

```
EFUN MOVE-LEFT-TOKEN (LAMBDA (
   TOKEN )
(SETQ TOKEN (CAR *COL*))
((NUMBERP TOKEN)
  ((ZEROP TOKEN)
    NIL )
  (RPLACA *COL* (SUB1 TOKEN))
  (RPLACD *COL* (CONS " " (CDR *COL*)))
  " " )
(SPLICE-TOKEN *COL*)
(SETQ *COL* (RDC$ (CADR *ROW*) *COL*))
TOKEN ))


EFUN SPLICE-TOKEN (LAMBDA (COL
  LST )
(ATOM (CAR COL))
  ((NOT (NUMBERP (CAR COL))))
  (POP LST)
  (LOOP
    ((OR (NULL LST) (NOT (SPACE$ (POP LST)))))
    (RPLACA COL (ADD1 (CAR COL)))
    (RPLACD COL LST) ) )
(OR (NULL (CDR COL)) (ATOM (CADR COL)))
 (RPLACA COL (PACK$ (CAR COL)))
 T )
RPLACA COL (PACK$ (APPEND (CAR COL) (CADR COL))))
RPLACD COL (CDDR COL))
IL ))
```



```
;***************** T E X T   P R I N T E R ********************;
  (DEFUN ROLL-UP-ROW (LAMBDA (LINE
      NUM LENGTH )
    (SETQ LENGTH (SUB1 (LENGTH LINE)))
    (SETQ NUM (REPL-ROW (CAR LINE) 0))
    (POP LINE)
    (LOOP
      ((NULL LINE))
      (SETQ NUM (REPL-ROW (POP LINE) NUM))
      (TERPRI) )
    (SPACES NUM)
    (UP-LIN$ LENGTH) ))

 (DEFUN ROLL-DWN-ROW (LAMBDA (ROW NUM)
   (TERPRI (SUB1 (LENGTH ROW)))
   (SETQ ROW (REVERSE ROW))
   (LOOP
     (SETQ NUM (REPL-ROW (POP ROW) NUM))
     ((NULL ROW))
     (UP-LIN$) )
   (PRIN1 CR) ))

 (DEFUN REPL-ROW (LAMBDA (COL LENGTH
     NUM )
   (PRIN1 CR)
   (PRT-ROW COL)
   (SETQ NUM (SPACES))
   (SPACES (DIFFERENCE LENGTH NUM))
   NUM ))

 (DEFUN DISP-TXT (LAMBDA (TEXT ROW COL)
   (TERPRI 47)                              % DISPLAY TEXT %
   (HOME$)                                  % CLEAR SCREEN %
   (PRT-TXT TEXT)                           % MOVE TO HOME$ %
   (MOVE-CUR TEXT ROW COL) ))               % PRINT TEXT %
                                            % RESTORE CURSOR %
 (DEFUN PRT-TXT (LAMBDA (TEXT)
   (MAPC (CDR TEXT) (QUOTE (LAMBDA (COL)    % PRINT TEXT %
     (PRT-ROW COL)
     (TERPRI) ))) ))

(DEFUN PRT-ROW (LAMBDA (COL)
  (SPACES (CAR COL))                        % PRINT A ROW OF TEXT %
  (MAPC (CDR COL) PRT-TOK) ))

(DEFUN MOVE-CUR (LAMBDA (TEXT ROW COL)
  (HOME$)
  (LOOP
    ((EQ ROW TEXT))                         % MOVE TO ROW %
    (TERPRI)
    (POP TEXT) )
  (SETQ TEXT (CADR ROW))
  (SPACES (CAR TEXT))
  LOOP

                                            % MOVE TO COL %
```

```
    ((EQ TEXT COL))
    (POP TEXT)
    (PRT-TOK (CAR TEXT)) ) ))

(DEFUN PRT-REST-LINE (LAMBDA (ROW COL NUM
    NUM0 )
  (SETQ NUM0 (SPACES))
  (MAPC (CDR COL) PRT-TOK)
  (SETQ NUM (DIFFERENCE (SPACES NUM) NUM0))
  ((LESSP NUM NUM0)
    (BCK-SPACE$ NUM) )
  (PRIN1 CR)
  (SETQ ROW (CADR ROW))
  (SPACES (CAR ROW))
  (LOOP
    ((EQ ROW COL))
    (SETQ ROW (CDR ROW))
    (PRT-TOK (CAR ROW)) ) ))

DEFUN PRT-TOK (LAMBDA (TOKEN)
 ((ATOM TOKEN)
    (PRIN1 TOKEN) )
 ((NULL (CDR TOKEN))
    ((NULL WRS)
      (PRIN1 (COND
        ((GET (CAR TOKEN) (QUOTE ALIAS)))
        ((CAR TOKEN)) )) )
    (PRIN2 (CAR TOKEN)) )
 ((NULL WRS)
    (MAPC TOKEN PRIN1) )
 (MAPC TOKEN PRIN2) ))

EFUN ROW-PRT-LEN (LAMBDA (COL
   NUM )
(SETQ NUM (POP COL))
(LOOP
   ((NULL COL) NUM)
   (SETQ NUM (PLUS NUM (TOK-PRT-LEN (POP COL)))) ) ))

EFUN TOK-PRT-LEN (LAMBDA (TOKEN)
((ATOM TOKEN) 1)
((NULL (CDR TOKEN))
   (LENGTH (CAR TOKEN)) )
(LENGTH TOKEN) ))
```



```
************** S-EXPRESSION TO TEXT TRANSLATOR **************%

(DEFUN DEF-TO-TXT (LAMBDA (VAR                          % TRANSLATE DEFINITIO
    TXT )
  (SETQ TXT (CONS))
  (LOOP
    (NCONC TXT (CDR
      (EXP-TO-TXT (GETD (CAR VAR)) (LIST (QUOTE DEFUN) (POP VAR))) ))
    ((NULL VAR) TXT)
    (NCONC TXT (CONS (CONS 0))) ) ))

(DEFUN SET-TO-TXT (LAMBDA (VAR                          % TRANSLATE VALUE %
    TXT )
  (SETQ TXT (CONS))
  (LOOP
    (NCONC TXT (CDR
      (EXP-TO-TXT (EVAL (CAR VAR)) (LIST (QUOTE SETQQ) (POP VAR))) ))
    ((NULL VAR) TXT)
    (NCONC TXT (CONS (CONS 0))) ) ))

(DEFUN PUT-TO-TXT (LAMBDA (VAR ATM)                     % TRANSLATE PROPERTY %
  (EXP-TO-TXT (GET VAR ATM) (LIST (QUOTE PUTQQ) VAR ATM)) ))

(DEFUN EXP-TO-TXT (LAMBDA (EXP LST
    *TEXT* *LINE* *LENGTH* TAB INDENT )
  (SETQ TAB 0)
  (SETQ INDENT 1)
  ( ((LESSP LIN-LEN$ 60))
    (SETQ INDENT 2) )
  (NEW-LIN TAB)
  (PUSH "(" *LINE*)
  (LOOP
    ((NULL LST))
    (EXP-TO-LIN (POP LST))
    (PUSH " " *LINE*) )
  (TSK-TO-TXT EXP TAB)
  (PUSH ")" *LINE*)
  (NEW-LIN TAB)
  (REVERSE *TEXT*) ))

(DEFUN TSK-TO-TXT (LAMBDA (TSK TAB)                     % TRANSLATE TASK %
  (SETQ TAB (PLUS TAB INDENT))
  ((ATOM TSK)
    (EXP-TO-LIN TSK) )
  ((ATOM (CAR TSK))
    ((MEMBER (CAR TSK) (QUOTE (LAMBDA NLAMBDA)))
      (PUSH "(" *LINE*)
      (EXP-TO-LIN (POP TSK))
      (PUSH " " *LINE*)
      (EXP-TO-LIN (POP TSK))
      (BDY-TO-TXT TSK TAB)
      (PUSH ")" *LINE*) )
    ((MEMBER (CAR TSK) (QUOTE (LOOP COND PROGN PROG1 AND OR)))
      (PUSH "(" *LINE*)
      (EXP-TO-LIN (POP TSK))
```

```
      (BDY-TO-TXT TSK TAB)
      (PUSH ")" *LINE*) )
    (EXP-TO-LIN TSK) )
  ((ATOM (CAAR TSK))
    (PUSH "(" *LINE*)
    (TSK-TO-TXT (POP TSK) TAB)
    ((AND TSK (ATOM (CAR TSK)) (NULL (CDR TSK)))
      (PUSH " " *LINE*)
      (EXP-TO-LIN (CAR TSK))
      (PUSH ")" *LINE*) )
    (BDY-TO-TXT TSK TAB)
    (PUSH ")" *LINE*) )
  (PUSH "(" *LINE*)
  (PUSH " " *LINE*)
  (BDY-TO-TXT1 TSK TAB)
  (PUSH ")" *LINE*) ))

DEFUN BDY-TO-TXT (LAMBDA (BDY TAB)
  ((NULL BDY))
  (NEW-LIN TAB)
  (BDY-TO-TXT1 BDY TAB) ))

DEFUN BDY-TO-TXT1 (LAMBDA (BDY TAB)
  (LOOP
    (TSK-TO-TXT (POP BDY) TAB)
    ((NULL BDY)
      (PUSH " " *LINE*) )
    (NEW-LIN TAB) ) ))

DEFUN EXP-TO-LIN (LAMBDA (EXP)
  ((ATOM EXP)
    (PUSH (CONS EXP) *LINE*) )
  (PUSH "(" *LINE*)
  (LOOP
    (EXP-TO-LIN (POP EXP))
    ((ATOM EXP)
      ((NULL EXP))
      (PUSH " " *LINE*)
      (PUSH "." *LINE*)
      (PUSH " " *LINE*)
      (PUSH (CONS EXP) *LINE*) )
    (PUSH " " *LINE*) )
  (PUSH ")" *LINE*) ))


EFUN NEW-LIN (LAMBDA (TAB)
(SETQ *LENGTH* TAB)
(SETQ *LINE* (REVERSE *LINE*))
(SETQ TAB (PLUS (CAR *LINE*) (TIMES 2 INDENT)))
CUT-LIN *LINE* (CAR *LINE*))
SETQ *LINE* (LIST *LENGTH*)) ))

FUN CUT-LIN (LAMBDA (LINE LENGTH)
PUSH *LINE* *TEXT*)
(NULL *LINE*))
```

% TRANSLATE BODY %

```
  (SETQ LENGTH (PLUS LENGTH (UNIT-LEN (CDR LINE))))
  (LOOP
    ((NOT (LESSP LENGTH LIN-LEN$))
      (SETQ *LINE* (CONS TAB (CDDR LINE)))
      (RPLACD LINE NIL)
      (CUT-LIN *LINE* (CAR *LINE*)) )
    (SETQ LINE (NXT-UNIT (CDR LINE)))
    ((NULL LINE))
    (SETQ LENGTH (PLUS LENGTH (ADD1 (UNIT-LEN (CDDR LINE))))) ) ))

(DEFUN UNIT-LEN (LAMBDA (LINE
    PRIN1 )
  ((OR (NULL LINE) (SPACE$ (CAR LINE))) 0)
  ((ATOM (CAR LINE))
    (ADD1 (UNIT-LEN (CDR LINE))) )
  ( ((NULL WRS)
      (SETQ PRIN1 T) ) )
  (PLUS (LENGTH (CAAR LINE)) (UNIT-LEN (CDR LINE))) ))

(DEFUN NXT-UNIT (LAMBDA (COL)
  (LOOP
    ((NULL (CDR COL)) NIL)
    ((SPACE$ (CADR COL)) COL)
    (SETQ COL (CDR COL)) ) ;)
```

```
************  TEXT TO S-EXPRESSION TRANSLATOR  ****************

DEFUN EVAL-TEXT (LAMBDA (TEXT
   COL TXT ERROR )
 (LOOP
   (SETQ TXT (NXT-TOK))
   ((NULL TXT) T)
   (SETQ TXT (TOK-TO-SEX TXT))
   ((NOT (NULL ERROR)) NIL)
   (EVAL TXT) ) ))

DEFUN TOK-TO-SEX (LAMBDA (TOKEN)
 ((ATOM TOKEN)
   ((EQ TOKEN "(")
     (LST-TO-SEX) )
   (PRIN1 "SYNTAX ERROR")
   (SETQ ERROR T) )
 ((NULL (CDR TOKEN))
   (CAR TOKEN) )
 (PACK TOKEN) ))

EFUN LST-TO-SEX (LAMBDA (
  TOKEN LST )
(SETQ TOKEN (NXT-TOK))
(LOOP
  ((EQ TOKEN ")")
    (REVERSE LST) )
  (PUSH (TOK-TO-SEX TOKEN) LST)
  ((EVAL ERROR))
  (SETQ TOKEN (NXT-TOK))
  ((EQ TOKEN ".")
    (SETQ TOKEN (TOK-TO-SEX (NXT-TOK)))
    ((EQ (NXT-TOK) ")")
      (NCONC (REVERSE LST) TOKEN) )
    (PRIN1 "SYNTAX ERROR")
    (SETQ ERROR T) ) ) ))

FUN NXT-TOK (LAMBDA (
 TOKEN )
LOOP
 ((NULL COL)
   (SETQ TEXT (CDR TEXT))
   ((NULL TEXT) NIL)
   (SETQ COL (CAR TEXT))
   (NXT-TOK) )
 (SETQ TOKEN (POP COL))
 ((NOT (OR (NUMBERP TOKEN) (SPACE$ TOKEN) (EQ TOKEN ",")))
   TOKEN ) ) ))
```



```
*********  T R A C E   D E B U G G I N G   P A C K A G E  **********

(DEFUN TRACE (LAMBDA (LST)
  (SETQ INDENT 0)
  (MAPC LST (QUOTE (LAMBDA (FUN BODY FUN#)
    (SETQ BODY (GETD FUN))
    (SETQ FUN# (PACK (LIST FUN #)))
    (MOVD FUN FUN#)
    ((MEMBER (CAR BODY) (QUOTE (LAMBDA NLAMBDA)))
      (PUTD FUN (LIST (CAR BODY) (CADR BODY)
        (LIST EVTRACE FUN (CADR BODY) FUN#) )) )
    (PRIN1 FUN)
    (PRINT " is not a LAMBDA defined function") ))) ))

(DEFUN UNTRACE (LAMBDA (LST)
  (MAPC LST (QUOTE (LAMBDA (FUN FUN#)
    (SETQ FUN# (PACK (LIST FUN #)))
    ((GETD FUN#)
      (MOVD FUN# FUN)
      (MOVD NIL FUN#) ) ))) ))

(DEFUN EVTRACE (NLAMBDA (FUN ARGS FUN#)
  (PRTARGS FUN ARGS)
  (PRTRSLT FUN (APPLY FUN# (MAKARGS ARGS))) ))

(DEFUN PRTARGS (LAMBDA (FUN ARGS)
  (SPACES INDENT)
  (SETQ INDENT (PLUS INDENT 1))
  (PRIN1 FUN)  (PRIN1 " [")
  ((NULL ARGS)
    (PRINT "]") )
  (LOOP
    ((ATOM ARGS)
      (SETQ ARGS (EVAL ARGS))
      (LOOP
        (PRIN1 (POP ARGS))
        ((ATOM ARGS))
        (PRIN1 ", ") ) )
    (PRIN1 (EVAL (POP ARGS)))
    ((NULL ARGS))
    (PRIN1 ", ") )
  (PRINT "]") ))

(DEFUN PRTRSLT (LAMBDA (FUN RSLT)
  (SETQ INDENT (DIFFERENCE INDENT 1))
  (SPACES INDENT)
  (PRIN1 FUN)  (PRIN1 " = ")  (PRINT RSLT)
  RSLT ))

(DEFUN MAKARGS (LAMBDA (ARGS)
  ((NULL ARGS) NIL)
  ((ATOM ARGS)
    (EVAL ARGS) )
  (CONS (EVAL (POP ARGS)) (MAKARGS ARGS)) ))
```

```
%File:  UTILITY.LIB                    08/11/80        The Soft Warehouse %

(PROG1 ""
  (PUTD DEFUN (QUOTE (NLAMBDA (FUNC DEF)
    (PUTD FUNC DEF)
    FUNC ))) )


% Function APPEND returns a list consisting of the elements of LST1
appended to LST2. %

(DEFUN APPEND (LAMBDA (LST1 LST2)
  ((NULL LST1) LST2)
  (CONS (CAR LST1) (APPEND (CDR LST1) LST2)) ))


% Function COPY returns a copy of its argument. %

(DEFUN COPY (LAMBDA (EXPN)
  ((ATOM EXPN) EXPN)
  (CONS (COPY (CAR EXPN)) (COPY (CDR EXPN))) ))


% Function UNION returns the union of LST1 and LST2. %

(DEFUN UNION (LAMBDA (LST1 LST2)
  ((NULL LST1) LST2)
  ((MEMBER (CAR LST1) LST2)
    (UNION (CDR LST1) LST2))
  (CONS (CAR LST1) (UNION (CDR LST1) LST2)) ))


% Function INTERSECTION returns the intersection of LST1 and LST2. %

(DEFUN INTERSECTION (LAMBDA (LST1 LST2)
  ((NULL LST1) NIL)
  ((MEMBER (CAR LST1) LST2)
    (CONS (CAR LST1) (INTERSECTION (CDR LST1) LST2)))
  (INTERSECTION (CDR LST1) LST2) ))


% Function SUBSET is a comparator returning T iff LST1 is a subset
of LST2. %

(DEFUN SUBSET (LAMBDA (LST1 LST2)
  ((NULL LST1))
  ((MEMBER (CAR LST1) LST2)
    (SUBSET (CDR LST1) LST2)) ))


% Function SUPERREVERSE returns a list of the elements of LST1
reversed at all levels. %

(DEFUN SUPERREVERSE (LAMBDA (LST1 LST2)
  ((NULL LST1) LST2)
  ((ATOM (CAR LST1))
    (SUPERREVERSE (CDR LST1) (CONS (CAR LST1) LST2)))
  (SUPERREVERSE (CDR LST1) (CONS (SUPERREVERSE (CAR LST1)) LST2)) ))


% Function REMBER is a constructor returning a list in which all
occurrences of ATM has been removed from LST. %

(DEFUN REMBER (LAMBDA (ATM LST)
  ((NULL LST) NIL)
  ((EQ ATM (CAR LST))
    (REMBER ATM (CDR LST)) )
  (CONS (CAR LST) (REMBER ATM (CDR LST))) ))


% Function SUBST is a constructor returning the expression resulting
from replacing all occurrences of OLD by NEW in EXPN. %

(DEFUN SUBST (LAMBDA (OLD NEW EXPN)
  ((EQUAL OLD EXPN) NEW)
  ((ATOM EXPN) EXPN)
  (CONS (SUBST OLD NEW (CAR EXPN)) (SUBST OLD NEW (CDR EXPN))) ))


% Function NTH is a selector which returns the result of
removing the first NUM elements from the list LST. %

(DEFUN NTH (LAMBDA (LST NUM)
  ((NOT (PLUSP NUM))
    LST )
  (LOOP
    (SETQ LST (CDR LST))
    (SETQ NUM (SUB1 NUM))
    ((ZEROP NUM)
      LST ) ) ))


% Function GENSYM is a constructor which returns a new name
of the form Gxxxx where xxxx is a number incremented each
time GENSYM is called. %

(SETQ GENSYM 0)

(DEFUN GENSYM (LAMBDA (NUM LST)
  (SETQ NUM (DIFFERENCE 4 (LENGTH GENSYM)))
  (LOOP
    ((ZEROP NUM))
    (PUSH 0 LST)
    (SETQ NUM (SUB1 NUM)) )
  (PROG1
    (PACK (NCONC (CONS (QUOTE G) LST) (LIST GENSYM)))
    (SETQ GENSYM (ADD1 GENSYM)) ) ))


% Function MAX returns the greater of two numbers. %
```

```
(DEFUN MAX (LAMBDA (M N)
  ((GREATERP M N) M)
  N ))

(DEFUN ADD1 (LAMBDA (NUM)
  (PLUS NUM 1) ))

(DEFUN SUB1 (LAMBDA (NUM)
  (DIFFERENCE NUM 1) ))

% Function DEPTH returns the maximum depth of an expression. %

(DEFUN DEPTH (LAMBDA (EXPN)
  ((ATOM EXPN) 0)
  (ADD1 (MAX (DEPTH (CAR EXPN)) (DEPTH (CDR EXPN)))) ))

% Function ABS returns the absolute value of NUM. %

(DEFUN ABS (LAMBDA (NUM)
  ((MINUSP NUM)
    (MINUS NUM) )
  NUM ))

% Function FACTORIAL returns NUM factorial. %

(DEFUN FACTORIAL (LAMBDA (NUM
    ANS )
  ((NOT (GREATERP NUM -1)) NIL)
  (SETQ ANS 1)
  (LOOP
    ((EQ NUM 0) ANS)
    (SETQ ANS (TIMES NUM ANS))
    (SETQ NUM (SUB1 NUM)) ) ))

% Function POWER returns NUM1 raised to the NUM2 power.  NUM3
is a local or temporary variable for the function POWER. %

(DEFUN POWER (LAMBDA (NUM1 NUM2
    NUM3 )
  (SETQ NUM3 1)
  (LOOP
    (SETQ NUM2 (DIVIDE NUM2 2))
    ( ((EQ (CDR NUM2) 1)
        (SETQ NUM3 (TIMES NUM1 NUM3)) ) )
    (SETQ NUM2 (CAR NUM2))
    ((ZEROP NUM2) NUM3)
    (SETQ NUM1 (TIMES NUM1 NUM1)) ) ))
```

```
% Function GCD returns the Greatest Common Divisor of NUM1 and
NUM2. %

(DEFUN GCD (LAMBDA (NUM1 NUM2
    NUM3 )
  (LOOP
    ((ZEROP NUM2) NUM1)
    (SETQ NUM3 NUM2)
    (SETQ NUM2 (REMAINDER NUM1 NUM2))
    (SETQ NUM1 NUM3) ) ))

% The following are examples of Mapping Functions equivalent to
the definitions found in LISP tutorials. %

(DEFUN MAPC (LAMBDA (LST FUN)
  (LOOP
    ((NULL LST) NIL)
    (FUN (POP LST)) ) ))

(DEFUN MAPCAR (LAMBDA (LST FUN)
  ((NULL LST) NIL)
  (CONS (FUN (CAR LST)) (MAPCAR (CDR LST) FUN)) ))

(DEFUN MAPLIST (LAMBDA (LST FUN)
  ((NULL LST) NIL)
  (CONS (FUN LST) (MAPLIST (CDR LST) FUN)) ))

(RDS)          % DELETE THIS LINE IF YOU WANT AN EVALQUOTE DRIVER %

% Function DRIVER is originally defined in machine language to be
an EVAL-LISP executive driver.  However, it may be redefined as
desired.  The following is an EVAL-QUOTE driver, which must be
used to load the remainder of the functions in this file. %

(DEFUN DRIVER (LAMBDA (RDS WRS)
  (LOOP
    (TERPRI)
    (PRIN1 (QUOTE "> "))
    (PRINT (APPLY (READ) (READ) (TERPRI))) ) ))

(DRIVER)
```