Microsoft® muLISP Artificial Intelligence Development System

Copyright Notice

Copyright ©, 1979, 80, 81, 82 and 83 by **The Soft Warehouse.** All Rights Reserved Worldwide.

No part of this manual may be reproduced, transmitted, transcribed, stored in a retrieval system, or translated into any human or computer language, in any form or by any means, electronic, mechanical, magnetic, optical, chemical, manual, or otherwise, without the express written permission of The Soft Warehouse, P.O. Box 11174, Honolulu, Hawaii 96828, U.S.A.

LIMITED WARRANTY

MICROSOFT CORPORATION and THE SOFT WAREHOUSE shall have no liability or responsibility to purchaser or any other person or entity with respect to any liability, loss or damage caused or alleged to be caused directly or indirectly by this product, including but not limited to any interruption of service, loss of business or anticipatory profits or consequential damages resulting from the use or operation of this product. This product will be exchanged within twelve months from date of purchase if defective in manufacture, labeling or packaging, but except for such replacement the sale or subsequent use of this program is without warranty or liability. Further, The Soft Warehouse reserves the right to revise this publication and to make changes from time to time in the contents hereof without obligation of Microsoft Corporation and The Soft Warehouse to notify any person or organization of such revision or changes.

THE ABOVE IS A LIMITED WARRANTY AND THE ONLY WARRANTY MADE BY MICROSOFT CORPORATION. ANY AND ALL WARRANTIES FOR MERCHANTABILITY AND/OR FITNESS FOR A PARTICULAR PURPOSE ARE EXPRESSLY EXCLUDED.

If you have comments about this documentation or the enclosed software, complete the Software Problem Report at the back of this manual and return it to Microsoft.

muLISP and muSTAR are trademarks of the Soft Warehouse.
MS-DOS is a registered trademark of Microsoft Corporation.
IBM PC is a registered trademark of International Business Machines Corporation.
CP/M is a registered trademark of Digital Research.
Apple II is a registered trademark of Apple Computer Inc.

Document Number: 8719-411-02
Part Number: 018-000-005

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 **Artificial Intelligence** (AI) community. The language and its many derivatives continue to monopolize all serious work in such diverse fields as robotics, expert systems, 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 structure in LISP is **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.

muLISP-83 represents the culmination of over six years of effort into designing a high performance, general purpose LISP programming system. It was developed on microcomputers for microcomputers. muLISP has been implemented for the 8080 and Z80 microprocessors, the 8086 and 8088 microprocessors, and the 6502 microprocessor.

muLISP was developed and implemented by Albert D. Rich, David R. Stoutemyer, and Roy Feldman. The design was influenced by the suggestions of Martin Griss, Peter Deutsch, Timothy Robinson, Richard Zipple, Steve Messick, Thomas Yonkman, Gary Rader and many others. The clarity and completeness of this manual is due in large part to the patience and persistence of Joan H. Rich.

Albert D. Rich
Applied Logician

TABLE OF CONTENTS

Title Page i
Copyright Notice ii
Preface iii
Table of Contents iv

1. INTRODUCTION

1.1 Major Features 1-1
1.2 muLISP System Contents 1-3
1.3 System Requirements 1-4
1.4 Master Diskette Backup 1-5
1.5 muLISP on the IBM Personal Computer 1-6
1.6 muLISP on CP/M-80 and Apple II SoftCard Computers . . . 1-7

2. BASIC OPERATING PROCEDURES

2.1 System Load and Run 2-1
2.2 The Executive Driver Loop 2-2
2.3 Line Editing Procedures 2-3
2.4 Programming Fundamentals 2-4
2.5 Reading Source Files 2-6
2.6 Environment SYS Files 2-7
2.7 The Debug Package 2-8

3. LEARNING muLISP

3.1 Demonstration Programs 3-1
3.2 Tutorial System 3-3
3.3 Utility Library Files 3-4

4. SYSTEM DESCRIPTION

4.1 Data Structures 4-1
4.2 Memory Management 4-4
4.3 The Break Package 4-6
4.4 Interrupting Execution 4-7
4.5 Error Messages 4-8

5. FUNCTIONS & CONTROL VARIABLES

5.1 Selector Functions 5-2
5.2 Constructor Functions 5-6
5.3 Modifier Functions 5-8
5.4 Recognizer Functions 5-11
5.5 Comparator Functions 5-14
5.6 Logical Functions 5-17

5.7 Assignment Functions 5-18
5.8 Property Functions 5-20
5.9 Flag Functions 5-22
5.10 Definition Functions 5-23
5.11 String Functions 5-25
5.12 Numerical Functions 5-29
5.13 Reader Functions, Control Variables, and Macros 5-32
5.14 Printer Functions and Control Variables 5-40
5.15 Evaluation Functions 5-46
5.16 Memory Management Functions 5-52
5.17 Environment Functions 5-54
5.18 User Interface Functions 5-56
5.19 Hardware Interface Functions 5-61
5.20 Special Functions 5-64

6. THE muSTAR AI DEVELOPMENT SYSTEM

6.1 Main Menu Commands 6-2
6.2 Text Editing 6-6
6.3 Console Customization 6-12
6.4 Theory of Operation 6-14

7. MACHINE LANGUAGE FUNCTIONS

7.1 8086 Machine Language Routines 7-2
7.2 8080 Machine Language Routines 7-5

8. LISP AND ARTIFICIAL INTELLIGENCE REFERENCES

8.1 Professional Societies 8-1
8.2 Periodicals 8-1
8.3 Books 8-2

9. GLOSSARY OF COMPUTER TERMINOLOGY

APPENDICES

Appendix A: IBM PC Extended Function Keys A-1
Appendix B: 8086 Base Page A-2
Appendix C: 8086 Machine Language Routine A-6
Appendix D: 8080 Base Page A-7
Appendix E: 8080 Machine Language Routine A-10

INDEXES

Concept Index I-1
Function and Variable Name Index I-5

1. INTRODUCTION

Congratulations on your purchase of the **muLISP-83 Artificial Intelligence Development System.** This system is a sophisticated software package for microcomputers. It has been designed to be capable of supporting a wide range of serious **artificial intelligence** (AI) research efforts. As might be expected, some degree of study and patience is required to properly use muLISP as a development tool for AI software projects.

The primary purpose of this manual is to serve as a reference guide. The best way to learn LISP is through "hands on" experience. The interactive muLISP Tutorial System, described in Chapter 3, will greatly aid the process. In addition, the LISP and AI Bibliography in Chapter 7 of this manual lists several good textbooks.

1.1 MAJOR FEATURES

1. Over 100 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.

2. Infinite precision integer arithmetic, expressed in any desired radix base from 2 through 36, is supported by a complete set of numerical functions.

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

4. Dynamic reallocation of data space boundaries occurs automatically to most efficiently use all available memory resources.

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 iterative programming.

6. A structured, non-local exit mechanism is provided by the functions CATCH and THROW. This greatly simplifies the handling of error and/or exceptional conditions in user programs.

7. muLISP functions can be defined as being either eval or no-eval, and as either spread or no-spread.

8. In addition to muLISP's interactive environment, program debugging is facilitated by the muSTAR resident display-oriented editor and a trace package.

9. Sequential and random access, streamed file I/O are fully

supported. Also, memory image files can be created to save muLISP environments so they can be reloaded at a later time.

10. The relatively small size of muLISP leaves the remainder of the computer's memory available for user data structures. A minimum system will run in as little as 48K bytes of computer memory (64K is required for 8088 and 8086 based systems).

11. 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.

12. 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.

13. Numerous I/O control variables have been included to handle such issues as upper/lower case conversion, console line edit mode, and the printing of quoted strings.

14. Text and language processing capabilities have been enhanced by the addition of efficient string functions. A hashing technique is used to rapidly intern new print name strings. Names that are unnecessary are automatically removed from the object list when a garbage collection occurs.

15. The writing of customized parse routines is facilitated by muLISP's table driven scanner, and read and splice macros.

16. When necessary for special applications, muLISP functions can be linked to user-defined, machine language subroutines.

17. Application programmers can sell programs written in muLISP without requiring the end-users to pay for a full muLISP system. Instead, a small royalty is paid for the use of the muLISP runtime system. To obtain a muLISP Runtime License Agreement, contact The Soft Warehouse.

The Soft Warehouse Newsletter, published by the authors of muLISP, provides users with the latest information on muLISP and muMATH, the symbolic math system for microcomputers. The Newsletter contains user contributed application programs, announcements of new releases or revisions to the software, and a forum for the growing community of muLISP and muMATH users. Registered muLISP owners can obtain a free copy of the Newsletter by writing

The Soft Warehouse
P.O. Box 11174
Honolulu, Hawaii 96828-0174
U.S.A.

Please include the serial number on your muLISP master diskette.

1.2 muLISP SYSTEM CONTENTS

The muLISP-83 Artificial Intelligence Development System (muLISP AIDS) consists of the following items:

1. The muLISP Reference Manual.

2. muLISP pseudo-code compiler and interpreter:

   MULISP.COM an executable, binary file.

3. muSTAR editor and debugging files:

   MUSTAR.SYS the muSTAR memory image SYS file.

   MUSTAR.LIB the muSTAR screen-oriented editor source file.

   CURSOR.* muSTAR console customization source files.

   DEBUG.LIB the muLISP debug package source file.

4. Utility library files:

   INTERLIS.LIB Interlisp utility functions.

   MACLISP.LIB Maclisp utility functions.

   GRAPHICS.LIB point, line, and turtle graphics routines.

5. Tutorial system files:

   LESSONS.LIB the basic lesson driver source file.

   MULISPn.LES muLISP on-line lesson files.

6. Demonstration program files:

   ANIMAL.LIB an animal guessing game.

   ANIMAL.MEM a prebuilt data tree for the animal game.

   DOCTOR.LIB an imitation of a psychologist.

   SCRIPT.LIB a text data file for DOCTOR.LIB.

   HANOI.LIB the three peg, Tower of Hanoi problem.

   METAMIND.LIB a secret code deciphering game.

1.3 SYSTEM REQUIREMENTS

muLISP-83 has been implemented on a wide variety of microcomputers. This section lists the minimum hardware and the operating system software required to run muLISP for each of the computers for which it has been implemented.

1.3.1 IBM Personal Computer

1. An IBM Personal Computer with keyboard and display monitor.
2. At least 64K (64 kilobytes) of RAM memory (the more the better).
3. At least one standard IBM PC, 5 1/4 inch floppy disk drive.
4. The diskette containing the IBM DOS software (either version 1.1 or 2.0) that is provided with the computer.

1.3.2 MS-DOS™ Computers

1. An 8086 or 8088 based microcomputer running Microsoft's MS-DOS operating system.
2. At least 64K (64 kilobytes) of RAM memory (the more the better).
3. At least one standard floppy disk drive capable of reading the muLISP master diskette (contact your muLISP dealer to determine if they have muLISP diskettes formatted for your disk drive).

1.3.3 Apple II and Apple IIe SoftCard™ Computers

1. An Apple II or Apple IIe computer with a display monitor.
2. At least 48K (48 kilobytes) of RAM memory (a 16K RAMcard or Language Card will allow larger programs to be run).
3. At least one standard Apple II, 5 1/4 inch floppy disk drive.
4. The CP/M-80 operating system diskette that is provided with the Z80 SoftCard.

1.3.4 CP/M-80™ Computers

1. An 8080, 8085, or Z80 based microcomputer running the CP/M-80 Disk Operating System.
2. At least 48K (48 kilobytes) of RAM memory.
3. At least one standard floppy disk drive capable of reading the muLISP master diskette (contact your muLISP dealers to determine if they have muLISP diskettes formatted for your disk drive).

1.4 MASTER DISKETTE BACKUP

As soon as possible after receipt of muLISP, make a copy of the master diskette for use as a **working copy.** This is a safeguard against the loss, damage or accidental erasure of the master diskette. Once copied, the master diskette should be stored in a safe, cool place separate from the working diskettes.

As stated in the license agreement, copies are for use on at most 5 computers at one location. Each copy of muLISP must have a label bearing the following notice:

> **(c) 1982 The Soft Warehouse. Permission for disclosure or duplication or other use hereof may be granted only by The Soft Warehouse.**

In addition to the copyright notice, it is a good idea to put on the label the version number and date, and the serial number of your muLISP system. Include these numbers in any inquiries you make concerning muLISP or muSTAR.

If your system has a diskette copy utility, use it to make duplicate copies of the master muLISP diskette. Your computer's operating system manuals should describe how to use the utility. For example, IBM PC users can use the **DISKCOPY** utility described in the IBM DOS Reference Manual.

For multiple drive systems another, though slower, way to make a copy is to perform the following steps:

1. Turn on the computer and <u>bootstrap</u> the disk operating system using a system diskette.

2. Insert a formatted diskette in drive B.

3. Insert the master muLISP diskette in drive A.

4. MS-DOS and IBM PC users enter the operating system command

   COPY A:*.* B:

   and press the <RETURN> key. CP/M users ensure there is a copy of the PIP utility on drive B and enter the operating system command

   B:PIP B:=A:*.*

   and press the <RETURN> key.

5. When all the files have been copied, the system will redisplay the operating system prompt.

6. Remove the diskette from drive B and place The Soft Warehouse copyright notice on it.

1.5 muLISP ON THE IBM PERSONAL COMPUTER

This section will be of interest only to those users running muLISP on the IBM Personal Computer or on an IBM "look-alike" computer. These computers have some nonstandard features and capabilities that effect the operation of muLISP.

1.5.1 Keyboard Notation

Some of the keys on the IBM PC keyboard are marked with descriptive symbols instead of spelled out names. Throughout this manual these keys will be referred to using names enclosed in angle brackets. The following table gives the names and the corresponding symbol used on the keyboard:

<RETURN>	↵
<LINEFEED>	CTRL ↵
<BACKSPACE>	←
<TAB>	→|
<SHIFT>	shift key
<SPACE>	space bar

1.5.2 Extended Function Keys

The IBM PC keyboard has a set of extended function keys. Examples include the ten function keys located at the left end of the keyboard and the cursor control keys located at the right end of the keyboard. See the IBM PC's Guide to Operations manual and the BASIC Language Programming manual for details.

If you are running the PC under IBM DOS Version 1.10 or later and console input is in the raw input mode (see the Reader Functions section of Chapter 5), muLISP programs can recognize the pressing of an extended function key by the user. Otherwise, the use of these keys will be indistinguishable from the other standard ASCII keys.

The IBM PC version of muSTAR has been enhanced to make use of some of the extended function keys for cursor control as follows:

←	left a character	CTRL ←	left a word
→	right a character	CTRL →	right a word
↑	up a line	Pg Up	up a screenful
↓	down a line	Pg Dn	down a screenful
Ins	insert mode toggle	Del	delete character

Receipt of an (ASCII 255) character by a muLISP program from the console indicates the use of an extended function key. The next character received from the console corresponds to a particular extended function key. See the Extended Function Key table given in Appendix A to this manual.

1.6 muLISP ON CP/M-80 AND APPLE II SOFTCARD COMPUTERS

The 64K address space of the 8080 and Z80 microprocessors imposes a very severe restriction on the maximum size of LISP programs. To ameliorate this problem for the CP/M-80 version of muLISP-83, two slightly different copies of the muLISP interpreter have been included on the Master Diskette: a full feature version and a smaller, limited feature version.

The file MULISP.COM contains all the primitive LISP functions and features as described in the muLISP-83 Reference Manual. The only exception is the GCD (greatest common divisor) function. If necssary, GCD can be defined as follows:

```
(PUTD 'GCD '(LAMBDA (N M)
  ((AND (NUMBERP N) (NUMBERP M))
    (LOOP
      ((ZEROP M)
        ((PLUSP N) N)
        (MINUS N) )
      (SETQ M (REMAINDER N (SETQ N M))) ) )
  (NONNUMERIC 'GCD (LIST N M)) ))
```

The file ML.COM is the limited feature version of the muLISP interpreter. By pruning out some of the less used LISP functions, ML.COM provides more than 1K of additional memory for user defined functions and data. muLISP-80 users will note that it contains essentially all the primitively defined functions provided in the '80 version of muLISP. The following is a list of the functions and features that have been left out of ML.COM:

1. List processing functions: NTH, LCONC (These functions can be defined as shown in Chapter 5 of this manual).

2. Numerical functions: EVEN, GCD (These functions can be defined as shown in Chapter 5 of this manual).

3. String functions: SUBSTRING, FINDSTRING (These functions can be defined using the functions PACK and UNPACK).

4. Random file I/O functions: READPTR, WRITEPTR.

5. Read table function: CHARTYPE (The default read table, located in the 8080 Base Page, can be modified manually).

6. Read and splice macros (The quote mark can still be used as an abbreviation for QUOTE as described in Section 5.15 of this manual).

7. Hardware interface functions: PORTIO, REGISTER, CALL.

8. Apple II SoftCard graphics functions: TEXT, GRAPHICS, PLOT, PADDLE, BEEP.

9. Comment function: COMMENT (This function can be defined as shown in Section 5.15 of this manual).

10. Garbage collection statistics when RECLAIM is NIL (See Section 5.16 of this manual).

11. The "Are you sure (Y/N)?" confirmation query presented after typing Ctrl-C or taking the "Restart" or "System" break option (See Section 4.3 of this manual).

Apple II SoftCard users, who have not already done so, may want to obtain the recently released software upgrade kit from Microsoft or your SoftCard dealer. The upgrade provides a 60K CP/M system rather than a 56K system allowing 4K more memory for running muLISP.

2. BASIC OPERATING PROCEDURES

This chapter provides the information necessary to load and run muLISP. The remainder of the manual provides a detailed explanation of muLISP data structures, the memory management scheme, and the 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 muLISP or the LISP programming language. The best way to learn muLISP is by exploratory use of the system using the muLISP Tutorial System (see Chapter 3). Several good references on the language are given in the bibliography to this manual.

2.1 SYSTEM LOAD AND RUN

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 the logged drive to the one with the diskette containing a copy of MULISP.COM. Following the operating system prompt, enter the command

MULISP

followed by a <RETURN>. After a few seconds of load time, the **header message**

muLISP-83 4.nn (mm/dd/yy)
xxxxxxxxxxx version
Copyright (C) 1982 The SOFT WAREHOUSE
Licensed by MICROSOFT Corp.

is displayed on the screen. Appropriate numbers will appear for the version number (4.nn), month (mm), day (dd), and year (yy). The **version number** and **version date**, along with your software **serial number**, should be included in all inquiries concerning the system. The xxxxxxxxxxx in the header message indicates the type of computer system on which this version of muLISP can be run.

Below the header message, the muLISP **prompt** will be displayed. The default prompt consists of a dollar sign ($) character followed by a space and the console cursor. The prompt indicates the system is ready to accept console input.

2.2 THE EXECUTIVE DRIVER LOOP

After a complete expression followed by a <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 the SYSTEM command is issued or <CTRL-C> is entered. In either case muLISP is terminated and control returns to the host operating system.

The default muLISP driver loop is an **eval-LISP** executive. Following the prompt, expressions written in list and/or dot notation can be entered from the console. Multi-line expressions can be entered since an expression is not evaluated until all parentheses are balanced. After the final <RETURN> is entered, the expression is read by the function READ, evaluated by EVAL, and the result displayed by PRINT. The following is a printout of a sample muLISP dialogue demonstrating the basic driver loop:

```
$ 'DOG
DOG

$ (PLUS 5 -2 4)
7

$ (EQUAL 'DOG 'CAT)
NIL

$ (MEMBER 'DOG '(CAT COW DOG PIG))
(DOG PIG)
```

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 (see Chapter 5).

2.3 LINE EDITING PROCEDURES

Since muLISP uses the line input routines of the host operating system, all the line editing features of the system are inherited by muLISP. These should include a means of deleting characters by backspacing the cursor. Usually backspacing is done by pressing the key marked <BACKSPACE>, <DELETE> or <RUB> on the keyboard. Entering <CTRL-H> (i.e. pressing the "H" key while holding down the key marked CTRL) is another means of backspacing.

A means of deleting the current line of input should also be provided. Some operating systems even have a means of restoring the previous line of input, making it available for reediting. See your computer's operating system reference manuals for a full description of all the line editing facilities available.

Only errors on the current line can be corrected, since there is no way to modify a line once <RETURN> has been entered. If an incorrect line of input has caused a "runaway" program, it can be interrupted as described in the Keyboard Interrupt section of Chapter 4.

muLISP output to the console can be "frozen" and then restarted using <CTRL-S> as a toggle. This is useful to prevent the scrolling of text off the screen before it can be read.

Most operating systems provide a means of echoing to a printer the output displayed on the console. This is a convenient means of making a "hardcopy" record of a muLISP session. Often <CTRL-P> is used as a toggle switch to turn printer echoing ON and OFF. IBM PC users use <CTRL-PrtSc> for the toggle. See the operating system manual for details.

2.4 PROGRAMMING FUNDAMENTALS

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

```
$ (PUTD 'FACTORIAL '(LAMBDA (N)
    (COND
      ((ZEROP N) 1)
      (T (TIMES N (FACTORIAL (DIFFERENCE N 1)))) ) ))

$ (FACTORIAL 5)
120
```

This 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.

As study of this manual and the muLISP source files supplied with the system shows, muLISP-83 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 was noncommittal. As discussed in later chapters, the extensions significantly increase program 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 a nonNIL 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 the explicit use of COND within function definitions.

For an example illustrating the extended capabilities of muLISP, compare the following definition of FACTORIAL with that given previously:

```
(PUTD 'FACTORIAL '(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 **nonrecursive loops.** This is achieved without the use of the totally unstructured GO feature.

2. As in the Interlisp dialect of LISP, 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-83 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 the Evaluation Functions section of Chapter 5 for a detailed explanation of the muLISP evaluation mechanism. (Note: The utility library file INTERLIS.LIB provides an interpreter for PROGs.)

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

```
(PUTD 'FACTORIAL '(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 '(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 nonrecursive definition (Note: It is not a workable muLISP function definition). The second is the equivalent muLISP-83 definition, also nonrecursive. You can decide for yourself which definition is more elegant and structured. It is interesting to note that the muLISP definition requires only 29 nodes; the conventional version requires 38. This is not an insignificant ratio for a LISP system running in a limited address space.

2.5 READING SOURCE FILES

muLISP **source files** are ASCII text files containing program and/or character data for muLISP. Several sample source files are provided on the muLISP Master Diskette. You can generate your own source files using either the muSTAR editor (see Chapter 6) or an external text editor appropriate for your computer's operating system.

Normally, character input to muLISP comes from the console. However, a source file can be selected as the source of input using an RDS (ReaD Select) command. Following the muLISP dollar sign prompt, an RDS command of the form

(RDS 'name 'type 'drive)

reads in the source file <name>.<type> on disk drive <drive>. <drive> is an optional argument that defaults to the currently logged drive. Note that the file name and type are separated by a space rather than a period, and that no colon is required following the drive.

For example, to run the animal guessing game program, ANIMAL.LIB, issue the RDS command:

(RDS 'ANIMAL 'LIB)

This command causes the animal program to be read in from the logged drive. After ANIMAL.LIB has been read in, the program will provide an explanation of how to play the animal game. The other source files on the master diskette can be read in in a similar fashion.

If you write your own source files using an external editor, normally you will want the last line of the file to be the RDS command:

(RDS)

This causes the input source for muLISP to return to the console after the file has been read in. Alternatively, you can automatically "chain-load" source files by using the RDS function to switch the input source to another file, instead of returning control to the console. See the Reader Functions section of Chapter 5 for a more detailed discussion of the RDS function and control variable.

2.6 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 command

(SAVE 'WHALE)

will generate a **SYS file** named WHALE.SYS on the logged drive. If the SAVE command is successful, T is returned; if not, NIL is returned.

The environment saved using a SAVE command can be restored at any later time by issuing a LOAD command. The command

(LOAD 'WHALE 'B)

loads WHALE.SYS from drive B. The environment present before the load is replaced with that stored in the SYS file.

Alternatively, SYS files can be loaded along with muLISP from the operating system level. For example, the following operating system command will load the SYS file WHALE.SYS from drive B after loading muLISP from the logged drive:

MULISP B:WHALE

When using this second method, if a SYS file of the given name does not exist, the following warning will be displayed immediately after the muLISP header message:

SYS file NOT found

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 reallocated according to the computer's current memory size. This means that the current memory size does <u>not</u> 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.

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 the identical program in each time can be tedious. If a SYS file is generated containing the source, the application program can be loaded at any time both quickly and conveniently.

2.7 THE DEBUG PACKAGE

The muLISP debug package is a powerful set of tools designed to help you develop and debug muLISP programs. It includes a trace facility, a break-point facility, and a statistics gathering facility. The function tracing facility displays the progress of function evaluation. The break-point facility stops function evaluation at critical points and lets you examine the environment. The statistics facility helps you identify the speed or depth critical portions of a large program.

The muLISP debugging facility is distributed as the source file DEBUG.LIB. It can be read into muLISP from the currently logged drive by the command

```
(RDS 'DEBUG 'LIB)
```

Throughout our discussion of the debug package, we will assume that the functions APP and REV have been defined as follows:

```
(PUTD 'APP '(LAMBDA (LST1 LST2)
    ((NULL LST1) LST2)
    (CONS (CAR LST1) (APP (CDR LST1) LST2)) ))

(PUTD 'REV '(LAMBDA (LST1)
    ((NULL LST1) NIL)
    (APP (REV (CDR LST1)) (LIST (CAR LST1))) ))
```

APP and REV are functionally equivalent to the primitive muLISP functions APPEND and REVERSE respectively. Since APP and REV are less efficient than their machine language counterparts, we have given them different names.

2.7.1 The Trace Facility

When a traced function is called, the function name and the function's formal and actual arguments are displayed. When a traced function returns, the function name and its value are displayed. You can constrain functions to display trace information only after a given number of function calls, or only between a minimum and a maximum level of recursive nesting, or only within the dynamic scope of specified functions.

The no-spread function TRACE is used to trace functions. It is called with one or more functions to be traced. For example, the command

```
(TRACE 'APP 'REV)
```

traces APP and REV. If you now enter the command

```
(REV '(A B))
```

the following trace will be displayed:

```
 1| REV [LST1: (A B)]
 2|  REV [LST1: (B)]
 3|   REV [LST1: NIL]
 3|   REV = NIL
 3|   APP [LST1: NIL, LST2: (B)]
 3|   APP = (B)
 2|  REV = (B)
 2|  APP [LST1: (B), LST2: (A)]
 3|   APP [LST1: NIL, LST2: (A)]
 3|   APP = (A)
 2|  APP = (B A)
 1| REV = (B A)
(B A)
```

Note how indentation is used to graphically indicate the nesting of function calls. The numbers down the left margin also give the level of nesting.

If trace output is scrolling too rapidly to read, use <CTRL-S> to temporarily freeze the display. Alternatively, the commands

```
(SETQ ECHO T)
(WRS 'filename 'filetype 'drive)
```

sends the entire trace output to the file <filename>.<filetype> on <drive> while echoing the trace output to the console. When you are finished tracing, issue the command

```
(WRS)
```

to close the file. Then you can review the file at your leisure using a standard text editor.

To avoid producing prodigious amounts of useless information, you can be selective about when a traced function is to produce output. The command

(SETQ TRACE NIL)

suppresses trace output from all functions until TRACE is reassigned a nonNIL value. The command

(REMFLAG 'TRACE 'name)

where <name> is the name of a traced function, suppresses trace output from that function until the command

(FLAG 'TRACE 'name)

is issued.

Trace output from all functions can be supressed based on the number of calls that have been made to traced functions since the last top-level interaction. If the control variable **MINCALL** has a positive integer value, all trace output is suppressed until the number of function calls is greater than or equal to that value.

Trace output from individual functions can also be supressed based on the number of calls that have been made to that function. The command

(PUT 'MINCALL 'name n)

where <name> is the name of a traced function and <n> is a positive integer, suppresses trace output from that function until it has been called <n> times.

Trace output from all functions can be supressed based on the level of function call nesting. If the control variable **MINLEVEL** has a positive integer value, all trace output is suppressed until the level of function nesting is greater than or equal to that value. Similarly, if **MAXLEVEL** has a positive integer value, all trace output is suppressed until the level of function nesting is less than or equal to that value.

Trace output from individual functions can also be supressed based on the level of recursive nesting of that function. The command

(PUT 'MINLEVEL 'name n)

where <name> is the name of a traced function and <n> is a positive integer, suppresses trace output from that function until its level of recursive nesting is greater than or equal to <n>. Similarly, the command

```
(PUT 'MAXLEVEL 'name n)
```

suppresses trace output from <name> until its level of recursive nesting is less than or equal to <n>. Setting the maximum trace level to 1 makes it possible to trace only the top-level call to recursive functions. For example, the commands

```
(PUT 'MINLEVEL 'APP 2)
(PUT 'MAXLEVEL 'REV 1)
(REV '(A B C))
```

would produce the following trace output:

```
 1| REV [LST1: (A B C)]
 4|    APP [LST1: NIL, LST2: (B)]
 4|    APP = (B)
 3|   APP [LST1: (B), LST2: (A)]
 4|    APP [LST1: NIL, LST2: (A)]
 4|    APP = (A)
 3|   APP = (B A)
 1| REV = (C B A)
(C B A)
```

Trace output from individual functions can also be suppressed unless the function is being called from within the dynamic scope of a given list of functions. This is often useful for tracing SETQs. The command

```
(PUT 'DEBUGIN 'name 'namelist)
```

where <name> is the name of a traced function and <namelist> is a list of traced functions, suppresses trace output from <name> until <name> is called from within the dynamic scope of one of the functions in <namelist>.

Even if one or more of the above means is used to supress trace output, you can display a trace of the most recent trace history. The debug package automatically maintains a history that can be used to reconstruct the trace output that had been supressed. A ring buffer of fixed length is used to store the history. The buffer length is the value of the control variable **HISTLEN.** The default value of HISTLEN is 15. If the command

```
(HISTORY)
```

is issued after the last example, the following is displayed:

```
 4|    APP [LST1: NIL, LST2: (C)]
 4|    APP = (C)
 3|   REV = (C)
 3|   APP [LST1: (C), LST2: (B)]
 4|    APP [LST1: NIL, LST2: (B)]
 4|    APP = (B)
```

```
3|   APP = (C B)
2|  REV = (C B)
2|  APP [LST1: (C B), LST2: (A)]
3|   APP [LST1: (B), LST2: (A)]
4|    APP [LST1: NIL, LST2: (A)]
4|    APP = (A)
3|   APP = (B A)
2|  APP = (C B A)
1| REV = (C B A)
```

The no-spread function CLEAR is used to untrace and unbreak functions, and reset the function call level counter. For example, the command

(CLEAR 'APP 'REV)

untraces APP and REV.

If you should abort an evaluation being traced, the function call level counter, LEVELCOUNT, will not be restored to zero as it should be at the top-level muLISP driver. In this event you should issue the command

(CLEAR)

which resets LEVELCOUNT to zero and performs some other related "house-keeping" chores.

2.7.2 The Break-point Package

When a break-pointed function is called, evaluation is suspended and the muLISP BREAK function is called. This lets you examine the environment present at the break-point (e.g. variable values), display a backtrace showing the current nesting of function calls, and make any desired changes to the environment prior to continuing. You can constrain break-pointed functions to break only at a specified level of recursive nesting, or only within the dynamic scope of specified functions, or only after a certain number of function calls.

The no-spread function BRK is used to break-point functions. It is called with one or more functions to be break-pointed. For example, the command

```
(BRK 'REV 'APP)
```

break-points APP and REV. If you now issue the commands

```
(REV '(A B C))
```

a break will occur and the following will be displayed:

```
Break-point:  REV [LST1: (A B C)]
Continue, Break, Executive, Restart, System?
```

The Break Package section of Chapter 4 describes the available options. Select the "B" option if you want to examine the current muLISP environment. While in the break, you can use the HISTORY function described in the previous section. If you want to determine the stack of function calls that got you to the break-point, use the command

```
(BACKTRACE)
```

The BACKTRACE command displays the functions currently being evaluated, along with their arguments, beginning from the top-level call. Only those functions that have been traced or break-pointed will be included in the backtrace.

When you are ready to proceed from the break, enter the command

```
RETURN NIL
```

Evaluation of the broken function will continue as before. If you do not want the function to be evaluated, instead of NIL in the above command, enter an expression to be evaluated instead. If you want to abort the evaluation entirely and return control to the top-level muLISP driver, enter the command

```
RETURN (THROW)
```

The CLEAR command, described in the previous section, unbreaks functions and restores their original definition.

You can be selective about when a break-pointed function is actually broken. The command

(SETQ BRK NIL)

suppresses all break-points until BRK is reassigned a nonNIL value. The command

(REMFLAG 'BRK 'name)

where <name> is the name of a break-pointed function, suppresses that function's break-point until the command

(FLAG 'BRK 'name)

is issued.

Break-points of all functions can be supressed based on the number of calls that have been made to break-pointed functions. If the control variable **MINCALL** has a positive integer value, all break-points are suppressed until the number of function calls is greater than or equal to that value.

Break-points of individual functions can also be supressed based on the number of calls that have been made to that function. The command

(PUT 'MINCALL 'name n)

where <name> is the name of a break-pointed function and <n> is a positive integer, suppresses break-points by that function until it has been called <n> times.

Break-points of all functions can be supressed based on the level of function call nesting. If the control variable **BRKLEVEL** has a positive integer value, functions are not broken until the level of function nesting is equal to that value.

Break-points of individual functions can also be supressed based on the level of recursive nesting of that function. The command

(PUT 'BRKLEVEL 'name n)

where <name> is the name of a break-pointed function and <n> is a positive integer, suppresses break-points of that function until its level of recursive nesting is equal to <n>. For example, the commands

```
(REMFLAG 'BRK 'REV)
(PUT 'BRKLEVEL 'APP 3)
(REV '(A B C))
```

would produce the following break:

```
Break-point:  APP [LST1: NIL, LST2: (A)]
Continue, Break, Executive, Restart, System?
```

The command

(HISTORY)

would produce

```
2|  REV [LST1: (B C)]
3|   REV [LST1: (C)]
4|    REV [LST1: NIL]
4|    REV = NIL
4|    APP [LST1: NIL, LST2: (C)]
4|    APP = (C)
3|   REV = (C)
3|   APP [LST1: (C), LST2: (B)]
4|    APP [LST1: NIL, LST2: (B)]
4|    APP = (B)
3|   APP = (C B)
2|  REV = (C B)
2|  APP [LST1: (C B), LST2: (A)]
3|   APP [LST1: (B), LST2: (A)]
4|    APP [LST1: NIL, LST2: (A)]
```

The command

(BACKTRACE)

would produce

```
REV [LST1: (A B C)]
APP [LST1: (C B), LST2: (A)]
APP [LST1: (B), LST2: (A)]
APP [LST1: NIL, LST2: (A)]
```

Break-points of individual functions can also be suppressed unless the function is being called from within the dynamic scope of a given list of functions. The command

(PUT 'DEBUGIN 'name 'namelist)

where <name> is the name of a break-pointed function and <namelist> is a list of traced or break-pointed functions, suppresses breaks by <name> until it is called from within the dynamic scope of one of the functions in <namelist>.

2.7.3 Statistics

After completion of an evaluation, you can determine the number of calls made to each function marked for debugging. The value of the variable **CALLCOUNT** is the total number of calls made to all marked functions since the last top-level interaction. The call count for individual functions is stored on the property list of CALLCOUNT. The command

(CDR 'CALLCOUNT)

displays all the marked functions paired with their call count.

After completion of an evaluation, you can determine the maximum level (i.e. depth) of function calls for each function marked for debugging. The value of the variable **MAXLEVELCOUNT** is the maximum level of function calls by the marked functions. The maximum level of recursion for individual functions is stored on the property list of MAXLEVELCOUNT. The command

(CDR 'MAXLEVELCOUNT)

displays all the marked functions paired with their maximum level of recursion.

From within a break-point, you can determine the current level (i.e. depth) of function calls for each function marked for debugging. The value of the variable **LEVELCOUNT** is the current level of function calls by the marked functions. The current level of recursion for individual functions is stored on the property list of LEVELCOUNT. The command

(CDR 'LEVELCOUNT)

displays all the marked functions paired with their current level of recursion.

3. LEARNING muLISP

3.1 DEMONSTRATION PROGRAMS

Several demonstration game programs are included with the muLISP system. These programs demonstrate numerous muLISP programming techniques such as manipulating data structures and providing a friendly user interface. All the programs can be loaded by issuing an RDS command following the muLISP prompt. For example, to load the animal game, issue the command

```
(RDS 'ANIMAL 'LIB)
```

ANIMAL.LIB Think of an animal. The computer will try to guess the animal by asking simple yes/no questions. It is an excellent example of the use of a data base set up as a binary discrimination net. If the computer should fail to guess the animal, it asks you how to distinguish the animal it guessed from the animal you were thinking of in order to extend its data base to include your animal. The more you play, the smarter the program becomes! The animal data base can be saved as a text file for use in a later session.

ANIMAL.MEM Is an optional, prebuilt data base for the animal game containing over 100 animals. It is automatically read in by ANIMAL.LIB.

DOCTOR.LIB Is the muLISP version of the ELIZA (sometimes called DOCTOR) program written by Joseph Weizenbaum of MIT. It turns your computer into an amateurish, but persistent, psychologist. It mimics the Rogerian school of psychology where the doctor asks leading questions, letting patients resolve their own problems.

The "P:" (patient) prompt indicates that the "doctor" is waiting for you to comment on your problem. Don't forget to terminate your sentence with a period, question mark, or exclamation point. The "D:" (doctor) herald precedes the doctor's responses to your statements. To exit the program, just type "Goodbye."

SCRIPT.LIB Contains the patterns and responses used as the script for DOCTOR.LIB. It can be elaborated upon or even replaced with a script on an entirely different subject.

HANOI.LIB Demonstrates the famous recursive solution to the Tower of Hanoi problem. There are three pegs, one of which has a stack of rings of successively smaller diameter from bottom to top (i.e. the smallest on the top). The objective is to transfer all the rings to one of the other pegs so they will be in the same order. Only one ring may be moved at a time, and at no time may a larger ring lie on a smaller ring.

The program uses the letters of the alphabet to represent the rings. The three stacks are displayed as a horizontal sequence of letters at the bottom of the screen. You can use <CTRL>-S to freeze the action as the stacks scroll up the screen. To run the program with, say, 5 rings, enter the muLISP command

```
(HANOI 5)
```

METAMIND.LIB Similar to the popular Master Mind^{tm} board game by Invicta. One player, the "code maker", chooses a secret code consisting of a sequence of four (not necessarily distinct) colors. The other player, the "code breaker", tries to break the code in a minimal number of guesses. In response to the code breaker's guess as to the code, the code maker tells him/her/it the number of exact matches (i.e. of the correct color in the correct position), and the number of correct colors that are in the wrong position. The computer can be either the code maker or breaker.

GRAPHICS.LIB One of the muLISP lessons systematically shows how to define point, line, and turtle graphics functions in muLISP. As a convenience to those wishing to experiment further with graphics, the functions defined in this lesson have been incorporated into the file GRAPHICS.LIB.

3.2 muLISP TUTORIAL SYSTEM

The muLISP Tutorial System is an interactive, on-line system designed to teach the fundamentals of muLISP programming. The source file LESSONS.LIB contains the muLISP program that presents the lessons. Each of the LESson files, named MULISPn.LES, contain the actual text and examples for each lesson.

To run the Tutorial System you will need to start up muLISP as described in Section 2.1 and then insert a diskette containing LESSONS.LIB and the LES files in the logged drive. After the muLISP dollar sign prompt, issue the RDS command:

```
(RDS 'LESSONS 'LIB)
```

After the lesson program is read in, it will display a menu listing the available lessons with a brief indication of their contents. To start with the first lesson, simply press the SPACE BAR. The screen should clear and the lesson will begin. From this point on the lessons should be self explanatory.

The first time through the lessons should be taken in order.

3.3 UTILITY LIBRARY FILES

One of the beauties of LISP is its extensibility (i.e. the ability to add to the language's basic set of primitive functions). Although this has led to numerous dialects, it also makes it relatively easy to extend one dialect of LISP so it will have the same functionality as another.

The two major dialects of LISP on mainframe computers are Maclisp and Interlisp. A large number of applications have been written in both dialects. The following utility library files contain a large number of equivalent muLISP function definitions to help convert programs written in these dialects to muLISP.

MACLISP.LIB: Contains the muLISP equivalent of over 125 Maclisp and Zetalisp (a derivative of Maclisp) functions. The functions are ordered in sections corresponding to those in The Lisp Machine Manual, by Daniel Weinreb and David Moon, published by Symbolics, Inc., 9600 De Soto Avenue, Chatsworth, California, 91311, U.S.A.

INTERLIS.LIB: Contains the muLISP equivalent of over 100 Interlisp functions. The functions are ordered in sections corresponding to those in the Interlisp Reference Manual, by Warren Teitelman, published by the Xerox Palo Alto Research Center, 3333 Coyote Hill Road, Palo Alto, California, 94304, U.S.A.

The files can be read into muLISP using the RDS command or into muSTAR using the R command. To save space, a text editor can be used to extract only those functions required for a particular application. Printed listings of these commented utility files will show numerous examples of good muLISP programming techniques.

4. SYSTEM DESCRIPTION

This chapter describes the primitive muLISP data structures, the memory management scheme, the break package, and the error messages.

4.1 DATA STRUCTURES

Names, numbers, and nodes make up the primitive muLISP **data objects.** Using the recognizer functions, the type of a data object can be determined. Each data object of a given type consists of a fixed number of **pointer cells.** These cells can point to (i.e. contain the memory address of) other data objects or to special purpose data structures such as character strings or binary number vectors. The set of all data objects form an interconnected network of pointers called the muLISP **data space.**

All three types of data objects have as a minimum a **car cell** and a **cdr cell.** These two cells can only point to other objects within the muLISP data space. Thus, the data space is "closed" in the sense that by following car and cdr cell pointers you cannot leave the space. The advantage of this closure is a simpler and more logical muLISP evaluation mechanism. For instance, it eliminates the need for time consuming run-time **type checking** by the functions that follow or manipulate these two pointer cells.

4.1.1 Names

Value	Property	Function	Pname

Internal Representation of a Name

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 four cells are used as follows:

Value cell: This cell contains a pointer to the name's current value. When a name is created, its value cell is set to the name itself. This automatic self-referencing of names is called **auto-quoting.** The assignment functions are used to change the contents of a name's value cell. The value cells of a function's formal parameters are temporarily reassigned when the function is called, and then restored to their original value when the function is exited.

Property list cell: This cell contains a pointer to the name's property list. This list is used and modified by the property and flag functions. **Flags** on a property list can be distinguished by their being atomic elements of the list (names or

numbers). A **property** consists of an indicator consed to a property value. When a name is created, its property list is set to NIL, indicating that no flags or properties are present.

Function cell: This cell contains a pointer to the name's definition. Function definitions are either machine language routines or compiled D-code. When a name is applied, the function cell is used by the muLISP evaluation mechanism to locate the name's definition. The function definition primitives are used to retrieve and modify function definitions. When a name is created, its function cell points to an undefined function trap.

Pname cell: This cell contains a pointer to the name's **print-name string.** The string consists of the ASCII characters used to print the name and it is terminated by a binary zero byte. The string can be of arbitrary length. Access to the pname cell and string is restricted to the I/O and string functions. When a print-name string is read in or generated by a string function, a hashing algorithm is used to determine 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.

4.1.2 Numbers

Itself	Sign	Length	Vector

Internal Representation of a Number

A **number** is a recognizable data object consisting of four pointer cells. Numbers are not uniquely stored, so numbers with identical vectors can coexist in the system. The four cells are used as follows:

Itself cell: This cell contains a pointer to the number itself. Thus numbers do not have to be quoted, since they evaluate to themselves. Although the contents of this cell can be changed using an assignment function, there is no reason to do so and it is not recommended.

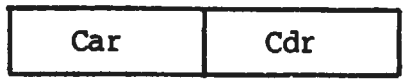
Sign cell: This cell is NIL if the number is positive or zero. If the number is negative, the sign cell is T. Although the contents of the sign cell can be changed using the function RPLACD, there is no reason to do so and it is not recommended.

Length cell: This cell contains the number's absolute value if the number is small enough to be stored in a word of memory (i.e. two bytes). Otherwise, the length cell contains the length of the number's vector in bytes (in words for 16 bit microprocessors).

Vector cell: This cell contains a 0 (a 1 for the 8086 version of muLISP) if the number has an absolute value that can be stored

in a word of memory. Otherwise, the vector cell contains a pointer to the least significant byte (word) of the number's vector. A **number vector** is a contiguous series of bytes (words) in memory used to store the unsigned magnitude of a muLISP number in binary. The bytes (words) of a number vector are stored in reverse order (i.e. more significant bytes (words) are stored in successively lower memory addresses).

4.1.3 Nodes



Internal Representation of a Node

Binary trees are the primary data structures in muLISP. Internally they are implemented as a network of cell pairs called **nodes.** Each node consists of a car cell and a cdr cell. As mentioned earlier, the cells of a node can only point to bona fide muLISP data objects: either names, numbers, or nodes.

Nodes are often called **dotted-pairs**, because of the **dot notation** sometimes used to represent LISP nodes. If X and Y are data objects, the expression

(X . Y)

denotes a node whose car cell points to X and whose cdr cell points to Y. Although cumbersome, dot notation is capable of showing the structure of any LISP data object. In most applications it is more convenient to think of a data structure as a linear list rather than as a deeply nested binary tree. Therefore data structures are usually expressed in **list notation.** The list

(X1 X2 ... Xn)

represents a chain of nodes linked through their cdr cells and whose car cells point to the elements of the list. To be a valid list, the cdr cell of the last node of the list must point to the atom **NIL.** The above structure expressed in dot notations is

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

The function READ accepts both dot and list notation. In list notation, the null or empty list is denoted by NIL. Therefore, when "( )" is read by READ, NIL is returned.

The functions PRIN1 and PRINT use list notation to the maximum extent possible when printing data structures. For example,

(DOG . (CAT . COW))

is output by PRIN1 and PRINT as (DOG CAT . COW).

4.2 MEMORY MANAGEMENT

Dynamic, automatic memory management gives LISP much of its inherent power. This frees the programmer from having to allocate, in advance, sufficient memory for a 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. During the execution of a program, all of the memory not actually required by the program should be available for the creation of new data structures. This is achieved in muLISP in phases, as described in the following subsections.

4.2.1 Initial Data Space Partition

During the initialization phase of muLISP, the amount of read/write memory available to the system is computed. Memory is then partitioned into four distinct data spaces. The proportion of memory initially assigned to each space is based on the relative demands made on the spaces by typical muLISP application programs. The following are the names of the four data spaces and their respective uses:

Atom Space: provides space for the four pointer cells required for each name and number.

Vector Space: provides space for each name's print-name string and each number's binary number vector.

Pointer Space: provides space for the two pointer cells required for each node and the D-code (distilled code) required for each function definition. Since nodes are the primary muLISP data structure, the pointer space is usually the largest data space.

Stack Space: provides space for the control stack and the variable value stack. The two stacks are based at opposite ends of the stack space.

4.2.2 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 that are no longer needed. New data objects are constantly being created during the execution of a muLISP program, while others are implicitly discarded when they are no longer referenced. When the creation process uses up the available resources in any one of the data spaces, a **garbage collection** is performed on all the spaces. Storage space vacated by discarded data structures is reclaimed so the program can continue.

A two-pass, mark-sweep garbage collection algorithm is used. The first pass consists of marking all data objects accessible by means of chaining through pointer cells beginning from the value and

property list cells of all system names, or from the variable stack, or from distilled code. Auto-quoted names that have no properties and have no current function definition are not marked. Such names are automatically removed from the object list during the second pass.

During the second pass of the garbage collection, all the marked data objects are collected and compacted into one end of their respective data spaces. This leaves the remainder of each data space available for newly created objects.

4.2.3 Data Space Reallocation

After a garbage collection one or more of the four muLISP data spaces may have an insufficient amount of free storage to permit an application program to continue execution, in spite of the fact that one or more of the other data spaces may have an abundance of free space. If this situation should occur, a data space reallocation routine is called to move the partitions dividing the spaces so as to allocate more memory to the exhausted space or spaces. Thus muLISP can respond to changing demands placed on the various data spaces by application programs.

Although garbage collection and data space reallocation are automatic, their occurrence is not entirely invisible to the user since they cause a short pause in the execution of a program. The exact amount of time required depends on the amount of muLISP data currently in the system. A garbage collection usually takes less than a second. Similarly, a data space reallocation usually takes less than a second. Normally this is of no concern to the user, but it should be considered in the design of **real time systems** using muLISP.

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

4.3 THE BREAK PACKAGE

The muLISP break package is called by the keyboard interrupt handler (see Section 4.4) and the error trap handlers (see Section 4.5). When a break occurs, the **option prompt** message

Continue, Break, Executive, Restart, System?

is displayed after the interrupt or error message. The system then waits for the user to select an option by typing its first letter (C, B, E, R, or S):

Continue: returns control to the program that invoked the break routine. If the break was caused by a keyboard interrupt, execution continues as if the break had not occurred. If the break was caused by an error trap, the value passed to the break by the error handler is returned as the value of the errant function.

Break: temporarily suspends program execution and calls the next level of the executive driver loop. This allows the user to examine and/or change the current muLISP environment before continuing. To exit the break and resume program execution, type

RETURN NIL

following the dollar sign prompt. See the User Interface Section of Chapter 5 for a detailed discussion of the executive driver loop.

Executive: aborts program execution, restores the original value to the formal arguments that are stored on the variable stack, makes the console the current input source (CIS) and output sink (COS), and returns control to the current level of the executive driver loop. Function definitions, property values, and global variable values are left unchanged.

Restart: restarts muLISP as if from scratch. All variable bindings, user-defined functions, and property values in the current muLISP environment are destroyed.

System: closes all open files, terminates execution of muLISP, and returns control to the host operating system.

4.4 KEYBOARD INTERRUPTS

At any time during the execution of a program, a user initiated keyboard **interrupt** will halt program execution. This may be necessary to stop a "runaway" or nonterminating program and return control to the console.

An interrupt is initiated by pressing the <ESC> key on the computer keyboard. If using a keyboard without an <ESC> key, the **interrupt character** can be generated by typing the left bracket key ([) while holding down the <CTRL> key. If that does not work, the character used to generate keyboard interrupts can be changed by the function CHARTYPE (see Chapter 5).

A keyboard interrupt displays the following **interrupt message** on the console screen:

```
Interrupt
Continue, Break, Executive, Restart, System?
```

The interrupt handler routine then calls the break package so the user can decide how to proceed (see Section 4.3).

If for some reason there is no response to a keyboard interrupt, it is always possible to interrupt the system by resetting the computer. However, a reset done in the middle of an evaluation leaves muLISP in an unpredictable state, making it impossible to restore the muLISP environment present at the time of the reset.

4.5 ERROR MESSAGES

This section explains the significance of the muLISP error messages and the options available to the user when the error occurs. When muLISP detects an error condition an error handler routine is called to display an error message. Most of the error handlers then call the muLISP break package to provide the user with options as to how to proceed (see Section 4.3).

As a general rule, primitive muLISP functions return a value appropriate for their arguments. For instance, if given a list, LENGTH returns the number of elements in the list; if given a name, LENGTH returns the number of characters required to print the name. The few remaining functions for which there is no meaningful value when given an argument of a certain data type simply return NIL. Therefore, with the exception of the "Nonnumeric Argument" error, muLISP has no **argument type errors.**

Directory Full: indicates that there is insufficient space in a diskette's directory for the new entry required by a muLISP SAVE or WRS command. WRS is set to NIL, making the console the current output sink (COS), and the option message is displayed.

If there are no other open input or output files on the diskette that caused the error, the diskette can be replaced with a diskette with sufficient directory space, and the SAVE or WRS command can be reissued.

Disk Full: indicates that there is insufficient space to store the data being written to a disk file. WRS is set to NIL, making the console the COS, and the option message is displayed.

If there are no other open input or output files on the diskette that caused the error, the diskette can be replaced with a diskette with sufficient free space to store the file, and the SAVE or WRS command can be reissued.

End-Of-File: indicates that an attempt was made to read from beyond the end, or from "holes" within, the current input file (CIF) (see the WRITEPTR function in Chapter 5 for an explanation of holes in randomly written files). The option message is displayed. If the Break option is selected, the variable BREAK is bound to the name 'EOF.

Insufficient Memory, Aborting: indicates that there is an insufficient amount of memory in which to load and run a muLISP environment. muLISP is aborted and control returns to the host operating system.

Note that a muLISP environment saved in a SYS file can be loaded into a computer having less memory than the computer on which it was created. The insufficient memory error occurs only if the computer on which the SYS file is being loaded does not have enough memory to store the muLISP environment. The only way to load the SYS file is to obtain more memory for the computer.

Memory Space Exhausted: indicates that there is insufficient memory to continue execution of the current muLISP program. Normally, the memory management system provides sufficient storage in each data space to continuously satisfy the demands of muLISP programs. If the memory requirements for storing data objects exhaust all available resources, the memory space exhausted error occurs. Statistics in the following form are displayed along with the error message:

GC: nnnn aaaa/aaaa vvvv/vvvv pppp/pppp ssss/ssss tttt/tttt

The hexadecimal numbers following "GC:" show the amount of space remaining in each of the four major data spaces. Thus, the data space responsible for the error can be determined. (**Note:** The pointer space on the 8086 and 8088 versions of muLISP is divided into two subregions either one of which can become exhausted before the other. Therefore, a Memory Space Exhausted error can occur even though the statistics show free space in all four data spaces.) See the Memory Management Functions section of Chapter 5 for a detailed explanation of the meaning of these statistics.

Below the statistics and the error message, the option message

Executive, Restart, System?

is displayed and the system waits for the user to select one of the following options by typing its first letter:

Executive: transfers control to the current executive DRIVER loop (see the User Interface section of Chapter 5).

Restart: restarts muLISP as if from scratch. All variable bindings, user-defined functions, and property values in the current muLISP environment are destroyed.

System: closes all open files, terminates execution of muLISP, and returns control to the host operating system.

Nonnumeric Argument: indicates that a numerical function was called with a nonnumeric argument. The option message is displayed. If the Break option is selected, the value of the variable BREAK can be examined to determine in which numerical function the error occurred and the nonnumerical argument that caused the error. The numerical functions include: GREATERP, LESSP, PLUS, MINUS, DIFFERENCE, TIMES, QUOTIENT, REMAINDER, DIVIDE, and GCD.

SYS File Not Found: indicates that the SYS file specified in the operating system command that initiated muLISP could not be found or is not a valid muLISP SYS file. A SYS file can only be loaded under the same version of muLISP that was used to save the file. Chapter 2 describes how muLISP and a muLISP SYS file can be simultaneously loaded using an operating system command.

SYS files can also be loaded from within muLISP using the LOAD command. When a LOAD command is issued from within muLISP and the SYS file is not found, NIL is returned instead of the "SYS File Not Found" message being displayed.

Undefined Function: indicates an attempt was made to evaluate an undefined function. The option message is displayed. If the Break option is selected, the value of the variable BREAK can be examined to determine the name of the undefined function that caused the error.

A common course of action at this point is to take the Break option, appropriately define the undefined function, and continue with the original program by issuing the command:

```
RETURN (EVAL BREAK)
```

Zero Divide: indicates that a division function was called with a divisor of zero. The option message is displayed. If the Break option is selected, the value of the variable BREAK can be examined to determine in which division function the error occurred. The division functions are QUOTIENT, REMAINDER, and DIVIDE.

5. FUNCTIONS AND CONTROL VARIABLES
CHAPTER CONTENTS

Sect	Page	Title	Functions
5.1	2	Selector	CAR, CDR, CxxxR, LAST, NTH, ASSOC
5.2	6	Constructor	CONS, LIST, APPEND, REVERSE, OBLIST
5.3	8	Modifier	RPLACA, RPLACD, NCONC, TCONC, LCONC
5.4	11	Recognizer	NAME, NUMBERP, ATOM, NULL, NONNULL, PLUSP, MINUSP, ZEROP, EVENP
5.5	14	Comparator	EQ, EQUAL, MEMBER, GREATERP, LESSP, ORDERP
5.6	17	Logical	NOT, AND, OR
5.7	18	Assignment	SET, SETQ, POP, PUSH
5.8	20	Property	PUT, GET, REMPROP
5.9	22	Flag	FLAG, FLAGP, REMFLAG
5.10	23	Definition	GETD, PUTD, MOVD
5.11	25	String	SUBSTRING, FINDSTRING, PACK, UNPACK, LENGTH, ASCII
5.12	29	Numerical	MINUS, PLUS, DIFFERENCE, TIMES, QUOTIENT, REMAINDER, DIVIDE, GCD
5.13	32	Reader	RDS, READPTR, READCH, RATOM, READ, READP, CHARTYPE
5.14	40	Printer	WRS, WRITEPTR, PRIN1, PRINT, TERPRI, SPACES, LINELENGTH, RADIX
5.15	46	Evaluation	QUOTE, EVAL, APPLY, LOOP, COND, PROG1, CATCH, THROW, COMMENT
5.16	52	Memory	RECLAIM
5.17	54	Environment	SAVE, LOAD, SYSTEM
5.18.1	56	Interface	DRIVER, BREAK
5.18.2	59	Error	UNDEFINED, NONNUMERIC, ZERODIVIDE, EOF
5.19	61	Hardware	MEMORY, PORTIO, REGISTER, INTERRUPT, CALL, EXECUTE
5.20	64	Special	CLRSCRN, CURSOR, TIME, TEXT, GRAPHICS, PLOT, PADDLE, BEEP

(This page intentionally left blank)

5. FUNCTIONS & CONTROL VARIABLES

This chapter describes all the built-in muLISP functions and control variables. They are primitively defined in machine language. Functions defined by the user must ultimately be defined in terms of these primitives.

The chapter is subdivided into the categories commonly used to characterize LISP functions. The following format is used to describe each primitively defined function:

1. The function name in bold face type, followed by a list of names enclosed in brackets. The names indicate typical arguments used in calls to the function.

2. A brief description of what the function does and/or the value it returns.

3. A formal definition of the function, written in muLISP. This definition is functionally equivalent to the actual machine language definition. **Auxiliary functions** (sometimes called "helper" functions), introduced to aid in the definition of the user accessible function, have names in lower case letters. Such auxiliary functions are not accessible to the user. When an irreducible, primitive concept is introduced in a definition, the concept is described in English using lower case letters.

   In order to more clearly show the essential aspects of functions that take an arbitrary number of arguments, these functions are formally defined for the two argument case. Extending the definitions to handle an arbitrary number of arguments is a matter of straightforward recursion.

4. If appropriate, examples of calls to the function. These are examples that can be entered and run by the user following the muLISP prompt. The value returned by the function is printed to the right of the call, following the symbol -->. Sometimes comments about an example are shown following a semicolon, which is the muLISP comment delimiter.

5. An informal discussion of subtle points and common uses of the function.

5.1 SELECTOR FUNCTIONS

The primary muLISP data structure is **binary trees** made up of nodes (branches) and terminated by atoms (leaves). Given such a tree, the selector functions CAR and CDR return the car (left) and cdr (right) branches of the tree respectively. Successive applications of these two functions is sufficient to traverse any tree. The other functions in this section are simply compositions of CAR and CDR. They are defined in machine language primarily for efficiency and convenience.

CAR [object] returns the data object pointed to by the car cell of <object>.

```
(PUTD 'CAR '(LAMBDA (OBJ)
     (the contents of the car cell of OBJ) ))

(CAR '(A B C D E))     -->  A
(CAR '(A . (B . C)))   -->  A
```

The correct interpretation of CAR[object] depends on whether <object> is an atom, and if not, whether it is regarded as a list or a binary tree. If <object> is an atom, then CAR[object] returns the current value of the atom. If <object> is a list, then CAR[object] returns the first element of the list. If <object> is a binary tree, then CAR[object] returns the left or **car branch** of the tree.

CDR [object] returns the data object pointed to by the cdr cell of <object>.

```
(PUTD 'CDR '(LAMBDA (OBJ)
     (the contents of the cdr cell of OBJ) ))

(CDR '(A B C D E))     -->  (B C D E)
(CDR '(A . (B . C)))   -->  (B . C)
```

As with CAR, the interpretation of CDR[object] depends on whether <object> is an atom, a list, or a binary tree. If <object> is a name, then CDR[object] returns the property list of the name. If <object> is a number, then CDR[object] returns the sign of the number (NIL if the number is positive or zero, T if the number is negative). If <object> is a list, then CDR[object] returns the tail of the list (i.e. everything but the first element of the list). If <object> is a binary tree, then CDR[object] returns the right or **cdr branch** of the tree.

CAAR [list] returns the CAR of CAR [list].

```
(PUTD 'CAAR '(LAMBDA (LST)
     (CAR (CAR LST)) ))
```

CADR [list] returns the CAR of CDR [list].

```
(PUTD 'CADR '(LAMBDA (LST)
     (CAR (CDR LST)) ))

(CADR '(A B C D E))  -->  B
```

CADR[list] returns the second element of <list>. If you prefer a more mnemonic name, use MOVD (see Section 5.10) to move the definition of CADR to SECOND.

CDAR [list] returns the CDR of CAR [list].

```
(PUTD 'CDAR '(LAMBDA (LST)
     (CDR (CAR LST)) ))
```

CDDR [list] returns the CDR of CDR [list].

```
(PUTD 'CDDR '(LAMBDA (LST)
     (CDR (CDR LST)) ))

(CDDR '(A B C D E))  -->  (C D E)
```

CDDR[list] returns <list> minus its first two elements.

CAAAR [list] returns the CAR of CAAR [list].

```
(PUTD 'CAAAR '(LAMBDA (LST)
     (CAR (CAAR LST)) ))
```

CAADR [list] returns the CAR of CADR [list].

```
(PUTD 'CAADR '(LAMBDA (LST)
     (CAR (CADR LST)) ))
```

CADAR [list] returns the CAR of CDAR [list].

```
(PUTD 'CADAR '(LAMBDA (LST)
     (CAR (CDAR LST)) ))
```

CADDR [list] returns the CAR of CDDR [list].

```
(PUTD 'CADDR '(LAMBDA (LST)
     (CAR (CDDR LST)) ))

(CADDR '(A B C D E))  -->  C
```

CADDR[list] returns the third element of <list>. If you prefer a more mnemonic name, use MOVD (see Section 5.10) to move the

definition of CADDR to THIRD.

CDAAR [list] returns the CDR of CAAR [list].

```
(PUTD 'CDAAR '(LAMBDA (LST)
     (CDR (CAAR LST)) ))
```

CDADR [list] returns the CDR of CADR [list].

```
(PUTD 'CDADR '(LAMBDA (LST)
     (CDR (CADR LST)) ))
```

CDDAR [list] returns the CDR of CDAR [list].

```
(PUTD 'CDDAR '(LAMBDA (LST)
     (CDR (CDAR LST)) ))
```

CDDDR [list] returns the CDR of CDDR [list].

```
(PUTD 'CDDDR '(LAMBDA (LST)
     (CDR (CDDR LST)) ))

(CDDDR '(A B C D E))  -->  (D E)
```

CDDDR[list] returns <list> minus its first three elements.

LAST [list] returns the last top-level node of <list>.

```
(PUTD 'LAST '(LAMBDA (LST)
     ((ATOM LST) NIL)
     ((ATOM (CDR LST)) LST)
     (LAST (CDR LST)) ))

(LAST '(A B C D))       -->  (D)        ;the LAST of a list
(LAST '(A B C . D))     -->  (C . D)    ;the LAST of an S-expr
(LAST 'FOO)             -->  NIL        ;the LAST of an atom
(CAR (LAST '(A B C)))   -->  C          ;the last element
```

Note that LAST[list] returns the last <u>node</u>, not the last <u>element</u>, of <list>. As the final example shows, the last element can be extracted by taking the CAR of LAST[list].

NTH [list, n] returns the tail of <list> beginning with the <n>th element, where the car of <list> is the first element. This is the same as the Interlisp definition of NTH.

NTH [n, list] returns the <n>th element of <list>, where the car of <list> is the zeroth element. This is the same as the Maclisp definition of NTH.

```
(PUTD 'NTH '(LAMBDA (ARG1 ARG2)
     ((OR (ZEROP ARG1) (PLUSP ARG1))
          ((ZEROP ARG1)                          ;The Maclisp NTH
               (CAR ARG2) )
          ((ATOM ARG1) NIL)
          (NTH (DIFFERENCE ARG1 1) (CDR ARG2)) )
     ((NOT (PLUSP ARG2)) NIL)
     ((EQ ARG2 1) ARG1)                          ;The Interlisp NTH
     ((ATOM ARG1) NIL)
     (NTH (CDR ARG1) (DIFFERENCE ARG2 1)) ))

(NTH 3 '(A B C D))  -->  D                       ;The Maclisp NTH
(NTH 6 '(A B C D))  -->  NIL
(NTH 2 '(A . B))    -->  NIL

(NTH '(A B C D) 3)  -->  (C D)                   ;The Interlisp NTH
(NTH '(A B C D) 6)  -->  NIL
(NTH '(A . B) 2)    -->  B
```

The two major dialects of LISP define NTH differently. Interlisp returns the tail of the list; Maclisp returns an element of the list. In addition, arguments in calls to NTH are passed in the opposite order by the two dialects. muLISP takes advantage of this ordering to provide compatibility with both dialects. In muLISP, if the first argument is a number, the Maclisp version of NTH is used. Otherwise, the Interlisp version is used.

NIL is returned by both versions if <list> has fewer than <n> elements.

ASSOC [key, alist] performs a linear search of the **association list**, <alist>, looking for a non-atomic element whose car cell is EQUAL to <key>. If found, the entire element is returned. Otherwise, NIL is returned.

```
(PUTD 'ASSOC '(LAMBDA (KEY ALIST)
     ((ATOM ALIST) NIL)
     ((ATOM (CAR ALIST))                    ;Skip atomic elements.
          (ASSOC KEY (CDR ALIST)) )
     ((EQUAL KEY (CAAR ALIST))              ;The EQUAL test.
          (CAR ALIST) )
     (ASSOC KEY (CDR ALIST)) ))

(SETQ CAPITALS '((USA . WASHINGTON) (FRANCE . PARIS) (JAPAN . TOKYO)))
(ASSOC 'FRANCE CAPITALS)      -->  (FRANCE . PARIS)
(ASSOC 'AUSTRALIA CAPITALS)   -->  NIL
```

5.2 CONSTRUCTOR FUNCTIONS

muLISP programs can dynamically generate new data structures using the constructor functions. These functions can create binary trees or linked lists that model the data structure of virtually any problem. The most primitive constructor function, CONS, creates new muLISP nodes. The storage required for new nodes 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 that are now inaccessible.

CONS [object$_1$, object$_2$] returns a new node whose car cell points to <object$_1$> and whose cdr cell points to <object$_2$>.

```
(PUTD 'CONS '(LAMBDA (OBJ1 OBJ2)
     (return a new node pointing to OBJ1 and OBJ2) ))

(CONS 'A '(B C D))  -->  (A B C D)
(CONS 'A 'B)        -->  (A . B)
```

The correct interpretation of CONS[object$_1$,object$_2$] depends on how <object$_2$> is regarded. If <object$_2$> is a list, it creates a list whose first element is <object$_1$> and whose tail is <object$_2$>. If <object$_2$> is an atom or a binary tree, it creates a tree whose left or car branch is <object$_1$>, and whose right or cdr branch is <object$_2$>. CONS simply creates a new node and does not alter either <object$_1$> or <object$_2$>.

LIST [object$_1$, object$_2$, ..., object$_n$] returns a list consisting of the elements <object$_1$>, <object$_2$>, through <object$_n$>.

```
(PUTD 'LIST '(LAMBDA (OBJ1 OBJ2)
     (CONS OBJ1 (CONS OBJ2 NIL)) ))

(LIST 'A 'B 'C 'D)   -->  (A B C D)
(LIST 'A '(B C) 'D)  -->  (A (B C) D)
```

The function LIST is used to create a list made up of its arguments. It is a no-spread function, so it can take an arbitrary number of arguments.

APPEND [list$_1$, list$_2$, ..., list$_n$] returns a list consisting of the elements of the lists <list$_1$>, <list$_2$>, through <list$_n$> in this order.

```
(PUTD 'APPEND '(LAMBDA (LST1 LST2)
     ((ATOM LST1) LST2)
     (CONS (CAR LST1) (APPEND (CDR LST1) LST2)) ))
```

```
(SETQ FOO '(D E F))                 -->  (D E F)
(APPEND '(A B C) FOO '(G H I))      -->  (A B C D E F G H I)
FOO                                 -->  (D E F)
```

In addition to appending lists, APPEND can be used to copy the top level of a list by calling it with a single argument.

Note that if APPEND and NCONC, defined in the next section, are given the same arguments, they will return EQUAL lists. However, APPEND uses conses to copy the top level of all but its last argument, whereas NCONC actually modifies all but its last argument.

REVERSE [list, object] returns the elements of <list> in reverse order, appended to <object>.

```
(PUTD 'REVERSE '(LAMBDA (LST OBJ)
     ((ATOM LST) OBJ)
     (REVERSE (CDR LST) (CONS (CAR LST) OBJ))) ))

(REVERSE '(A B C))    -->  (C B A)
(REVERSE '(A B) 'C)   -->  (B A . C)
```

Normally REVERSE is called with a single argument; however, it is often useful to call it with two arguments.

OBLIST [] returns the object list, a list of the currently active names in the system. (In muLISP it would be more appropriate to call it the **name list.**) The names are listed in the order in which they were read in and/or generated by the string functions: recent names are to the left of more primitive names. Note that unneeded names are automatically collected by the garbage collector (see the Memory Management section of Chapter 4 for details).

5.3 MODIFIER FUNCTIONS

Modifier functions actually redirect pointers in LISP data structures. Thus, modifier functions are used primarily for their effect rather than the value they return. By modifying an existing data structure, they can eliminate the need for CONSing together a whole new structure when only a small part needs to be changed. Since they can easily produce unwanted **side-effects**, these functions should be used only by experienced LISP programmers. For a good example of their use, study the source file for muSTAR.

RPLACA [object$_1$, object$_2$] replaces the car cell of <object$_1$> with a pointer to <object$_2$> and returns <object$_1$>.

```
(PUTD 'RPLACA (LAMBDA (OBJ1 OBJ2)
     (replace the car cell of OBJ1 with a pointer to OBJ2)
     OBJ1 ))

(SETQ FOO '(A B C))  -->  (A B C)
(RPLACA FOO 'D)      -->  (D B C)
FOO                  -->  (D B C)
```

If <object$_1$> is a list, then RPLACA[object$_1$,object$_2$] replaces the first element of the list by <object$_2$>. If <object$_1$> is a binary tree, then the left branch of the tree is replaced by <object$_2$>. If <object$_1$> is an atom, then the value of the atom is set to <object$_2$>. In all cases, the newly modified <object$_1$> is returned.

RPLACD [object$_1$, object$_2$] replaces the cdr cell of <object$_1$> with a pointer to <object$_2$> and returns <object$_1$>.

```
(PUTD 'RPLACD '(LAMBDA (OBJ1 OBJ2)
     (replace the cdr cell of OBJ1 with a pointer to OBJ2)
     OBJ1 ))

(SETQ FOO '(A B C))  -->  (A B C)
(RPLACD FOO '(D E))  -->  (A D E)
FOO                  -->  (A D E)
```

If <object$_1$> is a list, then RPLACD[object$_1$,object$_2$] replaces the tail of the list by <object$_2$>. If <object$_1$> is a binary tree, then the right branch of the tree is replaced by <object$_2$>. If <object$_1$> is an atom, then the property list of the atom is replaced by <object$_2$>. In all cases, the newly modified <object$_1$> is returned.

NCONC [list$_1$, list$_2$, ..., list$_n$] returns a list consisting of the elements of the lists <list$_1$>, <list$_2$>, through <list$_n$> in this order, by modifying the last cdr cells of <list$_1$> through <list$_{n-1}$>.

```
(PUTD 'NCONC '(LAMBDA (LST1 LST2)
     ((ATOM LST1) LST2)
     (RPLACD (LAST LST1) LST2)
     LST1 ))

(SETQ FOO '(D E F))                --> (D E F)
(NCONC '(A B C) FOO '(G H I))      --> (A B C D E F G H I)
FOO                                --> (D E F G H I)
```

Note that if NCONC and APPEND, defined in the previous section, are given the same arguments, they will return EQUAL lists. However, APPEND uses conses to copy all but its last argument, whereas NCONC actually modifies all but its last argument by redirecting their final cdr cells.

If the value of FOO is a list, (NCONC FOO FOO) will create a circular list. If an attempt is then made to display FOO, the display will continue indefinitely.

TCONC [pointer, object] adds <object> to the end of the list pointed to by the car of <pointer> by modifying the list.

```
(PUTD 'TCONC '(LAMBDA (PTR OBJ)
     (SETQ OBJ (LIST OBJ))
     ((ATOM PTR)
          (CONS OBJ OBJ) )
     ((ATOM (CDR PTR))
          (RPLACA PTR OBJ)
          (RPLACD PTR OBJ) )
     (RPLACD (CDR PTR) OBJ)
     (RPLACD PTR OBJ) ))

(SETQ FOO NIL)                     ;Starting with FOO = NIL
(SETQ FOO (TCONC FOO 'A))          ;SETQ FOO required
(SETQ FOO (TCONC FOO 'B))
(SETQ FOO (TCONC FOO 'C))
(CAR FOO)  -->  (A B C)

(SETQ FOO '(NIL))                  ;Starting with FOO = (NIL)
(TCONC FOO 'A)                     ;SETQ FOO not required
(TCONC FOO 'B)
(TCONC FOO 'C)
(CAR FOO)  -->  (A B C)
```

TCONC makes it possible to efficiently add elements to the tail end of a list, using RPLACD to modify the list. Its first argument is a dotted-pair, the car of which points to the beginning of a list and the cdr of which points to the end (i.e. the last node) of the list. TCONC's second argument is the item to be added to the end of the list. The function returns the dotted-pair whose cdr has been modified to point to the new end of the list. As shown above, the first time TCONC is called, its first argument should be the atom NIL or the list (NIL).

LCONC [pointer, list] adds <list> to the end of the list pointed to by the car of <pointer> by modifying the list.

```
(PUTD 'LCONC '(LAMBDA (PTR LST)
     ((ATOM LST) PTR)
     ((ATOM PTR)
          (CONS LST (LAST LST)) )
     ((ATOM (CDR PTR))
          (RPLACA PTR LST)
          (RPLACD PTR (LAST LST)) )
     (RPLACD (CDR PTR) LST)
     (RPLACD PTR (LAST LST)) ))

(SETQ FOO NIL)                          ;Starting with FOO = NIL
(SETQ FOO (LCONC FOO '(A B)))           ;SETQ required
(SETQ FOO (LCONC FOO '(C D)))
(SETQ FOO (LCONC FOO '(E F)))
(CAR FOO)  -->  (A B C D E F)

(SETQ FOO '(NIL))                       ;Starting with FOO = (NIL)
(LCONC FOO '(A B))                      ;SETQ not required
(LCONC FOO '(C D))
(LCONC FOO '(E F))
(CAR FOO)  -->  (A B C D E F)
```

LCONC makes it possible to efficiently add lists to the tail end of a list, using RPLACD to modify the list. Its first argument is a node, the car of which points to the beginning of a list and the cdr of which points to the end (i.e. the last node) of the list. LCONC's second argument is the item to be added to the end of the list. The function returns the dotted-pair whose cdr has been modified to point to the new end of the list. As shown by the examples, the first time LCONC is called, its first argument should be the atom NIL or the list (NIL).

5.4 RECOGNIZER FUNCTIONS

Recognizer functions are predicates used to identify LISP data objects. They all take a single argument and return a value of either **T** or **NIL**. Since these functions are used to identify objects, they do not invoke invalid argument error routines. For example, the numerical recognizer functions PLUSP, MINUSP, ZEROP, and EVENP, do not cause a "Nonnumeric Argument" error if given a name or a node as an argument.

NAME [object] returns T if <object> is a name, otherwise NIL.

```
(PUTD 'NAME '(LAMBDA (OBJ)
     ((if OBJ is a name) T)
     NIL ))

(NAME 'DOG)       -->  T
(NAME 100)        -->  NIL
(NAME '(A B C))   -->  NIL
```

NUMBERP [object] returns T if <object> is a number, otherwise NIL.

```
(PUTD 'NUMBERP '(LAMBDA (OBJ)
     ((if OBJ is an number) T)
     NIL ))

(NUMBERP 'DOG)       -->  NIL
(NUMBERP 100)        -->  T
(NUMBERP '(A B C))   -->  NIL
```

ATOM [object] returns T if <object> is an atom (i.e. a nonnode), otherwise NIL.

```
(PUTD 'ATOM '(LAMBDA (OBJ)
     (OR (NAME OBJ) (NUMBERP OBJ)) ))

(ATOM 'DOG)       -->  T
(ATOM 100)        -->  T
(ATOM '(A B C))   -->  NIL
```

NULL [object] returns T if <object> is the null list, (i.e. the atom NIL), otherwise NIL.

```
(PUTD 'NULL '(LAMBDA (OBJ)
     (EQ OBJ NIL) ))

(NULL NIL)       -->  T
(NULL ())        -->  T
(NULL '(NIL))    -->  NIL
```

NONNULL [object] returns NIL if <object> is the null list, (i.e. the atom NIL), otherwise it returns <object>.

```
(PUTD 'NONNULL '(LAMBDA (OBJ)
     ((EQ OBJ NIL) NIL)
     OBJ ))

(NONNULL NIL)          -->  NIL
(NONNULL 'DOG)         -->  DOG
(NONNULL '(A B C))     -->  (A B C)
```

NONNULL is really the identity function in disguise. The predicate of an implicit COND must be a function application, not merely a variable (see Section 5.15). Thus the function NONNULL is useful for testing if a variable is nonNIL.

PLUSP [object] returns T if <object> is a positive number, otherwise NIL.

```
(PUTD 'PLUSP '(LAMBDA (OBJ)
     ((NOT (NUMBERP OBJ)) NIL)
     (GREATERP OBJ 0) ))

(PLUSP 34)     -->  T
(PLUSP 0)      -->  NIL
(PLUSP -41)    -->  NIL
(PLUSP 'DOG)   -->  NIL
```

MINUSP [object] returns T if <object> is a negative number, otherwise NIL.

```
(PUTD 'MINUSP '(LAMBDA (OBJ)
     ((NOT (NUMBERP OBJ)) NIL)
     (LESSP OBJ 0) ))

(MINUSP 34)     -->  NIL
(MINUSP 0)      -->  NIL
(MINUSP -41)    -->  T
(MINUSP 'DOG)   -->  NIL
```

ZEROP [object] returns T if <object> is the number 0, otherwise NIL.

```
(PUTD 'ZEROP '(LAMBDA (OBJ)
     (EQ OBJ 0) ))

(ZEROP 34)     -->  NIL
(ZEROP 0)      -->  T
(ZEROP -41)    -->  NIL
(ZEROP 'DOG)   -->  NIL
```

EVENP [object] returns T if <object> is an even integer, otherwise NIL.

```
(PUTD 'EVENP '(LAMBDA (OBJ)
     ((NOT (NUMBERP OBJ)) NIL)
     (ZEROP (REMAINDER OBJ 2)) ))

(EVENP 34)     -->  T
(EVENP 0)      -->  T
(EVENP -41)    -->  NIL
(EVENP 'DOG)   -->  NIL
```

The function EVENP was called EVEN in muLISP-80 and muLISP-82.

5.5 COMPARATOR FUNCTIONS

Comparator functions are used to compare data structures. These predicates all require two arguments and, except for MEMBER, return a value of either **T** or **NIL**. Like the recognizer functions, they are used primarily as predicates to direct the flow of control in user defined functions.

EQ [object$_1$, object$_2$] returns T if <object$_1$> is identical to <object$_2$>, otherwise NIL.

```
(PUTD 'EQ '(LAMBDA (OBJ1 OBJ2)
    ((AND (NUMBERP OBJ1) (NUMBERP OBJ2))
        ((numerical value of OBJ1 = numerical value of OBJ2) T)
        NIL )
    ((OBJ1 and OBJ2 point to the same object) T) ))

(EQ 'A 'A)              -->  T
(SETQ FOO '(A B C))     -->  (A B C)
(EQ FOO '(A B C))       -->  NIL
(SETQ BAR FOO)          -->  (A B C)
(EQ FOO BAR)            -->  T
(EQ 7 (PLUS 3 4))       -->  T
```

EQ tests for the **identity** of its two arguments. Since muLISP names are stored uniquely, EQ uses an efficient pointer comparison to determine if two names are identical. Since numbers are not stored uniquely, EQ compares the number vectors to determine if two numbers are equal. Nonatomic arguments are EQ if and only if they point to the same node in memory.

EQUAL [object$_1$, object$_2$] returns T if <object$_1$> is equal to <object$_2$>, otherwise NIL.

```
(PUTD 'EQUAL '(LAMBDA (OBJ1 OBJ2)
     ((ATOM OBJ1)
          (EQ OBJ1 OBJ2) )
     ((ATOM OBJ2) NIL)
     ((EQUAL (CAR OBJ1) (CAR OBJ2))
          (EQUAL (CDR OBJ1) (CDR OBJ2)) ) ))

(EQUAL 'A 'A)                -->  T
(EQUAL '(A B C) '(A B C))    -->  T
(EQUAL '(A B C) '(C B A))    -->  NIL
```

EQUAL tests for the **equality** of its two arguments. If either or both its arguments are atomic, EQUAL is equivalent to the EQ test described above. Two nonatomic arguments are EQUAL if they are **isomorphic** data structures: the car branches are EQUAL and the cdr branches are EQUAL. Two EQUAL data structures look the same when printed. Since EQ is faster, it should be used in place of EQUAL in cases where at least one of the arguments is known to be atomic.

MEMBER [object, list] if <object> is an element of <list>, the tail of <list> beginning with the element <object> is returned, otherwise NIL is returned.

```
(PUTD 'MEMBER '(LAMBDA (OBJ LST)
     ((ATOM LST) NIL)
     ((EQUAL OBJ (CAR LST)) LST)
     (MEMBER OBJ (CDR LST)) ))

(MEMBER 'A '(B C D))  -->  NIL
(MEMBER 'A '(B A D))  -->  (A D)
```

MEMBER is normally used as a predicate to test whether an object is a member of a list. Note that EQUAL, not EQ, is used for the comparison test. If the object is found, the tail of the list is returned by MEMBER. This information is often of value.

GREATERP [n_1, n_2, ..., n_m] returns T if $n_1 > n_2 > \ldots > n_m$, otherwise NIL.

```
(PUTD 'GREATERP '(LAMBDA (N M)
     ((AND (NUMBERP N) (NUMBERP M))
          ((N is greater than M) T)
          NIL )
     (NONNUMERIC 'GREATERP (LIST N M)) ))

(GREATERP 9 5)   -->  T
(GREATERP -5 2)  -->  NIL
```

GREATERP is a multi-argument, greater-than predicate for numbers. If any argument is not a number, a "Nonnumeric Argument" error occurs (see Section 5.18).

LESSP [n_1, n_2, ..., n_m] returns T if $n_1 < n_2 < \ldots < n_m$, otherwise NIL.

```
(PUTD 'LESSP '(LAMBDA (N M)
     ((AND (NUMBERP N) (NUMBERP M))
          ((N is less than M) T)
          NIL )
     (NONNUMERIC 'LESSP (LIST N M)) ))

(LESSP 9 5)                    -->  NIL
(LESSP 47 (ASCII '"5") 58)     -->  T          ;Testing for
(LESSP 47 (ASCII '"=") 58)     -->  NIL        ;decimal digits.
```

LESSP is a multi-argument, less-than predicate for numbers. As the examples show, LESSP can be used to determine if a number lies within a given interval. Note that if any argument is not a number, a "Nonnumeric Argument" error occurs (see Section 5.18).

ORDERP [$atom_1$, $atom_2$] returns T if <$atom_1$> is ordered before <$atom_2$>, otherwise NIL.

```
(PUTD 'ORDERP '(LAMBDA (ATM1 ATM2)
     ((NUMBERP ATM1)
          ((NUMBERP ATM2)
               (LESSP ATM1 ATM2) )
          T )
     ((NUMBERP ATM2) NIL)
     (LESSP (address of ATM1) (address of ATM2)) ))
```

```
(ORDERP 5 9)          -->  T
(ORDERP 'DOG 5)       -->  NIL
(ORDERP 'DOG 'CAT)    -->  T              ;These result depends
(ORDERP 'CAT 'DOG)    -->  NIL            ;on the order of intro.
```

ORDERP provides a **chronological ordering** for atoms that is useful for writing sorting and searching routines. Names are ordered according to their order of introduction. Numbers are ordered in simple numerical order. When ORDERP is used with nonatomic arguments, no error occurs but the result is of no real significance.

5.6 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 **nonNIL** value is considered to be logically true.

NOT [object] returns T if its argument is NIL, otherwise NIL.

```
(PUTD 'NOT '(LAMBDA (OBJ)
     (EQ OBJ NIL) ))

(NOT NIL)    -->  T
(NOT '())    -->  T
(NOT 'FOO)   -->  NIL
```

Note that the function NOT is identical to the function NULL. However for program readability, it is better to use NOT when testing a truth value; use NULL when testing for the empty list.

AND [$object_1$, $object_2$, ..., $object_n$] returns NIL if any argument evaluates to NIL, otherwise <$object_n$> is returned.

```
(PUTD 'AND '(NLAMBDA LST
     ((NULL LST) 'AND)
     ((NULL (CDR LST))
          (EVAL (CAR LST)) )
     ((NOT (EVAL (CAR LST))) NIL)
     (APPLY 'AND (CDR LST)) ))
```

Note that AND is a no-eval function and that an argument is evaluated if and only if all the preceding arguments have evaluated to nonNIL values.

OR [$object_1$, $object_2$, ..., $object_n$] returns NIL if all the arguments evaluate to NIL, otherwise it returns the first argument that evaluates to a nonNIL value.

```
(PUTD 'OR '(NLAMBDA LST
     ((NULL LST) NIL)
     ((EVAL (CAR LST)))
     (APPLY 'OR (CDR LST)) ))
```

OR returns T if any one of its arguments evaluates to a nonNIL value. The arguments are successively evaluated, and if any evaluates to a nonNIL value, T is returned and none of the remaining arguments are evaluated.

5.7 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 parameter of the currently executing function), the assignment remains in effect even after the function is exited. As with the modifier functions (see Section 5.3), this phenomenon is called a **side-effect** of the function.

SET [name, object] replaces the value (i.e. car cell) of <name> with a pointer to <object> and returns <object>.

```
(PUTD 'SET '(LAMBDA (NAM, OBJ)
     (RPLACA NAM OBJ)
     OBJ ))

(SET 'FOO '(A B C))  -->  (A B C)        ;Equivalent to SETQ FOO
FOO                  -->  (A B C)
(SET 'PET 'DOG)      -->  DOG
(SET PET 'ROVER)     -->  ROVER
DOG                  -->  ROVER
PET                  -->  DOG
```

RPLACA and SET are identical as far as their effect on their first argument. However, RPLACA returns its first argument, whereas SET returns its second argument. As a general rule, SET should be used if the first argument is a name; RPLACA should be used if the first argument is a node.

SETQ [name, object] replaces the value (i.e. car cell) of <name> with a pointer to <object> and returns <object>. The first argument in a call to SETQ is not evaluated.

```
(PUTD 'SETQ '(NLAMBDA (NAM OBJ)
     (SET NAM (EVAL OBJ)) ))

(SETQ FOO '(D E F))   -->  (D E F)
FOO                   -->  (D E F)
(SETQ FOO (CDR FOO))  -->  (E F)
```

SETQ is used far more often than SET because assignments are usually made to a variable rather than to the value of a variable. SETQ will be familiar to programmers of more conventional languages where the assignment statement is used pervasively. Though powerful, indiscriminate use of SETQ will result in unstructured LISP programs that are difficult to understand and debug.

POP [name] returns the top of the "stack" list named <name> and updates the value of <name> to the remainder of the stack.

```
(PUTD 'POP '(NLAMBDA (NAM)
     ((NAME NAM)
          ((ATOM (EVAL NAM)) NIL)
          (PROG1 (CAR (EVAL NAM))
                 (SET NAM (CDR (EVAL NAM))) ) ) ))

(SETQ STACK-LIST '(A B C D E F))
(POP STACK-LIST)  -->  A
(POP STACK-LIST)  -->  B
(POP STACK-LIST)  -->  C
STACK-LIST        -->  (D E F)
```

This function provides the LISP analog of the pop operation often used in machine languages to retrieve information from the top of a stack.

PUSH [object, name] pushes <object> onto the top of the "stack" list named <name> and updates the value of <name>.

```
(PUTD 'PUSH '(NLAMBDA (OBJ NAM)
     ((NULL NAM) NIL)
     ((NAME NAM)
          (SET NAM (CONS (EVAL OBJ) (EVAL NAM))) ) ))

(SETQ STACK-LIST NIL)  -->  NIL
(PUSH 'A STACK-LIST)   -->  (A)
(PUSH 'B STACK-LIST)   -->  (B A)
(PUSH 'C STACK-LIST)   -->  (C B A)
STACK-LIST             -->  (C B A)
```

This function provides the LISP analog of the push operation used to save information on a stack.

5.8 PROPERTY FUNCTIONS

The property functions are used to manage **properties** associated with muLISP names. The cdr cell of a name points to a **property list** containing properties and flags (see Section 5.9). A property list is an ASSOCiation list of property keys paired with property values (see the description of ASSOC in Section 5.1). Since flags are atomic elements of the property list, they can be distinguished from properties.

The property and flag functions facilitate the construction of extendible data bases from which information can be easily and rapidly extracted.

PUT [name, key, object] places on the property list of <name> under the indicator <key> the value <object>.

```
(PUTD 'PUT '(LAMBDA (NAM KEY OBJ)
     ((NULL (ASSOC KEY (CDR NAM)))
          (RPLACD NAM (CONS (CONS KEY OBJ) (CDR NAM)))
          OBJ )
     (RPLACD (ASSOC KEY (CDR NAM)) OBJ)
     OBJ ))
```

Since ASSOC uses an EQUAL when searching the property list for a match to <key>, property keys need not be atomic. Note that any previous value under the indicator <key> is automatically discarded.

GET [name, key] returns the property value associated with <name> under the indicator <key>. If no such value exists, it returns NIL.

```
(PUTD 'GET '(LAMBDA (NAM KEY)
     ((NULL (ASSOC KEY (CDR NAM))) NIL)
     (CDR (ASSOC KEY (CDR NAM))) ))

(PUT 'USA 'CAPITAL 'WASHINGTON)  -->  WASHINGTON
(PUT 'FRANCE 'CAPITAL 'PARIS)    -->  PARIS
(PUT 'JAPAN 'CAPITAL 'TOKYO)     -->  TOKYO
(GET 'FRANCE 'CAPITAL)           -->  PARIS
(GET 'USA 'CAPITAL)              -->  WASHINGTON
(GET 'AUSTRALIA 'CAPITAL)        -->  NIL
```

REMPROP [name, key] removes from <name>'s property list the property associated with <key>. It returns the old property value.

```
(PUTD 'REMPROP '(LAMBDA (NAM KEY)
     ((ATOM (CDR NAM)) NIL)
     ((EQUAL (CAADR NAM) KEY)
          (SETQ KEY (CDADR NAM))
          (RPLACD NAM (CDDR NAM))
          KEY )
     (REMPROP (CDR NAM) KEY) ))

(PUT 'USA 'CAPITAL 'WASHINGTON)  -->  WASHINGTON
(PUT 'FRANCE 'CAPITAL 'PARIS)    -->  PARIS
(PUT 'JAPAN 'CAPITAL 'TOKYO)     -->  TOKYO
(REMPROP 'USA 'CAPITAL)          -->  WASHINGTON
(GET 'FRANCE 'CAPITAL)           -->  PARIS
(GET 'USA 'CAPITAL)              -->  NIL
```

Since REMPROP uses a RPLACD operation to remove an element from the property list, care should used when sharing property lists.

5.9 FLAG FUNCTIONS

The flag functions, like the property functions, can be used to store information on the property list of a name. The name is merely flagged as having an attribute by placing the attribute (usually a name) on the property list.

FLAG [name, attribute] flags <name> with <attribute> by making <attribute> the first element of the <name>'s property list.

```
(PUTD 'FLAG '(LAMBDA (NAM ATTRIB)
     ((FLAGP NAM ATTRIB) ATTRIB)
     (RPLACD NAM (CONS ATTRIB (CDR NAM)))
     ATTRIB ))
```

<attribute> is only added to <names>'s property list if not already present.

FLAGP [name, attribute] returns a nonNIL value if <attribute> is a member of <name>'s property list. Otherwise NIL is returned.

```
(PUTD 'FLAGP '(LAMBDA (NAM ATTRIB)
     (MEMBER ATTRIB (CDR NAM)) ))

(FLAG 'JOHN 'MALE)    -->  MALE
(FLAG 'SUE 'FEMALE)   -->  FEMALE
(FLAGP 'JOHN 'MALE)   -->  (MALE)
(FLAGP 'SUE 'MALE)    -->  NIL
```

REMFLAG [name, attribute] removes <attribute> from the property list of <name> and returns T. If the attribute is not found, NIL is returned.

```
(PUTD 'REMFLAG '(LAMBDA (NAM ATTRIB)
     ((ATOM (CDR NAM)) NIL)
     ((EQUAL ATTRIB (CADR NAM))
          (RPLACD NAM (CDDR NAM))
          T )
     (REMFLAG (CDR NAM) ATTRIB) ))

(FLAG 'JAN 'MALE)      -->  MALE
(FLAG 'JAN 'TALL)      -->  TALL
(REMFLAG 'JAN 'MALE)   -->  T
(FLAGP 'JAN 'MALE)     -->  NIL
(FLAGP 'JAN 'TALL)     -->  (TALL)
```

5.10 DEFINITION FUNCTIONS

The function definition functions are the only means of access to a name's function definition cell. When a function is defined, the definition is **pseudo-compiled** into an extremely dense form called **D-code** or **distilled code.** Compiling the definition results in about a three-fold increase in code density and a 20% improvement in interpretation speed over storing the definition as a linked list.

The inverse process of de-compiling a function definition back into a linked list also occurs automatically when GETD is used to retrieve a definition. Thus this process of **incremental compilation** and de-compilation is invisible to the user, and the interactive nature of LISP is not compromised despite the great improvement in efficiency and code density.

D-code can only represent valid lists. The compilation of an S-expression containing a cdr cell that points to a nonNIL atom results in the replacement of the atom by NIL in the D-code. Consequently, if a constant containing a dotted-pair with a nonNIL atomic cdr cell is needed within a compiled definition, the constant should be assigned to a global variable outside the definition. The variable can then be used in place of the constant within the definition. In fact, it is always more efficient to use global variables, rather than quoted non-atomic constants, within function definitions. This is because the constant must be de-compiled every time the value of the constant is required.

GETD [name, flag] decompiles and returns the function definition of <name>, if <flag> is NIL. Otherwise, it returns either LAMBDA, NLAMBDA, or NIL indicating whether <name> is an eval function, a no-eval function, or undefined.

```
(PUTD 'GETD '(LAMBDA (NAM FLG)
     ((NOT (NAME NAM)) NIL)
     ((undefined function NAM) NIL)
     ((NULL FLG)
          ((machine language routine NAM)
               (the memory address of the routine) )
          (the S-expression equivalent of the D-code defining
               the function NAM) )
     ((no-eval function NAM) 'NLAMBDA)
     'LAMBDA ))
```

If <flag> is NIL and <name> is a **user-defined** function, GETD decompiles the function and returns the linked list equivalent of the function's definition. The definition can then be manipulated just like any other list. If <name> is a **machine language** function, GETD returns the physical memory address of the routine. (If muLISP is running on an 8086 or 8088 based computer, GETD returns the offset of the routine from the base of the Code Segment.)

If <flag> is nonNIL, GETD returns LAMBDA if <name> is an eval function; NLAMBDA if <name> is a no-eval function; or NIL if <name> is undefined. If a program needs only to determine if <name> is a defined function, calling GETD with <flag> nonNIL is faster since no consing is required.

PUTD [name, definition] modifies the function definition cell of <name> to point to the compiled D-code for <definition>.

```
(PUTD 'PUTD '(LAMBDA (NAM DEF)
     ((NOT (NAME NAM)) NIL)
     ((NUMBERP DEF)
          (replace the contents of NAM's function cell with DEF)
          DEF )
     (replace the contents of NAM's function cell with a pointer
          to the D-code equivalent of DEF)
     DEF )
```

To link to user-defined, machine language routines, call PUTD with <definition> equal to the physical memory address of the routine. (If muLISP is running on an 8086 or 8088 based computer, use the routine's offset in the Code Segment.) For more details on implementing machine language routines, see Chapter 7.

MOVD [$name_1$, $name_2$] moves the function definition of <$name_1$> to <$name_2$> and returns <$name_2$>.

```
(PUTD 'MOVD '(LAMBDA (NAM1 NAM2)
     ((NOT (NAME NAM1)) NIL)
     ((NOT (NAME NAM2)) NIL)
     (replace the contents of NAM2's function cell with the
          contents of NAM1's function cell)
     NAM2 )

(MOVD 'CAR 'FIRST)
(MOVD 'CADR 'SECOND)
(MOVD 'CADDR 'THIRD)
(FIRST '(A B C))        -->  A
```

As shown by the example, MOVD can be used to provide more mnemonic names for the primitive LISP selector functions without sacrificing execution efficiency. In cases where MOVD is sufficient, it should be used instead of a combination of GETD and PUTD since no recompilation is required.

5.11 STRING FUNCTIONS

The string functions are primarily useful for text and language processing applications. They provide the basic operations necessary to manipulate print-name strings. The LENGTH function is included here since it can be used to find the length of print-name strings, as well as the length of lists.

SUBSTRING [atom, n, m] returns the <n>th through <m>th characters of <atom>'s print-name string.

```
(PUTD 'SUBSTRING '(LAMBDA (ATM N M)
     ((NOT (ATOM ATM)) NIL)
     ((NOT (NUMBERP N)) NIL)
     ((AND (NUMBERP M) (OR (EQ M N) (GREATERP M N)))
          (the name whose print-name consists of the Nth through
               Mth characters in the print-name of ATM) )
     (the name whose print-name consists of the Nth character to
          the end of the print-name of ATM) ))

(SUBSTRING 'ABCDEFG 3 5)    -->  CDE
(SUBSTRING 'ABCDEFG 3)      -->  CDEFG
(SUBSTRING 'ABCDEFG 1 5)    -->  ABCDE
(SETQ PRIN1 NIL)            -->  NIL
(SUBSTRING 1000 1)          -->  "1000"
```

<atom> must be an atom (either a name or a number), <n> is the position of the first desired character, and <m> is the position of the last desired character. All values of <n> less than 1 are equivalent to <n> being equal to 1. If no third argument is given, or if the third argument is less than the second argument, or if it is greater than the length of the string, all characters from <n> to the end of the string are returned. The empty string, "", is returned if <n> is greater than the length of <atom>'s print-name string.

FINDSTRING [$atom_1$, $atom_2$, n] returns the position of the first occurrence of <$atom_1$>'s print-name string in <$atom_2$>'s print-name. If <n> is a positive integer, the search begins with the <n>th character of <$atom_2$>. If <$atom_1$>'s print-name string is not found, NIL is returned.

```
(PUTD 'FINDSTRING '(LAMBDA (ATM1 ATM2 N)
     ((OR (NOT (ATOM ATM1)) (NOT (ATOM ATM2))) NIL)
     ((PLUSP N)
          ((NULL (FINDSTRING ATM1 (SUBSTRING ATM2 N))) NIL)
          (PLUS N (FINDSTRING ATM1 (SUBSTRING ATM2 N))) )
     ((if ATM1 occurs as a substring in ATM2)
          (the position of ATM1's first occurrence in ATM2) ) ))

(FINDSTRING 'XYZ 'ABCXYZDEFXYZGHI)     -->  4
(FINDSTRING 'XYZ 'ABCXYZDEFXYZGHI 5)   -->  10
(FINDSTRING 'XYZ 'ABCDEFGHI)           -->  NIL
```

PACK [list] returns the name whose print-name is a packed version of the print-names of the atoms in <list>.

```
(PUTD 'PACK '(LAMBDA (LST)
     ((ATOM LST) "")
     ((NAME (CAR LST))
          (the name whose print-name consists of the print-name
               of (CAR LST) concatenated with (PACK (CDR LST))) )
     ((NUMBERP (CAR LST))
          (the name whose print-name consists of the digits in
               the printed representation of (CAR LST)
               concatenated with (PACK (CDR LST))) )
     (PACK (CDR LST)) ))

(SETQ PRIN1 NIL)  -->  NIL
(PACK '(A B C))   -->  ABC
(PACK '(3 A 5))   -->  "3A5"
```

The current radix base is used to determine the print-name string of numbers. Note that PACK always returns a name, even if its print-name contains only digits. Thus PACK can be used to write a **GENSYM** (GENerate new SYMbol) function (see the muLISP utility source file). In some dialects of LISP, PACK is called **COMPRESS** or **IMPLODE.**

UNPACK [atom] returns a list of names whose one character print-names correspond to the successive characters in the printed representation of <atom>.

```
(PUTD 'UNPACK '(LAMBDA (ATM)
     ((NAME ATM)
          (a list of names whose print-names consist of the
               characters in the print-name of ATM) )
     ((NUMBERP ATM)
          (a list of names whose print-names consist of the
               digits in the printed represented of ATM) )
     NIL ))

(SETQ PRIN1 NIL)                          ;Enable quoted strings.
(UNPACK 'ABCDE)   -->  (A B C D E)        ;List of char names.
(SETQ FOO -216)
(UNPACK FOO)      -->  (- "2" "1" "6")    ;Decimal numerals.
(RADIX 16)
(UNPACK FOO)      -->  (- "0" D "8")      ;Hexadecimal numerals.
```

Note that NIL is returned if <atom> is nonatomic. When unpacking a number, the current radix is used to determine the digits representing the number. Such digits are converted into single character muLISP names rather than numbers. UNPACK is called **EXPLODE** in some dialects of LISP.

LENGTH [object] returns the length of <object> based on the data type of <object>.

```
(PUTD 'LENGTH '(LAMBDA (OBJ)
     ((AND (NULL OBJ) (NULL LENGTH))  0)
     ((ATOM OBJ)
          (LENGTH (UNPACK OBJ)) )
     (PLUS 1 (LENGTH (CDR OBJ))) ))

(LENGTH '(A B C D))  -->  4
(LENGTH 'MULISP)     -->  6
(LENGTH -13)         -->  3
```

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

a. If <object> is a name, LENGTH[object] returns the number of characters required to print <object>. The current value of the control variable PRIN1 is taken into account while computing this length. The effect of PRIN1 is described under the control variable subsection of Section 5.14.

b. If <object> is a number, LENGTH[object] returns the number of characters required to print <object>, taking into account the current radix base, leading "-" signs, and leading "0"s.

c. If <object> is nonatomic, LENGTH[object] returns the number of top level nodes (elements) in the list <object>. A cdr cell that points to an atom denotes the end of the list as far as LENGTH is concerned.

LENGTH: In LISP, NIL is used to denote both the 3 character name and the 0 element list. Thus the desired value for LENGTH[NIL] depends on the context in which it occurs. If the control variable **LENGTH** is NIL, LENGTH[NIL] is 3; otherwise, it is 0. By default, LENGTH has a nonNIL value.

```
(SETQ LENGTH T)
(LENGTH NIL)         -->  0            ;Length of empty list.
(SETQ LENGTH NIL)
(LENGTH NIL)         -->  3            ;Length of the name NIL.
```

ASCII [atom] returns the ASCII equivalent of <atom>.

```
(PUTD 'ASCII '(LAMBDA (ATM)
     ((NAME ATM)
          (the integer equivalent of the first character in the
               print-name of ATM) )
     ((AND (NUMBERP ATM) (LESSP -1 ATM 256))
          (the name whose print-name is the single character
               equivalent to ATM) ) ))
```

```
(ASCII 'A)     -->  65
(ASCII '" ")   -->  32
(ASCII 65)     -->  A
(ASCII 10)     -->  <the ASCII line-feed character>
```

If <atom> is a name, ASCII[atom] returns the integer value of the ASCII code used to denote the first character in the print-name of <atom>. If <atom> is a nonnegative number less than 256, ASCII[atom] returns the muLISP name whose print-name is the ASCII character <atom>. This function is especially useful for generating the ASCII control characters.

5.12 NUMERICAL FUNCTIONS

The numerical functions implement the basic mathematical operations using exact, infinite precision, integer arithmetic. The size of numbers that can be manipulated is limited only by the address space of the computer and the available memory.

If a numerical function is called with nonnumeric arguments, the NONNUMERIC error handler function is called. The default definition of NONNUMERIC breaks program execution and allows the user to determine what caused the error by examining the value of the variable BREAK. Similarly, the ZERODIVIDE error handler is called if an attempt is made to divide by a number by zero.

Several of the numerical functions accept a variable number of arguments. However, for simplicity's sake, the formal muLISP definition for these functions shows only two arguments.

MINUS [n] returns minus <n>.

```
(PUTD 'MINUS '(LAMBDA (N)
     ((NUMBERP N)
          (- N) )
     (NONNUMERIC 'MINUS (LIST N)) ))

(MINUS 10)  -->  -10
(MINUS -8)  -->  8
```

PLUS [n_1, n_2, ..., n_m] returns the sum of <n_1> through <n_m>.

```
(PUTD 'PLUS '(LAMBDA (N M)
     ((AND (NUMBERP N) (NUMBERP M))
          (N + M) )
     (NONNUMERIC 'PLUS (LIST N M)) ))

(PLUS 2 3 4)  -->  9
(PLUS -5 3)   -->  -2
```

DIFFERENCE [n_1, n_2, ..., n_m] returns the difference between <n_1> and the sum of <n_2> through <n_m>. If DIFFERENCE is given only a single argument, it returns minus that argument.

```
(PUTD 'DIFFERENCE '(LAMBDA (N M)
     ((AND (NULL M) (NUMBERP N))
          (MINUS N) )
     ((AND (NUMBERP N) (NUMBERP M))
          (N - M) )
     (NONNUMERIC 'DIFFERENCE (LIST N M)) ))

(DIFFERENCE 12 5)     -->  7
(DIFFERENCE 12 5 -3)  -->  10
```

TIMES [n_1, n_2, ..., n_m] returns the product of <n_1> through <n_m>.

```
(PUTD 'TIMES '(LAMBDA (N M)
     ((AND (NUMBERP N) (NUMBERP M))
          (N * M) )
     (NONNUMERIC 'TIMES (LIST N M)) ))

(TIMES 3 4 5)  -->  60
```

QUOTIENT [n, m] returns the truncated integer quotient of <n> divided by <m>. If <n> (i.e. the dividend) is negative and -<n> is not evenly divisible by <m>, the absolute value of the truncated quotient is incremented in order to preserve the fundamental identity between quotients and remainders (see the discussion following REMAINDER). If <m> (i.e. the divisor) is zero, the ZERODIVIDE error handler function is called.

```
(PUTD 'QUOTIENT '(LAMBDA (N M)
     ((AND (NUMBERP N) (NUMBERP M))
          ((ZEROP M)
               (ZERODIVIDE 'QUOTIENT (LIST N M)) )
          ((MINUSP M)
               (MINUS (QUOTIENT N (MINUS M))) )
          ((MINUSP N)
               ((-N is divisible by M)
                    (MINUS (QUOTIENT (MINUS N) M)) )
               (MINUS (PLUS 1 (QUOTIENT (MINUS N) M))) )
          (the truncated integer quotient of N and M) )
     (NONNUMERIC 'QUOTIENT (LIST N M)) ))

(QUOTIENT 7 3)     -->   2
(QUOTIENT 7 -3)    -->  -2
(QUOTIENT -7 3)    -->  -3
(QUOTIENT -7 -3)   -->   3
```

REMAINDER [n, m] returns the remainder of <n> divided by <m> (i.e. <n> modulo <m>).

```
(PUTD 'REMAINDER '(LAMBDA (N M)
     ((AND (NUMBERP N) (NUMBERP M))
          ((ZEROP M)
               (ZERODIVIDE 'REMAINDER (LIST N M)) )
          (DIFFERENCE N (TIMES M (QUOTIENT N M))) )
     (NONNUMERIC 'REMAINDER (LIST N M)) ))

(REMAINDER 7 3)     -->  1
(REMAINDER 7 -3)    -->  1
(REMAINDER -7 3)    -->  2
(REMAINDER -7 -3)   -->  2
```

Note that this definition of REMAINDER preserves the fundamental relation between quotients and remainders:

n = m QUOTIENT[n,m] + REMAINDER[n,m]

As shown in the examples, if the dividend is negative, nonnegative remainders are rotated about the origin in order to make them positive. Thus, remainders have the nice mathematical property of always being nonnegative and periodic in <n>, with period <m>. In other words, the REMAINDER function is equivalent to the **modulo** function, very useful for number theory and computer algebra applications.

If the second argument to REMAINDER (i.e. the divisor) is zero, the ZERODIVIDE error handler function is called.

DIVIDE [n, m] returns a node whose car cell points to the quotient of <n> and <m> and whose cdr cell points to the remainder of <n> and <m>.

```
(PUTD 'DIVIDE '(LAMBDA (N M)
     ((AND (NUMBERP N) (NUMBERP M))
          ((ZEROP M)
               (ZERODIVIDE 'DIVIDE (LIST N M)) )
          (CONS (QUOTIENT N M) (REMAINDER N M)) )
     (NONNUMERIC 'DIVIDE (LIST N M)) ))

(DIVIDE 7 3)    -->  (2 . 1)                ;Positive dividend.
(DIVIDE -7 3)   -->  (-3 . 2)               ;Negative dividend.
```

When both a quotient and remainder are required, using the DIVIDE function is more efficient than computing them separately.

If the second argument to DIVIDE (i.e. the divisor) is zero, the ZERODIVIDE error handler function is called.

GCD [n_1, n_2, ..., n_m] returns the greatest common divisor of its integer arguments.

```
(PUTD 'GCD '(LAMBDA (N M)
     ((AND (NUMBERP N) (NUMBERP M))
          ((ZEROP M)
               ((PLUSP N) N)
               (MINUS N) )
          (GCD M (REMAINDER N M)) )
     (NONNUMERIC 'GCD (LIST N M)) ))

(GCD 12 -16 20)  -->  4
(GCD 0 5)        -->  5
(GCD 0 0)        -->  0
```

The greatest common divisor of two integers is the largest nonnegative integer that evenly divides both integers.

5.13 READER FUNCTIONS, CONTROL VARIABLES, AND MACROS

The reader functions, control variables, and macros enable muLISP programs to read and interpret character data from various sources. The functions read characters from the **current input source (CIS)**. The reader control variables provide muLISP programs with control over various input options. A **read macro** is a special parse function assigned to a character that is called when the character is encountered by the LISP reader.

5.13.1 Reader Functions

RDS [filename, filetype, drive] selects as the **current input file (CIF)** the file named <filename>.<filetype> on <drive>, and sets RDS to <filename>, making the file the CIS as well. If <filename> is NIL and there is an active CIF, the file that is the CIF is closed for input, an internal CIF active flag is turned off, and RDS is set to NIL, making the console the CIS.

```
(PUTD 'RDS '(LAMBDA (FNAME FTYPE DRIVE)
     ((NULL FNAME)
          (if there is an active CIF, close the file for input)
          (SETQ RDS NIL) )
     ((NULL DRIVE)
          (RDS FNAME FTYPE <currently logged drive>) )
     ((AND (NAME FNAME) (NAME FTYPE) (ATOM DRIVE))
          ((if the file FNAME.FTYPE is a currently opened file)
               (make FNAME.FTYPE the active CIF)
               (SETQ RDS FNAME) )
          ((if the file FNAME.FTYPE exists on DRIVE)
               (open FNAME.FTYPE for input)
               (make FNAME.FTYPE the active CIF)
               (zero the file's read pointer)
               (SETQ RDS FNAME) )
          (SETQ RDS NIL) )
     (SETQ RDS NIL) ))

(RDS ANIMAL LIB D)  -->  ANIMAL          ;Read from D:ANIMAL.LIB
```

If RDS is called without a third argument, the currently logged drive is assumed. If <filename>.<filetype> has already been opened for input <u>or</u> output, the file is made the CIF and the file's read pointer is not changed. Otherwise, a search is made for <filename>.<filetype> on <drive>. If found, the variable RDS is set to the file's name and the file becomes the new active CIF. In either case if the file is found, RDS is set to <filename> and the file becomes the CIS. If the file is not found, RDS is set to NIL and the console becomes the CIS.

For more information on selecting the CIF and the CIS see the discussion of the control variable RDS in Section 5.13.2.

READPTR [pointer] sets the read pointer of the CIF to <pointer> and returns the previous read pointer. If <pointer> is not zero or a positive number, the current read pointer is returned. If <pointer> is the name **EOF**, the size of the CIF is returned.

```
(PUTD 'READPTR '(LAMBDA (PTR)
     ((if there is no active CIF) NIL)
     ((EQ PTR 'EOF)
          (return the file size in bytes) )
     ((AND (NUMBERP PTR) (LESSP -1 PTR (max file size in bytes)))
          (set the CIF read pointer to PTR)
          (return the previous CIF read pointer) )
     (return the current CIF read pointer) ))
```

A file's read pointer determines the point in the file at which the next character will be read. When RDS opens a file for input the file's read pointer is set to 0. When a character is read from the file, the file's read pointer is automatically incremented. If the file must be accessed randomly, use READPTR to change the CIF's read pointer. Note that READPTR operates on the CIF (current input file) rather than the CIS (current input source). Thus, READPTR can be used even if the console is the CIS.

READCH [peek-flag] reads and returns a single character from the CIS. If <peek-flag> is nonNIL and either READ or RATOM is used for the next read operation, the first character seen will be the same character last read by READCH. If a file is the CIS and an attempt is made to read past the end of the file, the end-of-file error handler, EOF, is called to provide a trap (see Section 5.18).

```
(PUTD 'READCH '(LAMBDA (FLAG)
     ((a file is CIS and the end-of-file has been reached)
          (EOF) )
     (SETQ RATOM (the next character from the CIS))
     ((NULL FLAG)
          RATOM )
     (store the character in the input buffer used by the
          function RATOM)
     RATOM ))
```

As a programming convenience, the atom returned by READCH is also assigned to the variable RATOM. Numbers are returned by READCH if the character is a decimal digit less than the current radix base.

If <peek-flag> is nonNIL, the character read by READCH remains in RATOM's (but not READCH's) input buffer. This makes it possible in parsing applications to use READCH to "look ahead" a single character, and then have RATOM re-read the character as part of the next token. This look ahead feature greatly simplifies the following definition of READ-LINE. (READ-LINE is NOT a primitively defined muLISP function.)

```
(PUTD 'READ-LINE '(LAMBDA (CHAR)
     (SETQ CHAR (READCH T))                  ;Peek ahead a character.
     ((EQ CHAR (ASCII 13)) NIL)              ;Finished if <RETURN>.
     ((EQ CHAR '" ")                         ;Skip past spaces.
          (READ-LINE) )
     (CONS (RATOM) (READ-LINE)) ))           ;Add token to list.
```

READ-LINE returns a list of tokens read from an input line. The line is terminated by <RETURN> (i.e. (ASCII 13)). Spaces are ignored except for the purpose of separating tokens. Note that RATOM, called in the last line of the definition, rereads the character CHAR.

RATOM [] reads a token from the CIS and returns the corresponding muLISP atom. If a file is the CIS and an attempt is made to read past the end of the file, the end-of-file error handler, EOF, is called to provide a trap (see Section 5.18).

```
(PUTD 'RATOM '(LAMBDA ()
     ((a file is CIS and the end-of-file has been reached)
          (EOF) )
     (SETQ RATOM (the next token read from the CIS)) ))

(RATOM)  DOG     -->  DOG
(RATOM)  123CAT  -->  123                    ;CAT remains for input.
```

A **token** is a string of characters delimited by either separator or break characters. RATOM returns a number if the token begins with a decimal digit less than the current radix base. Otherwise, RATOM returns a name. Number tokens are terminated by any character that is not a digit in the current radix base. As a programming convenience, the atom returned by the function RATOM is assigned to the variable RATOM.

Separator characters serve only to delimit tokens and are not returned by RATOM as atoms. The default RATOM separator characters are <space>, <return>, <line-feed>, and <tab>. The ASCII code for <return> is 0DH, <line-feed> is 0AH, and <tab> is 09H.

Break characters also delimit tokens, but unlike separators, they are returned by RATOM as single character muLISP names. The default RATOM break characters are

```
!  $  &  '  (  )  *  +  ,  -  .  /  @  :
<  =  >  ?  [  \  ]  ^  _  `  {  |  }  ~
```

CHARTYPE can be used to change the characters RATOM recognizes as separator and break characters. See the discussion following READ for details concerning the effect of comments and quoted strings on RATOM.

READ [] reads one complete symbolic expression from the CIS and returns the equivalent linked list. If a file is the CIS and an attempt is made to read past the end of the file, the end-of-file error handler, EOF, is called to provide a trap (see Section 5.18).

```
(PUTD 'READ '(LAMBDA ()
     (read-aux (read-ratom)) ))

(PUTD 'read-aux '(LAMBDA (TOKEN)
     ((EQ TOKEN '"(")
          (read-list (read-ratom)) )
     ((EQ TOKEN '"[")
          (read-bracket (read-list (read-ratom)) (read-ratom)) )
     ((OR (EQ TOKEN '")") (EQ TOKEN '".") (EQ TOKEN '"]"))
          (READ) )
     TOKEN ))

(PUTD 'read-bracket '(LAMBDA (LST TOKEN)
     ((OR (EQ TOKEN '")") (EQ TOKEN '"]"))
          LST )
     (CONS LST (read-bracket (read-list TOKEN) (read-ratom))) ))

(PUTD 'read-list '(LAMBDA (TOKEN)
     ((terminator TOKEN) NIL)
     ((EQ TOKEN '".")
          (read-dot (READ) (read-ratom)) )
     (CONS (read-aux TOKEN) (read-list (read-ratom))) ))

(PUTD 'read-dot '(LAMBDA (LST TOKEN)
     ((terminator TOKEN) LST)
     (CONS LST (read-list TOKEN)) ))

(PUTD 'terminator '(LAMBDA (TOKEN)
     ((EQ TOKEN '")") T)
     ((EQ TOKEN '"]")
          (unread "]" character back into CIS for the next
               call to read-ratom)
          T ) ))

(PUTD 'read-ratom '(LAMBDA ()
     ((a file is CIS and the end-of-file has been reached)
          (EOF) )
     (SETQ RATOM (the next token read from the CIS)) ))
```

<read-ratom> is a helper function that reads the next token from the CIS and returns the corresponding name or number. It is identical to RATOM except different separator and break characters are used. The <read-ratom> separator characters are <space>, <comma>, <return>, <line-feed>, and <tab>. The <read-ratom> break characters are <left parenthesis>, <right parenthesis>, <period>, <left bracket>, and <right bracket>.

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, extra right brackets, and extra dots in an expression are ignored.

Comments in a source file are ignored (i.e. skipped over) by both the functions RATOM and READ. (Note that READCH reads and returns comment characters the same as any other characters.) Comments can be either the single line or multi-line variety. Single line comments begin with a semicolon, and are terminated by the end of the line. Multi-line comments are delimited by matching percent signs. The characters used to delimit comments can be changed by the function CHARTYPE.

The **special characters** consist of the separator characters, the break characters, the comment delimiter characters, the read macro characters, the splice macro characters, and the double quote character. Sometimes it may be necessary to read these characters in as "normal" characters so that they can be interned as muLISP names or parts of names. The special reader properties of these characters can be defeated by means of **quoted strings.** Such strings must be delimited by double quote marks. The double quote mark can be included within the string by using two adjacent double quotes for each desired double quote. The character used to delimit quoted strings can be changed by the function CHARTYPE.

READP [] returns T if there is a character available for reading in the CIS. Otherwise NIL is returned.

```
(PUTD 'READP '(LAMBDA ()
     ((if there is a character in the input buffer) 'T)
     'NIL )
```

READP is used to determine the character ready status of the CIS. If the console is the CIS, this function can be used to test if a character has been entered by the user. If a source file is the CIS, this function can be used to test whether the end of the file has been reached.

CHARTYPE [character, n] assigns a character type of <n> to <character> and returns the original character type, provided <n> is a number between 0 and 8 inclusive. Otherwise, the character type of <character> is returned.

```
(PUTD 'CHARTYPE '(LAMBDA (CHAR NUM)
     ((NAME CHAR)
          (CHARTYPE (ASCII CHAR) NUM) )
     ((AND (NUMBERP CHAR) (LESSP -1 CHAR 128))
          ((AND (NUMBERP NUM) (LESSP -1 NUM 9))
               (assign CHAR a character type of NUM)
               (return the original character type of CHAR) )
          (return the character type of CHAR) ) ))
```

The following summarizes the eight possible character types:

C H A R A C T E R T Y P E T A B L E

Typ No.	Default type of characters	Meaning of Character type
1	<LINEFEED> <RETURN> <SPACE> <TAB>	RATOM separator character
2	! $ & ( ) * + , - . / : < = > ? @ [\ ] ^ _ ` { | } ~	RATOM break character
3	"	String delimiter
4	%	Multi-line comment delimiter
5	;	Single line comment delimiter
6	'	Read macro character
7	<none>	Splice macro character
8	<ESCAPE>	Keyboard interrupt character
0	<all other characters>	Non-special character

When a keyboard **interrupt character** is entered while a program is running, an interrupt will occur (see the Keyboard Interrupt section of Chapter 4). The console is checked periodically to see if an interrupt character has been entered. This has two possibly detrimental side effects. First, there is a minor speed degradation caused by the periodic checks. Second, and more importantly, non-interrupt characters entered from the console can be discarded by the interrupt checker before they can be read in by a muLISP program. If no character has a type of 8, both of these effects are eliminated at the price of disabling all keyboard interrupts.

5.13.2 Reader Control Variables

The reader control variables extend the flexibility of the reader functions by making several input options available to the user. Most operate as switches that are either on or off. With the exception of ECHO, these control variables have the same names as the related reader functions.

RDS: Normally the function RDS (see Section 5.13.1) can provide sufficient control over the source of character input to muLISP programs. However, after reading part of a file it may be necessary to receive input from the console before returning to the file. After using RDS to open a file and make it the **current input file (CIF)**, the console can be made the **current input source (CIS)** without changing the CIF by setting the variable RDS to NIL. A subsequent nonNIL assignment to RDS will again make the CIF the CIS, and reading of the file will resume from the point at which it had been suspended. Upon initial system startup and after interrupt or error traps, the muLISP DRIVER sets RDS to NIL so that the console is the default CIS.

READCH: When the console is the CIS, the console input mode is controlled by READCH. If READCH is nonNIL, console input is 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 <RETURN> is entered, the system's normal line editing procedures such as backspacing, line deletion, and printer output echoing are in force. If READCH is NIL, all buffering and input echoing is eliminated. This **raw input mode** is useful for applications, such as muSTAR, requiring a response to each character entered by the user.

READ: If READ is nonNIL, lowercase letters are legitimate and distinct from their uppercase counterparts. However, file names and types as used by the functions RDS and WRS are always converted to uppercase in order to eliminate conflicts with the host operating system's file naming convention. If READ is NIL, lowercase letters are converted to uppercase as they are read in (lowercase letters already in the system remain in lowercase).

ECHO: If a disk file is the CIS and ECHO is nonNIL, the characters read from the file are **echoed** to the current output sink (COS). Note that since comments are also echoed, English language text within a comment can be displayed conveniently without muLISP having to actually process the text. See Section 5.14 for the effect of ECHO when outputting characters to a file.

5.13.3 Read and Splice Character Macros

Character macros are special parse properties that are invoked when a macro character is read by READ. Not only do character macros make it possible to abbreviate frequently used code segments, but they can alter the syntax used in LISP source files to better suit the application at hand.

Read macros: When a read macro character is encountered by READ, the character's function definition is evaluated and the resulting expression is inserted into the input stream in place of the character. A read macro is established by providing a suitable function definition for the desired character and then using CHARTYPE to make it a character of type 6.

The most commonly used read macro is the single quote mark used to abbreviate a call to the function QUOTE. This is the only primitively defined read macro in muLISP. The following definition for the single quote character is equivalent to the one that is primitively defined in muLISP:

```
(PUTD (QUOTE "'") (QUOTE (LAMBDA ()
     (LIST (QUOTE QUOTE) (READ)) )))

(CHARTYPE (QUOTE "'") 6)

'(PLUS 3 4)  -->  (PLUS 3 4)
```

Splice macros: If a splice macro character is encountered while a list is being read, the character's function definition is evaluated and the resulting list is "spliced" into the input stream as possibly several (or no) elements. A splice macro is established by providing a suitable function definition for the desired character and then using CHARTYPE to make it a character of type 7.

Splice macros are most commonly used for splicing nothing into the input stream (i.e. a NIL)! For instance, the semicolon could be defined as a comment character as follows:

```
(SETQ CR (ASCII 13))                % Carriage-return character %

(PUTD '";" '(LAMBDA ()
     (LOOP
          ((EQ (READCH) CR) NIL) ) ))

(CHARTYPE '";" 7)

(PLUS 3 4                    ;What you are now reading is a comment!
5 6)      -->  18
```

5.14 PRINTER FUNCTIONS AND CONTROL VARIABLES

The printer functions and control variables enable muLISP programs to direct character output to various destinations or sinks. The functions output characters to the **current output sink (COS)**. The COS can be either the console, the system's hard-copy printer (if available), or a specified disk file. The printer control variables provide muLISP programs with control over various output options.

5.14.1 Printer Functions

WRS [filename, filetype, drive, reopen] selects as the **current output file (COF)** the file named <filename>.<filetype> on <drive>, and sets WRS to <filename> making the file the COS as well. If <filename> is NIL and there is an active COF, the COF is closed so there will be no active COF, and the console is made the COS.

```
(PUTD 'WRS '(LAMBDA (FNAME FTYPE DRIVE REOPEN)
     ((NULL FNAME)
          ((if there is an active COF)
               (close the COF for output, setting its length to
                    that of its write pointer)
               ((if the COF was given a temporary .$TYPE name)
                    ((if there is a file with COF's final name)
                         (rename that file to be a .BAK file)
                         (rename the temporary .$TYPE file
                              to its final name)
                         (SETQ WRS NIL) )
                    (rename the temporary .$TYPE file
                         to its final name)
                    (SETQ WRS NIL) )
               (SETQ WRS NIL) )
          (SETQ WRS NIL) )
     ((NULL DRIVE)
          (WRS FNAME FTYPE (currently logged drive)) )
     ((AND (NAME FNAME) (NAME FTYPE) (ATOM DRIVE))
          ((if the file FNAME.FTYPE is an already opened file)
               (make FNAME.FTYPE the active COF)
               (SETQ WRS FNAME) )
          ((NULL REOPEN)
               (delete FNAME.BAK if it exists)
               (create FNAME.$TYPE on DRIVE)
               (make FNAME.$TYPE the active COF)
               (zero the file's write pointer)
               (SETQ WRS FNAME) )
          ((if the file FNAME.FTYPE exists on DRIVE)
               (open FNAME.FTYPE for output)
               (make FNAME.FTYPE the active COF)
               (zero the file's write pointer)
               (SETQ WRS FNAME) )
          (create FNAME.FTYPE on DRIVE)
```

```
        (make FNAME.FTYPE the active COF)
        (zero the file's write pointer)
        (SETQ WRS FNAME) )
    (SETQ WRS NIL) ))
```

If <drive> is NIL, the file is opened on the currently logged drive. If the <reopen> flag is NIL, WRS opens a temporary file for output, leaving the original file untouched until the temporary file is successfully closed. This single level **file backup** feature helps prevent a catastrophic loss of information should a disk error occur. The WRS command first deletes the file named <filename>.BAK if it should exist. Then a new file is created with the name <filename>.<filetype> but with the first letter of <filetype> replaced by a "$". Later, when another WRS command is issued to close the file, the temporary file is closed, the original file is renamed <filename>.BAK, and the temporary file is renamed <filename>.<filetype>.

If the <reopen> flag is nonNIL, WRS reopens for output the file named <filename>.<filetype> if it already exists. If it does not exist, a new file is created. This makes it possible to simply update certain parts of a previously existing file rather than having to re-write the entire file. This feature is particularly useful for managing large data base files. **Warning:** There is no file backup feature if the WRS reopen option is used.

If WRS attempts to create a new file and there is insufficient space to store a new directory entry, the "Directory Full" warning message is displayed on the console. BREAK is then called to ask the user how to proceed (see Section 5.18).

If <filename> is NIL, the COF is closed. Note that muLISP programs can reduce the size of an existing file by adjusting the file's write pointer. Files can be deleted by closing the file with its write pointer equal to 0. **Warning:** To save the entirety of a randomly written file, remember to set the write pointer to the end of the file before closing the file.

WRITEPTR [pointer] sets the write pointer of the COF to <pointer> and returns the previous write pointer. If <pointer> is not zero or a positive number, the current write pointer is returned. If <pointer> is the name **EOF** (end-of-file), the size, in characters, of the COF is returned.

```
(PUTD 'WRITEPTR '(LAMBDA (PTR)
    ((if there is no active COF) NIL)
    ((EQ PTR 'EOF)
        (return the file size in bytes) )
    ((AND (NUMBERP PTR) (LESSP -1 PTR (max file size in bytes)))
        (set the COF write pointer to PTR)
        (return the previous COF write pointer) )
    (return the current COF write pointer) ))
```

A file's write pointer determines the point in the file at which the next character will be written. When WRS opens a file for output, the file's write pointer is set to 0. When a character is written to the file, the file's write pointer is automatically incremented. If output to the file must be non-sequential, use WRITEPTR to change the COF's write pointer. Note that WRITEPTR operates on the COF, not the COS, so it can be used even if the console is the COS.

WRITEPTR makes it possible to create files containing "holes" where no characters have been written. An attempt to read characters from such holes will cause an "End-Of-File" error.

PRIN1 [object] sends a character representation of <object> to the COS, using **list notation** where possible and **dotted-pair notation** where necessary (see Chapter 4 for an explanation of these notations). <object> is returned as the value of the function.

```
(PUTD 'PRIN1 '(LAMBDA (OBJ)
     ((NAME OBJ)
          (output OBJ's print-name string to the COS) )
     ((NUMBERP OBJ)
          (output OBJ's numerical value in the current
               radix base) )
     (PRIN1  '"("  )
     (LOOP
          (PRIN1 (POP OBJ))
          ((NULL OBJ))
          (SPACES 1)
          ((ATOM OBJ)
               (PRIN1 '".")
               (SPACES 1)
               (PRIN1 OBJ) ) )
     (PRIN1  '")")
     OBJ ))
```

If the COS is a disk file and there is insufficient space to store the characters being output, the "Disk Full" warning message is displayed on the console. BREAK is then called to ask the user how to proceed (see Section 5.18).

PRINT [object] also sends a character representation of <object> to the COS, and then follows it by a return and a line feed to terminate the line. <object> is returned as the value of the function.

```
(PUTD 'PRINT '(LAMBDA (OBJ)
     (PRIN1 OBJ)
     (TERPRI)
     OBJ ))
```

TERPRI [n] sends <n> carriage return and line feed character pairs to the COS. If <n> is neither 0 nor a positive number, one pair is sent. TERPRI returns NIL.

```
(PUTD 'TERPRI '(LAMBDA (N)
     ((ZEROP N) NIL)
     (PRIN1 (ASCII 13))              ;A carriage return character
     (PRIN1 (ASCII 10))              ;A line feed character
     ((PLUSP N)
          (TERPRI (DIFFERENCE N 1)) ) ))
```

SPACES [n] sends <n> spaces (i.e. blank characters) to the COS. If <n> is not a positive number, no spaces are sent. In either case, the resulting cursor column position is returned.

```
(PUTD 'SPACES '(LAMBDA (N)
     ((PLUSP N)
          (PRIN1 '" ")
          (SPACES (DIFFERENCE N 1)) )
     (return the current cursor column position) ))

(PUTD 'TAB '(LAMBDA (N)              ;A horizontal tab function.
     (SPACES (DIFFERENCE N (SPACES))) ))
```

LINELENGTH [n] sets the length at which output lines will automatically be terminated to <n> characters and returns the previous line length. If <n> is not a positive integer, the current line length is returned.

```
(PUTD 'LINELENGTH '(LAMBDA (N)
     ((PLUSP N)
          (set maximum output line length to N)
          (return the previous line length) )
     (return the current line length) ))
```

The default line length is set as appropriate for the host computer and operating system. The control variable LINELENGTH can be used to override the automatic line termination feature.

For most consoles, the muLISP line length should be made one less than the actual width of the screen. This prevents unwanted blank lines that result from the automatic scrolling of the screen that occurs when a character is displayed on the rightmost column of the screen.

RADIX [n] sets the **radix base** in which numbers are expressed for both input and output, and returns the previous base (which will be expressed in the new radix base). If <n> is not a number between two and thirty-six inclusive, the current radix base is returned. The default radix base is decimal ten.

```
(PUTD 'RADIX '(LAMBDA (N)
     ((LESSP 1 N 37)
          (set the radix base to N)
          (return the previous radix base)
     (return the current radix base) ))

(RADIX 16)        -->  0A              ;Use hexadecimal
(PLUS 3F6 0A4E)   -->  0E44            ;A hex calculator
(RADIX 0A)        -->  16              ;Return to base ten
```

5.14.2 Printer Control Variables

The printer control variables enhance the control muLISP programs have over character output. They function as switches to control the COS, case conversion, printing of quoted strings, and output echoing of characters to the console.

WRS: Normally the function WRS, as described in Section 5.14.1, can provide sufficient control over the destination of character output from muLISP programs. Sometimes, however, after writing part of a file, it may be necessary to send output to the console before returning to the file. After using WRS to open a file and make it the **current output file** (**COF**), the console can be made the **current output sink** (**COS**) without changing the COF by setting the variable WRS to NIL. A subsequent nonNIL assignment to WRS will again make the COF the COS and writing to the file will resume at the point at which it had been suspended. Upon initial system startup and after interrupt or error traps, DRIVER sets WRS to NIL, so the console is the COS.

If the value of WRS is PRINT, the system's hard-copy printer becomes the COS. This makes it possible to direct output to the printer under program control. Note that input, including console input, is not sent to the printer. Thus, to produce a hard copy printout of a muLISP session, use the host operating system's convention for sending what appears on the console to the printer as well. Most operating systems use <CTRL-P> as a toggle switch to send output to the printer.

PRINT: If PRINT is NIL, uppercase letters are converted to lowercase as they are output. Note that this **case conversion** is only done as the characters are being output. It does not affect the actual print-name strings stored in memory. If PRINT is nonNIL, letters are output to the COS as stored. Initially, PRINT has a nonNIL value.

PRIN1: If PRIN1 is NIL, print-names that contain special characters (see Section 5.13) are enclosed within double quote marks when output. When such **quoted strings** are subsequently read back into the system, they will be interned with the same print-

name. This is essential for applications, such as muSTAR, that generate muLISP source files. If PRIN1 is nonNIL, print-names are output without double quotes, unless a double quote happens to be part of the name. Initially, PRIN1 has a nonNIL value.

ECHO: If ECHO is nonNIL and a disk file is the COS, characters sent to the file are also sent to the console. If ECHO is NIL, characters are not echoed to the console. Initially ECHO is NIL. See Section 5.13 for the effect of ECHO when reading from a file.

LINELENGTH: If LINELENGTH is NIL, all automatic line termination is suppressed. This feature is useful for applications in which automatic line terminations may be undesirable, such as the muSTAR screen editor or sending a very large number to a disk file. If LINELENGTH is nonNIL, output lines are automatically terminated as determined by the LINELENGTH function. Initially, LINELENGTH has a nonNIL value.

5.15 EVALUATION FUNCTIONS

The evaluation functions are used for expression and function body evaluation. The algorithm used to evaluate function bodies in muLISP has been enhanced to use the tree structure of the body as the basis for the flow of control. This makes function definitions clean, concise, and efficient.

QUOTE [object] is a no-eval function that simply returns its argument without evaluation.

```
(PUTD QUOTE '(NLAMBDA (OBJ)
     OBJ ))

(SETQ FOO 125)   -->  125
FOO              -->  125
(QUOTE FOO)      -->  FOO

(CAR (CONS 4 7))           -->  4
(CAR (QUOTE (CONS 4 7)))   -->  CONS
```

As the last example shows, QUOTE makes it possible to directly pass a muLISP data object to a function without evaluation. The single quote mark, ', is a primitively defined read macro that can be used as a convenient abbreviation for QUOTE (see Section 5.13).

EVAL [object] evaluates and returns the value of the expression <object>.

```
(PUTD 'EVAL '(LAMBDA (OBJ)
     ((ATOM OBJ) (CAR OBJ))
     ((undefined function (CAR OBJ))
          ((EQ (CAR OBJ) (EVAL (CAR OBJ)))
               (APPLY 'UNDEFINED OBJ) )
          (EVAL (CONS (EVAL (CAR OBJ)) (CDR OBJ))) )
     ((no-eval function (CAR OBJ))
          (APPLY (CAR OBJ) (CDR OBJ)) )
     (APPLY (CAR OBJ) (EVLIS (CDR OBJ))) )

(PUTD 'EVLIS '(LAMBDA (OBJ)
     ((ATOM OBJ) NIL)
     (CONS (EVAL (CAR OBJ)) (EVLIS (CDR OBJ))) ))
```

If <object> is an atom, the content of <object>'s value cell is returned. Note that new names are auto-quoted when they are interned.

If the car of <object> is not a function and does not evaluate to a function, UNDEFINED is called to handle the **undefined function error** (see Section 5.18). If the car of <object> is a no-eval function or if it is the name of a no-eval function, the function is applied to the arguments (i.e. the cdr of <object>) without evaluation. Otherwise, the car of <object> is applied to a list of the evaluated arguments.

APPLY [function, argument-list] applies <function> to <argument-list> and returns the result. If <function> is the name of a machine language routine, control passes to the routine. If <function> is a LAMBDA defined function, the formal arguments of <function> are temporarily bound to the **actual arguments** (i.e. the elements of <argument-list>), the function body is evaluated, the original values of the formal arguments are restored, and the value of the function body evaluation is returned.

```
(PUTD 'APPLY '(LAMBDA (FUNC ARGS)
     ((undefined function FUNC)
          ((EQ FUNC (EVAL FUNC))
               (APPLY 'UNDEFINED FUNC) )
          (APPLY (EVAL FUNC) ARGS) )
     ((machine language function FUNC)
          (execute the machine language routine using
               ARGS for the arguments) )
     (bind FUNC's formal arguments to ARGS)
     (eval-body NIL <FUNC's function body>)
     (restore the original value to FUNC's formal arguments)
     (return the result of the eval-body evaluation) )

(PUTD 'eval-body '(LAMBDA (RSLT BODY)
     ((ATOM BODY) RSLT)
     ((OR (ATOM (CAR BODY)) (ATOM (CAAR BODY)))
          (eval-body (EVAL (CAR BODY)) (CDR BODY)) )
     ((ATOM (CAAAR BODY))
          (SETQ RSLT (EVAL (CAAR BODY)))
          ((NOT RSLT)
               (eval-body NIL (CDR BODY)) )
          (eval-body RSLT (CDAR BODY)) )
     (eval-body (eval-body RSLT (CAR BODY)) (CDR BODY)) ))
```

The linked list used to define a muLISP function consists of a function type specifier, the **formal arguments**, and the function body. The first element of the list is the function type, either LAMBDA or NLAMBDA. A function defined with LAMBDA is called an **eval function.** When an eval function is called, each of the arguments are evaluated and only the resulting values are passed to the function.

A function defined with NLAMBDA is called a **no-eval function.** When a no-eval function is called, the arguments are passed to the function without evaluation just as they were listed in the

call. Since no-eval functions are used much less frequently than eval functions, the novice LISP programmer need not be concerned with learning to define them until the other features of the language have been mastered.

The second element of a function definition is either the function's formal argument list or formal argument name. A function defined with a list of formal arguments, possibly NIL, is called a **spread function.** When a spread function is called, the actual arguments are bound to (i.e. spread over) each element of the formal argument list.

A function defined with a formal argument name, rather than a formal argument list, is called a **no-spread function.** When a no-spread function is called, a list of the actual arguments is bound to the formal argument name. Thus no-spread functions can accept an arbitrary number of actual arguments. Note that in muLISP the decision to make a function spread or no-spread can be made independently of the decision to make it eval or no-eval.

The remaining elements of a function definition comprise the **function body.** A function body is a list of **tasks** to be performed when the function is called. The auxiliary function <eval-body> successively evaluates the tasks of a function body and returns the value of the last task performed. How a task is performed depends on the structure of the task, as follows:

a. If the task is an atom, the value of the task is the value of the atom.

b. If the car of the task is an atom, that atom 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. If the function is an eval function, the arguments are evaluated before the application.

c. If the car of the car of the task is an atom, the car of the task is considered to be a **predicate.** If the value of the predicate is NIL, the value of the task is NIL. However, if the predicate's value is nonNIL, the original function body is abandoned and evaluation proceeds using the cdr of the task as the new function body.

d. 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 a function body to recombine at a later time.

LOOP [$task_1$, $task_2$, ..., $task_n$] iteratively evaluates the tasks until a nonNIL predicate is encountered and returns the result of evaluating the predicate's body.

```
(PUTD 'LOOP '(NLAMBDA TASK-LIST
     (eval-loop TASK-LIST TASK-LIST) ))

(PUTD 'eval-loop '(LAMBDA (FULL-LIST CURR-LIST
          % Local: % RSLT)
     ((ATOM CURR-LIST)
          (eval-loop FULL-LIST FULL-LIST) )
     ((OR (ATOM (CAR CURR-LIST)) (ATOM (CAAR CURR-LIST)))
          (EVAL (CAR CURR-LIST))
          (eval-loop FULL-LIST (CDR CURR-LIST)) )
     ((ATOM (CAAAR CURR-LIST))
          (SETQ RSLT (EVAL (CAAR CURR-LIST)))
          ((NOT RSLT)
               (eval-loop FULL-LIST (CDR CURR-LIST)) )
          (eval-body RSLT (CDAR CURR-LIST)) )
     (eval-body NIL (CAR CURR-LIST))
     (eval-loop FULL-LIST (CDR CURR-LIST)) ))
```

The muLISP evaluation scheme is very powerful but it does not have any provisions for nonrecursive program control structures. Such **iterative** capability is provided by the LOOP function. LOOP evaluates its arguments sequentially in a manner identical to the evaluation of the tasks in a function body (see the discussion following APPLY). However, if all the arguments of LOOP are evaluated without a conditional having been satisfied, evaluation begins again with the first argument. Although the above definition is recursive, in actuality LOOP is a nonrecursive, machine language routine.

The LOOP function returns the value of the last task evaluated following the nonNIL predicate. Note that any number of predicates can occur within a loop, and they can be located at the beginning, middle, or end of the loop. This flexible loop construct preserves the structured nature and clarity of programs written in muLISP while greatly improving their performance. The source files distributed with muLISP contain numerous examples of the LOOP function.

COND [$conditional_1$, $conditional_2$, ..., $conditional_n$) successively evaluates the car of each conditional (i.e. a **predicate**) until either a nonNIL value is encountered or all the predicates have been evaluated. In the former case the cdr of the conditional is evaluated as a function body (see the discussion following APPLY). In the latter case NIL is returned by COND.

```
(PUTD 'COND '(NLAMBDA COND-LIST
     (eval-cond COND-LIST) ))

(PUTD 'eval-cond '(LAMBDA (COND-LIST RSLT)
     ((ATOM COND-LIST) NIL)
     (SETQ RSLT (EVAL (CAAR COND-LIST)))
     ((NOT RSLT)
          (eval-cond (CDR COND-LIST)) )
     (eval-body RSLT (CDAR COND-LIST)) ))
```

```
(SETQ CIS (COND ((NULL WRS) 'CONSOLE)        ;Assigns the current
                (WRS) ))                     ;input source to CIS
```

In muLISP the implicit COND has greatly diminished the need to explicitly use the COND function. However, as the example shows, COND is useful when making a conditional statement within a nest of function calls.

PROG1 [$task_1$, $task_2$, ..., $task_n$] successively evaluates the tasks and returns the result of the evaluation of <$task_1$>.

```
(PUTD 'PROG1 '(NLAMBDA TASK-LIST
     (eval-prog1 (EVAL (CAR TASK-LIST)) TASK-LIST) ))

(PUTD 'eval-prog1 '(LAMBDA (RSLT TASK-LIST)
     ((ATOM TASK-LIST) RSLT)
     (EVAL (CAR TASK-LIST))
     (eval-prog1 RSLT (CDR TASK-LIST)) ))

(SETQ FOO '(A B C D))                      -->  (A B C D)
(PROG1 (CAR FOO) (SETQ FOO (CDR FOO)))     -->  A
FOO                                        -->  (B C D)
```

PROG1 avoids the need for introducing a temporary variable to store a result while other expressions are being evaluated.

CATCH [label, body] returns the value of evaluating <body> as a function body, unless a throw to <label> occurs during the evaluation. If such a throw occurs, evaluation of <body> is terminated and the call to CATCH returns the value specified by the throw.

```
(PUTD 'CATCH '(NLAMBDA ARG-LIST
     ((NULL ARG-LIST) NIL)
     (catch-aux (EVAL (CAR ARG-LIST)) (CDR ARG-LIST)) ))

(PUTD 'catch-aux '(LAMBDA (LABEL BODY RSLT)
     (SETQ RSLT (eval-body NIL BODY))
     ((if throw flag is inactive) RSLT)
     ((OR (NULL LABEL) (EQ LABEL (throw's label)))
          (deactivate throw flag)
          THROW ) )
     RSLT ))
```

The <label> argument in a call to CATCH is evaluated before the <body> is evaluated. If <label> is NIL, all throws from the evaluation of <body> are caught by the CATCH. The top-level muLISP executive driver loop also catches all throws and deactivates the throw flag.

THROW [label, expression] terminates execution of the current function body and "throws" control back to the most recent CATCH with the same <label>. <expression> is returned as the value of that CATCH.

```
(PUTD 'THROW '(LAMBDA (LABEL EXPN)
     (SETQ THROW EXPN)
     (activate the throw flag)
     (unwind the variable and control stacks until a CATCH with
          a matching label or the NIL label is found, or the
          top-level muLISP executive loop is reached) ))
```

```
(CATCH 'FOO (PRINT 'DOG)
            (CATCH 'BAR (PRINT 'CAT)
                        (THROW 'FOO 'COW)
                        (PRINT 'PIG) )
            (PRINT 'RAT) )
DOG
CAT
COW
```

The <label> and <expression> arguments in a call to THROW are evaluated before the throw occurs. The global variable named THROW is assigned the value <expression> so that CATCH will be able to return <expression> as its value.

COMMENT [comments] ignores its arguments and returns NIL.

```
(PUTD 'COMMENT '(NLAMBDA () NIL))
```

```
(PUTD 'FOO '(LAMBDA (X)
     (COMMENT The definition of FOO contains a comment that
          refers to itself!)
     (CONS X X) ))
```

COMMENT provides a means of including comments within muLISP function definitions. When a function's definition is to be written out to a file, COMMENT function comments are automatically written out along with the rest of the definition, since the comments are part of the function body. They are normally used when developing muLISP programs using the muSTAR resident editor.

In contrast, single and multi-line comments, delimited by comment characters, are ignored by the muLISP reader when they are read in (see Section 5.13). These comments do not require storage space and, therefore, are not saved when a function is written out. They are normally used when developing muLISP programs using an external text editor.

5.16 MEMORY MANAGEMENT FUNCTIONS

The function RECLAIM forces a garbage collection and returns the resulting amount of free data space. Since memory management is fully automatic in muLISP, RECLAIM is normally only used to determine the amount of free space. The muLISP memory management system is described in Chapter 4.

RECLAIM [] forces a garbage collection to reclaim the inactive data structures in the four muLISP data spaces. If required, a memory reallocation is also performed. The total number of bytes available in the atom, vector, pointer, and stack spaces is returned.

```
(PUTD 'RECLAIM '(LAMBDA ()
     ( ((NULL RECLAIM)
          (display pre-GC statistics) ) )
     (perform a garbage collection on all data spaces)
     ( ((NULL RECLAIM)
          (display post-GC statistics) ) )
     ( ((if the available free space is not optimally proportioned
                between the four data spaces)
          (optimally reallocate free memory among the spaces)
          ((NULL RECLAIM)
                (display post-allocation statistics) ) ) )
     (return the amount of free space expressed in bytes) ))

(RECLAIM)  -->  94968                    ;The amount of free space.
```

RECLAIM: If the control variable RECLAIM is NIL, informative statistics are displayed on the console before and after all garbage collections and memory reallocations. The first line of the statistics shows the available space just before the garbage collection. The second line shows the available space after the collection. If a third line is displayed, it indicates that there has been a data space reallocation and shows the available space after the reallocation. The format for a line of statistics is as follows:

GC: nnnn aaaa/aaaa vvvv/vvvv pppp/pppp ssss/ssss tttt/tttt

<nnnn>, <aaaa>, <vvvv>, <pppp>, <ssss>, and <tttt> denote hexadecimal numbers. <nnnn> is the number of garbage collections since muLISP began execution. The first <aaaa>, <vvvv>, <pppp>, and <ssss> of each pair of numbers is the amount of free space in bytes in the atom, vector, pointer, and stack spaces respectively. Chapter 4 describes the purpose of each of these spaces. The first <tttt> of the pair is the total amount of free space (i.e. the sum of <aaaa>, <vvvv>, <pppp>, and <ssss>). The numbers following the "/" are the total size of the respective spaces, including both the free and used space.

Two bytes are required for each muLISP pointer cell in the atom, pointer, and stack spaces. One byte is required for each character in print-name strings and one byte is required for the binary zero used to terminate strings. Number vectors use only as many bytes as necessary to store the number in binary.

These statistics are heavily dependent on the implementation of muLISP being run and should normally be of no concern to users. However, they are sometimes useful for determining what is causing the system to "thrash" (i.e. to conduct an excessive number of garbage collections for very little free space returned). The statistics are also displayed when a **Memory Space Exhausted error** occurs to show which space was responsible (see Chapter 4).

```
(SETQ RECLAIM NIL)                    ;Turn on GC statistics flag
(LOOP (OBLIST))                       ;Use lots of nodes

GC: 0001  2258/2600  1C8C/1C8E  00000/119C0  1AAA/1AC0  0598C/1770E
GC: 0001  2258/2600  1C8C/1C8E  117D0/119C0  1AAA/1AC0  1715C/1770E
GC: 0001  09C0/0D68  09C2/09C4  13CFE/13EEE  20DE/20F4  1715C/1770E

GC: 0002  09C0/0D68  09C2/09C4  00002/13EEE  20DE/20F4  03460/1770E
GC: 0002  09C0/0D68  09C2/09C4  13E1A/13EEE  20DE/20F4  17278/1770E

GC: 0003  09C0/0D68  09C2/09C4  00002/13EEE  20DE/20F4  03460/1770E
GC: 0003  09C0/0D68  09C2/09C4  13E1A/13EEE  20DE/20F4  17278/1770E
```

The three lines of statistics for garbage collection #1 indicate that a memory reallocation occurred. More memory was allocated to the pointer space because of the heavy demand by OBLIST for new nodes. Subsequent garbage collections did not perform reallocations because the data space sizes were already optimized. However, if the program changed and started using the atom space, for example, another reallocation would occur to expand the size of that region at the expense of the others.

This example was done using the 8086/8088 version of muLISP running in 128 kilobytes of memory. The numbers shown above will vary greatly depending on the host computer's memory size.

5.17 ENVIRONMENT FUNCTIONS

The current muLISP **environment** consists of all the variable values, function definitions, and property values presently in the system. These values and definitions could have been entered directly from the console, read from a source file, or loaded from a SYS file. The environment functions make it possible to load and save muLISP environments as compact, memory image **SYS files.** A SYS file is usually created after a program has been thoroughly debugged. Loading a SYS file is much faster than reading in the equivalent source files.

SAVE [filename, drive, flag], if <flag> is NIL, saves the current muLISP environment as <filename>.SYS on <drive>. If <flag> is nonNIL, it saves the muLISP interpreter along with the current environment as <filename>.COM on <drive>.

```
(PUTD 'SAVE '(LAMBDA (FNAME DRIVE FLAG)
     ((NULL DRIVE)
          (SAVE FNAME (currently logged drive) FLAG) )
     (close all currently open input and output files)
     ((AND (NAME FNAME) (ATOM DRIVE))
          (RECLAIM)
          ((NULL FLAG)
               (save a compact memory image of the current
                    environment as FNAME.SYS on DRIVE)
               T )
          (disable the muLISP pseudo-code compiler)
          (save muLISP along with the current environment as
               FNAME.COM on DRIVE)
          T ) ))

(SAVE 'WHALE 'B)                ;Saves WHALE.SYS on drive B:
(SAVE 'WHALE NIL T)             ;Saves WHALE.COM on logged drive
```

If SAVE is called without a second argument, the currently logged drive is used. Prior to saving a system, all the active data structures are automatically compacted so the resulting SYS file is of minimum size. Therefore, the size of the SYS file increases with the amount of active data structures in the system at the time the SAVE command is issued.

If <flag> is nonNIL, a COM file is generated that contains the muLISP interpreter and the current environment. This COM file can be executed as an operating system command. To prevent users of the COM file from defining any more functions using PUTD, muLISP's pseudo-code compiler is disabled. For this reason, SYS files rather than COM files should be used for the routine saving of muLISP environments.

COM files allow programmers to sell applications written in muLISP to end-users without requiring the end-users to pay for a full muLISP system. Instead, the application programmer pays a small royalty for the use of the muLISP system (minus the

pseudo-code compiler) that is contained within the COM file. To obtain a License Agreement permitting you to distribute the muLISP Runtime System imbedded in your application program, contact The Soft Warehouse, P.O. Box 11174, Honolulu, Hawaii, 96828-0174, U.S.A.

LOAD [filename, drive] restores the muLISP environment present at the time <filename>.SYS was created.

```
(PUTD 'LOAD '(LAMBDA (FNAME DRIVE)
     ((NULL DRIVE)
          (LOAD FNAME (currently logged drive)) )
     (close all currently open input and output files)
     ((AND (NAME FNAME) (ATOM DRIVE))
          ((if FNAME.SYS exists on DRIVE)
               (load the environment stored in FNAME.SYS)
               (begin execution with the DRIVER function) ) ) ))

(LOAD 'WHALE 'B)                    ;Loads WHALE.SYS from drive B:
```

If LOAD is called without a second argument, the SYS file is loaded from the currently logged drive. LOAD does not return a value if the SYS file is successfully loaded, but begins execution of the DRIVER function that has just been loaded. If the SYS file is not found, NIL is returned.

SYSTEM [byte-value] closes all open files, terminates execution of muLISP, and returns control to the host operating system.

If running under MS-DOS version 2.00 or better, muLISP can return an exit code (also called a return code) of <byte-value> to its host. The exit code can be interrogated by the batch command "IF ERRORLEVEL" (see the Batch Processing Commands section of the MS-DOS Reference Manual for details). If no <byte-value> is specified, muLISP returns the default exit code stored in the 8086 Base Page (see Appendix B).

```
(PUTD 'SYSTEM '(LAMBDA (BYTE)
     (close all currently open input and output files)
     ((LESSP -1 BYTE 256)
          (exit to host returning the exit code BYTE) )
     (exit to the host operating system) ))
```

If while typing a line of text you enter a <CTRL-C>, the prompt

```
Are you sure (Y/N)?
```

will be displayed on the screen. If you respond in the affirmative, all open files are closed and control returns to the host operating system. Otherwise, you can continue typing as before.

5.18 USER INTERFACE FUNCTIONS

5.18.1 Executive Driver and Break Functions

The basic user interface to muLISP is provided by the <read-eval-print-loop> driver routine. It is used by the top level muLISP executive and by the break debugging package. This internal, machine language driver routine is only indirectly accessible to the user via the functions DRIVER and BREAK.

```
(PUTD 'read-eval-print-loop '(LAMBDA (expr)
     (flush the line-edit buffer)
     (LOOP
          ( ((ZEROP break-level))
            (PRIN1 break-level) )          ;Print break level.
          (PRIN1 '"$ ")                    ;Print prompt.
          (SETQ expr (READ))               ;Read an expression.
          ((EQ expr 'RETURN))              ;Exit if 'RETURN.
          ( ((EQ expr 'EOF)
               (SETQ RDS NIL) )
            (PRINT (CATCH NIL (EVAL expr))) )    ;Eval & print.
     (SETQ expr (READ))
     (flush the line-edit buffer)
     (EVAL expr) ))
```

After flushing (i.e. discarding) any text remaining in the console's line-edit buffer, this routine enters the driver loop. If <break-level> is not 0 (i.e. not at the top-most driver loop), <break-level> is displayed immediately preceding the muLISP dollar sign prompt. The system reads in the next object from the CIS, evaluates it, and outputs the result to the COS. Note that all throws are caught by the driver loop. The loop is repeated until RETURN is entered in response to the prompt.

After exiting the loop, a final expression is read in, the line-edit buffer is flushed, and the result of evaluating the expression is returned to the function that called <read-eval-print-loop>.

DRIVER [] establishes the console as the current input source (CIS) and current output sink (COS), sets ECHO to NIL, and calls <read-eval-print-loop>.

```
(PUTD 'DRIVER '(LAMBDA ()
     (SETQ RDS NIL)                        ;Make console CIS.
     (SETQ WRS NIL)                        ;Make console COS.
     (SETQ ECHO NIL)                       ;Turn off echo flag.
     (read-eval-print-loop) ))             ;Driver loop.
```

The top level muLISP executive sets <break-level> to 0, deactivates the throw flag, and calls the current definition of DRIVER. The default DRIVER function is a machine language

routine. However, as with any other function, the user can redefine DRIVER to suit the application. Once a new DRIVER function has been defined, type RETURN and NIL following the dollar sign prompt. As inspection of the definition of <read-eval-print-loop> shows, this will exit the default driver loop and return control to top level muLISP executive. This in turn will call your new DRIVER function. For example, some LISP users may prefer the following **eval-quote LISP** driver loop:

```
$ (PUTD 'DRIVER '(LAMBDA ()
     (SETQ RDS NIL)
     (SETQ WRS NIL)
     (SETQ ECHO NIL)
     (LOOP
          (TERPRI)
          (PRIN1 '">")
          (SPACES 1)
          (PRINT (APPLY (READ) (READ))) ) ))

$ RETURN NIL

> CONS (DOG (CAT COW PIG))
(DOG CAT COW PIG)
```

BREAK [object] breaks program execution and provides the user with options on how to proceed.

```
(PUTD 'BREAK '(LAMBDA (BREAK
       % Local: % RDS WRS READCH char)
     (TERPRI)
     (PRIN1 "Continue, Executive, Break, Restart, System? ")
     (LOOP
          (PRIN1 (ASCII 7))
          (SETQ char (READCH))
          ((EQ char 'C) BREAK)
          ((EQ char 'B)
               (SETQ READCH 'READCH)
               (SETQ break-level (PLUS break-level 1))
               (SETQ BREAK (read-eval-print-loop))
               (SETQ break-level (DIFFERENCE break-level 1))
               BREAK )
          ((EQ char 'E)
               (THROW NIL BREAK) )
          ((EQ char 'R)
               (restart a fresh muLISP system) )
          ((EQ char 'S)
               (SYSTEM) ) ) ))
```

RDS and WRS are set to NIL, to establish the console as the CIS and COS. READCH is set to NIL so console input will be in the raw-input mode. Then BREAK displays the break **option prompt**

Continue, Break, Executive, Restart, System?

and waits for the user to select an option by entering its first letter (C, B, E, R, or S):

Continue: continues execution as before with BREAK's argument returned as the value of the function that caused the error.

Break: temporarily suspends execution for a break. READCH is set to 'READCH so console input will be in the line-edit mode and the internal counter, <break-level> is incremented. The <read-eval-print-loop> driver routine is called to permit the user to examine the environment present at the time the break occurred. The driver routine displays the break level in the break prompt. By examining the value of the variable BREAK, you can determine what caused the error.

To exit the break and return to the calling program, following the break prompt type the word RETURN and an expression to be evaluated, and press the <RETURN> key. Upon exit from the driver routine, <break-level> is decremented; RDS, WRS, READCH, and BREAK are restored to their original values; and the value of the expression you entered following RETURN is returned as the value of BREAK.

Executive: throws control to the most recent catch of NIL, which is usually the executive driver loop.

Restart: abandons the current muLISP environment and starts up a fresh muLISP system.

System: closes all open files, terminates execution of muLISP, and returns control to the host operating system.

When an error condition is detected by muLISP, an error handling function is called. The error handler displays an appropriate error message and then calls BREAK to let the user decide how to proceed. BREAK and/or the individual error handlers can be redefined by the user to suit the application.

By inserting calls to BREAK at key points within function definitions, the break package makes an excellent debugging tool. It permits you to examine and change variable values and function definitions before proceeding with a computation.

5.18.2 Error Handling Functions

Error handling functions are invoked when an error condition is detected while running a muLISP program. The error handlers display a descriptive diagnostic message on the console and then call BREAK to ask the user how to proceed. The value returned by BREAK is the value returned by the function that detected the error.

If necessary, the error handler functions can be redefined to better suit the application. The only errors that cannot be handled by the user are those caused by the (unfortunately) finite size of computer memory resources: the memory space exhausted, disk full, and directory full errors.

UNDEFINED [list] displays the "Undefined Function" error message on the console screen and calls BREAK with <list> as the argument.

```
(PUTD 'UNDEFINED '(NLAMBDA LST
     (TERPRI)
     (PRIN1 "Undefined Function")
     (BREAK LST) ))
```

This error handler is invoked when an attempt is made to apply a name that has not been assigned a function definition. The list being evaluated is passed as the argument to BREAK.

NONNUMERIC [function, arguments] displays the "Nonnumeric Argument" error message on the console screen and calls BREAK with (CONS <function> <arguments>) as the argument.

```
(PUTD 'NONNUMERIC '(LAMBDA (FUNC ARGS)
     (TERPRI)
     (PRIN1 "Nonnumeric Argument")
     (BREAK (CONS FUNC ARGS)) ))
```

This error handler is invoked when PLUS, MINUS, DIFFERENCE, TIMES, QUOTIENT, REMAINDER, DIVIDE, GCD, GREATERP, or LESSP is called with a nonnumeric argument.

ZERODIVIDE [function, arguments] displays the "Zero Divide" error message on the console screen and calls BREAK with (CONS <function> <arguments>) as the argument.

```
(PUTD 'ZERODIVIDE '(LAMBDA (FUNC ARGS)
     (TERPRI)
     (PRIN1 "Zero Divide")
     (BREAK (CONS FUNC ARGS)) ))
```

This error handler is invoked when QUOTIENT, REMAINDER, or DIVIDE is called with 0 as the second argument.

EOF [] displays the "End-Of-File" error message on the console screen and calls BREAK with 'EOF as the argument.

```
(PUTD 'EOF '(LAMBDA ()
     (TERPRI)
     (PRIN1 "End-Of-File")
     (BREAK 'EOF) ))
```

This error handler is invoked when a file is the CIS (current input source) and READ, RATOM, or READCH attempt to read past the end of the file.

5.19 HARDWARE INTERFACE FUNCTIONS

The functions in this section facilitate communication between muLISP programs and the host computer's hardware and operating system. The computer's technical reference and operating system manuals contain the information required to make full use of these muLISP functions.

MEMORY [address, byte-value] stores <byte-value> at memory location <address> and returns the original contents of <address>.

```
(PUTD 'MEMORY '(LAMBDA (ADDR BYTE)
     ((LESSP -1 ADDR (the address space of the processor))
          ((LESSP -1 BYTE 256)
               (replace the contents of ADDR with BYTE)
               (return the original contents of ADDR) )
          (return the contents of ADDR) ) ))
```

```
(RADIX 16)          -->  0A              ;Switch to hexadecimal
(MEMORY 100)        -->  0C3             ;Examine 100H
(MEMORY 100 41)     -->  0C3             ;Store 41H at 100H
(MEMORY 100)        -->  41              ;Examine 100H
(MEMORY 100 0C3)    -->  41              ;Restore original value
(RADIX 0A)          -->  16              ;Return to decimal
```

This function permits any computer memory location to be examined or changed. If no second argument is given or if it is not a single byte number, only the value of the location is returned.

> **WARNING: The MEMORY function should be used with care since there is no protection against inadvertently destroying the muLISP program or the host operating system.**

PORTIO [port-number, byte-value] outputs <byte-value> to <port-number>, if a <byte-value> is given. Otherwise, it inputs a byte from <port-number> and returns its integer value.

```
(PUTD 'PORTIO '(LAMBDA (PORT BYTE)
     ((LESSP -1 PORT (the number of I/O ports for the processor))
          ((LESSP -1 BYTE 256)
               (output BYTE to PORT)
               NIL )
          (input a byte value from PORT) ) ))
```

```
(RADIX 16)          -->  0A              ;Switch to hexadecimal
(PORTIO 72)         -->  <?>             ;Input byte from port 72H
(PORTIO 72 10)      -->  NIL             ;Output 10H to port 72H
(RADIX 0A)          -->  16              ;Return to decimal
```

This function is used to send byte values to the output ports, and to receive byte values from input ports. This can be

useful for muLISP programs that need to communicate with external devices such as sensors, robots, etc.

REGISTER [n, m] stores <m> in muLISP pseudo-register <n>, if <m> is a nonnegative integer less than 65536, and returns NIL. If <m> is not such an integer, the content of pseudo-register <n> is returned.

```
(PUTD 'REGISTER '(LAMBDA (N M)
     ((LESSP -1 N 10)
          ((LESSP -1 M 65536)
               (store M in register N)
               NIL )
          (the integer value of register N) ) ))

(REGISTER 2)       -->  0                ;Examine 2's contents
(REGISTER 2 100)   -->  NIL              ;Store 100 in register 2
(REGISTER 2)       -->  100              ;Examine 2's contents
```

If muLISP is running on an 8086 or 8088 based computer, the default value of Pseudo-register 0 is the muLISP code segment base paragraph; all other pseudo-registers are 0. If muLISP is running on another microprocessor, the default value for all the pseudo-registers is 0.

INTERRUPT [n] loads the microprocessor's registers from the muLISP pseudo-registers, executes interrupt number <n>, stores the register values returned by the interrupt in the pseudo-registers, and returns NIL. **Note:** INTERRUPT is only available if muLISP is running on an 8086 or 8088 based computer.

There are 10 muLISP pseudo-registers numbered 0 through 9. They correspond to the 8086 registers as follows:

Pseudo-register:	0	1	2	3	4	5	6	7	8	9
8086 register:	AX	BX	CX	DX	SI	DI	BP	DS	ES	FLAG

The functions REGISTER and INTERRUPT enable muLISP programs to communicate with external **machine language** routines via the 256 interrupt vectors provided by the 8086 microprocessor. The computer's technical reference and operating system manuals will describe the services provided by the interrupt vectors.

For example, Interrupt 33 (i.e. 21H) is used by MS-DOS^{tm} and IBM-DOS for system calls. If register AH is 44 (i.e. 2AH), Interrupt 33 will return the year in CX (1980-2099), the month in DH (1-12), and the day in DL (1-31).

```
(RADIX 16)                 -->  0A        ;Hexadecimal
(REGISTER 0 2A00)          -->  NIL       ;Sets AH to 44
(INTERRUPT 21)             -->  NIL       ;Execute interrupt 33
(RADIX 0A)                 -->  16        ;Decimal
```

```
(REGISTER 2)                  -->  1983       ;Year
(DIVIDE (REGISTER 3) 256)     -->  (6 . 1)    ;Month and day
```

CALL [n] loads the microprocessor's registers from the muLISP pseudo-registers, calls location <n>, and stores the register values returned by the interrupt in the pseudo-registers. **Note:** CALL is only available if muLISP is running on an 8080, 8085, or Z80 based computer.

There are 10 muLISP pseudo-registers numbered 0 through 9. The first four correspond to the 8080 registers as follows:

Pseudo-register:	0	1	2	3
8080 register:	BC	DE	HL	PSW

The functions REGISTER and CALL make it possible for muLISP programs to communicate with external **machine language** routines via machine language calls. The computer's technical reference manuals will describe how to make operating system calls.

For example, memory location 5 is used by CP/M-80^{tm} for system calls based on the value of register C. If C is 3, a call to location 5 will input an ASCII character from the logical reader device and return it in the A register.

```
(REGISTER 2 3)                -->  NIL        ;Sets C reg to 3
(CALL 5)                      -->  NIL        ;Input reader char
(QUOTIENT (REGISTER 3) 256)   -->  <?>        ;Contents of A reg
```

EXECUTE [program, command line] compacts and suspends muLISP, then loads and executes <program> passing it <command line> as argument. When <program> terminates, control reverts to muLISP and EXECUTE returns <program>'s exit code. If <program> is not found, EXECUTE returns NIL. **Note:** EXECUTE is only available if muLISP is running on an 8086 or 8088 based computer that is operating under **MS-DOS version 2.00** or better.

A common use for EXECUTE is to invoke a secondary command processor. (See the "Invoking A Secondary Command Processor" section of the MS-DOS Reference Manual for details.) For example, if the command processor file COMMAND.COM is on drive A:, the muLISP command

```
(EXECUTE  "A:COMMAND.COM"  "/C DIR B:")
```

will display the file directory on drive B. Using EXECUTE, the full range of facilities provided by the operating system and application programs can be accessed, all from within muLISP. For instance, another copy of muLISP can be loaded and run.

5.20 SPECIAL FUNCTIONS

5.20.1 Console Screen Functions

The console screen functions CLRSCRN and CURSOR provide a standard convention enabling muLISP programs to perform full screen operations on different console screens. The CONSOLE files distributed with muLISP contain definitions of these functions for several different computers and console terminals. If muLISP is being run on the IBM PC, these two functions are already primitively defined in machine language.

CLRSCRN [] clears the console screen and moves the cursor to the top left corner of the screen (i.e. the "home" position).

```
(PUTD 'CLRSCRN '(LAMBDA ()
     (clear the display screen)
     T ))
```

This function is useful for clearing the console screen before full screen operations.

CURSOR [row, col] repositions the cursor to the specified row and column on the screen if <row> and <col> are within range.

```
(PUTD 'CURSOR '(LAMBDA (ROW COL)
     ((AND (LESSP -1 ROW 24) (LESSP -1 COL 80))
          (position the cursor to row ROW and column COL) ) ))
```

The top left corner of the screen is row 0, column 0. The maximum value of <row> is one less than the number of rows on the screen. The maximum value of <col> is one less than the number of columns on the screen.

IBM PC version only: the built-in CURSOR function returns cursor's current row number. To determine the cursor's current row number without moving the cursor, call CURSOR without any arguments.

5.20.2 Real-time Clock Function

The function TIME is only available if muLISP is running under an operating system, like MS-DOS, which provides support for a real-time clock.

TIME [reset-flag] returns the elapsed time since the system clock was last reset, expressed in 100ths of a second. If <reset-flag> is nonNIL, the clock is reset to 0.

```
(PUTD 'TIME '(LAMBDA (RESET)
     ((NULL RESET)
          (return the time in 100ths of a second) )
     (set the time counter to time 0)
     0 ))

(TIME T) (RECLAIM) (TIME)  -->  50      ;Time required for GC in
                                        ;100ths of a second.
```

TIME can be used to <u>bench mark</u> programs and help determine the most efficient <u>algorithm</u> to accomplish a particular task. Resetting the time counter to zero does not affect the operating system's time counter, only an internal muLISP time counter.

5.20.3 Apple II Functions

A special implementation of muLISP has been made for Apple II systems that have a Z80 SoftCard[tm]. The softcard version of muLISP has additional primitively defined functions to take advantage of the computer's special hardware. Low-resolution graphics, the game control paddles, and the speaker are supported in a manner very similar to BASIC.

TEXT [] puts the Apple screen in full-screen text mode and moves the cursor to the last line of the screen.

```
(PUTD 'TEXT '(LAMBDA ()
     (set the screen to full-screen text mode: 40 columns by
           24 lines)
     (move the cursor to the last line of the screen)
     NIL ))
```

TEXT is used to return to the normal text mode following the use of the screen in the graphics mode. Like the BASIC version of the TEXT command, it does not clear the screen.

GRAPHICS [mode, color] clears the screen using color number <color> and sets up the screen for low-resolution graphics operations.

```
(PUTD 'GRAPHICS '(LAMBDA (MODE COLOR)
     ((NULL MODE)
          ((AND (NUMBERP COLOR) (LESSP -1 COLOR 16))
               (set the screen for mixed mode, 40x40,
                    low-resolution graphics)
               (fill the screen with the COLOR color number)
               (move the cursor to the four line text window)
               NIL )
          (set the screen for mixed mode, 40x40, low-
               resolution graphics)
          (clear the screen)
          (move the cursor to the four line text window)
          NIL )
     ((AND (NUMBERP COLOR) (LESSP -1 COLOR 16))
          (set the screen for full-screen mode, 40x48,
               low-resolution graphics)
          (fill the screen with the COLOR color number)
          NIL )
     (set the screen for full-screen mode, 40x48, low-
          resolution graphics)
     (clear the screen)
     NIL ))
```

If the first argument to GRAPHICS is NIL, a four line window at the bottom of the screen is left in the text mode as a programming convenience. Otherwise full-screen mode graphics is established. The second argument to GRAPHICS controls the color used to clear the screen. If it is not a valid color

number, black is assumed. Note that when GRAPHICS is called with no arguments (e.g. (GRAPHICS)) it has the same effect as the APPLESOFT BASIC **GR** command.

PLOT [x, y, color] plots a low-resolution pixel using the color numbered <color> at coordinates {x, y}. muLISP conveniently combines in its single PLOT function the low-resolution graphics capability provided in APPLESOFT BASIC by the **COLOR, PLOT, HLIN, VLIN** and **SCRN** commands. In addition, rectangles of arbitrary dimension, location, and color can easily be painted.

```
(PUTD 'PLOT '(LAMBDA (X Y COLOR)
     ((AND (NUMBERP COLOR) (LESSP -1 COLOR 16))
          ((AND (NUMBERP X) (LESSP -1 X 40))
               ((AND (NUMBERP Y) (LESSP -1 Y 48))
                    (place a pixel of color number COLOR at
                         coordinates {X, Y})
                    T )
               ((AND (LESSP -1 (CAR Y) 48) (LESSP -1 (CADR Y) 48))
                    (draw a vertical line of color number COLOR
                         at x-coordinate X and between the
                         y-coordinates (CAR Y) and (CADR Y))
                    T ) )
          ((AND (LESSP -1 (CAR X) 40) (LESSP -1 (CADR X) 40))
               ((AND (NUMBERP Y) (LESSP -1 Y 48))
                    (draw a horizontal line of color number
                         COLOR at y-coordinate Y, between the
                         x-coordinates (CAR X) and (CADR X))
                    T )
               ((AND (LESSP -1 (CAR Y) 48) (LESSP -1 (CADR Y) 48)
                    (draw a rectangle of color number COLOR
                         between the x-coordinates (CAR X) and
                         (CADR X), and between the y-coordinates
                         (CAR Y) and (CADR Y)
                    T ) ) )
     ((AND (NUMBERP X) (LESSP -1 X 40)
               (NUMBERP Y) (LESSP -1 Y 48))
          (the color number of the pixel at {X, Y}) ) ))
```

The first two arguments of the PLOT function are used to specify the <x> and <y> coordinates on the 40 by 48 graphics screen. If these arguments are positive integers, they specify a single point on the screen. However, if either or both of them are lists consisting of two positive integers, they indicate a range of values for the respective coordinates. This range is then used for the plot and provides a means for drawing horizontal lines, vertical lines, and rectangles.

The optional third argument to PLOT is used to specify an Apple II color. If no third argument is given or if it is not a valid color number, the PLOT function simply returns the current color number of the point specified by the first two arguments (i.e. {x, y}). This provides the equivalent of the

SCRN command in BASIC. However if it is a color number, the function will "plot" a graphics point of that color at those coordinates.

The following is a list of the available colors and associated numbers:

0 black	4 dark green	8 brown	12 light green
1 magenta	5 grey 1	9 orange	13 yellow
2 dark blue	6 medium blue	10 grey 2	14 aquamarine
3 purple	7 light blue	11 pink	15 white

Example: The following function continuously plots points of a given COLOR on the screen using the game paddles to determine the X and Y coordinates:

```
(PUTD 'PLOTTER '(LAMBDA (COLOR)
     (LOOP
          (PLOT (QUOTIENT (PADDLE 0) 7)
                (QUOTIENT (PADDLE 1) 6)
                COLOR ) ) ))
```

PADDLE [n] returns the current position of game control paddle <n> as an integer between 0 and 255 inclusive.

```
(PUTD 'PADDLE '(LAMBDA (N)
     ((AND (NUMBERP N) (LESSP -1 N 4))
          (the current value of game control paddle N) ) ))
```

The value returned by PADDLE is the same as that of the **PDL** function found in Apple BASICs. The problem described in the APPLESOFT Manual of too rapidly reading different paddles has been resolved. Thus **no** artificial delay is required to obtain accurate readings.

BEEP [pitch, duration] creates a tone of pitch <pitch> lasting <duration> long.

```
(PUTD 'BEEP '(LAMBDA (PITCH DURATION)
     ((AND (NUMBERP PITCH) (LESSP -1 PITCH 256)
                (NUMBERP DURATION) (LESSP -1 DURATION 256))
          (create a tone of pitch PITCH and duration DURATION)
          T ) ))
```

The arguments to BEEP should be numbers between 0 and 255. The first argument specifies the pitch of the tone; 0 being the highest pitch, 255 being the lowest. The second argument specifies the duration of the tone; 0 being shortest, 255 being the longest (approximately a second). This function is intended for making sound effects only. No attempt has been made to match specific pitches with specific musical notes.

Example: The following loop produces a continuous tone whose pitch and duration are controlled by the game paddle controls:

```
(LOOP (BEEP (PADDLE 0) (PADDLE 1)))
```

Notes to Apple II SoftCard Users

In order to take full advantage of the Apple II hardware, there are a few minor differences between the standard CP/M version of muLISP and the SoftCard version:

Since the standard Apple keyboard has no way of generating the left bracket (i.e. "["), the SoftCard version of CP/M automatically translates **<CTRL-K>** into the **left bracket.** This is useful for entering the left super-parentheses described in the Reader Functions section of Chapter 5. However, it also means <CTRL-K> is no longer available for muLISP programs such as muSTAR. Thus, the Apple II version of muSTAR uses **<SHIFT CTRL-N>** in all places where **<CTRL-K>** occurs in the muSTAR chapter of this manual.

Similarly, SoftCard CP/M translates <CTRL-B> into the back slash (i.e. "\") character. Thus, **<CTRL-I>** should be used in all places where **<CTRL-B>** occurs in the muSTAR chapter of this manual.

As described in Chapter 5 of the **Apple II Reference Manual**, the Apple has a large number of "special memory locations" which are used to access the built-in I/O hardware. The muLISP **MEMORY** function can be used to read or toggle these locations as required. Since the SoftCard's **memory address translation** alters the locations of these I/O functions, the addresses given in the Apple II manual for them are invalid. The following table lists some of the special memory locations and gives both their original 6502 address and the Z80 address in decimal and hexadecimal.

I/O Function	6502 Address Hex	6502 Address Decimal	Z80 Address Hex	Z80 Address Decimal
Keyboard Data Input	$C000	49152	$E000	57344
Clear Keyboard Strobe	$C010	49168	$E010	57360
Cassette Output	$C020	49184	$E020	57376
Speaker Toggle	$C030	49200	$E030	57392
Utility Strobe	$C040	49216	$E040	57408
Set GRAPHICS Mode	$C050	49232	$E050	57424
Set TEXT Mode	$C051	49233	$E051	57425
No Mix TEXT and GRAPHICS	$C052	49234	$E052	57426
Mix TEXT and GRAPHICS	$C053	49235	$E053	57427
Display Primary Page	$C054	49236	$E054	57428
Display Secondary Page	$C055	49237	$E055	57429
Display LO-RES graphics	$C056	49238	$E056	57430
Display HI-RES graphics	$C057	49239	$E057	57431
Annunciator Output #0 OFF	$C058	49240	$E058	57432
Annunciator Output #0 ON	$C059	49241	$E059	57433
Annunciator Output #1 OFF	$C05A	49242	$E05A	57434
Annunciator Output #1 ON	$C05B	49243	$E05B	57435
Annunciator Output #2 OFF	$C05C	49244	$E05C	57436
Annunciator Output #2 ON	$C05D	49245	$E05D	57437
Annunciator Output #3 OFF	$C05E	49246	$E05E	57438
Annunciator Output #3 ON	$C05F	49247	$E05F	57439
Cassette Input	$C060	49248	$E060	57440
Pushbutton Input #1	$C061	49249	$E061	57441
Pushbutton Input #2	$C062	49250	$E062	57442
Pushbutton Input #3	$C063	49251	$E063	57443
Game Control (Paddle) #1	$C064	49252	$E064	57444
Game Control (Paddle) #2	$C065	49253	$E065	57445
Game Control (Paddle) #3	$C066	49254	$E066	57446
Game Control (Paddle) #4	$C067	49255	$E067	57447

6. THE muSTAR AI DEVELOPMENT SYSTEM

muLISP source files can be generated using a standard text editor suitable for your computer and disk operating system. Program development using an editor external to muLISP consists of repeatedly editing, loading, and testing source files. The principal advantage of an **external editor** is that you are already experienced in its use. Although this is a 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 by the **muSTAR Artificial Intelligence Development System.** muSTAR greatly reduces program development time by taking advantage of the interactive nature of LISP. Using the **resident editor** and **tracing facilities** provided by muSTAR, function definitions can be created, tested, and thoroughly debugged all within muLISP. This encourages the incremental approach to programming and avoids the cumbersome edit-debug-reedit cycle. Thus program development is both faster and more enjoyable.

When it comes time to save a program developed in this way, muSTAR can produce a pretty-printed source file containing all the variable values, function definitions, and property values you have specified. Since this source file can then be read into a stand alone muLISP system, the memory space required by muSTAR is freed for use by your application program.

The procedure for loading the muSTAR system and the options provided by the main menu are explained in Section 6.1. The next section describes how to use the muSTAR editor to write and modify function definitions, property values, etc. The function editing commands provide for cursor control, scrolling of the display, text insertion, text deletion, and the copying of lines of text.

The muSTAR screen-oriented editor will work on virtually any modern computer console or terminal. muSTAR must be able to perform simple cursor operations on the screen, including the ability to clear the screen, move the cursor up a line, and move the cursor to the home position (i.e. the upper left-hand corner of the screen).

The MS-DOS version of muSTAR is customized for the **IBM Personal Computer.** The **Apple II Computer** version of muSTAR is customized for the Apple II screen. The standard CP/M version of muSTAR is customized for the **SOROC IQ-120** terminal. If muSTAR has not been customized for your console, Section 6.3 describes how to write the simple muLISP functions needed to perform the cursor operations required by muSTAR.

For the information of those users who wish to modify or extend the capabilities of muSTAR, Section 6.4 describes how the system stores and manages the text displayed on the screen.

6.1 MAIN MENU COMMANDS

muSTAR is distributed both as the SYS file MUSTAR.SYS, and as the source files MUSTAR.LIB and CURSOR.LIB. The SYS file can be loaded by either method described in Chapter 2 of this manual. Probably the easiest way is to use the operating system command:

MULISP MUSTAR

This command first loads muLISP and displays the standard muLISP header message. Next the muSTAR system is loaded and its executive driver is called. The driver displays the following menu of options available to the user:

```
                    O P T I O N S

                  F  EDIT FUNCTION
                  V  EDIT VARIABLE
                  P  EDIT PROPERTY
                  E  EVAL LISP
                  T  TRACE FUNCTION
                  U  UNTRACE FUNCTION
                  R  READ FILE
                  W  WRITE FILE
                  X  EXIT TO DOS
ENTER CHOICE:
```

The facilities available in muSTAR fall into four categories, as reflected in the list above. The first three options call the display-oriented editor. The next three are used for program debugging. The next two fall into the category of disk file storage. The final option exits muSTAR and muLISP, and returns control to the host operating system.

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 entering <RETURN> following the prompt displayed after an option is selected.

When the menu has been memorized and is no longer needed, the control variable **MENU** can be set to NIL to suppress its output.

6.1.1 Editor Options

F **EDIT FUNCTION:** This option is used to create and/or edit muLISP function definitions. When "F" is pressed, the following prompt is displayed:

Function Name(s):

Enter as many function names as desired followed by <RETURN>. The system will then 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 muSTAR editor can then be used to modify the text of the definitions as described in Section 6.2. Once editing is complete, the function(s) can be redefined using the modified definition(s).

V **EDIT VARIABLE:** This option is used to assign and/or edit variable values. When "V" is pressed, the following prompt is displayed:

Variable Name(s):

Enter as many variable names as desired followed by <RETURN>. The system will then display the variable value(s) in pretty-printed form with the cursor at the beginning of the text. The editor can then be used to modify the value(s) of the variable(s).

P **EDIT PROPERTY:** This option is used to create and/or edit a name's property value. When "P" is pressed, the following prompt is displayed:

Name:

Enter the desired name (additional names will be ignored) followed by <RETURN>. The following prompt will be displayed:

Indicator:

Again only one property indicator can be specified. After <RETURN> is entered, the system displays the variable name, indicator, and current property value. After the property value is edited, it can be reassigned to the name under the indicator.

6.1.2 Debugging Options

E **EVAL LISP:** This option is used to initiate an eval-LISP driver loop (see Chapter 5). This eval-LISP loop will continue until **RETURN NIL** is entered following the dollar sign prompt.

T **TRACE FUNCTION:** This option is used to trace a LAMBDA or NLAMBDA defined function for debugging purposes. When "T" is pressed, the following prompt is displayed:

Function Name(s):

Enter as many function names as desired, followed by <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 function names to be entered for untracing.

6.1.3 Disk I/O Options

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

File Name:

Enter a drive letter, a colon, a file name, a period, a file type extension (e.g. B:ANIMAL.LIB), and <RETURN>. The drive is optional, defaulting to the currently logged drive. The file type is also optional, defaulting to type "LIB". If the file is found, muSTAR will read the specified file. If the file is not found, the following message will be displayed and 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 using an external editor, the last statement of the file should be a call to the function RDS, since this returns 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 "W" is pressed, the following prompt is displayed:

File Name:

Enter a drive letter, a colon, a file name, a period, a file type extension (e.g. B:ANIMAL.LIB), and <RETURN>. The drive is optional, defaulting to the currently logged drive. The file type is also optional, defaulting to type "LIB". muSTAR will create a **pretty-printed** source file of the function definitions and variable values that are marked for saving.

SAVELIST is a list of the items that will be saved by the next muSTAR Write command. If the value of the variable **SAVE** is T, functions defined using **DEFUN**, variables assigned using **SETQQ**, properties assigned using **PUTQQ**, and flags set by **FLAGQQ** are automatically added to SAVELIST. Items can be manually added or

deleted from the list if desired. When the development phase is complete, files produced by muSTAR can be read in by muLISP alone. muSTAR need not be loaded.

6.1.4 Exit muSTAR Option

X **EXIT TO DOS:** muSTAR and muLISP are terminated and control returns to the host operating system. The operating system prompt will be displayed.

6.2 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 effect immediately.

Because muSTAR is written entirely in muLISP, each user can extend the system to his/her own taste. When an ASCII control character is entered, the corresponding function is called to perform the desired operation. Naturally this function can be redefined to perform whatever task is desired. This is more fully explained in Section 6.4.

Function and variable names used in muSTAR which might conflict with user program names are terminated with a "$". Avoid the use of such names in your programs if you intended to develop them with muSTAR.

The general directions 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.

6.2.1 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 primarily using the control characters that are entered using your left hand. On most consoles, control characters are entered by pressing 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:

```
                    U P
                     ↑
                     E
LEFT  ←——A  ←—S        D—→  F——→  RIGHT
                     X
                     ↓
                  D O W N
```

The arrows indicate the direction the cursor will move when the corresponding control character is entered. The lengths of the arrows indicate the relative magnitude of cursor motion. The best way to get the "feel" of these commands is to experiment using the system.

<CTRL-D> and **<CTRL-S>** move the cursor one **character** to the right and left respectively. <CTRL-S> has the same effect as typing **<CTRL-H>** or pressing the **<BACKSPACE>** key). **<CTRL-F>** and **<CTRL-A>** move the cursor one **word** to the right and left respectively, both leaving 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 reached. Alternatively, after the beginning of a line is reached when backspacing, the cursor will move to the end of the previous line.

Entering **<CTRL-E>** will move the cursor up one line. **<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 line-feed 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>**. (Apple II SoftCard users use **<CTRL-U>** instead.)

A <RETURN> (i.e. **<CTRL-M>**) typed when in the Non-insert mode (see Section 6.2.3) advances the cursor to the beginning of the next line. A new line is added if the cursor is already at the bottom of the text. See Section 6.2.3 for the effect of <RETURN> when in the Insert mode.

Cursor control commands can only move the cursor to places on the screen that are 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 <RETURN> to insert new text instead of trying to use the cursor control characters.

6.2.2 Display Control

When long or multiple definitions are displayed on the screen, only part of the text can be shown at one time. The text can be thought of as being "written" on a long scroll. The screen is then a "window" on the text, which shows only a portion of the scroll. Since our window cannot 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>** scrolls the text UP a line. This is equivalent to moving the window DOWN a line. To see the previous line of text, typing **<CTRL-W>** scrolls the text DOWN a 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.

6.2.3 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 **<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:

Initial display:	(CONS BETA)
After entering "A":	(CONS ABETA)
After entering "L":	(CONS ALBETA)
After entering "P":	(CONS ALPBETA)
After entering "H":	(CONS ALPHBETA)
After entering "A":	(CONS ALPHABETA)
After entering " ":	(CONS ALPHA BETA)

When muSTAR is in the insert text mode, entering a <RETURN> causes a new line to be inserted in the text. The cursor will end up immediately following the new line. See Section 6.2.1 for a description of what happens when <RETURN> is entered from the Non-insert mode. In either mode, **<CTRL-N>** inserts a blank line in the text; however unlike <RETURN>, the cursor ends up on the new blank line.

The single quote mark can be used as a convenient shorthand for quoting an expression. The quote mark is a prefix operator that takes a single operand. For example, entering

'(THE IDES OF MARCH)

is equivalent to entering

(QUOTE (THE IDES OF MARCH))

<CTRL-P> is the escape control character used to allow delimiter characters to be used as names rather than delimiters. To use spaces, parentheses, periods, brackets, and the single quote mark as names, enter <CTRL-P> before typing the delimiter character.

6.2.4 Deleting and Moving 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 character to the immediate left of the cursor, enter **<CTRL-\>** or **<RUB>** (sometimes called <DELETE>).

4. To delete the word to the right of the cursor, type **<CTRL-T>**;

5. The current line can be deleted by typing **<CTRL-Y>**.

The text that has most recently been deleted using <CTRL-Y> can be re-inserted at any time by entering **<CTRL-I>**. While this is useful if a line is deleted by mistake, its main purpose is to allow lines to be moved or copied. To move a line, place the cursor on the line to be moved and enter <CTRL-Y> to delete the line. If the line is to be copied rather than moved, enter <CTRL-I>. Then move the cursor to the new location for the line and enter <CTRL-I>.

6.2.5 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 <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>**.

6.2.6 Exiting Editor

When the edit of a variable, function, or property is completed, entering **<CTRL-K>** will cause the following menu to be displayed on the console (Apple II SoftCard users use **<SHIFT CTRL-N>** in place of <CTRL-K>):

```
                    O P T I O N S

                  E  EVALUATE TEXT
                  A  ABANDON TEXT
                  C  CONTINUE EDIT

        ENTER CHOICE:
```

Normally the "E" option is chosen 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.

6.2.7 Editor Command Summary

Control Char	Function Name	Editing Operation
<CTRL-A>	LFT-TOK	Move cursor left one word
<CTRL-B>	MOV-RHT-END	Move cursor to end of current line*
<CTRL-C>	DWN-PAG	Scroll text up a screenful
<CTRL-D>	RHT-CHR	Move cursor right one character
<CTRL-E>	UP-LIN	Move cursor up one line
<CTRL-F>	RHT-TOK	Move cursor right one word
<CTRL-G>	DEL-RHT-CHR	Delete current character
<CTRL-H>	LFT-CHR	Move cursor left one character
<CTRL-I>	GET-LIN	Insert previously deleted line of text
<CTRL-J>	CAR-RET-TAB	Move cursor down one line following blanks
<CTRL-K>		Exit editor command**
<CTRL-L>	RPT-SRC	Repeat last search command
<CTRL-M>	CAR-RET	Move cursor down one line to beginning of line
<CTRL-N>	INS-LIN	Insert a blank line, do not move cursor
<CTRL-O>	SRC-TOK	Find a name
<CTRL-P>	ESC-CHR	Escape character for special characters as names
<CTRL-Q>	MOV-LFT-END	Move to beginning of current line
<CTRL-R>	UP-PAG	Scroll text down a screenful
<CTRL-S>	LFT-CHR	Move cursor left one character
<CTRL-T>	DEL-RHT-TOK	Delete word to the right of cursor
<CTRL-U>		UNUSED (Used by Apple II SoftCard for <CTRL-B>)
<CTRL-V>	INS-TOG	Toggle insert mode switch
<CTRL-W>	SCR-DWN	Scroll text down one line
<CTRL-X>	DWN-LIN	Move cursor down one line
<CTRL-Y>	DEL-LIN	Delete current line
<CTRL-Z>	SCR-UP	Scroll text up one line
<CTRL-\>	DEL-LFT-CHR	Delete character to left of cursor
<RUBOUT>	DEL-LFT-CHR	Delete character to left of cursor
<DELETE>	DEL-LFT-CHR	Same as <RUBOUT>
<RETURN>	CAR-RET	Same as <CTRL-M>
<LINEFEED>	CAR-RET-TAB	Same as <CTRL-J>

* Apple II and IIe SoftCard users use <CTRL-U> instead of <CTRL-B>.

** Apple II SoftCard users use <SHIFT CTRL-N> instead of <CTRL-K>.
Apple IIe SoftCard users use <CTRL-6> instead of <CTRL-K>.

6.3 CONSOLE CUSTOMIZATION

The muSTAR SYS file distributed with the **MS-DOS** version of muLISP is customized for the **IBM Personal Computer.** The muSTAR SYS file distributed with the **Apple II** version of muLISP is customized for the Apple II screen. The muSTAR SYS file distributed with the standard **CP/M** version of muLISP is customized for **Soroc IQ-120** and **Televideo 912** terminals. If you are using one of these computer systems, or a system that mimics one of them, the muSTAR SYS file can be loaded and run as described in Section 6.1 without modification, and the remainder of this section can be ignored.

The muLISP distribution diskette includes the muSTAR source file MUSTAR.LIB. Also included are a number of CURSOR source files that contain the muSTAR cursor control functions customized for various computers and terminals. For example, CURSOR.ZEN contains the cursor control functions required by the Zenith Z-100 computer. A comment at the top of each CURSOR file indicates what computer or terminal each file is customized for.

The CURSOR file named CURSOR.LIB is the one that will actually be read in when building muSTAR using MUSTAR.LIB. If one of the CURSOR files included with muLISP is right for your computer and terminal, simply rename that file CURSOR.LIB using a standard operating system command.

To actually build a customized muSTAR system, start with a copy of the muLISP master diskette that contains the files MULISP.COM, MUSTAR.LIB, and CURSOR.LIB. After starting up the muLISP system as described in Chapter 2, issue the following muLISP command:

```
(RDS 'MUSTAR 'LIB)
```

Asterisks will be displayed as each muSTAR function is defined. After MUSTAR.LIB has been read in, it automatically reads in CURSOR.LIB from the logged drive. If CURSOR.LIB is not found on the logged drive, drive A: is tried. If the file is still not found, the warning message

```
CURSOR.LIB not found
```

is displayed on the console. If this should occur, you will need to exit muLISP, and either rename an existing CURSOR file or create your own CURSOR.LIB file.

Once MUSTAR.LIB and CURSOR.LIB have been read in, issue the following muLISP command to create your own customized muSTAR SYS file:

```
(SAVE 'MUSTAR)
```

Finally, to actually start muSTAR running, you can use the procedure described in Section 6.1 or simply issue the command:

```
RETURN NIL
```

If there is not a CURSOR file suitable for your computer and terminal, you will need to make a customized version of CURSOR.LIB. It must contain definitions for the following functions and control variables applicable for your computer and terminal.

CLRSCRN [] clears the screen and sends the cursor to the top left corner (home position) of the screen.

HOME [] moves the cursor to the top left corner of the screen (home position) without affecting the characters being displayed on the screen.

UPLINE [n] moves the cursor to the beginning of the current line and up <n> lines. If <n> is zero, it should exit immediately without affecting the cursor. If <n> is not an integer, it should move the cursor up one line.

BACKSPACE [n] moves the cursor <n> characters to the left. If <n> is zero, it should exit immediately without affecting the cursor. If <n> is not an integer, it should backspace the cursor one place.

LINELEN: is the line length used by the pretty-printer to automatically terminate lines to prevent text from reaching the right margin of the screen. It should be set to one less than the maximum number of characters on a screen line.

PAGELEN: is the number of lines of text that can be displayed on your console screen.

INDENT: is the number of spaces used by the pretty-printer to indent for each level of program nesting.

Use your computer's text editor to make a customized version of the file CURSOR.LIB. The CURSOR files distributed with muLISP should serve as a guide. It will probably require less than a page of muLISP code. Once your own CURSOR.LIB file is made, build and save a customized muSTAR SYS file as described earlier in this section.

6.4 THEORY OF OPERATION

6.4.1 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 list 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 is 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 subsection. 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 and writes out the basic functions needed to read in the remainder of the file (i.e. DEFUN, SETQQ, PUTQQ, and/or FLAGQQ). Then the function definitions, values, properties, and flags on the list **SAVELIST** are pretty-printed.

6.4.2 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 features of your console screen or terminal. The first step is to become thoroughly familiar with how the text is stored and modified. Then the 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 **PAGELEN** is set to the maximum allowable number of lines on the screen. Its value is established in the file CURSOR.LIB. The length of the list *TEXT* can never be greater than PAGELEN.

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 nonnegative 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 delimiters are shown here enclosed in double quotes:

" " "," "(" ")" "." "[" "]" "|"

These 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 **UNPACK$**. When the cursor leaves the name, the characters are repacked or "compressed" into a new name using the function **PACK$**.

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.

6.4.3 Text Primitives

When the text editor is running, console input is in the raw input mode as described in Chapter 5. 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 nonNIL, the character is added to the text instead of overwriting old text.

When MUSTAR.LIB is read in, each of the ASCII control characters is assigned to the editing function that implements the desired operation for that character. For example, control-A is attached to the function **LFT-TOK** by the assignment

```
(SET (ASCII 1) LFT-TOK)
```

When the muSTAR editor is running and control-A is entered, the function LFT-TOK is evaluated and the console cursor and the internal text pointer are each moved one token to the left.

The Command Summary at the end of Section 6.2 lists each control character, the corresponding muSTAR function, and the effect of the function on the text. To customize muSTAR to more closely approximate your favorite text editor, simply attach the control characters to the editing function of your choice. You can also write your own editing functions using the existing primitives in the muSTAR source file. The purpose of each primitive function should be apparent from the function name.

7. MACHINE LANGUAGE ROUTINES

This chapter is for the small number of muLISP users who find it necessary to write machine language routines and call them from muLISP. There are three major reasons for writing such routines: a) to control some computer hardware or peripheral device; b) to communicate with the host operating system; c) to implement some critical muLISP function in machine language for efficiency reasons. The services provided by the Hardware Interface Functions (see Chapter 5) are intended to minimize as much as possible the need to write such machine language routines.

The remainder of this chapter describes how to write and link to machine language routines if you need to do so. There is a separate section for each microprocessor for which muLISP is implemented.

The appendices to this manual contain a muLISP **base page** assembly listing for each microprocessor. The base page consists of the first 180H bytes in MULISP.COM. It contains the muLISP pseudo-registers, the default values for several system variables (e.g. line length, radix base, etc.), the default reader parse table, and space for entry points into user-defined machine language routines. The commented listings show the addresses of these items in the event you need to change their default value. This can be done using the debugger supplied with your computer's operating system (see the operating system reference manual).

7.1 8086 MACHINE LANGUAGE ROUTINES

All user-defined machine language routines will be eval, spread functions (see Evaluation Functions section of Chapter 5). Arguments to machine language routines are the addresses of muLISP data objects (see Chapter 4). The first argument is passed in the SI register, the second in the DI register, and a list of the remaining arguments, if any, in the BX register.

The easiest way to pass and/or return small integer values to/from machine language routines is via the muLISP pseudo-registers (see the Hardware Interface Functions section of Chapter 5). The pseudo-registers are stored in the 8086 Base Page.

Machine language routines should be terminated with an intra-segment RET instruction. The value returned by the routine is the object pointed to by the DI register. DI must contain the address of a bona fide muLISP data object. If the routine returns no special value, DI should be set to the address of NIL which is 200H.

Machine language routines must save the CX, BP, and the four 8086 segment registers, so that they can be restored to their original value before a call is made to a muLISP primitive and before control is returned to muLISP.

The pointer cells making up a muLISP data object are stored at the same offset in different segments. An object's car cell is stored in the data segment; the cdr cell in the extra segment; a name's function cell and a number's length cell in the stack segment; and a name's print name cell and a number's vector cell in the code segment. For example, if SI contains a name, the value of the four cells of a name can be found as shown by the following code segment:

```
MOV     AX,[SI]             ;Set AX to the name's value
MOV     BX,ES:[SI]          ;Set BX to the name's plist
MOV     DX,SS:[SI]          ;Set DX to the name's defn
MOV     DI,CS:[SI+MCOFF]    ;Set DI to the name's pname
```

MCOFF (machine code offset) is a constant offset to be added to SI when accessing print name cells and vector cells. The value of MCOFF is stored in the 8086 Base Page.

The address of a muLISP object determines its data type as either a name, number, or node. The following code segment jumps to one of three routines based upon the type of object pointed to by the SI register:

```
CMP     SI,[NXNAME]
JB      NAMTYPE             ;Jump if SI is a name
CMP     SI,[ENDATM]
JB      NUMTYPE             ;Jump if SI is a number
JMP     NODTYPE             ;SI is a number
```

The current values of the global variables NXNAME and ENDATM are stored in the data segment at offsets 0 and 2 respectively.

Calls to muLISP routines that do CONSing or generate numbers can invoke the garbage collector. Garbage collections will destroy unreferenced muLISP data objects and can change the address of referenced objects. To ensure that pointers to objects are saved and updated during a garbage collection, they must be pushed onto the muLISP variable stack. The BP register is the variable stack pointer. The following code segment illustrates how the pointer in BX is pushed on and later popped off the variable (BP) stack for a call to the muLISP CONS routine:

```
MOV     [BP],BX             ;Push BX on BP stack
INC     BP                  ;Increment BP a word
INC     BP
CALL    CONS                ;Set DI to CONS(SI,DI)
DEC     BP                  ;Decrement BP a word
DEC     BP
MOV     BX,[BP]             ;Pop updated BX from BP stack
```

User-defined routines can call the large number of primitively defined muLISP functions. These functions use the same register conventions to receive arguments and return values as those described for user routines. The offset address in the code segment of the functions can be determined from within muLISP using the function GETD (GET Definition).

```
(RADIX 16)      -->  0A        ;Set hexadecimal
(GETD 'CAR)     -->  34F       ;Address of CAR
(GETD 'ATOM)    -->  51A       ;Address of ATOM
(RADIX 0A)      -->  16        ;Restore decimal
```

To call a muLISP routine from a user routine, put the offset of the muLISP routine in the AX register and execute an 8086 inter-segment CALL to the muLISP code segment at offset 103H.

To link a muLISP name to a user-defined routine, call PUTD (PUT Definition) with the routine's name and entry address (i.e. its offset in the code segment).

```
(RADIX 16)                  ;Set hexadecimal
(PUTD 'FOO 200)             ;Link FOO to 200H
(RADIX 0A)                  ;Restore decimal
```

Note that muLISP uses **address typing** to determine a function's type as being either machine language or D-code. Therefore, all user-defined machine language routines must have entry addresses in low memory (i.e. in the 8086 Base Page). As a convenience to the user, the 8086 Base Page contains sixteen entries for user routines, beginning at 200H. Each entry is preset to call an 8086 interrupt vector (80H through 8FH). These vectors can be changed if desired.

```
(PUTD COPY '(LAMBDA (TREE)
     ((ATOM TREE) TREE)
     (CONS (COPY (CAR TREE)) (COPY (CDR TREE))) ))
```

COPY is a simple LAMBDA defined function for copying muLISP data objects. The equivalent 8086 assembly language code for COPY is printed in an appendix to this manual. The listing illustrates most of the points discussed in this section. If the file is assembled and turned into the COM file named COPYTREE.COM, it can be loaded into memory and linked to the muLISP name COPY as follows:

```
A>COPYTREE

A>MULISP

$ (RADIX 16)                    ;Switch to hexadecimal
0A

$ (PUTD 'COPY 200)              ;Linking COPY to routine
200

$ (RADIX 0A)                    ;Return to decimal
16

$ (COPY '(A (B . C) D))         ;Calling the COPY routine
(A (B . C) D)
```

7.2 8080 MACHINE LANGUAGE ROUTINES

All user-defined machine language routines will be eval, spread functions (see Evaluation Functions section of Chapter 5). Arguments to machine language routines are the addresses of muLISP data objects (see Chapter 4). The first argument is passed in the HL register, the second in the DE register, and a list of the remaining arguments, if any, in the BC register.

The easiest way to pass and/or return small integer values to/from machine language routines is via the muLISP pseudo-registers (see the Hardware Interface Functions section of Chapter 5). The pseudo-registers are stored in the 8080 Base Page.

Machine language routines should be terminated with a RET instruction. The value returned by the routine is the object pointed to by the HL register. HL must contain the address of a bona fide muLISP data object. If the routine returns no special value, HL should be set to the address of NIL. The address of NIL is stored in the 8080 Base Page.

The pointer cells making up a muLISP data object are stored in consecutive words of memory. For example, if HL contains a name, the following code segment places its function definition in DE:

```
        INX     H           ;Set HL to HL+4
        INX     H
        INX     H
        INX     H
        MOV     E,M         ;Set DE to defn cell ptr
        INX     H
        MOV     D,M
```

The address of a muLISP object determines its data type as either a name, number, or node. The following code segment jumps to one of three routines based upon the type of object pointed to by the DE register:

```
        LHLD    NXNAME
        CALL    CMPDH
        JC      NAMTYPE     ;Jump if DE is a name
        LDA     ENDATM+1
        CMP     D
        JC      NUMTYPE     ;Jump if DE is a number
        JMP     NODTYPE

CMPDH:  MOV     A,D         ;Compare DE with HL
        CMP     H
        RNZ
        MOV     A,E
        CMP     L
        RET
```

The addresses of the variables NXNAME and ENDATM are stored in the 8080 Base Page.

Calls to muLISP routines that do CONSing or generate numbers can invoke the garbage collector. Garbage collections will destroy unreferenced muLISP data objects and can change the address of referenced objects. To ensure that pointers to objects are saved and updated during a garbage collection, they must be pushed onto the muLISP variable stack. The BP pseudo-register is the variable stack pointer. The following code segment illustrates how the pointer in DE is pushed on and later popped off the variable (BP) stack for a call to the muLISP CONS routine:

```
PUSH    H
LHLD    BP
MOV     M,E         ;Push DE on BP stack
INX     H
MOV     M,D
INX     H
SHLD    BP          ;Increment BP a word
POP     H
CALL    CONS        ;Set HL to CONS(HL,DE)
PUSH    H
LHLD    BP
MOV     D,M         ;Pop updated DE from BP stack
DCX     H
MOV     E,M
DCX     H
SHLD    BP          ;Decrement BP a word
POP     H           ;Set HL to CONS(HL,DE)
```

User-defined routines can call the large number of primitively defined muLISP functions. These functions use the same register conventions to receive arguments and return values as those described for user routines. The address of the muLISP functions can be determined from within muLISP using the function GETD (GET Definition).

```
(RADIX 16)      -->  0A       ;Set hexadecimal
(GETD 'CAR)     -->  34F      ;Address of CAR
(GETD 'ATOM)    -->  51A      ;Address of ATOM
(RADIX 0A)      -->  16       ;Restore decimal
```

To link a muLISP name to a user-defined routine, call PUTD (PUT Definition) with the routine's name and entry address.

```
(RADIX 16)                    ;Set hexadecimal
(PUTD 'FOO 200)               ;Link FOO to 200H
(RADIX 0A)                    ;Restore decimal
```

Note that muLISP uses **address typing** to determine a function's type as being either machine language or D-code. Therefore, all user-defined machine language routines must have entry addresses in low memory (i.e. in the 8080 Base Page). As a convenience to the user, the 8080 Base Page contains sixteen entries for user routines, beginning at 200H. Each entry is a dummy JUMP instruction that can be redirected to a user routine.

The best place to store user routines is just below the host operating system. The amount of memory that muLISP will use can be reduced by adjusting a word in the 8080 base page. Another alternative is to store the user routines above the operating system. For CP/M computers such space can be reserved by creating a smaller CP/M system than you normally use. See the CP/M reference manuals for details.

```
(PUTD COPY '(LAMBDA (TREE)
     ((ATOM TREE) TREE)
     (CONS (COPY (CAR TREE)) (COPY (CDR TREE))) ))
```

COPY is a simple LAMBDA defined function for copying muLISP data objects. The equivalent 8080 assembly language code for COPY is printed as an appendix to this manual. The listing illustrates most of the points discussed in this section. If the code is assembled and loaded into memory at, say, 0C000H, the following illustrates how to link it to the muLISP name COPY:

```
$ (RADIX 16)                              ;Switch to hexadecimal
0A

$ (MEMORY 201 (REMAINDER 0C000 100))      ;Setting up JMP address
0

$ (MEMORY 202 (QUOTIENT 0C000 100))
C0

$ (PUTD 'COPY 200)                        ;Linking COPY to routine
200

$ (RADIX 0A)                              ;Return to decimal
16

$ (COPY '(A (B . C) D))
(A (B . C) D)
```

(This page intentionally left blank)

8. LISP AND ARTIFICIAL INTELLIGENCE REFERENCES

After learning muLISP, you may be anxious to learn more about AI (artificial intelligence) programming techniques. There is an extensive body of literature on AI and LISP programming. This chapter lists a few of the AI professional societies, periodicals, and books. They can serve as an entry point to this exciting field.

8.1 PROFESSIONAL SOCIETIES

The American Association for Artifical Intelligence (AAAI)
445 Burgess Drive, Menlo Park, CA, 94025; phone (415) 328-3123.

The ACM Special Interest Group on Artificial Intelligence (SIGART)
ACM, 1133 Avenue of the Americas, New York, NY, 10036.

8.2 PERIODICALS

The AI Magazine, 445 Burgess Drive, Menlo Park, CA, 94025; phone (415) 328-3123.

The Applied Artificial Intelligence Reporter, P.O. Box 248235, Coral Gables, FL, 33124.

Artificial Intelligence, P.O. Box 211, 1000 AE Amsterdam, The Netherlands.

IEEE Transactions on Pattern Analysis and Machine Intelligence, Institute of Electrical and Electronics Engineers Inc., 345 East 47th St., New York, NY, 10017.

Machine Intelligence, Ellis Horwood Ltd., Market Cross House, Cooper St., Chichester, West Sussex, PO19 1EB, England.

Pattern Recognition, Pergamon Press Inc., Maxwell House, Fairview Park, Elmsford, NY, 10523.

Robotics Age, Strand Building, 174 Concord St., Peterborough, NH, 03458; phone (603) 924-7136.

8.3 BOOKS

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

2. Barr, A., and Feigenbaum, E. A., (eds),
The Handbook of Artificial Intelligence,
William Kaufmann, Inc., Los Altos, CA, 1982.

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

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

5. Charniak, E., et al, Artificial Intelligence Programming,
Lawrence Erlbaum Associates, 365 Broadway, Hillsdale, NJ, 1980.

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

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

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

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

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

11. Rich, E., Artificial Intelligence,
McGraw-Hill Book Company, New York, NY, 1983.

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

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

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

15. Winston, P. H. & Horn, B. K. P., LISP,
Addison-Wesley Publishing Co., Reading, MA, 1981.

9. GLOSSARY OF COMPUTER TERMINOLOGY

Address typing: a method of determining the type of a data object (e.g. name, number, list, etc.) by its address in memory. This method of data typing does not require the use of tag bits, nor is there a need to actually examine the object merely to determine its type.

Algorithm: a set of rules for solving a problem in a finite number of steps.

Argument: an expression whose value is used by a function to compute a result. When a function is called, the arguments are evaluated and the values are passed to the function.

Artificial Intelligence (AI): a field of computer science that attempts to make computers perform tasks that are normally done by humans such as reasoning, learning, and creating. This can further our understanding of intelligence itself.

ASCII (American Standard Code for Information Interchange): this code is the widely used standard for transmitting and storing character data by computers. The letters of the alphabet, decimal digits, punctuation marks, and other special characters are each represented by a unique 7 bit code.

Assembly language: the low-level computer language for a processor, each statement of which corresponds one-for-one to a machine language instruction of the processor. Assembly language is used to write high-level languages such as muLISP since maximum efficiency can be achieved through its use.

Binary: a condition in which there are but two alternatives. Thus the binary number system (i.e. base 2) has but two values for its digits, either 0 or 1.

Bit: a binary digit, the smallest unit of computer data. It can be either 0 or 1. However, collections of bits can represent arbitrarily large and complex numbers and data.

Boolean logic: a binary system of logic named for George Boole. It includes operators such as AND, OR, and NOT which return a truth value based upon the truth values of their operands. Thus boolean operators are widely used in programming to combine logical values to determine the desired course of action.

Bootstrap loader: a program usually in ROM which is designed to load the operating system and then turn control over to the freshly loaded system. Initiating the loader on a computer may either be automatic when the system is turned on or reset, or it may be necessary to perform a number of actions manually.

Byte: a unit of data made of 8 bits. The meaning of the information stored depends entirely on the way in which the byte is interpreted. For instance, a byte can represent an ASCII character or an integer in the range from 0 to 255.

Compiler: a program that converts programs written in a higher level computer language into a lower-level form that is more easily "understood" by the computer. Compilers can produce either directly executable machine code, or easily interpreted and portable pseudo-code. Compilers do not actually execute programs. Program execution is the job of the microprocessor or, in the case of pseudo-code, the interpreter.

Control character: a nonprintable ASCII character code used primarily to communicate control directives to the receiving device. Examples include the carriage return, linefeed, and bell characters.

CRT (Cathode Ray Tube): a terminal which uses an electronic vacuum tube to display characters and/or graphics.

Cursor: a rectangular block or horizontal bar of light, often blinking, on a CRT screen, whose position indicates where the next character will be displayed. The cursor position is controlled by special control characters sent from the computer to the terminal.

Disk drive: a computer accessory device that rotates disks (hard or floppy) and reads and writes information on the disk by means of a magnetic head sensor.

Environment: the current state of a program in the computer. For muLISP programs it includes the values and properties of all program variables and the definitions of all functions.

File: A block of data stored on the mass storage device under a given name. It consists of a collection of records treated as units.

Floppy disks: flexible plastic disks coated with an iron oxide film that are capable of storing magnetically recorded information. They are used in conjunction with floppy disk drives as a mass storage device.

Function: a procedure for computing from a given set of arguments a unique result. In most programming languages, functions can have side effects which permanently affect the environment.

Global variable: a variable whose value has not been assigned by its use in the formal argument list of a function. Any change to such a variable is recognized by all functions which use it.

Hard disks: rigid disks covered with a magnetic oxide film capable of storing large amounts of data (tens of millions of bytes). Since the disks rotate at a high rate of speed, hard disk

drives can access data in a very short amount of time. Currently hard disk systems are more expensive and hence less common than floppy disk storage systems.

Hardware, computer: the physical components making up a computer system. It includes the computer's power supply, electronic logic circuits, and the frame housing the computer.

Head alignment: in order for a disk drive to read or write information from or to a given track of a floppy disk, the magnetic head sensor must be accurately positioned over the track. If this is not the case and disk errors are occurring, an alignment must be performed.

Hexadecimal: a notation for representing numbers using base 16 rather than base 10. Since 16 is a power of 2, it easy to convert from the binary representation used by computers to the hexadecimal notation more easily understood by humans.

Interpreter: a computer program that implements a computer language by executing while decoding individual program statements written in that language.

I/O (Input/Output): refers to the transfer of data into and out of a device such as a disk drive or a computer.

Kilobyte: 1024 bytes of storage. This number is used as the standard unit for measuring computer memory sizes since it is an even power of 2.

Local variable: a variable listed in a function's formal argument list. A function's local variables can be modified within that function without having any effect outside the function. This is a tremendous advantage since it frees the programmer from concerns about undesirable global effects.

Logged drive: the disk drive which will be used by default in the event a specific drive is not specified in a command related to disk file I/O. The currently logged drive can be changed by an operating system command.

Lower-case: the letters of the alphabet expressed as noncapitals.

Machine code: instructions that are directly executable by a microprocessor.

Mass storage device: hardware designed to provide permanent storage of computer information, such as floppy or hard disk drives. Although not as fast as RAM, it is usually capable of storing large amounts of information, and for the most part, the data is immune to computer malfunctions.

Megahertz (MHz): a unit of frequency equal to a million cycles per second.

Microprocessor: the electronic device which actually performs the logical and arithmetic operations in a microcomputer. It consists of a small silicon chip housed in a ceramic or plastic dual in-line package (DIP) usually having forty or more connector pins.

Monitor: a small supervisor program, usually implemented in ROM. It provides a primitive means of communicating with the computer to allow such operations as examining and changing memory.

Nonvolatile memory: computer memory which retains its information even after the computer has been turned off. Thus it provides safer, more permanent storage than volatile memory such as RAM.

On-line: information which can be directly read by a computer. Thus data on a mass storage device is on-line whereas printed matter is off-line.

Operating System: a complete system which provides the basis for user interaction with a computer. It usually provides facilities for the generation, testing, and running of programs and the capability for permanently storing such programs for future use.

Parse: to determine the syntactic relationships between the components of a statement in a given computer language.

Pseudo-code: a code for storing function definitions that is extremely compact and can be efficiently interpreted by a computer. Pseudo-code has the additional advantage over machine code of being "portable" since it is relatively easy to program different microprocessors to interpret the same pseudo-code.

RAM (Random Access Memory): the primary means of fast-access read-write data storage within microcomputers. Data stored in RAM is accessed by specifying the address of the data. In contrast to sequential access memory, the time for such an access is independent of the location of the data.

Rational arithmetic: has for its domain or area of applicability all numbers which can be expressed as the ratio of two integers. Thus it includes integer arithmetic as a subset.

Record: a block of data from a disk file which is stored as a unit on a mass storage device.

Register: a storage cell located in a microprocessor that is used by the processor for logical and arithmetic operations. The capacity of a register is measured by the number of bits it can store. Common microprocessor register sizes include 8 bits and 16 bits.

Reset: a procedure to return a computer to the initial condition it

was in when first turned on. If a computer program goes wild, reset the computer to initiate the bootstrap loader process. A reset is usually initiated by pressing a switch on the computer or a conbination of keys on the keyboard.

ROM (Read Only Memory): this form of data storage can only be read from, but not changed, by the computer. However, it has an advantage over most RAM's in that it is a nonvolatile memory. Thus, it is commonly used to store bootstrap loader programs. ROM is random access memory; however, it is not called RAM since that term is reserved for the volatile, read-write memory.

Sector: a wedge shaped area on the surface of a floppy or hard disk. That portion of a track which is contained in one sector is capable of storing one record of data.

Software, computer: the program or set of instructions which specify how a computer is to accomplish a given task. This is considered "soft" since the same computer hardware can execute many different programs.

Stack, push-down: a data structure used to temporarily store information. The PUSH operation stores data on top of a stack pushing older data "down". The POP operation retrieves the most recently pushed data from the top of a stack. Stacks are heavily used in recursive languages like LISP and muSIMP.

Syntax: the patterns of symbols or tokens that constitute all the valid expressions or statements of a computer language.

Track: one of many concentric circles around a disk in which data is stored. That portion of a track which is contained in one sector is capable of storing one record of data.

Truth value: either the value TRUE or FALSE. Truth values are the subject of Boolean Logic and play a fundamental role in program control.

Upper-case: the letters of the alphabet expressed in capital letters.

Variable: a name which is used by a computer program to store a value. The value can be explicitly made using an assignment statement, or implicitly made when a function call assigns values to the function's local variables.

(This page intentionally left blank)

Appendix A: IBM PC Extended Function Keys

Second Character	Key Pressed
(ASCII 3)	CTRL 2
(ASCII 15)	SHIFT ←
(ASCII 16)	ALT Q
(ASCII 17)	ALT W
(ASCII 18)	ALT E
(ASCII 19)	ALT R
(ASCII 20)	ALT T
(ASCII 21)	ALT Y
(ASCII 22)	ALT U
(ASCII 23)	ALT I
(ASCII 24)	ALT O
(ASCII 25)	ALT P
(ASCII 30)	ALT A
(ASCII 31)	ALT S
" "	ALT D
!	ALT F
"	ALT G
#	ALT H
$	ALT J
%	ALT K
&	ALT L
'	ALT Z
-	ALT X
.	ALT C
/	ALT V
0	ALT B
1	ALT N
2	ALT M
;	F1
<	F2
=	F3
>	F4
?	F5
@	F6
A	F7
B	F8
C	F9
D	F10
G	Home
H	↑
I	Pg Up
K	←
M	→
O	End
P	↓
Q	Pg Dn
R	Ins
S	Del

Second Character	Key Pressed
T	SHIFT F1
U	SHIFT F2
V	SHIFT F3
W	SHIFT F4
X	SHIFT F5
Y	SHIFT F6
Z	SHIFT F7
[	SHIFT F8
\	SHIFT F9
]	SHIFT F10
^	CTRL F1
_	CTRL F2
`	CTRL F3
a	CTRL F4
b	CTRL F5
c	CTRL F6
d	CTRL F7
e	CTRL F8
f	CTRL F9
g	CTRL F10
h	ALT F1
i	ALT F2
j	ALT F3
k	ALT F4
l	ALT F5
m	ALT F6
n	ALT F7
o	ALT F8
p	ALT F9
q	ALT F10
s	CTRL ←
t	CTRL →
u	CTRL End
v	CTRL Pg Dn
w	CTRL Home
x	ALT 1
y	ALT 2
z	ALT 3
{	ALT 4
|	ALT 5
}	ALT 6
~	ALT 7
(ASCII 127)	ALT 8
(ASCII 128)	ALT 9
(ASCII 129)	ALT 0
(ASCII 130)	ALT -
(ASCII 131)	ALT =
(ASCII 132)	CTRL Pg Up

Appendix B: 8086 Base Page

```
        ORG     100H            ;Base address of muLISP
0100    JMP     LISP0           ;muLISP entry point
0103    JMP     FARCALL         ;Entry point for inter-segment calls
                                ;to muLISP routines.  AX contains
                                ;the offset address of the routine.

0106    DB      CR,LF
        DB      'muLISP-83 4.10 (03/07/84)',CR,LF
        DB      'MS-DOS Version',CR,LF
        DB      'Copyright (C) 1982 The SOFT WAREHOUSE',CR,LF
        DB      'Licensed by MICROSOFT Corp.',LF,'$'

        ORG     180H            ;TABLE DRIVEN PARSER READ TABLE

;               @ A B C D E F G H I J K L M N O     ASCII CONTROL CHARS
0180    DB      0,0,0,0,0,0,0,0,0,1,1,0,0,1,0,0

;               P Q R S T U V W X Y Z  [  \ ] ^ _   ASCII CONTROL CHARS
0190    DB      0,0,0,0,0,0,0,0,0,0,2,128,0,0,0,0

;                 ! " # $ % &  ' ( ) * + , - . /
01A0    DB      1,2,4,0,2,8,2,32,2,2,2,2,2,2,2,2

;               0 1 2 3 4 5 6 7 8 9 :  ; < = > ?
01B0    DB      0,0,0,0,0,0,0,0,0,0,2,16,2,2,2,2

;               @ A B C D E F G H I J K L M N O
01C0    DB      2,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0

;               P Q R S T U V W X Y Z [ \ ] ^ _
01D0    DB      0,0,0,0,0,0,0,0,0,0,0,2,2,2,2,2

;               ` a b c d e f g h i j k l m n o
01E0    DB      2,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0

;               p q r s t u v w x y z {,|,},~,RUB
01F0    DB      0,0,0,0,0,0,0,0,0,0,0,2,2,2,2,0
```

```
        ORG     200H            ;Prebuilt entry points for user
0200    INT     80H             ;defined routines.  The interrupts
0202    RET                     ;can be changed as desired.

0203    INT     81H
0205    RET

0206    INT     82H
0208    RET

0209    INT     83H
020B    RET

020C    INT     84H
020E    RET

020F    INT     85H
0211    RET

0212    INT     86H
0214    RET

0215    INT     87H
0217    RET

0218    INT     88H
021A    RET

021B    INT     89H
021D    RET

021E    INT     8AH
0220    RET

0221    INT     8BH
0223    RET

0224    INT     8CH
0226    RET

0227    INT     8DH
0229    RET

022A    INT     8EH
022C    RET

022D    INT     8FH
022F    RET
```

```
        ORG     230H
0230    DB      -1,-1           ;READ XLAT TABLE
0232    DB      -1,-1           ;If a character being input matches
0234    DB      -1,-1           ;the first character of a pair, the
0236    DB      -1,-1           ;second character of the pair is
0238    DB      -1,-1           ;passed to muLISP instead.
023A    DB      -1,-1
023C    DB      -1,-1
023E    DB      -1,-1

0240    DB      -1,-1           ;WRITE XLAT TABLE
0242    DB      -1,-1           ;If a character being output matches
0244    DB      -1,-1           ;the first character of a pair, the
0246    DB      -1,-1           ;second character of the pair is
0248    DB      -1,-1           ;passed to the operating system
024A    DB      -1,-1           ;instead.
024C    DB      -1,-1
024E    DB      -1,-1

                                ;PSEUDO-REGISTERS
0250    DW      0               ;0      AX
0252    DW      0               ;1      BX
0254    DW      0               ;2      CX
0256    DW      0               ;3      DX
0258    DW      0               ;4      SI
025A    DW      0               ;5      DI
025C    DW      0               ;6      BP
025E    DW      0               ;7      DS
0260    DW      0               ;8      ES
0262    DW      0               ;9      FLAGS

                                ;MINIMUM FREE SPACE ALLOWED AFTER GC
0264    DW      200H            ;Minimum Atom Space bytes
0266    DW      200H            ;Minimum Node Space bytes
0268    DW      200H            ;Minimum Vector Space bytes
026A    DW      400H            ;Minimum Stack Space bytes
026C    DW      32              ;Ratio of total free space to stack
                                ;free space

                                ;REALLOCATION SENSITIVITY
026E    DW      4               ;Reallocation occurs after a garbage
0270    DW      3               ;collection if for any space, the
                                ;ratio of the size of the space to
                                ;the optimum size of the space is
                                ;greater than [026E]/[0270] or less
                                ;than [0270]/[026E].

0272    DW      MCOFF           ;MACHINE CODE OFFSET
                                ;The offset to add when accessing a
                                ;name's print name pointer or a
                                ;number's vector pointer.

0274    DW      79              ;DEFAULT LINE LENGTH
                                ;Should be one less than the console
                                ;screen width.
```

```
0276    DB      10          ;DEFAULT RADIX BASE

0277    DB      7FH         ;CHARACTER MASK
                            ;The mask used on characters input
                            ;from files to strip parity bit.

0278    DB      7           ;BELL CHARACTER
                            ;The character output by muLISP to
                            ;sound the console bell.  Set byte to
                            ;0 to deactivate bell.

0279    DB      100         ;KEYBOARD INTERRUPT SENSITIVITY
                            ;Controls the frequency at which the
                            ;console is checked for an escape char.
                            ;Make larger for less frequent.

027A    DW      0           ;FIRST PARAGRAPH AFTER END OF MEMORY
                            ;A user imposed limit to the amount
                            ;of memory muLISP will use.
                            ;0 to use all available memory.

027C    DB      0           ;DEFAULT EXIT CODE
                            ;This byte is the exit code returned
                            ;by muLISP if the function SYSTEM is
                            ;called without arguments, if the break
                            ;option "System" is chosen, or if exit
                            ;is via a <CTRL-C>.
```

Appendix C: 8086 Machine Language Routine

```
        ORG     100H            ;Entry address for MS-DOS programs.
        MOV     AX,0
        MOV     DS,AX           ;Set DS to 0.
        MOV     [200H],COPYX    ;Set up interrupt vector 80H to
        MOV     [202H],CS       ;       point to CS:COPYX.
        MOV     DX,LENGTH       ;Set DX to length to protect.
        INT     27H             ;Terminate this program but leave
                                ;       routine resident.

                                ;Args:  SI: TREE
                                ;Rslt:  DI: COPY(TREE)
COPYX:  POP     CS:[IPREG]      ;To remove return address and flags
        POP     CS:[CSREG]      ;from SP stack.
        POP     CS:[FLAGS]
        CALL    COPY            ;Call to COPY routine.
        PUSH    CS:[FLAGS]      ;To restore return address and flags
        PUSH    CS:[CSREG]
        PUSH    CS:[IPREG]
        IRET

ENDATM  EQU     02H             ;Offset of the variable ENDATM in DS
CONS    EQU     441H            ;Offset of the routine CONS in CS

COPY:   CMP     SI,[ENDATM]
        JB      COPY1           ;Jump if ATOM(TREE)
        MOV     [BP],SI         ;Push TREE on BP stack
        INC     BP
        INC     BP
        MOV     SI,[SI]         ;Set SI to CAR(TREE)
        CALL    COPY            ;Set DI to COPY(CAR(TREE))
        XCHG    DI,[BP-2]       ;Pop TREE, and push DI
        MOV     SI,ES:[DI]      ;Set SI to CDR(TREE)
        CALL    COPY            ;Set DI to COPY(CDR(TREE))
        DEC     BP              ;Restore BP
        DEC     BP
        MOV     SI,[BP]         ;Set SI to COPY(CAR(TREE))
        MOV     AX,OFFSET CONS
        CALL    L,CS:[FARCAL]   ;Set DI to CONS(SI,DI)
        RET

COPY1:  MOV     DI,SI           ;Set DI to TREE
        RET

FARCAL  DW      103H            ;CSREG:FARCAL: MULISP ENTRY POINT
CSREG   DW      0               ;       FOR INTER-SEGMENT CALLS

IPREG   DW      0               ;Temporary storage for IP & FLAGS
FLAGS   DW      0

LENGTH  EQU     $               ;Length of routine
        END
```

Appendix D: 8080 Base Page

```
        ORG     100H            ;Base address of muLISP
0100    JMP     LISP0           ;muLISP entry point
0103    JMP     $               ;DUMMY JUMP INSTRUCTION

0106    DB      CR,LF
        DB      'muLISP-83 4.10 (03/07/84)',CR,LF
        DB      'CP/M-80 Version',CR,LF
        DB      'Copyright (C) 1981 The SOFT WAREHOUSE',CR,LF
        DB      'Licensed by MICROSOFT Corp.',LF,'$'

        ORG     180H            ;TABLE DRIVEN PARSER READ TABLE

;               @ A B C D E F G H I J K L M N O    ASCII CONTROL CHARS
0180    DB      0,0,0,0,0,0,0,0,0,1,1,0,0,1,0,0

;               P Q R S T U V W X Y Z  [  \ ] ^ _  ASCII CONTROL CHARS
0190    DB      0,0,0,0,0,0,0,0,0,0,2,128,0,0,0,0

;                 ! " # $ % &  ' ( ) * + , - . /
01A0    DB      1,2,4,0,2,8,2,32,2,2,2,2,2,2,2,2

;               0 1 2 3 4 5 6 7 8 9 :  ; < = > ?
01B0    DB      0,0,0,0,0,0,0,0,0,0,2,16,2,2,2,2

;               @ A B C D E F G H I J K L M N O
01C0    DB      2,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0

;               P Q R S T U V W X Y Z [ \ ] ^ _
01D0    DB      0,0,0,0,0,0,0,0,0,0,0,2,2,2,2,2

;               ` a b c d e f g h i j k l m n o
01E0    DB      2,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0

;               p q r s t u v w x y z {,|,},~,RUB
01F0    DB      0,0,0,0,0,0,0,0,0,0,0,2,2,2,2,0
```

```
        ORG     200H            ;Sixteen dummy JMP instructions
0200    JMP     $               ;for use as entry to user routines.
0203    JMP     $
0206    JMP     $
0209    JMP     $

020C    JMP     $
020F    JMP     $
0212    JMP     $
0215    JMP     $

0218    JMP     $
021B    JMP     $
021E    JMP     $
0221    JMP     $

0224    JMP     $
0227    JMP     $
022A    JMP     $
022D    JMP     $


0230    DB      -1,-1           ;READ XLAT TABLE
0232    DB      -1,-1           ;If a character being input matches
0234    DB      -1,-1           ;the first character of a pair, the
0236    DB      -1,-1           ;second character of the pair is
0238    DB      -1,-1           ;passed to muLISP instead.
023A    DB      -1,-1
023C    DB      -1,-1
023E    DB      -1,-1

0240    DB      -1,-1           ;WRITE XLAT TABLE
0242    DB      -1,-1           ;If a character being output matches
0244    DB      -1,-1           ;the first character of a pair, the
0246    DB      -1,-1           ;second character of the pair is
0248    DB      -1,-1           ;passed to the operating system
024A    DB      -1,-1           ;instead.
024C    DB      -1,-1
024E    DB      -1,-1


                                ;PSEUDO-REGISTERS
0250    DW      0               ;0      BC
0252    DW      0               ;1      DE
0254    DW      0               ;2      HL
0256    DW      0               ;3      PSW
0258    DW      0               ;4
025A    DW      0               ;5
025C    DW      0               ;6
025E    DW      0               ;7
0260    DW      0               ;8
0262    DW      0               ;9
```

```
                                ;MINIMUM FREE SPACE ALLOWED AFTER GC
0264    DW      100H            ;Minimum Atom Space bytes
0266    DW      100H            ;Minimum Node Space bytes
0268    DW      80H             ;Minimum Vector Space bytes
026A    DW      400H            ;Minimum Stack Space bytes
026C    DW      24              ;Ratio of total free space to stack
                                ;free space

                                ;REALLOCATION SENSITIVITY
026E    DW      4               ;Reallocation occurs after a garbage
0270    DW      3               ;collection if for any space, the
                                ;ratio of the size of the space to
                                ;the optimum size of the space is
                                ;greater than [026E]/[0270] or less
                                ;than [0270]/[026E].

0272    DW      BP              ;Address of memory word that contains
                                ;the variable (BP) stack pointer.

0274    DW      79              ;DEFAULT LINE LENGTH
                                ;Should be one less than the console
                                ;screen width.

0276    DB      10              ;DEFAULT RADIX BASE

0277    DB      7FH             ;CHARACTER MASK
                                ;The mask used on characters input
                                ;from files to strip parity bit.

0278    DB      7               ;BELL CHARACTER
                                ;The character output by muLISP to
                                ;sound the console bell.  Set byte to
                                ;0 to deactivate bell.

0279    DB      100             ;KEYBOARD INTERRUPT SENSITIVITY
                                ;Controls the frequency at which the
                                ;console is checked for an escape char.
                                ;Make larger for less frequent.

027A    DW      NXNAME          ;Address of the variable NXNAME.

027C    DW      ENDATM          ;Address of the variable ENDATM.

027E    DW      0               ;FIRST BYTE AFTER END OF TPA
                                ;A user imposed limit to the amount
                                ;of memory muLISP will use.
                                ;0 to use all available memory.
```

Appendix E: 8080 Machine Language Routine

```
ENDATM  EQU     4BA3H           ;END OF ATOM SPACE ADDRESS
BP      EQU     4BB7H           ;BP STACK PTR ADDRESS
CONS    EQU     745H            ;ENTRY ADDRESS TO CONS

                                ;ARGS:  HL: TREE
                                ;RSLT:  HL: COPY(TREE)
COPY:   LDA     ENDATM+1
        CMP     H
        RC                      ;Return if ATOM(TREE)
        MOV     E,M             ;DE: CAR(TREE)
        INX     H
        MOV     D,M
        INX     H
        MOV     C,M             ;BC: CDR(TREE)
        INX     H
        MOV     B,M
        LHLD    BP
        MOV     M,C
        INX     H
        MOV     M,B             ;Push CDR(TREE) on BP stack
        INX     H
        SHLD    BP              ;Update BP stack ptr
        XCHG                    ;HL: CAR(TREE)
COPY1:  CALL    COPY            ;HL: COPY(CAR(TREE))
        XCHG                    ;DE: COPY(CAR(TREE))
        LHLD    BP
        DCX     H
        MOV     B,M
        MOV     M,D
        DCX     H
        MOV     C,M             ;BC: CDR(TREE)
        MOV     M,E             ;Push COPY(CAR(TREE)) on BP stack
        MOV     H,B
        MOV     L,C             ;HL: CDR(TREE)
COPY2:  CALL    COPY            ;HL: COPY(CDR(TREE))
        XCHG                    ;DE: COPY(CDR(TREE))
        LHLD    BP
        DCX     H
        MOV     E,M
        DCX     H
        MOV     D,M             ;DE: COPY(CAR(TREE))
        SHLD    BP              ;Restore BP
        XCHG                      ;HL: COPY(CAR)      DE: COPY(CDR)
        JMP     CONS            ;Return HL: CONS(COPY(CAR),COPY(CDR))

LENGTH  EQU     $-USER          ;LENGTH OF USER ROUTINE
        END
```

CONCEPT INDEX

actual arguments 5-47
address typing 7-3,6
AI bibliography 8-1
AIDS 1-1; 6-1
ANIMAL game 2-6; 3-1
Apple II Computer
  control characters 5-69
  functions 5-66
  hardware requirements 1-4
  low-resolution graphics 5-67
  muLISP on 1-7
  special memory locations 5-69
arithmetic 5-29
argument type errors 4-8
Artificial Intelligence iii; 8-1
assignment functions 5-18
association list 5-5
atom space 4-4
attribute 5-22
auto-quoting 2-4; 4-1
auxiliary functions 5-1

backup, diskette 1-5
base page 7-1
binary trees iii; 4-3; 5-2
brackets 5-36
break characters 5-34
Break option 4-6; 5-58
break package 4-6; 5-56,58

car branch 5-2
car cell 4-1
case conversion 5-38,44
cdr branch 5-2
cdr cell 4-1
chain load 2-6
character type table 5-37
chronological ordering 5-16
circular list 5-9
COM files 5-54,55
comments 5-36,51
comparator functions 5-14
compiling 5-23
COMPRESS 5-26
COND, implicit 5-49,50
conditional 5-49
constants 5-23
constructor functions 5-6
Continue option 4-6; 5-58
control characters 5-28
control variables 5-38,44
copyright notice ii; 1-5
CP/M-80 1-4,5,7; 5-63; 7-7
current input file (CIF) 5-32,38
current input source (CIS) 5-32,38
current output file (COF) 5-40,44
current output sink (COS) 5-40,44

D-code 5-23
data compaction 4-5
data objects 4-1
data space 4-1
data space partition 4-4
data structures 4-1
decompiling 5-23
definition functions 5-23
demonstration programs 3-1,2
Directory Full Error 4-8; 5-41,59
Disk Full Error 4-8; 5-42,59
diskette backup 1-5
distilled code 5-23
DOCTOR program 3-1
dot notation 4-3; 5-42
dotted-pairs 4-3
driver loop 2-2; 5-56,57

echo 5-38,45
editor, muSTAR 6-1
empty list 4-3
End-Of-File Error 4-8; 5-42,60
environment 2-7; 5-54
environment functions 5-54
environment SYS files 2-7; 5-54
equality comparison 5-14
error handling functions 5-59
error messages 4-8
eval function 5-47
eval-LISP 2-2
eval-quote LISP 2-2; 5-57
evaluation functions 5-46
executive driver loop 2-2; 5-56,57
Executive option 4-6; 5-58
exit code 5-55,63; A-5
exiting muLISP 2-2; 5-55
EXPLODE 5-26
extended COND 2-4
external editor 2-6; 5-51; 6-1

file backup 5-41
flag functions 5-22

garbage collection 4-4; 5-6,52
GENSYM 5-26
glossary 9-1
greatest common divisor 5-31

HANOI, Tower of 3-2
hardware functions 5-61
header message 2-1
helper functions 5-1
hexadecimal numbers 5-52

IBM Personal Computer
  hardware requirements 1-4
  keyboard notation 1-6
  extended function keys 1-6; A-1
  CURSOR function 5-64
identity comparison 5-14
implied COND 2-4, 5-49,50
IMPLODE 5-26
incremental compilation 5-23
indicators 5-20
Insufficient Memory 4-8
interaction cycle 2-2
interactive environment iii
Interlisp 3-4
interrupt character 4-7; 5-37
interrupt message 4-7
interrupt vectors 5-62; 7-3
isomorphic structures 5-14
iterative loop 5-48,49

keyboard interrupt 4-7

LAMBDA definitions 5-47
length cell 4-2
line edit mode 5-38
line editing 2-3
line length 5-43,45
linked list 4-3
LISP iii; 2-4; 8-1
LISP bibliography 8-1
list notation 4-3; 5-42
LOAD command 2-7; 5-55
local variables 2-5; 5-18
logical functions 5-17
lower case conversion 5-44

machine language 5-23,62,63; 7-1
Maclisp 3-2,4
macros 5-32,39
master diskette 1-3
memory management 4-4; 5-52
Memory Space Exhausted 4-9; 5-53
METAMIND game 3-2
modifier functions 5-8
modulo 5-31
muLISP
  demonstration programs 3-2
  features 1-1
  header message 2-1
  lessons 3-3
  programming 2-4
  prompt 2-1
  runtime system 1-2; 5-55
  serial number 2-1
  system contents 1-3
  system requirements 1-4
  Tutorial System 3-3
  utility library files 3-4
  version date 2-1
  version number 2-1

muSTAR editor and debugger
  command summary 6-11
  console customization 6-12
  cursor control 6-6
  debugging options 6-3
  deleting text 6-9
  disk I/O options 6-4
  display control 6-7
  editor options 6-2
  entering text 6-8
  exit muSTAR option 6-5
  exiting editor 6-10
  finding names 6-9
  Insert mode 6-8
  leading blanks 6-15
  main menu 6-2
  moving text 6-9
  muSTAR Executive 6-14
  Non-insert mode 6-8
  pretty-printer 6-4
  quote operator 6-9
  text editing 6-6
  text translation 6-14
  theory of operation 6-14
  token 6-15
  tracing options 6-4
  translator 6-14
  word 6-7

name list 5-7
names 4-1
NIL 4-3; 5-11,17
NLAMBDA definitions 5-47
no-eval function 5-47
no-spread function 5-48
nodes 4-3

names 4-1
NIL 4-3; 5-11,17
NLAMBDA definitions 5-47
no-eval function 5-47
no-spread function 5-48
nodes 4-3
non-local exit 1-1
nonNIL 5-17
Nonnumeric Argument 4-9; 5-59
nonrecursive loops 2-5; 5-49
null list 4-3
number vector 4-2
numbers 4-2
numerical functions 5-29

object list 5-7
option prompt 4-6; 5-57

pname cell 4-2
pointer cells 4-1
pointer space 4-4; 5-6
predicate 5-48,49
primitive functions 5-1
print-name string 4-2; 5-25
printer control var. 5-44
printer functions 5-40
printout, hardcopy 2-3; 5-44
property 4-1; 5-20
property functions 5-20
property list 4-1; 5-20
property list cell 4-1
property value 5-20
pseudo-compiler 5-23

quoted strings 5-36,44

radix base 5-43
raw input mode 5-38
RDS command 2-6
read macros 5-32,39
reader control var. 5-38
reader functions 5-32
reading source files 2-6
real time systems 4-5
recognizer functions 5-11
reset, computer 4-7
resident editor 5-51; 6-1
Restart option 4-6; 5-58
return code 5-55
runtime system 1-2; 5-55

SAVE command 2-7; 5-54
selector functions 5-2
separator characters 5-34
side-effects 5-8,18
sign cell 4-2
Soft Warehouse, The ii; 2-1
source files 2-6
special characters 5-36
special functions 5-64
splice macros 5-39
spread function 5-48
stack space 4-4
stack operations 5-19
string functions 5-25
super-parentheses 5-36
SYS files 2-7; 5-54
SYS file NOT found 2-7; 4-10
system contents 1-3
System option 4-6; 5-58
system requirements 1-4

T 5-11,17
tasks 5-48
thrashing 4-5; 5-53
token 5-34; 6-15
truth values 5-17
type checking 4-1

Undefined Function 4-10; 5-47,59
upper case conversion 5-38
user interface functions 5-57
utility library files 3-4

value cell 4-1
vector cell 4-2
vector space 4-4

Zero Divide Error 4-10; 5-59

(This page intentionally left blank)

FUNCTION & VARIABLE NAME INDEX

This is an index of all the primitive functions, control variables, and constants in muLISP. The brackets following each function name enclose a list of names denoting the function's arguments. An asterisk, *, indicates an argument that is not evaluated prior to being passed to the function (see the discussion of no-eval functions in Chapter 5). All entries are followed by a category and the page number in Chapter 5 on which the entry is described.

Name [arguments]	Category	Page
AND [obj_1*, obj_2*, ..., obj_n*]	Logical	17
APPEND [$list_1$, $list_2$, ..., $list_n$]	Constructor	6
APPLY [function, argument-list]	Evaluator	47
ASCII [atom]	String	27
ASSOC [key, alist]	Selector	5
ATOM [object]	Recognizer	11
BEEP [pitch, duration]	Apple II	68
BREAK [object]	Interface	57
CAAAR [list]	Selector	3
CAADR [list]	Selector	3
CAAR [list]	Selector	2
CADAR [list]	Selector	3
CADDR [list]	Selector	3
CADR [list]	Selector	3
CALL [address]	Hardware	63
CAR [object]	Selector	2
CATCH [label, body*]	Evaluator	50
CDAAR [list]	Selector	4
CDADR [list]	Selector	4
CDAR [list]	Selector	3
CDDAR [list]	Selector	4
CDDDR [list]	Selector	4
CDDR [list]	Selector	3
CDR [object]	Selector	2
CHARTYPE [character, n]	Reader	37
CLRSCRN []	Special	64
COMMENT [comments ...]	Evaluator	51
COND [$cond_1$*, $cond_2$*, ..., $cond_n$*]	Evaluator	49
CONS [obj_1, obj_2]	Constructor	6
CURSOR [row, column]	Special	64
DIFFERENCE [n_1, n_2, ..., n_m]	Numerical	29
DIVIDE [n, m]	Numerical	31
DRIVER []	Interface	56
ECHO	Reader	38
ECHO	Printer	45
EOF []	Error	60
EQ [obj_1, obj_2]	Comparator	14

Name [arguments]	Category	Page
EQUAL [obj_1, obj_2]	Comparator	14
EVAL [object]	Evaluator	46
EVENP [object]	Recognizer	13
EXECUTE [program, command line]	Hardware	63
FINDSTRING [$atom_1$, $atom_2$, n]	String	25
FLAG [name, attribute]	Flag	22
FLAGP [name, attribute]	Flag	22
GCD [n_1, n_2, ..., n_m]	Numerical	31
GET [name, key]	Property	20
GETD [name]	Definition	23
GRAPHICS [mode, color]	Apple II	66
GREATERP [n_1, n_2, ..., n_m]	Comparator	15
INTERRUPT [vector number]	Hardware	62
LAMBDA	Constant	47
LAST [list]	Selector	4
LCONC [pointer, list]	Modifier	10
LENGTH [object]	String	26
LENGTH	String	27
LESSP [n_1, n_2, ..., n_m]	Comparator	15
LINELENGTH [n]	Printer	43
LINELENGTH	Printer	45
LIST [obj_1, obj_2, ..., obj_n]	Constructor	6
LOAD [file, drive]	Environment	55
LOOP [$task_1$, $task_2$, ..., $task_n$]	Evaluator	48
MEMBER [object, list]	Comparator	15
MEMORY [address, byte]	Hardware	61
MINUS [n]	Numerical	29
MINUSP [object]	Recognizer	12
MOVD [$name_1$, $name_2$]	Definition	24
NTH [list, n]	Selector	4
NTH [n, list]	Selector	4
NAME [object]	Recognizer	11
NCONC [$list_1$, $list_2$, ..., $list_n$]	Modifier	8
NIL	Constant	11
NLAMBDA	Constant	47
NONNULL [object]	Recognizer	12
NONNUMERIC [function, arguments]	Error	59
NOT [object]	Logical	17
NULL [object]	Recognizer	11
NUMBERP [object]	Recognizer	11
OBLIST []	Constructor	7
OR [obj_1*, obj_2*, ..., obj_n*]	Logical	17
ORDERP [$atom_1$, $atom_2$]	Comparator	16
PACK [list]	String	26

Name [arguments]	Category	Page
PADDLE [n]	Apple II	68
PLOT [x, y, color]	Apple II	67
PLUS [n_1, n_2, ..., n_m]	Numerical	29
PLUSP [object]	Recognizer	12
POP [name*]	Assignment	19
PORTIO [port, byte]	Hardware	61
PRIN1	Printer	44
PRIN1 [object]	Printer	42
PRINT	Printer	44
PRINT [object]	Printer	42
PROG1 [$task_1$*, $task_2$*, ..., $task_n$*]	Evaluator	50
PUSH [object, name*]	Assignment	19
PUT [name, key, object]	Property	20
PUTD [name, definition]	Definition	24
QUOTE [object*]	Evaluator	46
QUOTIENT [n, m]	Numerical	30
RADIX [n]	Printer	43
RATOM []	Reader	34
RDS [file, type, drive]	Reader	32
RDS	Reader	38
READ []	Reader	35
READ	Reader	38
READCH [peek-flag]	Reader	33
READCH	Reader	38
READP []	Reader	36
READPTR [pointer]	Reader	33
RECLAIM []	Memory	52
RECLAIM	Memory	52
REGISTER [n]	Hardware	62
REMAINDER [n, m]	Numerical	30
REMFLAG [name, attribute]	Flag	22
REMPROP [name, key]	Property	20
REVERSE [list, object]	Constructor	7
RPLACA [obj_1, obj_2]	Modifier	8
RPLACD [obj_1, obj_2]	Modifier	8
SAVE [file, drive]	Environment	54
SET [name, object]	Assignment	18
SETQ [name*, object]	Assignment	18
SPACES [n]	Printer	43
SUBSTRING [atom, n, m]	String	25
SYSTEM []	Environment	55
T	Constant	11
TCONC [pointer, object]	Modifier	9
TERPRI [n]	Printer	43
TEXT []	Apple II	66
THROW [label, expression]	Evaluator	51
TIME [reset-flag]	Special	65
TIMES [n_1, n_2, ..., n_m]	Numerical	30

Name [arguments]	Category	Page
UNDEFINED [list]	Error	59
UNPACK [atom]	String	26
WRITEPTR [pointer]	Printer	41
WRS [file, type, drive, reopen]	Printer	40
WRS	Printer	44
ZERODIVIDE [function, arguments]	Error	59
ZEROP [object]	Recognizer	12
' []	Read Macro	39

MICROSOFT®
10700 Northup Way, Bellevue, WA 98004

Software Problem Report

Name _____

Street _____

City _____ State _____ Zip _____

Phone _____ Date _____

Instructions

Use this form to report software bugs, documentation errors, or suggested enhancements.
Mail the form to Microsoft.

Category

____ Software Problem

____ Software Enhancement

____ Documentation Problem
(Document # _____)

____ Other

Software Description

Microsoft Product _____

Rev. _____ Registration # _____

Operating System _____

Rev. _____ Supplier _____

Other Software Used _____

Rev. _____ Supplier _____

Hardware Description

Manufacturer _____ CPU _____ Memory _____ KB

Disk Size _____" Density: Sides:

Single _____ Single _____

Double _____ Double _____

Peripherals _____

_____

Problem Description

Describe the problem. (Also describe how to reproduce it, and your diagnosis and suggested correction.) Attach a listing if available.

Microsoft Use Only

Tech Support _____ Date Received _____

Routing Code _____ Date Resolved _____

Report Number _____

Action Taken: