TR-678
July 1978

N00014-76C-0477
NSG-7253

Maryland LISP Reference Manual

Philip Agre
Computer Science Department
University of Maryland
College Park, Maryland 20742

This work was supported in part by the Office of Naval Research under grant number N00014-76C-0477 and the National Aeronautics and Space Administration under grant number NSG-7253. Their support is gratefully acknowledged.

Table of Contents

Section 1 Introduction 2

Section 2 Maryland LISP Intrinsic Functions 19

Section 3 Maryland LISP Function Packages 112

Section 4 Maryland LISP Assembler 180

Section 5 Appendices 234

Section 1

Introduction

Contents

1. About This Document 4
2. Highlights of Maryland LISP 5
3. History and Acknowledgements 7
4. LISP and UNIVAC EXEC-8 8
4.1. EXEC-8 8
4.2. Files 8
4.3. Character Sets and Special Characters 9
4.4. Control Cards of Use in LISP 10
4.5. Interrupting LISP 11
5. Maryland LISP Objects and Syntax 12
5.1. Cons Nodes, Lists, and Extended Lists 12
5.2. Numeric Types 12
5.3. Atomic Symbols 13
5.4. Strings 13
5.5. Other LISP Data Structures 13
5.6. LISP Directives 14
6. Maryland LISP Structures and Concepts 15
6.1. Variable Binding 15
6.2. Maryland LISP Functions 15
6.3. Control Structures 15
6.4. Storage Management 16
7. Bibliography 17

1. About This Document

This report documents Maryland LISP, a dialect of the LISP language developed at the Universities of Wisconsin and Maryland. The reader is assumed to have some knowledge of LISP. There exist several excellent texts on LISP, including [Wei67], [Rie74], and [Sik76].

This report is divided into several sections, covering all aspects of the use of Maryland LISP. Section 1 serves as an introduction. Section 2 describes the various functions and directives available to the Maryland LISP user. Section 3 documents Maryland LISP's extensive library packages. The final section documents the Maryland LISP assembler and provides insights into the implementation of Maryland LISP. There are also several appendices on various topics.

2. Highlights of Maryland LISP

Maryland LISP is available on the UNIVAC 1100/40 and 1108 machines at the University of Maryland. It can be called by:

```
@LISP*LIB.LISP .
```

If the D option is specified, Maryland LISP will run in demand mode. The B option causes batch mode operation. If neither of these options is specified, Maryland LISP will run in whatever mode the user is in. The CAR and CDR functions can be made not to check if their arguments are cons nodes if the Z option is specified. The F option can be used to cause the stack unwinding routines (see, for example, :PEEK on page 108) to print out every object on the stack rather than just alists and s-expressions being EVALuated. The X option can be used to cause LISP to terminate if an error condition is found. In addition, the V option, if specified on the 1100/40, causes Maryland LISP to use a form of virtual memory described in an appendix (see page 236).

All the programs which accompany Maryland LISP reside in LISP*LIB., and the user can get a list of what is there by doing:

```
@PRT,T LISP*LIB.
```

With regard to implementation, Maryland LISP uses a deep-binding strategy together with a global binding capability for system constants. There are eleven data types, numbered according to this table:

type #	data type
0	cons node
1	integer
2	octal
3	floating-point number
4	system code
5	compiled code
6	linker node
7	atomic symbol
8	string node
9	buffer
10	unallocated page

Functions are implemented as "linker nodes" rather than being identified with particular atoms. Therefore, a function definition is represented as the value of an atom, avoiding many of the ambiguities associated with "special-cell" function bindings. Data nodes of like types are collected together on 128-word "pages", and the type of a given object is determined by its address through a table of page types. Evaluation data is maintained on two stacks, a control stack and a value stack. All

Maryland LISP data transactions, including strings (and excluding special-purpose control code I/O), are in UNIVAC's 64-character FIELDATA character set.

Some of the more important and interesting features of Maryland LISP include:

(1) Several general-purpose debugging and error-handling facilities,

(2) String-handling functions,

(3) Extensive I/O capabilities, including random- and sequential-access file I/O,

(4) Optional virtual memory extension to 128K,

(5) A program-callable interface to the University of Maryland Text Editor,

(6) A facility (LOAD/DUMP) for saving and restoring the binary representations of structures and compiled code without reinterpretation or recompilation, and

(7) An extensive library of functions, including an assembler, array and matrix facilities, a bignum manipulation package, implementations of Stanford MLISP and Micro-Planner, a prettyprinter, a compiler, and a debugging package.

3. History and Acknowledgements

The program which has evolved into Maryland LISP was begun in 1969 by Eric Norman of the Univerity of Wisconsin. Several basic implementation decisions made in the writing of Wisconsin LISP remain in Maryland LISP, though there have been many changes and extensions. The bulk of the code for Maryland LISP's control structures, as well as the compiler, prettyprinter, debugger, and Micro-Planner programs are revised versions of routines written at Wisconsin.

Wisconsin LISP arrived at the University of Maryland in 1974 and was modified and adapted by, among others, Chuck Rieger and Milt Grinberg. Mache Creeger added the string functions, random-access file I/O, the text editor interface, ASCII character I/O, an implementation of Stanford MLISP, the Suspend-Resume Package, the contingency routines, and numerous other extensions. Phil Agre added the sequential file I/O functions, a new garbage collector, the virtual memory option, the assembler, the array, matrix, and bignum packages, and the core dumping and mail routines.

The authors wish to thank numerous individuals for their efforts, advice, and criticisms. Primary among these are Chuck Rieger, Hanan Samet, Milt Grinberg, Phil London, Steve Small, Rich Wood, and Gyorgy Fekete.

4. LISP and UNIVAC EXEC-8

This chapter contains the information about UNIVAC's EXEC-8 operating system that one needs to know to use Maryland LISP effectively. The University of Maryland Computer Science Center (CSC) publishes two useful manuals on this subject, [Doy76] and [Pro76], and these are available from the Center's Program Library.

4.1. EXEC-8

After logging into one of Maryland's UNIVAC machines, one is talking to the EXEC-8 operating system. Commands to EXEC-8 begin with at-signs ("@"), and are not read until a carriage return is entered. Normally, any such "control card" will cause whatever is happening to terminate and whatever is specified in the command to begin. Exceptions are few and include @ADD, @EOF, and @ENDX, which are explained later. LISP manages to defeat this mechanism, as we shall see.

4.2. Files

For the LISP user's purposes, there are three kinds of files, program, data, and random-access. In general, an EXEC-8 file is an unstructured area of mass storage space. Any structure which is placed on the file is completely defined by the user's software. Most text files are in the system Standard Data Format (SDF), in which each line of text is a separate record with a (usually invisible) header word. It is this format with which the Text Editor [Hay77] deals.

The syntax of a file name is:

[<qualifier>*]<name>.

where the default <qualifier> is the user's project identification. An important concept for LISP users is the "internal name". This is a one to six character alphanumeric string by which the system can uniquely refer to a file. If <qualifier> is the user's project identification, then the <name> field will serve as an internal name for the file. Also, one can establish an alternate name for the file with the EXEC-8 processor @USE:

@USE <new-name>.,<file>.

where <new-name> is some 1 to 12 character alphanumeric name. The Maryland LISP functions which deal with file names all require internal file names. These functions are LOAD, DUMP, FIOPEN, COPYIN, and COPYOUT.

A program file is roughly the EXEC-8 equivalent of what is usually known as a user directory, a collection of smaller files known as "elements" under a table of contents. The syntax of an element is:

[<file-name>.]<elt-name>[/<version>]

where the default <file-name> is the user's workspace file. The <version> field is an optional extension to <elt-name>. Both <elt-name> and <version> should be 1-to-12 character alphanumeric names. Elements come in four types, symbolic, relocatable, absolute, and omnibus. The ordinary LISP user has need for only symbolic and omnibus element types. A symbolic element contains text in Standard Data Format [Uni78]. It is usually created by entering the Text Editor with the name of an element which does not exist. In this case, the Editor creates the element automatically. Omnibus elements have no predefined structure; the user's software can impose its own when the traditional record structure is inconvenient. LISP uses omnibus elements with the binary dumping function DUMP and its reloader LOAD.

A data file is a block of mass storage which can be used by a user program for any purposes in any format. Usually, data is in Standard Data Format. A random access file is just a data file with a structure imposed by the random access routines, allowing arbitrary records to be accessed.

4.3. Character Sets and Special Characters

All LISP data transactions (except those of the AREAD and AXMIT device control functions) are in UNIVAC's infamous FIELDATA character set (described fully in the appendix on page 237). FIELDATA is a 64-character set including upper-case letters only and no control codes.

There are several special control characters which one should know about before using LISP. The line delete character is question-mark ("?"). Typing this character will cause the current line to be ignored by the system. This can be changed by doing:

@@TTY L,<new-character>

The character delete character (called the rubout or backspace on other systems) is back-arrow or underline ("_"). This is frequently changed to the backspace character control-H (denoted ^H) by doing:

@@TTY C,^H

Any such system-defined special character can be entered literally (ie, without any special meaning) by preceding it with the escape code (marked ESC on most terminals).

LISP defines several special characters of its own. Three levels of parenthesization are allowed, using the square-bracket ("[]"), angle-bracket ("<>"), and parentheses ("()") characters. A closing delimiter (one of "]", ">", ")") matches all opening delimiters back to one of its own type. This is useful for closing off a long s-expression typed at one's terminal, for example:

```
(PLUS (QUOTIENT A B) (TIMES C (PLUS E (ADD1 F]
```

The period character (".") has two uses in LISP, in the syntaxes for dotted pairs and floating-point numbers. The distinction is explained in the paragraph on the syntax of Maryland LISP objects. The double-quote character (""") is used to delimit strings. In addition, by means of the READMAC and DELIM functions one may assign special meanings to other characters. Any character may be used literally in LISP by prefixing it with the LISP escape character, exclamation-point ("!"). For example, the following will be interpreted as a single LISP atomic symbol which will print as (A . B):

```
!(A! !.! B!)
```

4.4. Control Cards of Use in LISP

Maryland LISP has an arrangement with EXEC-8 whereby all control cards with the exception of "@ED", "@EOF", "@ADD", and "@ENDX" will be rejected as input. These four have special meaning in LISP.

An @ED card can be entered at any time when LISP wants to read input to enter the Text Editor. The Text Editor will return back to LISP with all LISP data structures intact. The user should take care that any @ED control card submitted in LISP has the correct format and refers to an existing file. Otherwise, the Text Editor will exit in an error condition and trigger the LISP contingency routines. Often, LISP finds this serious enough to ask the user whether LISP should be aborted, though this is usually not necessary.

The @ADD control card is one of UNIVAC's few contributions to the general welfare of mankind. When @ADD <file>., @ADD <element>, or @ADD <file>.<element> is entered at the user's terminal, the contents of that file or element are inserted into the input stream as though they had been typed in by the user. This is most useful for entering files full of LISP function definitions into LISP. Indeed, each of the chapters in the section documenting the Maryland LISP function packages gives an @ADD card which can be used to load the functions being discussed.

The @EOF card has basically the same effect as the :STOP

directive; it causes LISP to exit normally.

The @ENDX card is used to turn off the mode which allows LISP to ignore all but the four control cards described here. If this is done, any control card but @ADD will cause LISP to exit and do whatever it is the card indicates.

4.5. Interrupting LISP

To interrupt a LISP computation, enter the character sequence @@X. If LISP is printing output, it can be halted by hitting the BREAK key prior to entering @@X. This causes control to be transferred to the LISP contingency routine, which allows any garbage collection or bank swapping which may have been in effect to finish, and then perform any action the user might have defined for such a case with BRKCON (see page 72) or, if none, return to the most recent READ-EVAL-PRINT loop.

To cause LISP to be immediately terminated, enter @@X TIO, wait for a cryptic message from EXEC-8, and type @ENDX.

5. Maryland LISP Objects and Syntax

This chapter explains the syntaxes of the various things one can type at LISP, namely the various data structures and the directives. Implementation details on each are given starting on page 206. There are eleven Maryland LISP data types, six of which can be represented symbolically and interpreted by the read routines. These are cons nodes, integers, octal numbers, real numbers, atomic symbols, and strings.

5.1. Cons Nodes, Lists, and Extended Lists

Cons nodes form the structure of the s-expressions which are the basic data structure of LISP. Syntactically, a cons node can participate in either a "list" or an "extended list", of which the "dotted pair" is a special case. An expression of the form (S1 S2 ... Sn) is called a list. An extended list is a construct of the form (S1 S2 ... Sn . T), where T is referred to as the "tail" of the list. For example, we have:

(A (B C) D) is a list of 3 elements
(A (B C) . D) is an extended list
((A B) . (C D)) is a dotted pair

We define the term "atom" to mean any LISP data structure which is not a cons node. This term is sometimes loosely used to mean "atomic symbol", which is a specific data type described below.

5.2. Numeric Types

Maryland LISP provides three numeric types, integers, octals, and reals. Integers and octals are both "fixed-point numbers", and as such are treated identically by all routines except the I/O routines.

Syntactically, integers are denoted as strings of decimal digits, possibly with a sign prefix ("-" or "+", with the latter the default).

An octal number is of the form <n>Q[<e>], where <n> is a string of octal digits, and <e> is an optional integer suffix denoting the number of implied zeros following <n>. For example, 17Q3=17000Q. Octals are always printed with trailing zeros rather than the <e> notation.

A real (floating-point) number is of the form [<s>]<n>.<d>[E<e>], where <s> is an optional sign, <n> and <d> are non-null sequences of decimal digits, and <e> is a (possibly signed) scientific notation exponentiation factor, where <x>E<e> represents "<x> times 10 to the <e> power". Reals are printed in this format, where <n> is always a single digit, and an E<e>

suffix is provided if necessary. For example, 345.27 is printed as 3.4527E2, and 0.00458 is printed as 4.58E-3. The forms 3908. and .045 are not legal, since the decimal points might then also be interpretable as "cons" points. As a general rule, any non-escaped period will be interpreted as a cons point unless there are no opening delimiters preceding it or it has all digits to the left and right.

These numbers are all single-precision. If fixed-point numbers larger than 36 bits are required, the bignum package (see page 128) can be used.

5.3. Atomic Symbols

Maryland LISP uses the usual LISP definition of an atomic symbol; an "indivisible" node which has associated with it a value, a property list, and a print name, and which is represented syntactically by its print name. Maryland LISP uses separate fields for these three parts of an atomic symbol, mainly for purposes of efficiency. Basically, any input object which cannot be interpreted as being any other kind of data type is assumed to be an atomic symbol. By using the LISP escape character, one can have atomic symbols with arbitrarily unusual print names, although it is generally not a good idea to construct an atomic symbol whose print name exceeds 80 characters in length. The following are examples of atomic symbols:

```
ABCDEF
SECOND-ENTRY-IN-SCHEME-TABLE
1098372642R
!4
!]! !.XYZ!!!
!>X
```

5.4. Strings

Maryland LISP has a FIELDATA string facility, in which the strings may be of any length. A Maryland LISP string is a sequence of characters enclosed by double-quotes (`"`). No characters have special meanings inside of strings since LISP does not attempt to parse strings. In order to include a double-quote in a string, enter two consecutive double-quotes (e.g., "HE SAID ""HI""."). The characters in the string are numbered beginning with 1 for the first character, so that the number of the last character coincides with the length of the string. The atomic symbol NIL serves as the string of length 0.

5.5. Other LISP Data Structures

Maryland LISP has eleven data structures in all, the six

described above and five which cannot be parsed as input. Two of the five, unallocated pages and system code, are only called data types for the convenience of LISP hackers. The other three, compiled code, linker nodes, and buffers, are printed by LISP as either [<a>], where <a> is some atomic symbol which is bound constantly to the object, or [<t>:<addr>], where <t> is the numeric type of the object and <addr> is its octal address in core. They are described in the notes on Maryland LISP implementation (see page 206).

5.6. LISP Directives

There are several other things one can type at LISP. The directives (see page 105) begin with a colon (":") in column 1 and perform operations relating to the LISP environment, such as turning output listing and input echoing echoing on and off, returning to the top level of the current level of supervision, and interfacing with EXEC-8 file-handling commands.

6. Maryland LISP Structures and Concepts

This chapter attempts to explain to an experienced user of another dialect of LISP how things are done in Maryland LISP. Emphasis is placed on those Maryland LISP conventions which differ from those of other dialects.

6.1. Variable Binding

Maryland LISP uses two binding strategies, global "constant" bindings and local "fluid" bindings. A constant binding is a pointer stored in the value field of an atomic symbol. Constant bindings are usually used for implicit functions and for other values which need not change during a LISP run. Maryland LISP's fluid binding strategy is deep binding, implemented through a standard cons node association list. Because retrieval of constant bindings must be a fast operation, a variable may not have both a constant and a fluid binding; if a constant binding is present, all functions dealing with that atomic symbol's "value" will refer to the constant binding. Attempting to associate a fluid value with a constantly bound variable (through function application or PROG entry) results in an error message.

6.2. Maryland LISP Functions

In Maryland LISP, a function is embodied by a "linker node", which is a data object returned by LAMBDA, LAMDA, or FUNCTION, and which contains the information necessary to use the function. A linker node may or may not be bound to an atomic symbol, and if it is bound to one, the binding is implemented as a simple pointer to the linker node, just like normal bindings.

There are three kinds of functions in Maryland LISP: regular functions, special forms, and macros (known roughly as expr's, fexpr's, and macro's, respectively, in some other dialects). Regular function linker nodes may be passed as arguments to functions in the same manner as any other data object; specifying "'EQ" as an argument will pass the atomic symbol EQ, wheras specifying "EQ" will pass the function definition attached to EQ. The FUNCTION function serves only as a saver of enviroments, and it is not necessary to apply FUNCTION to every functional argument. The layout of a linker node is described in detail on page 203.

6.3. Control Structures

Maryland LISP has two stacks, the control stack and the value stack, described in the implementation notes on page 212. Several levels of control can be set up with the LISP function (see page 26), which calls the standard READ-EVAL-PRINT loop with

a user-defined read routine and establishes a "level of supervision", below which error exits of functions cannot fall.

Given an s-expression (F A1 A2 ... An) to evaluate, the following algorithm is performed in the Maryland LISP implementation:

```
evaluate F
if F did not evaluate to a function
   then perform the action specified through FUNCON
           to get a function
endif
if F evaluated to a macro or special form
   then push the argument list (A1 ... An) intact
        jump to F's value
   else evaluate each Ai and push the results
        jump to F's value
endif
```

6.4. Storage Management

Maryland LISP's data area is partitioned into 128-word pages, each of which is unallocated or is dedicated to a single data type. The data type of an arbitrary node is computed by looking up its page number in a master page table (see page 218). In one mode, LISP will operate a virtual memory scheme to allow the user access to 128K of memory. This is documented in an appendix (see page 236). The oblist in Maryland LISP is a hash table with 64 entries pointing to chains of atomic symbols with the same hash code.

7. Bibliography

[All78] Allen, J., The Anatomy of LISP, McGraw-Hill, New York, 1978

[Bau72] Baumgart, B., Micro-Planner Alternate Reference Manual, Stanford AI Lab Operating Note No. 67, April 1972

[Bob72] Bobrow, R. J., R. R. Burton, and D. Lewis, UCI LISP Manual, Information and Computer Science Dept., University of California at Irvine, Technical Report 21, October 1972

[Doy76] Doyle, J., Getting Started on the University of Maryland Univac Computers, Note CN17, Computer University of Maryland Computer Science Center, May 1976

[Fri74] Friedman, D. P., The Little LISPer, Science Research Associates, Palo Alto, California, 1974

[Gre77] Greenblatt, R., et. al., LISP Machine Progress Report, AI Memo 444, Artificial Intelligence Lab, MIT, August 1977

[Hag77] Hagerty, P. E., and K. E. Sibbald, Text Editor User's Guide, Computer Note CN7.11, University of Maryland Computer Science Center, May 1977

[Knu68] Knuth, D. E., The Art of Computer Programming, Volume 1, Reading, Mass., 1968

[Knu69] Knuth, D. E., The Art of Computer Programming, Volume 2, Addison-Wesley, Reading, Mass., 1969

[McC60] McCarthy, J., Recursive Functions of Symbolic Expressions and Their Computation by Machine, Communications of the ACM, vol. 3, no. 4 (1960), 184-195

[McC62] McCarthy, J., LISP 1.5 Programmer's Manual, MIT Press, Cambridge, Mass., 1962

[McD74] McDermott, D. V., and G. J. Sussman, The Conniver Reference Manual, AI Memo 259a, Artificial Intelligence Lab, MIT, January 1974

[Mos70] Moses, J., The Function of FUNCTION in LISP, or Why the Funarg Problem Should be Called the Environment Problem, AI Memo 199, Artificial Intelligence Lab, MIT, 1970

[Pro76] Prochazka, J. C., and E. U. Putnam, University of Maryland Univac 1100 Series Reference Manual, Computer Note CN13.3, University of Maryland Computer Science Center, January 1976

[Rie74] Rieger, C., Everyman's LISP, unpublished lecture notes, 1974

[Sik76] Siklossy, L., Let's Talk LISP, Prentice-Hall, Englewood Cliffs, N.J., 1976

[Smi70] Smith, D. C., MLISP, Stanford Artificial Intelligence Memo AIM-135, October 1970

[Ste76] Steel, G., Lamuda, the Ultimate Imperative, AI Memo 353, Artificial Intelligence Lab, MIT, March 1976

[Sus71] Sussman, G., T. Winograd, and E. Charniak, Micro-Planner Reference Manual, MIT AI Lab Memo 203a, December 1971

[Tei74] Teitelman, W., InterLISP Reference Manual, Xerox Palo Alto Research Center, 1974

[Uni74] UNIVAC 1100 Series Assembler Programmer Reference, UNIVAC Publication UP-4040 Rev. 4, Sperry-Univac, 1974

[Uni78] UNIVAC 1100 Series Executive System Programmer Reference, UNIVAC Publication UP-4144.11, Sperry-Univac, 1978

[Wei74] Weissman, C., LISP 1.5 Primer, Dickenson, Encino, California, 1974

Section 2

Maryland LISP Intrinsic Functions

Contents

1. Introduction 25
2. LISP Control Functions 26
2.1. quote 26
2.2. eval 26
2.3. do 26
2.4. lisp 26
2.5. oblist 26
2.6. type 27
2.7. stack 27
2.8. alist 28
2.9. rand 28
3. S-Expression Manipulation Functions 30
3.1. Basic Functions 30
3.1.1. car 30
3.1.2. cdr 30
3.1.3. cons 30
3.1.4. *car 31
3.1.5. *cdr 31
3.2. S-Expression Predicates 31
3.2.1. eq 31
3.2.2. equal 31
3.2.3. atom 32
3.2.4. null 32
3.2.5. member 32
3.3. Function Application Functions 33
3.3.1. into 33
3.3.2. mapc 33
3.3.3. onto 34
3.3.4. map 34
3.3.5. index 35
3.3.6. ondex 35
3.4. S-Expression Pseudo-Functions 36
3.4.1. rplaca 36
3.4.2. rplacd 37
3.4.3. nconc 37
3.5. Other S-Expression Functions 38
3.5.1. list 38
3.5.2. append 38
3.5.3. length 38
3.5.4. subst 39
3.5.5. reverse 39
3.5.6. nth 40
3.5.7. assoc 40
4. Atom Manipulation Functions 42
4.1. atom 42
4.2. gensym 42
4.3. string 42
4.4. atsymb 42
4.5. *car 42
4.6. *cdr 43
4.7. compress 43
4.8. explode 43

4.9. explode2 44
4.10. remob 44
4.11. erase 45
5. Value and Function Definition Functions 46
5.1. Value Assignment Functions 46
5.1.1. csetq 46
5.1.2. cset 46
5.1.3. setq 46
5.1.4. set 46
5.1.5. define 46
5.2. Function Assignment Functions 47
5.2.1. defspec 47
5.2.2. defmac 47
5.3. Function Definition Functions 48
5.3.1. lambda 48
5.3.2. lamda 49
5.3.3. function 50
6. Input-Output Functions 52
6.1. Input Functions 52
6.1.1. Input Parsing Functions 52
6.1.1.1. read 52
6.1.1.2. token 52
6.1.1.3. readmac 53
6.1.1.4. delim 53
6.1.2. Input Buffer Manipulation Functions 54
6.1.2.1. clearbuff 54
6.1.2.2. curchar 54
6.1.2.3. setcol 54
6.1.2.4. backsp 55
6.1.3. Record and Character Reading Functions 55
6.1.3.1. readch 55
6.1.3.2. readrec 56
6.1.4. Miscellaneous Input Functions 56
6.1.4.1. enuchar 56
6.2. Output Functions 56
6.2.1. S-Expression Printing Functions 56
6.2.1.1. print 56
6.2.1.2. prin1 56
6.2.1.3. prin2 57
6.2.2. Output Buffer Manipulation Functions 57
6.2.2.1. currcol 57
6.2.2.2. terpri 57
6.2.3. Output Formatting Functions 57
6.2.3.1. space 57
6.2.3.2. digits 57
6.2.3.3. plimit 58
6.2.4. Print Length Functions 58
6.2.4.1. plength 58
6.2.4.2. plength2 59
6.3. ASCII I/O Functions 59
6.3.1. axmit 59
6.3.2. axmit1 60
6.3.3. aread 60
6.4. Sequential File I/O Functions 60

6.4.1. fiopen 61
6.4.2. ficlose 61
6.4.3. clearbuff 61
6.4.4. terpri 61
6.5. Random Access File I/O Functions 62
6.5.1. fiopen 63
6.5.2. fiprint 63
6.5.3. firead 64
6.5.4. fierase 64
6.5.5. fidrop 64
6.5.6. fipack 65
6.5.7. fitoc 65
6.5.8. fiprintrec 65
6.5.9. fireadrec 65
6.5.10. copyout 65
6.5.11. copyin 66
7. Debugging and Error Handling Functions 67
7.1. Error Handling Functions 67
7.1.1. attempt 67
7.1.2. error 68
7.1.3. conlim 68
7.1.4. backtr 69
7.2. Contingency Definition Functions 69
7.2.1. offender 70
7.2.2. eofcon 71
7.2.3. misscon 71
7.2.4. funcon 71
7.2.5. oinocon 71
7.2.6. carcon 71
7.2.7. brkcon 72
7.3. Tracing Functions 72
7.3.1. break 72
7.3.2. unbreak 73
8. Property List Functions 74
8.1. put 74
8.2. get 74
8.3. remprop 74
8.4. flag 75
8.5. ifflag 75
8.6. unflag 75
8.7. prop 76
9. Logical and Conditional Functions 77
9.1. cond 77
9.2. and 77
9.3. or 78
9.4. not 78
10. Program Feature Functions 79
10.1. prog 79
10.2. go 79
10.3. return 80
11. Numeric Functions 81
11.1. Arithmetic Predicates 81
11.1.1. numberp 81
11.1.2. zerop 81

11.1.3. equal 82
11.1.4. greaterp 82
11.1.5. lessp 82
11.1.6. minusp 83
11.1.7. fixp 83
11.1.8. floatp 83
11.2. Arithmetic Functions 83
11.2.1. plus 83
11.2.2. times 84
11.2.3. difference 84
11.2.4. quotient 84
11.2.5. remainder 85
11.2.6. add1 85
11.2.7. sub1 85
11.2.8. minus 86
11.3. Fortran Library Math Routines 86
11.3.1. sin 86
11.3.2. cos 86
11.3.3. log 86
11.3.4. power 87
11.4. Bitwise Logical Functions 87
11.4.1. logor 87
11.4.2. logand 87
11.4.3. logxor 88
11.4.4. leftshift 88
11.5. Miscellaneous Numeric Functions 88
11.5.1. fuzz 88
11.5.2. entier 89
12. String Functions 90
12.1. size 90
12.2. stringp 90
12.3. srev 90
12.4. match 91
12.5. cat 91
12.6. substring 91
12.7. readstr 92
12.8. string 92
12.9. atsymb 92
13. Executive Interface Functions 94
13.1. exec 94
13.2. pct 94
13.3. twait 95
13.4. ea 95
13.5. dump 96
13.6. load 96
14. Program Statistics Functions 97
14.1. Timing Functions 97
14.1.1. time 97
14.1.2. gctime 97
14.2. Time/Date Functions 97
14.2.1. date 97
14.2.2. dtime 98
14.3. Memory Management Functions 98
14.3.1. trash 98

14.8.2. memory 98
14.8.3. grow 98
14.3.4. swaps 99
14.4. *pack 99
15. Compiler Functions 100
15.1. Function Definition Retrieval Functions 100
15.1.1. *def 100
15.1.2. *spec 100
15.1.3. *macro 101
15.1.4. *chain 102
15.2. Code Generating Functions 102
15.2.1. *begin 102
15.2.2. *emit 102
15.2.3. *org 103
15.3. Other Compiler-Oriented Functions 103
15.8.1. *ept 103
15.3.2. *exam 103
15.3.3. *deposit 103
15.3.4. manifest 104
15.8.5. buffer 104
16. LISP Directives 105
16.1. :LOAD 105
16.2. :END 105
16.8. :LIST 105
16.4. :UNLIST 105
16.5. :CKPT 105
16.6. :RSTR 106
16.7. :LISP 106
16.8. :EXEC 106
16.9. :CODE 107
16.10. :BANK 107
16.11. :TIME 107
16.12. :BACK 107
16.13. :PEEK 108
16.14. :STOP 108
16.15. :OOPS 108
16.16. :DATA 108
17. Alphabetic Index 110

1. Introduction

This section is a description of the intrinsic functions of Maryland LISP, that is, the functions which are available in a minimal system called up via "@LISP*LIB.LISP". They are divided somewhat arbitrarily into an outline according to their various purposes, and an alphabetic index is provided in the last chapter for easy reference. In the syntax descriptions provided with each function, angle brackets (ie., "<...>") delimit logical tokens whose types and meanings are explained in the text defining the purpose of each function. A function's description may be accompanied by examples which, in addition to giving some idea of the purpose of the function in question, make clear the function's behavior in special cases or give some interesting application of the function.

In addition to the function names listed below, three other atomic symbols have constant bindings when LISP is loaded. These are NIL (value NIL), F (value NIL), and T (value T). One only tampers with these values at one's own risk.

The user should note that most of these functions do very little or no checking of the correctness of their arguments, so that a function given erroneous (eg., not enough or too many, or of the wrong type) arguments will most likely return nonsensical results or have unpredictable effects.

The author wishes to acknowledge the efforts and advice of Mache Creeger, who wrote the text describing the ASCII character I/O and random access I/O functions, as well as many of the facilities that this section documents.

2. LISP Control Functions

The functions documented in this chapter are the basic control functions of Maryland LISP. They are: QUOTE, EVAL, DO, LISP, OBLIST, TYPE, STACK, ALIST, and RAND.

2.1. quote

(quote <sexp>) or '<sexp>

QUOTE is a special form which returns its one argument unevaluated.

2.2. eval

(eval <sexp>)

Performs a LISP evaluation on its argument and returns the result of the evaluation. Note that EVAL is not a special form, so that <sexp> is evaluated once by the automatic argument handling mechanism before being handed to EVAL.

Examples.

```
(eval (list 'times 4 5 (add1 3))) = 80
((eval (cons 'lambda '((X Y) Y))) 'Y1 'Y2) = Y2
(eval (quote x)) = the value of x, for all x
```

2.3. do

(do <sexp1> . . . <sexpn>)

DO is a special form which applies EVAL to each of its arguments and returns the result of evaluating the last expression, <sexpn>.

2.4. lisp

(lisp <fn>)
(lisp)

The LISP function establishes a new level of LISP supervision. That is, a read-eval-print loop is entered with <fn>, which should be a function of no arguments, doing the prompting (if any) and reading. Also, any errors caused by evaluations at the new level will be routed to the new "latest level of supervision", so that the only way to escape from the new level is to do a RETURN outside of a PROG. The default value for <fn> could be written as (lambda nil (*read "EVAL: ")).

2.5. oblist

```
(oblist <fn>)
(oblist)
```

The OBLIST function takes one argument, which itself should be a function of one argument, <fn>, which it applies to every atomic symbol currently on the oblist. All atomic symbols except those created by GENSYM are always on the oblist. The default value for <fn> is a function which will perform a PRIN1 on its argument and do a (TERPRI) at the end of each linked hash bucket.

Examples.

```
(csetq allatoms (lambda nil
     (prog (val)
           (oblist (lambda (x)
              (setq val (cons x val))))
           (return val))))
```

This defines a function which constructs a list of all atomic symbols in the system.

2.6. type

```
(type <sexp>)
```

This function returns the type of its argument as an integer as listed in the table of type numbers given in the introduction to this manual. It should be noted that the STRINGP, NUMBERP, FIXP, FLOATP, and ATOM functions could be written using TYPE and EQUAL.

Examples.

```
(type (cons 'A 'B)) = 0
(type 47) = 1
(type 177Q) = 2
(type -3.4091E7) = 3
(type car) = 4
(type fn) = 5 if fn is a compiled function
(type (lambda nil nil)) = 6
(type 'XYZ123) = 7
(type "ABCDEFGHIJKLM") = 8
(type (buffer)) = 9
```

2.7. stack

```
(stack <l>)
```

STACK is a pseudo-function which takes as its argument a list <l> of s-expressions and expands it so as to have the members of <l> sitting on the stack just as though they had been

arguments of the function calling STACK. In some sense, STACK is the inverse of LIST. Note that giving a call on STACK as an argument to a special form does not usually result in the desired substitution of arguments.

Examples.

```
(list 'A 'B (stack '(C D E)) 'F) = (A B C D E F)
(or NIL (stack (list NIL 'TRUBLU NIL)) NIL) = NIL
```

If a call on STACK is EVAL'ed, the result is NIL. STACK and special forms don't mix.

2.8. alist

(alist)

This function returns the current association list, which is a list of dotted pairs pairing fluidly bound variables with their bindings at all levels on the stack.

(The user doing sophisticated alist manipulations will note that the alist also contains one or more atoms which print as "[]". This is a marker which is placed on the alist whenever a new level of LISP supervision is entered (ie., at the beginning of the run and whenever the LISP function is used) to mark where the levels of supervision are on the stack and to assure that the alist is never empty. Since ASSOC skips over non-cons nodes when looking through an association list, these markers pose no problem for ASSOC, though user-defined functions should watch out too.)

Examples.

```
(csetq lookup (lambda (atm)
     (assoc atm (alist))))
```

This function looks its argument up on the alist, and returns NIL if it is not there, and its dotted pair if it is.

2.9. rand

(rand <a1> <a2> . . . <an>)

This function returns one of its arguments at random.

Examples.

```
(csetq die (lambda nil
     (rand 1 2 3 4 5 6)))
```

This defines a function which will roll an imaginary die and return the result.

3. S-Expression Manipulation Functions

The functions described in this chapter are used for manipulating LISP s-expressions. They are: CAR, CDR, CONS, *CAR, *CDR, EQ, ATOM, NULL, MEMBER, EQUAL, MAPC, MAP, INTO, ONTO, INDEX, ONDEX, RPLACA, RPLACD, NCONC, LIST, APPEND, LENGTH, SUBST, REVERSE, NTH, and ASSOC.

The user should note that in addition to the CAR and CDR functions, it is possible to use C<ad>R as a function name, where <ad> stands for any string of 0 to 35 A's and D's, in any combination.

Examples.

```
(car '(A . B)) = A
(cr '(A . B)) = (A . B)
(cadddr '(A B C D E)) = D
(cadadadr '(A (B (C D)))) = D
(cadr x) = (car (cdr x))
(caddaar x) = (car (cdr (car (car (cdr x)))))
```

Of the functions in this chapter which may take lists as arguments, the following functions also allow extended lists (e.g., (A B C . D)), and handle the non-NIL "tail" as though it is NIL:

MEMBER	MAPC	INTO
ONTO	NCONC	APPEND
LENGTH	REVERSE	NTH
ASSOC		

3.1. Basic Functions

3.1.1. car

(car <obj>)

Returns the left pointer of the cons node <obj>. An error results if <obj> is not a cons node.

3.1.2. cdr

(cdr <obj>)

Returns the right pointer of the cons node <obj>. An error results if <obj> is not a cons node.

3.1.3. cons

(cons <obj1> <obj2>)

Creates a new cons node whose left pointer is <obj1> and whose right pointer is <obj2>, where both arguments are arbitrary LISP objects.

3.1.4. *car

(*car <obj>)

Same as CAR except no type-checking is performed on the argument <obj>. The *CAR of an atomic symbol, for example, is its value cell.

3.1.5. *cdr

(*cdr <obj>)

Same as CDR except no type-checking is performed on the argument <obj>. The *CDR of an atomic symbol, for example, is its property list.

3.2. S-Expression Predicates

3.2.1. eq

(eq <s1> <s2>)

Returns T if <s1> and <s2> are pointers to the same memory location, and NIL otherwise.

Examples.

```
(eq x x) = T if x is an atomic symbol
(eq (cons 'A 'B) (cons 'A 'B)) = NIL
((lambda (a) (eq a a)) x) = T for any x
```

3.2.2. equal

(equal <s1> <s2>)

See also the definition on page 82.

Returns T if <s1> and <s2> have the same structure and content, and NIL otherwise.

Examples.

```
(equal '(A . B) (cons 'A 'B)) = T
(equal (add1 47) (sub1 49)) = T
(equal "S" 'S) = NIL
```

```
(equal (lambda nil nil) (lambda nil nil)) = NIL
If (eq x y) = T then (equal x y) = T.
```

3.2.3. atom

```
(atom <a>)
```

Returns NIL if <a> is a cons node and T otherwise.

Examples.

```
(atom '(A . B)) = NIL
(atom 'XYZ) = T
(atom 452) = T
(atom "THIS IS A STRING") = T
(atom (lambda (x) x)) = T
```

3.2.4. null

```
(null <a>)
```

Returns T if <a> is the atomic symbol NIL, and NIL otherwise.

3.2.5. member

```
(member <a> <l>)
```

If <a> is EQUAL to one of the top-level members of the list <l>, this function returns that sublist of <l> which begins with the first occurence of <a>. Otherwise, NIL is returned. If <a> is an atomic symbol, then a "fast" EQ is used to make the equality test. Otherwise, an explicit call on EQUAL is made. The MEMBER function's implementation could be coded as:

```
(csetq member (lambda (a l)
     (member1 (cond ((equal (type a) 7) eq)
                    (t equal))
              a l)))

(csetq member1 (lambda (fn a l)
     (cond ((atom l) nil)
           ((fn a (car l)) l)
           (t (member1 fn a (cdr l))) )))
```

Examples.

```
(member 'A '(B D C E X)) = NIL
(member '(A B C) '(D E A B C F)) = NIL
(member '(X 3) '((A 3) (B 9) (X 3) (B 1))
      = ((X 3) (B 1))
(member 'W '(A B W X Z T)) = '(W X Z T)
(member x NIL) = NIL for any x.
```

```
(member 'A '(B (A C))) = NIL
```

3.3. Function Application Functions

3.3.1. into

```
(into <l> <fn>)
```

Applies the function of one argument <fn> to each member of the list <l>, returning a list of the results. The function could be written this way:

```
(csetq into (lambda (l fn)
     (cond ((atom l) nil)
           (t (cons (fn (car l))
                    (into (cdr l) fn))) )))
```

Examples.

```
(into '(A B C D) list) = ((A) (B) (C) (D))
(into '((A . B) (C . 4) (WYS$ . XX))
      car) = '(A C WYS$)
(into NIL fn) = NIL for any function fn.
```

3.3.2. mapc

```
(mapc <l> <fn>)
```

Applies the function of one argument <fn> to each member of the list <l>, and returns NIL. MAPC could be written as:

```
(csetq mapc (lambda (l fn)
     (cond ((atom l) nil)
           (t (fn (car l))
              (mapc (cdr l) fn)) )))
```

Examples.

```
(mapc '(F1 F2 F3) (lambda (fn) (cset fn nil)))
```

This sets each of the atomic symbols F1, F2, and F3 to NIL and returns NIL.

```
(csetq message (lambda msgs
     (mapc msgs prin1) (terpri)))
```

This function takes any number of arguments, and prints them all out on the same line.

MAPC is often an efficient way to replace loop

constructs in PROGs by making the LISP system do the looping. For example, the REV function given as an example under PROG could be rewritten as:

```
(csetq rev (lambda (l)
     (prog (r)
           (mapc l (lambda (x)
                (setq r (cons x r))))
           (return r))))
```

Similar constructions can be done with ONTO. In any event, because of the overhead involved with PROGs and SETQs, a recursive algorithm is usually competitive (provided there is no danger of running out of stack space).

3.3.3. onto

(onto <l> <fn>)

This function applies the function <fn> to <l>, and to each successive CDR of <l>, and returns a list of the results. The function could be written as:

```
(csetq onto (lambda (l fn)
     (cond ((atom l) nil)
           (t (cons (fn l) (onto (cdr l) fn))) )))
```

Examples.

```
(onto l car) = l for any list l.
(onto NIL fn) = NIL for any function fn.
(onto '(1 2 3 4) (lambda (l) (times (stack l))))
     = (24 24 12 4)
(onto '(X Y W U) reverse)
     = ((U W Y X) (U W Y) (U W) (U))
```

3.3.4. map

(map <l> <fn>)

This function applies the function <fn> to <l> and to each successive CDR of <l>, and returns NIL. The function could be written as:

```
(csetq map (lambda (l fn)
     (cond ((atom l) nil)
           (t (fn l) (map (cdr l) fn)) )))
```

Examples.

```
(map '(1 2 3 4) print) would print:
(1 2 3 4)
(2 3 4)
(3 4)
(4)
```

3.3.5. index

(index <l> <e> <fn>)

This function defies verbal explanation. Basically, given a non-NIL list <l>, any LISP object <e>, and a two-argument function <fn>, it computes:

(fn (car l) (fn (cadr l) (... (fn (cad...dr l) e) ...))).

The function can be written as:

```
(csetq index (lambda (l e fn)
     (cond ((atom l) e)
           (t (fn (car l) (index (cdr l) e fn))) )))
```

Examples.

If L = (X1 X2 X3 X4), then (INDEX L E FN) =
(FN X1 (FN X2 (FN X3 (FN X4 E))))

```
(index '((A B C)(D E F)(G H I)) NIL append)
     = (A B C D E F G H I)
(index '(A B C D E) 'F cons)
      = (A B C D E . F)
(index '(1 2 3 4 5) 0 plus) = 15

(csetq minval (lambda (nl)
     (index (cdr nl) (car nl)
        (lambda (x y)
           (cond ((lessp x y) x)
                 (t y))))))
```

This function returns the smallest of a list of numbers.

3.3.6. ondex

(ondex <l> <e> <fn>)

This function is similar to INDEX, and it also defies verbal explanation. Given arguments of the same description as those required for INDEX, it computes:

```
(fn l (fn (cdr l) ( ... (fn (cd...dr l) e) ... ))).
```

It can be written as:

```
(csetq ondex (lambda (l e fn)
     (cond ((atom l) e)
           (t (fn l (ondex (cdr l) e fn))) )))
```

Examples.

```
If L = (X1 X2 X3 X4) then (ONDEX L E FN) =
     (FN '(X1 X2 X3 X4) (FN '(X2 X3 X4)
                  (FN '(X3 X4) (FN '(X4) E))))
(ondex '(1 2 3 4) NIL list)
     = ((1 2 3 4)((2 3 4)((3 4)((4) NIL))))

(csetq artist (lambda (subs)
     (ondex (cdr subs) '(1) (lambda (lp lr)
            (cons (times (stack lp)) lr)))))
```

This function is used by the Maryland LISP Array package to compute the coeficients to be used when computing an address in a multi-dimensional array. See [Knu68], Section 2.2.6, Page 296 on "Sequential Allocation" of arrays. This function computes the a[r]'s.

3.4. S-Expression Pseudo-Functions

3.4.1. rplaca

```
(rplaca <cn> <obj>)
```

Replaces the left-side pointer of the LISP object <cn> by a pointer to the LISP object <obj>, returning the new value of <cn>.

Examples.

```
(csetq makeunbound (lambda (atm)
     (rplaca atm (*car 0q))))
```

This function removes the constant binding from an atomic symbol.

If the following s-expressions were EVALuated:

```
(csetq X1 '(A . B))
(csetq X2 X1)
(rplaca X1 'C)
```

then both X1 and X2 will have the value '(C . B).

3.4.2. rplacd

(rplacd <cn> <obj>)

Replaces the right-side pointer of the LISP object <cn> by a pointer to the LISP object <obj>, returning the new value of <cn>.

Examples.

```
(csetq ::T1 (cons 'A 'B))
(rplaca ::T1 ::T1)
(rplacd ::T1 ::T1)
```

This creates a cons node whose pointers both point at itself, making it unprintable and generally quite dangerous.

```
(csetq circularize (lambda (lst)
     (cond ((null lst) nil)
           (t (circularize1 lst lst)) )))
(csetq circularize1 (lambda (start lst)
     (cond ((null (cdr lst))
               (rplacd lst start)
               start)
           (t (circularize1 start (cdr lst))) )))
```

This defines a function which, given a list, makes a circular list out of it. Even though such objects are sometimes useful, they should be used with great care. For instance, the following would result in an infinite loop:

```
(length (circularize '(A B C D E)))
```

3.4.3. nconc

(nconc <l1> <l2>)

Changes the CDR of the last cons node in the list <l1> to point to the list <l2> and returns the new <l1>.

Examples.

If the following s-expressions are EVALuated:

```
(csetq X1 '(A B C))
(csetq X2 '(D E F))
(csetq X3 X1)
(csetq X1 (nconc X1 X2))
```

then X1 and X2 would both be bound to (A B C D E F).

```
(csetq circularize (lambda (lst)
     (nconc lst lst)))
```

This is a faster way to create a circular list.

3.5. Other S-Expression Functions

3.5.1. list

```
(list <e1> . . . <ek>)
```

Returns a list containing the elements <e1> through <ek>. Note that this function could be written as:

```
(csetq list (lambda elts elts))
```

Examples.

```
(list 'A 'B '(C D E) 34) = (A B (C D E) 34)
(list 'XYZATOM) = (XYZATOM)
(list) = NIL
```

3.5.2. append

```
(append <l1> <l2>)
```

Returns a list whose members are <l1>'s members followed by <l2>'s. In the result, the cons nodes representing the list <l1> have been copied while those forming the list <l2> are part of the result. This function could be written this way:

```
(csetq append (lambda (l1 l2)
     (cond ((atom l1) l2)
           (t (cons (car l1) (append (cdr l1) l2))) )))
```

Examples.

```
(append '(F 0 0) '(B A Z)) = (F 0 0 B A Z)
       (we give equal time to foobar and foobaz.)
(append NIL x) = x for any list x.
(append x NIL) = x for any list x.
(append NIL NIL) = NIL
(append 'A '(B C)) and (append '(A B) 'C) are illegal
```

3.5.3. length

```
(length <l>)
```

Returns the number of elements of the list <l>. The function could be written as:

```
(csetq length (lambda (l)
     (cond ((atom l) 0)
           (t (add1 (length (cdr l)))) )))
```

Examples.

```
(length '(A B C)) = 3
(length NIL) = 0
(length '(A B C D . E)) = 4
```

3.5.4. subst

(subst <new> <old> <sexp>)

Returns a copy of the s-expression <sexp> in which all occurances of <old> have been replaced by <new>. This function could be written as:

```
(csetq subst (lambda (new old sexp)
     (cond ((equal sexp old) new)
           ((atom sexp) sexp)
           (t (cons (subst new old (car sexp))
                    (subst new old (cdr sexp)))) )))
```

Examples.

```
(subst 'X 'Y '(((Y W) Y Y) (W X Y)))
       = (((X W) X X) (W X X))
(subst '(A . B) NIL '(A B C D)) = (A B C D A . B)
(subst '(L1 L3) '(4 R) '((4 T) (4 (4 R)) (4 R)))
       = ((4 T) (4 (L1 L3)) (L1 L3))
(eval (subst 4 'X '(plus 1 2 X 5))) = 12
```

3.5.5. reverse

(reverse <l>)

Returns a copy of the list <l> with its elements in reverse order. The REVERSE function could be implemented through these definitions:

```
(csetq reverse (lambda (l)
     (reverse1 l nil)))

(csetq reverse1 (lambda (l s)
     (cond ((atom l) s)
           (t (reverse1 (cdr l) (cons (car l) s))) )))
```

Examples.

```
(reverse '(A B C D)) = (D C B A)
(reverse (reverse l)) = l for any list l.
(reverse NIL) = NIL
(reverse '(X)) = (X)
(reverse '((A B) (C D))) = ((C D) (A B))
```

3.5.6. nth

(nth <l> <n>)

Returns the <n>th sublist of <l>. If <n>, which must be an integer, is positive, then the result is what would be computed by taking <n>-1 CDR's to <l>. If <n>-1 CDR's cannot be taken, then NIL is returned. If <n> is zero, the result is NIL. If <n> is negative, then <s>+<n> CDR's are taken, where <s> is the LENGTH of <l>, so that a list consisting of the last <n> elements of <l> is returned. The function could be written as:

```
(csetq nth (lambda (l n)
     (cond ((zerop n) nil)
           ((lessp n 0) (nth1 l (plus (length l) n)
                                (length l)))
           (t (nth1 l (sub1 n) (length l))) )))

(csetq nth1 (lambda (l n s)
     (cond ((greaterp n s) nil)
           (t (nth2 l n)))))

(csetq nth2 (lambda (l n)
     (cond ((zerop n) l)
           (t (nth2 (cdr l) (sub1 n))) )))
```

Examples.

```
(nth l 0) = NIL for any list l.
(nth l 1) = l for any list l.
(nth '(W3 P02 G67 6 A) 2) = (P02 G67 6 A)
(nth '(F 0 0 B A R) -1) = (R)
(nth '(E1 E2 E3 E4) 7) = NIL
(nth '(A B . C) 2) = (B . C)
(nth '(A B . C) 3) = C
(nth '(A B . C) 4) = NIL
```

3.5.7. assoc

(assoc <e> <l> <n>)
(assoc <e> <l>)

This function is used to look up values on association lists. The second argument, <l>, should be a list of s-expressions. Any atoms in <l> are ignored. The result is the

<n>th element of <l> which has <e> as its CAR. The default value for <n> is 1. If there are not <n> such elements, NIL is returned. If <e> is an atomic symbol, ASSOC works in a "fast" mode in which single-instruction EQ tests are made for equality. Otherwise, an explicit call on EQUAL is made, so that the ASSOC function's implementation could be coded as:

```
(csetq assoc (lambda (e l . n)
     (assoc1 (cond ((equal (type e) 7) eq)
                   (t equal))
             l (cond (n (car n)) (t 1)))))

(csetq assoc1 (lambda (fn e l n)
     (cond ((atom l) nil)
           ((atom (car l)) (assoc1 fn e (cdr l) n))
           ((eq e (caar l))
              (cond ((equal n 1) (car l))
                    (t (assoc1 fn e (cdr l) (sub1 n)))))
           (t (assoc1 fn e (cdr l) n)) )))
```

Examples.

```
(assoc 'X '((T . 9)(LSSC . 140)(X . P)(7 7)))
       = (X . P)
(assoc 'W '(A B (W Y) (E . R) X)) = (W Y)
(assoc x NIL) = NIL for any x.
(assoc 94 '((45 . ADX$) (94) (RR . PL3))) = (94)
(assoc 'W '((W . 4) (X . 3) (W . 9)) 2) = (W . 9)
(assoc 'RR '((RR . RR) (RR . E) (4 . PLW))
       3) = NIL
```

4. Atom Manipulation Functions

The functions in this chapter are used to manipulate atomic symbols. They are: ATOM, GENSYM, STRING, ATSYMB, *CAR, *CDR, COMPRESS, EXPLODE, EXPLODE2, REMOB, and ERASE.

4.1. atom

(atom <sexp>)

See the definition on page 32.

4.2. gensym

(gensym <a>)
(gensym)

This function returns a unique atomic symbol. The symbol which is returned will have a print name which is an integer, but will be printed as the print name of the atomic symbol <a> followed by the integer. The default value for <a> is the atomic symbol G. Symbols produced by GENSYM do not exist in the oblist and will not be recognized by their printed names by the input routine. They are also subject to garbage collection when no longer in use.

Examples.

```
(gensym) = G1  (if this is the first call)
(gensym 'NEW-ATOM#) = NEW-ATOM#2   (next call)
(string (gensym)) = 3 (next call)
```

4.3. string

See the definition on page 92.

4.4. atsymb

See the definition on page 92.

4.5. *car

See also the definition on page 31.

When its argument is an atomic symbol, *CAR will return its value, if any, and garbage otherwise.

Examples.

```
(csetq boundp (lambda (atm)
     (not (and (null (assoc atm (cdr (alist))))
               (eq (*car atm) (*car 0q))))))
```

This defines a function which returns T if its argument is a fluidly or constantly bound atomic symbol, and NIL otherwise. The CDR of the alist is used so that ATM will not appear bound unless it was bound before the call on BOUNDP.

4.6. *car

See also the definition on page 31.

When its argument is an atomic symbol, *CDR returns its property list. When initially created, all atomic symbols have null property lists.

Examples.

Suppose A has a null property list. If we do:

```
(put 'A 'I 'V)
(csetq x (*cdr 'A))
(put 'A 'I 'W)
```
then X will be bound to ((I . W)).

4.7. compress

(compress <l>)

This function takes a list of single-character atomic symbols <l> and produces an atomic symbol whose print name is produced by making a string of the print names of the various atomic symbols in <l>. If all the atomic symbols on the list <l> have print names which are digits, then the result is an integer.

Examples.

```
(compress '(F 0 0 B A R)) = F00BAR
(compress '(!1 R !, T !9 U)) = 1R,T9U
(compress '(1 2 3 4)) is trash, but
(compress '(!1 !2 !3 !4)) = 1234, an integer.
(compress '(AB CD EF GH)) = ACEG  (but don't do this)
```

4.8. explode

(explode <a>)

This function produces a list of atomic symbols which can be

COMPRESSed to form the atomic symbol <a>. The print routines are used to do this.

Examples.

```
(explode 'ABCDEFG) = (A B C D E F G)
(compress (explode x)) = x
```

This works for any atomic symbol x not created by GENSYM. (For gensyms, see the INTERN example below.) It also works for a number of other things that it wasn't really intended for.

```
(compress (append '(A T O M) (explode 137))) = ATOM137
(explode 12345) = (1 2 3 4 5)
```

But be careful, because the result (1 2 3 4 5) is a list of atomic symbols, not a list of numbers.

```
(csetq intern (lambda (gen)
     (compress (explode gen))))
```

This function takes a GENSYM-created atomic symbol and returns an atomic symbol which has a string print-name field and which is linked into the oblist. Thus (intern (gensym)) will always return a non-garbage-collectable atomic symbol which will be recognized on input.

4.9. explode2

(explode2 <a>)

This function is similar to EXPLODE, except it uses the PRIN2 print routines to produce escape characters where needed in the exploded result.

Examples.

```
(explode2 'AB!(!!X) = (A B !! !( !! !! X)
```

where the escape characters ("!") are the ones which would be PRIN2'ed.

4.10. remob

(remob <at>)

This function, given an atomic symbol, removes it from the oblist. If the argument is either a gensym or not an atomic symbol, NIL is returned, otherwise a pointer to the atom itself is returned. Note that REMOB'ing an atom will not cause a

dangling pointer, since the symbol will not be garbage collected until no other LISP structures point at it. However, REMOB'ing an atom and then attempting to read it in before it is garbage-collected will cause two copies of the atom to be present in the system, so that this is not a good practice. Needless to say, one should not seriously attempt to apply REMOB to NIL or any other necessary atomic symbol.

4.11. erase

(erase <l>)

ERASE reverts all the atomic symbols on the list <l> to the default condition, that is, no value and null property list. ERASE always returns NIL.

The function could be written as:

```
(csetq erase (lambda (al)
     (mapc al (lambda (atm)
        (rplaca atm (*car 0q)) (rplaca atm nil)))))
```

Examples.

(erase '(V1 V2 XRTVAR))

This eliminates the values and property lists of the atomic symbols V1, V2, and XRTVAR.

(oblist (lambda (atm) (erase (list atm))))

This would destroy most of LISP before either ERASE or LIST becomes unusable.

5. Value and Function Definition Functions

The functions documented in this chapter are used to define functions and other objects as the values of atomic symbols. They are: CSETQ, CSET, SETQ, SET, DEFINE, DEFSPEC, DEFMAC, LAMBDA, LAMDA, and FUNCTION.

5.1. Value Assignment Functions

5.1.1. csetq

(csetq <a> <v>)

This is a special form which makes the atomic symbol <a> constantly bound to the value <v>. The first argument is not evaluated but the second one is.

5.1.2. cset

(cset <a> <v>)

This is the same as CSETQ, except both arguments are evaluated. Thus, whatever is given for <a> should evaluate to an atomic symbol.

5.1.3. setq

(setq <a> <v>)

This special form makes the atomic symbol <a> fluidly bound to the value <v>. Like CSETQ, the first argument is not evaluated, but the second is. If <a> already has a constant binding, SETQ acts like CSETQ.

5.1.4. set

(set <a> <v>)

This function is like SETQ, except that both arguments are evaluated.

5.1.5. define

(define <pl>)

The DEFINE function, given a list <pl> of the form ((<at1> <val1>) ... (<atn> <valn>)) assigns <ati> the value <vali> for each i, and returns a list of the <ati>'s. Each <vali> is EVAL'ed before its assignment is made (eg., first <val1> is EVAL'ed and <at1> is assigned that value, then <val2> is EVAL'ed and <at2> is assigned that value, etc.). The RPLACA routine is used to make the assignments, so that special forms and macros

may not be created this way. The DEFINE function may be written as:

```
(csetq define (lambda (pl)
     (into pl (lambda (avpr)
        (cset (car avpr) (eval (cadr avpr)))
        (car avpr)))))
```

5.2. Function Assignment Functions

5.2.1. defspec

(defspec <a> <v>)

This function defines <a> to be a special form of the function <v>. The first argument, <a>, is not evaluated, and it is given a global binding. This binding consists of a special form linker node which in turn points at the result of EVALuating <v>, which should be a regular function's linker node or a pointer at compiled or system code.

Examples.

```
(defspec sfcons cons)
(defspec foobar (lambda (x y z) (list z x y)))
```

Given these definitions, the following are true:

```
(sfcons A B) = (A . B)
(sfcons X (EVAL L)) = (X EVAL L)
(foobar A W K) = (K A W)
```

5.2.2. defmac

(defmac <a> <v>)

This is like DEFSPEC, except that it defines macros rather than special forms.

Examples.

```
(defmac defun (lambda (name typ args . stmts)
     (list (cond ((eq typ 'expr) 'csetq)
                 ((eq typ 'fexpr) 'defspec)
                 (t 'defmac))
           name (cons 'lambda (cons args stmts)) )))
```

This defines the DEFUN function, which is the standard way of defining functions in many other LISPs. As an example of its use, consider the following:

```
(defun push fexpr (atm lstname)
```

```
(cset lstname (cons atm (eval lstname))))
```

5.3. Function Definition Functions

5.3.1. lambda

(lambda <args> <body1> . . . <bodyn>)

This special form creates a function definition which accepts arguments as given by <args>, and evaluates <body1> through <bodyn>, returning <bodyn>'s result as its value. The first argument, <args>, should be either a list of atomic symbols or a single atomic symbol. In the former case, each named symbol is bound to an argument which is passed to the defined function. If there are not enough, an error message will result. If there are too many arguments, the extras will be thrown away, unless the list <args> is an extended list, in which case a list containing the extra arguments is fluidly bound to the "extended" entry of the argument list. If <args> is an atomic symbol rather than a list, the defined function will take any number of arguments, and all the arguments are put on a list which is fluidly bound to this lone atomic symbol.

Examples.

```
(csetq add3 (lambda (x y z)
     (plus x y z 3)))
```

This is the normal use of LAMBDA, wherein the argument list is just a list of atomic symbols.

Note that an atomic symbol which has a constant binding may not be specified in a LAMBDA or LAMDA argument list or in a PROG local variable list. Because of this, the following is in error:

```
(csetq foobar (lambda (f) . . .))
```

(Remember that F has a constant binding, namely NIL.)

```
(csetq asm (lambda (ccde . options) . . .))
```

This is how the Maryland LISP assembler handles optional arguments, in this case, assembler options. If it is called by (asm mycode 'X 'Y 'N), then OPTIONS gets the binding (X Y N).

```
(csetq tracefn (lambda (atm fn . args)
     (mapc (list "CALLING: " atm ", ARGUMENTS: " args)
           prin1) (terpri)
     (fn (stack args))))
```

This is a typical sort of function one would send to BREAK to trace the entry of a certain function.

```
(csetq nargs (lambda lst (length lst)))
```

This function returns the number of its arguments.

5.3.2. lamda

```
(lamda <args> <body1> . . . <bodyn>)
```

This function is just like LAMBDA, except that it ensures that whenever the function it defines is entered, the alist which will be used is the one which was present when LAMDA was entered to define the function. LAMDA should be thought of as follows:

```
(defmac lamda (lambda args
     (list 'function (cons 'lambda args))))
```

Examples.

```
(csetq compose (lambda (fn1 fn2)
     (lamda args (fn1 (fn2 (stack args))))))
```

This defines a function which, given two regular function definitions (no special forms or macros allowed), returns a third which is a composition of them. For example,

```
(csetq composelist (lambda (fnlist)
     (cond ((null (cdr fnlist)) (car fnlist))
           (t (compose (car fnlist)
                       (composelist (cdr fnlist)))) )))
```

would define a function which, given a list of regular functions, would return a composition of them all. Of course, all but the last function in the list is assumed to take just one argument, and the composed function takes the same number of arguments as the last function in the list.

As another example of COMPOSE, consider the following:

```
(csetq f1 (lambda (y) (list args y)))
(csetq f2 plus)
(csetq f3 (lambda (args)
      ((compose f1 f2) 1 2 3)))
```

The function F3 will not work as desired because of the free variable in the definition of F1. See the examples under FUNCTION for the solution to this.

5.3.3. function

(function <fn>)

This takes as its single argument a function <fn>, and returns a version of <fn> which will substitute the alist in effect when FUNCTION was entered for the current one while it is running. The concept of FUNCTION is one of the most important in current control structure research. A good reference on the intention and possibilities of FUNCTION is [Mos70].

Examples.

The problem referred to in the example under LAMDA occurs because free variables in functions being composed might accidentally refer to variables bound in COMPOSE itself. To fix F3, one might do:

```
(csetq f3 (lambda (args)
    ((compose (function f1) (function f2)) 1 2 3)))
```

Given the following definitions:

```
(csetq bump (lambda nil
     (setq x (add1 x)) x))
(csetq skip (lambda (x y)
     (caller (function bump) y)))
(csetq caller (lambda (fn x)
     (fn)))
```

then (skip 3 7) = 4, since BUMP changes the X in SKIP's frame rather than the one in CALLER's.

An alternative to the FUNCTION function is what is known in the LISP Machine implementation of LISP [Gre77] as the CLOSURE function. This function, given a list of variables and a function definition, returns a function which, when executed, will attach the CLOSURE-time values of the variables given to CLOSURE to the front of the current alist, thus guaranteeing that the given variables will have the proper values in the function. In Maryland LISP, the CLOSURE function could be written as:

```
(defmac closure (lambda (vars fn)
     (list 'lambda (args fn)
        (list 'eval (list 'quote (cons fn (args fn)))
           (list 'append (list 'quote (into vars
                    (lambda (var) (cons var (eval var)))))
              '(alist)))) ))
```

(ARGS is a function which, given a linker node, returns the argument list for the node.) As an illustration of the CLOSURE function, consider the following example, made comprehensible in [Ste76] and [Gre77]:

```
(csetq generate-sqrt-of-given-extra-tolerance
     (lambda (factor)
             (closure '(factor)
                 (lambda (x)
                         ((lambda (epsilon) (sqrt x))
                          (times epsilon factor))))))
```

In this example, SQRT is a function which computes the square root of its argument to within a tolerance given as the free variable EPSILON. This function returns a version of SQRT whose error tolerance is greater than EPSILON by a factor of FACTOR, where FACTOR is evaluated once when CLOSURE is called and EPSILON is evaluated each time the new square-root function is called. As an example,

```
(setq epsilon 0.01)
(setq sqrt4 (generate-...-tolerance 4))
(sqrt 9.0) = 2.99  where |9.0^0.5 - 2.99| ≤ 0.01
(sqrt4 16.0) = 4.04  where |16.0^0.5 - 4.04| ≤ 0.04
(setq epsilon 0.02)
(sqrt 2.0) = 1.395  where |2.0^0.5 - 1.395| ≤ 0.02
(sqrt4 0.25) = 0.42  where |0.25^0.5 - 0.42| ≤ 0.08
```

6. Input-Output Functions

The functions defined here are READ, READREC, READCH, TOKEN, CLEARBUFF, CURCHAR, SETCOL, BACKSP, READMAC, DELIM, ENDCHAR, PRINT, PRIN1, PRIN2, CURRCOL, TERPRI, SPACE, DIGITS, PLIMIT, PLENGTH, PLENGTH2, AXMIT, AXMIT1, AREAD, FIOPEN, FICLOSE, FIPRINT, FIREAD, FIERASE, FIDROP, FIPACK, FITOC, FIPRINTREC, FIREADREC, COPYOUT, and COPYIN.

6.1. Input Functions

6.1.1. Input Parsing Functions

6.1.1.1. read

(read)
(read <prompt>)

Reads the next s-expression from the input stream, or if CLEARBUFF has been used to read from a sequential file, from the file. The result is the object read. If READ encounters an end-of-file while reading from the normal input stream, LISP will terminate, and if an end-of-file is found during file input, an (ERROR -11) condition will be generated.

If an argument is given to READ, it should be a string which will be printed as a prompt. The prompt will not be printed if file I/O is taking place.

6.1.1.2. token

(token)
(token <prompt>)

Returns the next token in the input stream. A token is either a delimiter, a string, or a sequence of digits, letters, and non-delimiter symbols surrounded by delimiters. The result is an atomic symbol, a number, or a string.

If an argument is given, it should be a string which will be printed as a prompt. The prompt will not be given if file I/O is occurring.

Examples.

Suppose the following function were defined:

```
(csetq tokengrabber (lambda nil
     (prog ((lst nil))
      loop (setq lst (cons (token) lst))
           (cond ((null (car lst))
                  (return (cdr lst))))
           (go loop))))
```

Then the following exchange might take place:

```
EVAL:  (DO (PRIN2 (TOKENGRABBER)) NIL)
(A "B *" C (23 'E . F)) NIL                    <- user
(!) !) F !. E !' 23 !( C "B *" A !()           <- LISP
VALUE: NIL
```

6.1.1.3. readmac

```
(readmac <c> <fn>)
(readmac <c>)
```

This function establishes a status for the character (ie., one-character string) <c> in the readmacro table used by the LISP scanner. The second argument, <fn>, should be a function of no arguments which should be applied whenever the character <c> is parsed in the input stream. The result of the function is the old readmacro table entry for <c>. If <fn> is omitted, the current entry will be returned and there will be no effect. The character <c> can be cancelled as a readmacro by submitting NIL as the value of <fn>. Initially, the readmacro characters are "'", " ", ",", and "?", where the first is the "quote" character, the next two are treated as all-purpose delimiters, and the last is used for comments.

Examples.

The readmacros for the four characters just mentioned could be defined by:

```
(readmac "'" (lambda nil (list (quote quote) (read))))
(readmac " " read)
(readmac "," (readmac " "))
(readmac "?" (lambda nil (clearbuff) (read)))
```

6.1.1.4. delim

```
(delim <c> <v>)
(delim <c>)
```

This function sets or retrieves the delimiter table entry for the character (ie., one-character string) <c>. The second argument <v>, if given, should be either T or NIL, meaning that <c> should or should not (respectively) be a delimiter. A delimiter is a character which is always parsed as a separate token, provided it does not occur in a string and it is not preceded by the escape character, "!". The value is the old

delimiter table entry for <c>. If <v> is omitted, the current entry is returned and no change is made.

Examples.

The DELIM function helps make the Maryland LISP scanner very versatile. For example, the Stanford MLISP program uses the normal scanning functions (especially TOKEN) and a straightforward recursive descent parser to handle an ALGOL-type syntax. In order to scan expressions properly, the scanner must cause TOKEN to scan out arithmetic, logical, and list-manipulation operators as separate tokens. To do this for the arithmetic operators, MLISP just executes the following calls before starting to scan a program:

```
(delim "+" T)
(delim "-" T)
(delim "/" T)
(delim "*" T)
```

6.1.2. Input Buffer Manipulation Functions

6.1.2.1. clearbuff

(clearbuff)

Clears the input buffer so no more input tokens can be parsed from the current line. This function has an alternate use for file reading which is explained below.

6.1.2.2. curchar

(curchar)

This function returns as its value the number of the next position in the input buffer to be processed as input. Buffer positions are numbered starting at 1.

Examples.

If this is EVALuated:

```
(DO (READ) (CSETQ X (CURCHAR)) (READ))
```

and this is entered to be read:

```
(A . B) ATOMICSYMBOL
```

then X will get the value 5.

6.1.2.3. setcol

(setcol <n>)

Sets the input pointer to position <n> in the current input line.

6.1.2.4. backsp

(backsp)

Moves the input pointer back one position in the current input line. There is no effect if the pointer is already at the start of the line. The value is T, unless the input pointer was already at the start of the line, if which case NIL is returned.

Examples.

The input pointer can be reset to the beginning of the current line by doing:

```
(prog nil
 loop (cond ((backsp) (go loop))))
```

Of course, (setcol 1) would have been a little faster.

The following example is based on the algorithm used by the bignum package to read in long strings of digits and process them into bignum representation. When the readmacro for "^" is activated, it calls READCH until it finds a non-digit, whereupon it backspaces and sends the list of digits to a function for creating bignums.

```
(readmac "^" (lambda nil
     (prog ((lst nil) (ctr 1) (sgn (readch)))
           (cond ((eq sgn '-) (setq sgn -1))
                 ((eq sgn '+) (setq sgn 1))
                 (t (backsp) (setq sgn 1)))
      loop (setq lst (cons (readch) lst))
           (cond ((digitp (car lst)) (go loop)))
           (backsp) (setq lst (cdr lst))
           (return (makebignum lst sgn)) )))
```

Note the handling of the optional sign. DIGITP, of course, is just another bignum package function for checking whether its argument is the atomic symbol representation of a decimal digit.

6.1.3. Record and Character Reading Functions

6.1.3.1. readch

(readch)
(readch <prompt>)

Scans the next character in the input stream and returns it as an atomic symbol. If an argument is given to READCH, it should be a string which will be printed as a prompt. The prompt will not be printed if the function is reading from a file.

6.1.3.2. readrec

(readrec)
(readrec <prompt>)

Reads the entire next line and returns it unparsed as a string. If an argument is given, it should be a string which will be printed as a prompt. If the function is reading from a file, the prompt will not be printed.

6.1.4. Miscellaneous Input Functions

6.1.4.1. endchar

(endchar <c>)
(endchar)

Declares or retrieves a character which, whenever scanned as input, should be interpreted as an end-of-file. The result is the old endchar, the default being the "_" character. If the argument <c>, which should be a single-character string, is omitted, then the current endchar is returned and no change is made.

6.2. Output Functions

6.2.1. S-Expression Printing Functions

6.2.1.1. print

(print <obj>)

This function edits the LISP object <obj> into the current output buffer and prints out the whole buffer. Its value is <obj>.

6.2.1.2. prin1

(prin1 <obj>)
(prin1 <obj> <col>)

This function edits the LISP object <obj> into the current output buffer starting at column <col>. The default for <col> is the first column in the output buffer into which no output character has been edited. If necessary, PRIN1 will print the current output buffer to make room for extra output. If <col> is not a positive integer, output editing will start in column 1.

6.2.1.3. prin2

(prin2 <obj>)
(prin2 <obj> <col>)

This function is like PRIN1 except it supplies escape characters and string delimiters to make its output READable by the LISP input routines.

6.2.2. Output Buffer Manipulation Functions

6.2.2.1. currcol

(currcol)

This function returns the number of the next position in the output buffer to receive an output character.

6.2.2.2. terpri

(terpri)

This function causes the current output buffer to be printed and a new one set up. It has an alternate use for file output which is described below.

6.2.3. Output Formatting Functions

6.2.3.1. space

(space <n>)

This function causes <n> blank lines to be printed. If <n>=-0, a page-eject will be inserted if output is being breakpointed into a print file and 1 blank line will be printed otherwise.

6.2.3.2. digits

(digits <n>)
(digits)

This function specifies the number of digits to be printed on output for a floating-point number. The result is the old number of digits, and if <n> is omitted, this number is returned

without being changed. The initial number of output digits is 6.

6.2.3.5. plimit

(plimit <p>)
(plimit)

This function sets depth and length limits for the outputting of LISP lists. The argument <p> should be a dotted pair of integers (<d> . <w>), where <d> is the number of levels of nesting the output routines will tolerate before printing "&&" and refusing to go deeper, and <w> is the number of elements a list being printed may have before the output routines print "--" and refuse to go further. The value of the function is the old pair of plimits. If <p> is omitted, the current pair will be returned unchanged. The initial PLIMIT pair is (10 . 50).

Examples.

If the print limits were set to 3 and 4 by:

(plimit '(3 . 4))

then

(print '((((A B) C) D E) F G H I))

would print:

((&& D E) F G H --))

6.2.4. Print Length Functions

6.2.4.1. plength

(plength <obj>)

This function returns the number of output positions the LISP object <obj> would take up if printed by PRIN1 or PRINT. Note that for long (or deep) s-expressions, the print length is limited by the current print limits, as set through PLIMIT.

Examples.

(plength s) = (size s) for any non-NIL string s.
(plength a) = (size (string a))
for any atomic symbol not created by GENSYM.
(plength 1726351Q) = 8
(plength '(A . B)) = 7

If n is an integer, (plength n) can be used to find out how many digits it has.

6.2.4.2. plength2

(plength2 <obj>)

This function returns the number of output positions the LISP object <obj> would take up if printed by PRIN2. Again, the print length of a composite s-expression is limited by the print limits as set by PLIMIT.

Examples.

(plength2 "ABCD") = 6
(plength2 'ABC!)D) = 6

6.3. ASCII I/O Functions

Since Maryland LISP does all its normal I/O in UNIVAC's infamous FIELDATA character set, it is not possible via the standard I/O functions to use ASCII control codes to control special devices. Because of this, Maryland LISP has three functions to perform ASCII character I/O.

6.3.1. axmit

(axmit <oct1> <oct2> ... <octn>)

The AXMIT function takes as its arguments any number of octal words, each of which represents four ASCII characters. It puts these characters into the ASCII output buffer and then dumps the buffer.

There are several things which are important to note. First the buffer used by AXMIT (and AXMIT1) is different from the standard LISP output buffer. This allows the user to have some measure of asynchrony (subh as in device control) when outputting from LISP. The second and most important item is a notion peculiar to Maryland's computers called FUNNY MODE.

When one requests that an internal buffer be printed on a terminal, EXEC 8 does not allow the programmer to add a CARRIAGE RETURN (CR) LINEFEED (LF) called CRLF at the end of his image; it is done automatically. In the FIELDATA character set, no provisions have been made to represent these characters. For purposes of sending commands to a graphics device it might be nice not to have spurious CRLF's floating around to screw up the picture hence the inception of FUNNY MODE. In order to prevent having a CRLF sequence tacked onto the end of an ASCII line image, the user should place a flag (octal 777) in the first quarter word of the buffer to be printed.

EXAMPLE: Send seven bells to the terminal with no trailing CRLF.

EVAL: (AXMIT 7770070070070Q 0070070070070Q)

Note that 777 is only in the first quarter word of the first word of the buffer to be sent.

VALUE: NIL

6.3.2. axmit1

(axmit1 <oct1> <oct2> ... <octn>)

Sometimes the user might wish just to keep sending a large number of characters (graphics applications) and let the system figure out when to dump the buffer. This is a perfectly reasonable thing to do and if the AXMIT1 function is used instead of AXMIT, LISP will dump the buffer if it detects imminent overflow. The new buffer is automatically prefaced with 777177177177Q, the command to enter FUNNY MODE and send three RUB OUTs (nops). (It is assumed that the CRLF sequence is generally not desired if the user is sending a stream of characters to a device.)

6.3.3. aread

(aread)

Along with AXMIT and AXMIT1 there is an ASCII read facility to accept characters from a specific device. AREAD is a function of no arguments which returns a list of octal numbers (four characters per word, 9 bits per character), representing the characters sent. The utilization of AREAD, AXMIT and AXMIT1 allows for two way communication between an arbitrary device and LISP.

6.4. Sequential File I/O Functions

This section describes the Maryland LISP functions for sequential file I/O. (The random access file I/O functions are described in the next section.) In sequential file I/O, the user defines file descriptors through FIOPEN, and when it is desired to use the file, CLEARBUFF (input files) and TERPRI (output files) are used to redirect the normal flow of LISP I/O to or from the file. If a file is read from or written to on several occasions using the same file descriptor, at each point the reading or writing continues where it left off. This process may be ended (and an end-of-file inserted for output files) by sending the file descriptor to FICLOSE. There may be several input file descriptors for a file, each reading independently, but there may be only one output file descriptor at a time for a file. It is not presently possible to perform sequential file I/O on program files.

As an example of the use of the functions to be described, consider the following function which, given the internal names of two data files (see page 8 for the definition of "internal

name"), copies the contents of the first to the second.

```
(csetq fcopy (lambda (infn outfn)
     (prog ((infd (fiopen infn t))
            (outfd (fiopen outfn nil)))
           (clearbuff infn) (terpri outfn)
      loop (attempt (prin2 (read))
                    (-11 (go cont)))
           (terpri)
           (go loop)
      cont (clearbuff nil) (terpri nil)
           (ficlose infd) (ficlose outfd)
           (return))))
```

6.4.1. fiopen

(ficpen <n> <m>)

The FIOPEN function for sequential files takes two arguments. The first, <n>, should be a string which is the internal name of an existing sequential data file, and the second, <m>, should be either T or NIL, indicating that the file is to be read from or printed to, respectively. Its result is a list of three elements, the file's internal name <n> and two buffers which are used during the I/O process. This is a file descriptor, and it should be passed to CLEARBUFF or TERPRI when the file is to be used.

6.4.2. ficlose

(ficlose <fd>)

This function should be used to close off a sequential file before the file is used for any other purposes. The argument, <fd>, should be the file's file descriptor which was initially created by FIOPEN.

6.4.3. clearbuff

(clearbuff <fd>)

For purposes of sequential file I/O, the CLEARBUFF function is used to specify the sequential file from which input is to be read until another is given or LISP returns to the top-level READ-EVAL-PRINT loop. The argument <fd> should be either NIL, specifying that input should be taken from the user's terminal (or from the main input stream in batch mode), or a file descriptor as described under FIOPEN. When used in this manner, CLEARBUFF returns NIL, indicating that input had previously been read from the user's terminal, or the file descriptor which was specified on the last such call to CLEARBUFF.

6.4.4. terpri

(terpri <fd>)

In the context of sequential file I/O, TERPRI is much like CLEARBUFF, specifying a file to which output is to be directed until another is given or control returns to the top-level loop. Again, <fd> should be either NIL, specifying that output should be done to the user's terminal (or to the main output stream in batch mode), or a file descriptor as described under FIOPEN. TERPRI returns either NIL, indicating that output had previously been sent to the user's terminal, or the file descriptor which was specified on the last such call to TERPRI.

6.5. Random Access File I/O Functions

Maryland LISP has a general-purpose random access file system. The design concept of the system is based around the UNIVAC concept of a "program file". (Indeed, the program file would have been used directly if not for the large amount of core required to implement it.)

LISP random access files are divided into records (analogous to program elements). A record contains the symbolic representation of a single LISP object, and is identified by a positive integer. It is important to note that the record number has no correlation with the physical location of that record within the file. Internal to records are the physical images which represent the information stored. These images are stored sequentially in Standard Data Format (see [Uni78]).

Once a "file descriptor" has been established for a random access file, each operation on the file is self-contained, so that the "open", I/O, and "close" operations of sequential file I/O are each done in every random access file I/O function.

A system convention has been established to use record 1 as a table of contents, represented as an association list. It is therefore advisable if one wishes to interface with other LISP support routines to maintain an association list with the appropriate titles and record numbers in record 1. For example, if the file contains function definitions, the table of contents in record 1 might be:

((FACT . 2) (FOO . 9) (BAR . 4))

When one of these functions encounters an error condition, an error message is given, and an error number is provided to explain the nature of the problem. A list of current codes is given below:

Error Number	Error Type
-1	BAD FILE TABLE INDEX (File not LISP random access.)
-2	RECORD NUMBER OUT OF RANGE
-3	RECORD UNWRITTEN
-4	RECORD ALREADY DELETED
-5	NO RECORDS AVAILABLE (File may need to be packed.)

6.5.1. fiopen

(fiopen <n>)

In the context of random access files, FIOPEN takes only one argument, a string which is the internal name of an existing data file. It returns a four-member list which serves as a random access "file descriptor" and which is to be used to refer to the file when using the random access file I/O functions described below.

Examples.

To open a file called LISP*FILE.,

First create an @USE relation via:

```
:EXEC @USE F.,LISP*FILE.
```

and then assign the file:

```
:EXEC @ASG,A F.
```

(Note that the EXEC function could also be used.) Now use FIOPEN to open it:

```
EVAL:  (CSETQ FILE (FIOPEN "F"))
VALUE: (F [9:61000Q] [9:61200Q] [9:61400Q])
```

where the [9:...]'s are buffers to be used by the I/O routines as a workspace for operations involving this new file descriptor.

6.5.2. fiprint

(fiprint <obj> <fd> <n>)

Prints out the LISP object <obj> as the <n>th record of the random access file for which <fd> is the descriptor. The value is <obj>. The PRIN2 routine is used to generate the text to be written to the file, so that string quotes and escape characters will be generated where necessary for subsequent FIREADing. Note

that the printed forms of objects such as linker nodes and buffers are still unreadable, whether one has printed them to a file or a terminal.

Examples.

Given the example under FIOPEN...

To store the list

("TOM" "DICK" "HARRY")

just EVALuate:

(FIPRINT '("TOM" "DICK" "HARRY") FILE 2 T)

which will return the list of strings. Note that since FIPRINT does a PRIN2 out to the file, the double-quotes will remain around the strings.

6.5.3. firead

(firead <fd> <n>)

Reads and returns as its value the <n>th record of the random access file for which <fd> is the descriptor. If no <n>th record exists, an error message with error number -3 will be given and control will return to the latest level of supervision. (See list of random access file I/O error codes above.)

Examples.

Referring to the above example...

EVAL: (FIREAD FILE 2)
VALUE: (TOM DICK HARRY)

This is a list of three strings, as PRIN2 would show.

6.5.4. fierase

(fierase <fd>)

Erases all records in the random access file for which <fd> is the descriptor. The value is T.

6.5.5. fidrop

(fidrop <fd> <n>)

Deletes the <n>th record in the random access file for which <fd> is the descriptor. The value is T. Note that after a

FIDROP is done, the locations assigned to the dropped record have not been deallocated but are merely flagged as being deleted. After a large number of FIDROPs, it is advisable to use FIPACK to clean up the extra garbage.

6.5.6. fipack

(fipack <fd>)

Reorders the contents of the file <fd> so as to eliminate dropped records and minimize the size of the file. FIPACK does not renumber the records, however. The value is T.

It is important to note that if the operating system crashes during a FIPACK there is the possibility of losing at most one record. Although this problem cannct be avoided, the damage is usually not irreparable, and a FILEDIT expert might be able to fix it.

6.5.7. fitoc

(fitoc <fd>)

Returns a list of integers which represent those records in the random access file <fd> which are currently being used. If the file is empty, NIL is returned.

6.5.8. fiprintrec

(fiprintrec <obj> <fd> <n>)

This function, together with FIREADREC, allows the user to do random access file I/O with records of arbitrary format, treated as strings. The first argument, <obj>, should be a list of strings to be written to record number <n> in the file represented by file descriptor <fd>. Note that objects written by FIPRINTREC may not be recognizable by FIREAD. The result of this function is always <obj>.

6.5.9. fireadrec

(fireadrec <fd> <n>)

This function reads the <n>th record in the file represented by file descriptor <fd> and returns it as a list of strings without attempting to parse it.

6.5.10. copyout

(copyout <fd> <n> <fn>)

This function, given a random access file descriptor <fd>, a record number <n>, and an internal name (atomic symbol or string) of an assigned file <fn>, will copy the indicated record out into

the file in Standard Data Format.

6.5.11. copyin

(copyin <fd> <n> <fn>)

This function is similar to COPYOUT, except that it copies the text from the sequential file <fn> to record number <n> of the random access file represented by the file descriptor <fd>.

7. Debugging and Error Handling Functions

Maryland LISP has extensive features for the handling of error contingencies, as described in this chapter. Two error-handling schemes are used by Maryland LISP. The first is implemented through the stack "unwinding" process in which LISP follows pointers down the stacks looking for "traps" which are applicable to the situation. Although several normal LISP control structures are handled this way (the PROG feature among them), the user has access to this facility only through the ATTEMPT and ERROR functions, described in the first section here.

The second error-handling scheme is the contingency expression mechanism. In this case, when an error occurs, rather than giving up and unwinding the stack, LISP transfers control to a user- or system-specified routine to allow debugging and possibly specification of a corrective measure that will allow the computation to continue safely.

When Maryland LISP encounters an error it cannot handle, such as a Guard Mode violation during garbage collection, it describes the error to the user and asks the user to specify if processing should continue. There is usually little risk in continuing, but LISP makes no guarantees about its performance in this case. In any event, a problem of this kind should be reported to a LISP hacker.

The functions documented in this chapter are ATTEMPT, ERROR, CONLIM, BACKTR, OFFENDER, EOFCON, MISSCON, FUNCON, BINDCON, CARCON, BRKCON, BREAK, and UNBREAK.

7.1. Error Handling Functions

7.1.1. attempt

```
(attempt <exp> (<c-1> <e-1-1> ... <e-1-n1>) ...
               (<c-m> <e-m-1> ... <e-m-nm>))
```

This special form is used to evaluate an expression and handle any errors which it may provoke. The first argument to ATTEMPT, <exp>, should be an EVALuatable s-expression. The succeeding arguments should be n-tuples of the form described above, where each <c-i> is an integer, and the <e-i-1>...<e-i-ni> form a set of expressions which will be handed to DO (and thus EVALuated in order), should a stack unwind which is looking for trap number <c-i> occur.

If <exp> is EVALuated successfully, its result will be the value of ATTEMPT. Otherwise, if an unwind occurs on trap number <c-i>, then the value returned by DO, ie., <e-i-ni>, will be returned. In the case that <exp> provokes an unwinding on a trap number not included as a <c-i>, the unwinding will continue down

the stack, past the ATTEMPT, until such a trap is found or, failing that, to the latest level of LISP supervision.

All positive integers (less than 2^17-1) are available to the user as trap numbers. LISP, though, has its own meanings for several negative trap numbers, specifically:

number	purpose
0	general-purpose EXEC error (eg., arithmetic error)
-1	return to last prog because of a RETURN
-2	return to last prog because of a GO
-3	return because of break-key interrupt
-4	error trap not found
-5	return to READ routine after :OOPS
-6	return because of non-transparent control card
-7	initiate backtrace because of :PEEK
-11	sequential end of file
-12	no more compiled code area

For example,

```
(csetq fread (lambda nil
     (attempt (read) (-11 "END OF FILE"))))
```

would define a function which behaves just like READ for purposes of reading sequential files until an end-of-file is encountered, when it returns the string "END OF FILE".

In addition, several of LISP's library packages make use of this facility. The user should check the appropriate documentation to make sure there are no conflicts between the user's and the package's trap numbers.

7.1.2. error

```
(error <n>)
```

The ERROR function is used in conjunction with ATTEMPT to cause an unwinding of the stack on trap number <n>. If a trap by that number resides on the stack, control will revert to it in the manner described above. Otherwise, control will revert to the latest level of LISP supervision. The user should not make use of any of the negative reserved trap numbers listed in the table above so as to avoid confusing LISP.

7.1.3. conlim

```
(conlim <n>)
(conlim)
```

Sometimes, a complicated error handling routine will itself

provoke an error which it or some other routine must handle. In order to avoid an infinite loop or similar explosion resulting from such an occurrence, there is a limit (initially 25) on the number of such occurrences which can happen during error handling. If the user finds the current value of this number inappropriate, it can be changed using the CONLIM function. This function takes as its argument the new value of the contingency limit (default is the old value), and returns in either case the old value.

7.1.4. backtr

(backtr <n>)

This function is used to specify the extent of the listing which is provided when an error causes the stack to be unwound back to an error trap. If <n> is T, a full backtrace is provided, if <n> is NIL, no backtrace is provided, and if <n> is an integer, up to <n> stack entries will be dumped when a backtrace occurs. Initially, LISP is set to give no backtrace.

7.2. Contingency Definition Functions

The six functions which are the subject of this section are used to transfer control to a specified s-expression in case one of the following kinds of errors occurs: 1) EOFCON - end of file, 2) MISSCON - missing argument to function, 3) FUNCON - object specified as function is not executable, ie., either a linker node or compiled or system code, 4) BINDCON - attempt to get value of unbound atom, 5) CARCON - attempt to take CAR of CDR of atom, or 6) BRKCON - BREAK-@@X sequence used by user to interrupt LISP. Since all five are used in basically the same manner, one example is given here to demonstrate how these functions can be used.

Example.

Suppose the atomic symbol "A" has no value.

```
EVAL: A

This would normally return...

UNBOUND A
HELP:
```

However, suppose we had previously the following:

```
(BINDCON '(DO (PRINT "UNBOUND ATOM, RETURNING NIL")
              NIL))
VALUE: NIL
```

Then,

```
EVAL: A

Now we get...

UNBOUND ATOM, RETURNING NIL
VALUE: NIL
```

7.2.1. offender

(offender)

In the case that an error is detected and control is passed to a user-defined contingency function, (OFFENDER) returns the object whose inappropriate properties provoked the error.

Examples.

Now we can embellish the example by adding OFFENDER:

```
EVAL: (BINDCON '(DO (PRINT (OFFENDER))
                    (PRINT " IS UNBOUND, RETURNING NIL") NIL))
VALUE: (DO (PPINT UNBOUND ATOM, RETURNING NIL) NIL))
```

Note that the old BINDCON expression is returned.

Now let's try it again...

```
EVAL: A
```

And we get the friendly message:

```
A IS UNBOUND, RETURNING NIL
VALUE: NIL
```

Another approach would be to enter a READ-EVAL-PRINT loop to see what might be wrong.

```
EVAL: (BINDCON '(LISP (LAMBDA NIL (*READ "->"))))
```

And the old value is returned.
Now if something is unbound,

```
EVAL: A
->
```

And :PEEK T or other means can be used to poke around and see what the trouble is.

Each of these functions takes as its lone argument the new value of the contingency expression for the given kind of contingency. The default argument is the current value of the expression, NIL if none. The value returned by each of these expressions is the old value of the given contingency expression,

and NIL if the system default was in place.

If one of these functions has been used to define an action to take in the event of an error, if a function call causes such an error, the indicated s-expression will be EVALuated, and its result used as the value of the offending function call.

7.2.2. eofcon

(eofcon <sexp>)
(eofcon)

This function defines the s-expression to be EVALuated in case an end-of-file is encountered during sequential file reading. The system default is to do an (ERROR -11).

7.2.3. misscon

(misscon <sexp>)
(misscon)

This function defines the s-expression to be EVALuated in case some argument to a function is missing. In this case, the (OFFENDER) is the unbound argument. The system default is just to give a warning message and allow the atom to remain unbound.

7.2.4. funcon

(funcon <sexp>)
(funcon)

This function defines the s-expression to be EVALuated in case the CAR of an s-expression being EVALuated is not an executable object, ie., not a linker node, pointer into compiled code, or pointer at system code. The default is to print "X IS NOT A FUNCTION" for the OFFENDER X, and to request a function value via a "HELP:" prompt.

7.2.5. bindcon

(bindcon <sexp>)
(bindcon)

This function is used to define the s-expression to be EVALuated in case an atomic symbol being EVALuated has neither a constant nor fluid binding. The system default is to print an "UNBOUND X" message for the OFFENDER X, and request a value via a "HELP: " prompt.

7.2.6. carcon

(carcon <sexp>)
(carcon)

This function is used to define the s-expression to be EVALuated in case LISP catches itself trying to take the CAR or CDR of something which is not a cons node (especially an atomic symbol). The system default is to print "CANNOT TAKE CAR OR CDR OF X" message for the OFFENDER X.

7.2.7. brkcon

(brkcon <sexp>)
(brkcon)

This function is used to define the s-expression to be EVALuated in case the user enters a BREAK-@@X sequence to interrupt LISP (see page 11). When used, the value of this s-expression is ignored since the computation should be able to continue unharmed if the user determines that nothing is wrong. This facility is useful when an infinite loop is suspected or in a case where a large program is executing and the user just wishes to see how far it has gotten.

7.3. Tracing Functions

These functions are used mainly by debugging routines to trace function entries and exits, or otherwise trace program execution.

7.3.1. break

(break <a> <fn>)

This function establishes a function <fn> which will intercept any call on the function bound to the atomic symbol <a>. The function <fn> should take a number of arguments which is two more than the number <a>'s function takes. Whenever <a>'s function is called, <fn> will receive control with <a> followed by <a>'s old value and all the arguments which were given for <a> as its arguments.

Examples.

```
(break 'cons (lambda (atm cns arg1 arg2)
     (prin1 "CONSING ") (prin1 arg1)
     (prin1 " AND ") (prin1 arg2) (print ".")
     (cns arg1 arg2)))
```

This causes a message to be printed out whenever the function CONS is called. While this BREAK is in effect,

```
(cons 'X '(A B C))
```

would print the message:

CONSING X AND (A B C).

7.3.2. unbreak

(unbreak <a>)

This function removes the effects of the last BREAK done on the atomic symbol <a>.

Examples.

(unbreak 'cons)

This will undo the BREAK done in the above example.

8. Property List Functions

Property lists in Maryland LISP are lists attached to individual atomic symbols which store information in one of two ways. The usual use of a property list, the storage of indicator/value pairs, is implemented as dotted pairs of indicators and their associated values in the list. There are also "flags", which are single atomic symbols on a property list whose simple presence or absence is useful.

The functions documented in this chapter are used to manipulate property lists. They are: PUT, GET, REMPROP, FLAG, IFFLAG, UNFLAG, and PROP.

8.1. put

(put <a> <i> <v>)

Associates the value <v> with the indicator <i> on the property list for the atomic symbol <a>. The value is <a>.

Examples.

```
(put 'TOMATO 'COLOR 'RED)
(put 'HADRON 'SIZE 'REALLY-SMALL)
(put 'LXM 'OPCODE 266)
(put 'QUOTE 'FUNC-TYPE 'SPECIAL-FORM)
```

8.2. get

(get <a> <i>)

Returns the value associated with the indicator <i> on the property list of the atomic symbol <a>. If the indicator <i> is not present on the property list, NIL is returned. (Note that the converse is not true, that is, if GET returns NIL, it is not necessarily true that the indicator is not present; perhaps it is there and is paired with NIL.)

Examples.

Given the examples under PUT,

```
(get 'TOMATO 'COLOR) = RED
(get 'HADRON 'SIZE) = REALLY-SMALL
(get 'LXM 'OPCODE) = 266
(get 'QUOTE 'NON-EXISTENT-PROPERTY) = NIL
```

8.3. remprop

(remprop <a> <i>)

Removes the indicator <i> and its associated value from the

property list for the atomic symbol <a>, if they are present. The value is <a>.

Examples.

Given the above examples, doing:

```
(remprop 'HADRON 'SIZE)
```

causes:

```
(get 'HADRON 'SIZE) = NIL
```

8.4. flag

(flag <a> <f>)

Places the flag <f> on the property list for the atomic symbol <a>, if it is not already present. The value is <a>.

Examples.

```
(flag 'APPLE 'TASTY)
(flag 'SULFUR-DIOXIDE 'CARCINOGENIC)
(flag 'YELLOW-BELLIED-SAP-SUCKER 'ISA-BIRD)
```

8.5. ifflag

(ifflag <a> <f>)

Returns T if the flag <f> can be found on the property list for the atomic symbol <a>, and NIL otherwise.

Examples.

Given the above examples,

```
(ifflag 'APPLE 'CARCINOGENIC) = NIL
(ifflag 'SULFUR-DIOXIDE 'TASTY) = NIL
(ifflag 'YELLOW-BELLIED-SAP-SUCKER 'ISA-BIRD) = T
```

8.6. unflag

(unflag <a> <f>)

Removes the flag <f> from the property list for the atomic symbol <a>, provided it is actually present. The value is <a>.

Examples.

Given the above examples, the damage can be undone by:

```
(unflag 'APPLE 'TASTY)
(unflag 'SULFUR-DIOXIDE 'CARCINOGENIC)
```

```
(unflag 'YELLOW-BELLIED-SAP-SUCKER 'ISA-BIRD)
```

8.7. prop

(prop <a> <i> <fn>)

If the indicator <i> has a value associated with it on the property list for the atomic symbol <a>, the value is returned. Otherwise the function <fn> of no arguments is called and its result is returned.

Examples.

```
(csetq getl (lambda (atm ind)
     (prop atm ind (lambda nil 'not-present))))
```

Something like this can be used to distinguish the situation where an indicator is associated with NIL on a property list from the situation where the indicator is not on the property list at all.

9. Logical and Conditional Functions

Unlike most languages, LISP has very few constructs for selectively executing program text. These and the other logical functions are documented in this chapter. They are: COND, AND, OR, and NOT.

9.1. cond

```
(cond (<exp-1> <res-1-1> ... <res-1-m1>) ...
      (<exp-n> <res-n-1> ... <res-n-mn>)))
```

This is the basic LISP conditional function, a special form of an indeterminate number of arguments. It evaluates each <exp-i> in order until it finds one which evaluates non-NIL. Then the <res-i-1>, . . ., <res-i-mi> are evaluated in order, and the result of evaluating <res-i-mi> is returned as COND's value. If there are no <res-i-j>'s, then the value of <exp-i> is returned, and if none of the <exp-i>'s are non-NIL, then NIL is returned.

Examples.

```
(csetq cdrassoc1 (lambda (atm lst)
     (cond ((csetq ::t1 (assoc atm lst)) (cdr ::t1))
           (t nil))))

(csetq cdrassoc2 (lambda (atm lst)
     ((lambda (val)
          (cond ((null val) nil)
                (t (cdr val))))
      (assoc atm lst))))
```

These two functions demonstrate two methods for computing a quantity as a condition of a COND, and then to use the quantity only if it is non-NIL.

9.2. and

```
(and <sexp1> . . . <sexpn>)
```

This is a special form which evaluates is arguments in order until one evaluates to NIL, or until all have been found to evaluate non-NIL. In the former case, NIL is returned, and in the latter case, the result of evaluating <sexpn> is returned. is returned.

Examples.

```
(and T '(A . B) NIL 'X) = NIL
(and) = T
(and x) = x for all x.
```

(and (csetq abc '(A B C)) (csetq ww NIL) (csetq R87 1))

This will change the values of ABC and WW, but not R87, and return NIL.

9.3. or

(or <sexp1> . . . <sexpn>)

This is a special form which evaluates its arguments in order until one evalutates non-NIL or until all have been found to evaluate to NIL. In the former case, the first non-NIL value is returned, and in the latter case, NIL is returned.

Examples.

(or NIL NIL '(A . B) (csetq WW 'E)) = (A . B)
The value of WW will not be changed.

(or (zerop x) (greaterp (quotient 1 x) 0.2))
If x=0, the second expression will not be evaluated.

(or (atom x) (eq (car x) 'A))
If x is an atom, then (CAR X) will not be evaluated.

(or) = NIL
(or x) = x for all x.

9.4. not

(not <sexp>)

Returns T if <sexp> is NIL, and NIL otherwise. This is the same function as NULL. In Maryland LISP, the values denoting truth and falsehood (e.g., of predicates) are the atomic symbols T and NIL. Other LISP systems ([McC62]) prefer to use other atoms or even non-atoms to represent these concepts. The theoretical treatment of LISP in [All78] defines a different class of objects to be the values of predicates, so that the functions NOT and NULL are quite different.

10. Program Feature Functions

The three functions, PROG, GO, and RETURN, described in this chapter implement the standard LISP PROG facility. Since the standard LISP references cited in the introduction describe this facility in great detail, only features specific to Maryland LISP are discussed here.

10.1. prog

(prog <locs> <sexp1/lab1> . . . <sexpn/labn>)

This is the PROG function, a special form of an indeterminate number of arguments. As usual, <locs> is the local list. It should be a list of s-expressions, each of which is an atom or a list containing an atom and an s-expression which can be EVALuated. In the case of an atom standing alone, its initial value is set to NIL, and when an atom is given in a 2-member list, its initial value is set to the result of EVALuating the s-expression it is paired with.

Each of the other arguments should be an atomic symbol, denoting a label, or an s-expression which can be EVALuated so as to have some effect on the world. If control "falls out the end" of a PROG before encountering a RETURN, NIL is returned as the value of the PROG call. The two special functions relating to progs, GO and RETURN, are described next, and they may appear only in cases when control is inside a PROG.

Examples.

```
(csetq rev (lambda (l)
     (prog (r)
      loop (cond ((null l) (return r)))
           (setq r (cons (car l) r))
           (setq l (cdr l))
           (go loop))))
```

Could be used to define a function which reverses a list.

10.2. go

(go <l>)

This is the standard GO function, a special form which transfers program control to the label <l> in the last PROG in which it occurred. GOing to a non-existent label causes an error message at the latest level of LISP supervision.

Examples.

```
(csetq go1 (lambda (lab)
     (eval (list 'go lab))))
```

Could be used to define a function which behaves like GO, except that it EVALuates its argument.

10.3. return

(return <val>) (return)

This is the standard function of one optional argument which declares a value for the most recently entered PROG or level of LISP supervision, entered via the LISP function. (If the RETURN function is given outside a PROG at the lowest level of LISP supervision, LISP will be exited. This way, it is possible to write a LISP function that will exit LISP.) If the argument is not given, NIL is returned.

11. Numeric Functions

Maryland LISP provides several functions for doing arithmetic. There are three numeric cata types in Maryland LISP, integers, octals, and reals (type numbers 1, 2, and 3, respectively). Except for the input/output routines, all LISP functions treat integers and octals the same. The arithmetic functions provide type conversions between these numeric types when necessary. However, most of the functions do no type checking beyond real/non-real; a cons node, for instance, will usually be treated as a large, fixed-point number. Thus some degree of caution is required.

The functions defined in this chapter are NUMBERP, ZEROP, EQUAL, GREATERP, LESSP, MINUSP, FIXP, FLOATP, PLUS, TIMES, DIFFERENCE, QUOTIENT, REMAINDER, ADD1, SUB1, MINUS, SIN, COS, LOG, POWER, LOGOR, LOGAND, LOGXOR, LEFTSHIFT, FUZZ, and ENTIER.

11.1. Arithmetic Predicates

An arithmetic predicate function takes numeric arguments and returns a value of T or NIL.

11.1.1. numberp

(numberp <n>)

Returns T if <n> is an integer, octal, or real and NIL otherwise.

Examples.

```
(numberp 4) = T
(numberp 3.141592E14) = T
(numberp '!3) = NIL
(numberp 470362Q) = T
(numberp -0) = T
(numberp '(A . B)) = NIL
```

11.1.2. zerop

(zerop <n>)

Returns T if <n> is zero. If <n> is an integer, the test is <n>=0, if <n> is an octal, the test is <n>=0Q, and if <n> is a real, the test is <n>=0.0.

Examples.

```
(zerop 0) = T
(zerop 0.0) = T
(zerop 1) = NIL
```

(zerop 0Q) = T

11.1.3. equal

(equal <n1> <n2>)

See also the definition on page 31.

If <n1> and <n2> are numeric, EQUAL returns T if they are numerically equal. If either <n1> or <n2> is real, then the other is converted to real, otherwise an integer comparison is made.

Examples.

(equal 4 4) = T
(equal 4 4.0) = T
(equal 12 14Q) = T
(equal -0 777777777777Q) = T
(equal 3.4E2 340.0) = NIL

Even though this isn't FORTRAN, one still needs to worry about microscopic floating-point round-off errors which make "obviously" EQUAL floating-point quantities not EQUAL. In future examples involving floating-point results, "=" should be taken to mean "is approximately EQUAL to".

11.1.4. greaterp

(greaterp <n1> <n2>)

Returns T if <n1> is numerically greater than <n2>. Type conversion is similar to the discussion under EQUAL.

Examples.

(greaterp 4 3) = T
(greaterp 0 -0) = NIL
(greaterp 3.01 3) = T
(greaterp 14Q 12) = NIL

11.1.5. lessp

(lessp <n1> <n2>)

Returns T if <n1> is numerically less than <n2>. Again, type conversion is similar to the discussion under EQUAL.

Examples.

(lessp 3 4) = T
(lessp 3.1415 3.1415E0) = NIL

```
(lessp 12 14Q) = NIL
(lessp -0 0) = T
```

Note that GREATERP says that -0=0, but that LESSP says that -0<0. Which comparison function one uses depends on whether -0=0 or not in the given application.

11.1.6. minusp

(minusp <n>)

Returns T if the argument is negative. Integers and octals are tested as integers, and reals are tested as reals.

Examples.

```
(minusp -34) = T
(minusp -3.4) = T
(minusp -34Q) = T
(minusp -0) = T
(minusp 1) = NIL
```

11.1.7. fixp

(fixp <n>)

Returns NIL if <n> is a real number, and T otherwise.

Examples.

```
(fixp 3) = T
(fixp 3.4392) = NIL
(fixp '(A . B)) = T    (Be careful!)
(fixp 14Q) = T
```

11.1.8. floatp

(floatp <n>)

Returns T if <n> is a real number, and NIL otherwise.

Examples.

```
(floatp 3.493E3) = T
(floatp 3Q) = NIL
(floatp 19372) = NIL
(floatp '(A . B)) = NIL
```

11.2. Arithmetic Functions

11.2.1. plus

(plus <n1> <n2> . . . <nk>)

Returns the numeric sum of its arguments. The arguments are added as integers until one is found to be real, in which case the addition continues as a real addition. If k=0, the result is 0, and if k=1, the result is <n1>.

Examples.

(plus 1 2 3) = 6
(plus) = 0
(plus 4.5) = 4.5
(plus 1 2 3 4.0) = 10.0
(plus 1Q 2Q 3Q) = 6

11.2.2. times

(times <n1> <n2> . . . <nk>)

Returns the numeric product of its arguments. Type conversion conventions are as in PLUS. If k=0, the result is 1, and if k=1, the result is <n1>.

Examples.

(times 2 3 4) = 24
(times) = 1
(times 45) = 45
(times 2 9 1.01) = 1.818E1
(times 12Q 10Q) = 80

11.2.3. difference

(difference <n1> <n2>)

Returns the numeric difference of its arguments. If either argument is real, the result is real, otherwise it is an integer.

Examples.

(difference 45 26) = 19
(difference 47.0 12) = 3.5E1
(difference 12Q 10Q) = 2
(difference NIL NIL) = 0

11.2.4. quotient

(quotient <n1> <n2>)

Returns the numeric quotient of its arguments. Type conversion conventions are as in DIFFERENCE. If <n2>=0, an (ERROR 0) will result.

Examples.

```
(quotient 4 2) = 2
(quotient 12.0 4) = 3.0
(quotient <anything> 0) produces an 'ARITHMETIC ERROR'
(quotient <anything> 0.0) likewise, an (ERROR 0)
(quotient 100Q 10Q) = 8
```

11.2.5. remainder

(remainder <n1> <n2>)

Returns the integer remainder when <n1> is divided by <n2>. Both arguments must be of type integer or octal.

Examples.

```
(remainder 5 3) = 2
(remainder 5.0 3) produces "garbage"
(remainder 5 3.0) ditto.
(remainder 10Q 3Q) = 2
(remainder 999 9) = 0
```

11.2.6. add1

(add1 <n>)

Returns the quantity <n>+1. If <n> is real then the result is real, otherwise the result is an integer.

Examples.

```
(add1 1) = 2
(add1 1.0) = 2.0
(add1 7Q) = 8
```

11.2.7. sub1

(sub1 <n>)

Returns the quantity <n>-1. If <n> is real then the result is real, otherwise the result is an integer.

Examples.

```
(sub1 2) = 1
(sub1 2.7) = 1.7
(sub1 10Q) = 7
(sub1 (add1 x)) = (plus x 0), for any object x
```

(plus x 0) instead of x because octals are converted to integers by ADD1 and SUB1, as are any non-floating point objects. For example, if x=NIL in the last example, the EQUALity holds.

11.2.8. minus

(minus <n>)

Returns the quantity -<n>. If <n> is real the result is real, otherwise the result is an integer.

Examples.

```
(minus 4) = -4
(minus 0) = -0
(minus 4.3) = -4.3
(minus -9) = 9
(minus 123Q) = 777777777654Q
```

11.3. Fortran Library Math Routines

11.3.1. sin

(sin <n>)

Computes the sine of <n>, which should be in radian measure. The result is always real.

Examples.

```
(sin 0) = 0.0
(sin 3.1415926535) = 9.13912E-8  (ie., about 0.0)
(sin (quotient 3.1415926535 2)) = 1.0
```

11.3.2. cos

(cos <n>)

Computes the cosine of <n>, which should be in radian measure. The result is always real.

Examples.

```
(cos 0.0) = 1.0
(cos (quotient 3.1415926535 2)) = 4.56956E-8
```

11.3.3. log

(log <n>)

Computes the natural logarithm of <n>, which should be a positive number. The result is always real.

Examples.

```
(log x) produces an error for x <= 0.
(log 1.0) = 0.0
```

```
(log 2.7182818284) = 9.99999E-1

(csetq logn (lambda (n b)
     (quotient (log n) (log b)))
```

Defines a function which takes the logarithm of n to the base b, where n and b are positive numbers.

11.3.4. power

(power <b> <e>)

Raises <b> to the <e>th power and returns the result as a real number. The arguments can be of any numeric type, but <b> may never be negative.

Examples.

```
(power 2.0 10.0) = 1.024E3
(power 2.7182818284 (log x)) = x for all positive x
(power x 0) = 1.0 for all positive x
```

11.4. Bitwise Logical Functions

The arguments to these functions are treated as bit masks rather than as numeric quantities, so that any argument which is not an octal or real will be used in its internal form. This is useful for getting at the mantissa of a real number, or a specific part of an atom or linker node, for example.

11.4.1. logor

(logor <w1> <w2> . . . <wk>)

Performs a bitwise logical "or" on its arguments and returns an octal result.

Examples.

```
(logor 17Q 10000Q 317000000Q) = 317010017Q
(logor 5555Q 2222Q) = 7777Q
(logor 1234777Q 7771234Q) = 7775777Q
```

11.4.2. logand

(logand <w1> <w2> . . . <wk>)

Performs a bitwise logical "and" on its arguments and returns an octal result.

Examples.

```
(logand 0Q x) = 0Q for all x.
(logand 1234323Q 177Q) = 123Q
(logand 777777000000Q x) masks out the lower half of x
```

11.4.3. logxor

(logxor <w1> <w2> . . . <wk>)

Performs a bitwise logical "exclusive or" on its arguments and returns an octal result.

Examples.

```
(logxor 77600Q 377Q) = 77577Q
(logxor 777777777777Q x) gets the complement of x.
(logxor 0 x) = an octal translation of x.
```

11.4.4. leftshift

(leftshift <w> <n>)

If <n> is positive, LEFTSHIFT returns <w> logically shifted <n> to the left. Otherwise, the result is <w> circularly shifted -<n> to the right. The result is always octal. The argument <n> should be an integer or octal.

Examples.

```
(leftshift x n) = 0Q for any x if n >= 36.
(leftshift x -36) = x for any octal x.
(leftshift (leftshift x 18) -18)
      = (logand 777777Q x) for any octal x.
(leftshift 17Q 3) = 170Q
```

11.5. Miscellaneous Numeric Functions

11.5.1. fuzz

(fuzz <n>)

This function specifies an amount of rounding which is to be performed on all real arithmetic results. Specifically, the <n> lowest-order bits of every real result will be zeroed. The result will be the "fuzzing factor" being replaced. If the argument is omitted, the result is the fuzzing factor currently in force. The default fuzzing factor is 0.

The FUZZ function is used to help reduce the effects of round-off error when applying EQUAL and GREATERP, for example, to floating-point arguments.

Note that the mantissa of a UNIVAC single-precision floating-point number takes up the low-order 27 bits of a word.

11.5.2. entier

(entier <n>)

If <n> is real, the result is the greatest integer which is less than or equal to <n>. If <n> is integer or octal, it is returned as the result.

Examples.

```
(entier 4) = 4
(entier 14Q) = 14Q
(entier 127.3931) = 127
(entier -1.467) = -2  (not -1)
```

12. String Functions

Maryland LISP has a FIELDATA (upper-case only) string facility, in which the strings may be of any length. Strings are delimited by double-quotes (´"´), and they may not contain at-signs (´@´). In order to have a double-quote in a string, type two double-quotes together (eg., "HE SAID, ""HI"".") . The characters in a string are numbered so that the first one is number 1, and the number of the last one is returned by the SIZE function. The atom NIL serves as the string of length zero. Note that EQUAL should be used instead of EQ to test equality of strings.

The functions described in this section are useful for manipulating strings. They are: SIZE, STRINGP, SREV, MATCH, CAT, SUBSTRING, READSTR, STRING, and ATSYMB.

12.1. size

(size <s>)

Returns, as an integer, the length of the string <s> in characters. If <s>=NIL then the result is 0.

Examples.

```
(size "ABCDE FGH") = 9
(size NIL) = 0
(size """") = 1
```

12.2. stringp

(stringp <s>)

Returns T if <s> is a string, and NIL otherwise.

Examples.

```
(stringp NIL) = T
(stringp 'ABCDEF) = NIL
(stringp "ABCDEF") = T
(stringp """""") = T
(stringp 123) = NIL
```

12.3. srev

(srev <s>)

Returns the string <s> in a reversed form, that is, with the characters in reverse order.

Examples.

```
(srev NIL) = NIL
(srev "ABCD") = "DCBA"
(srev "XY&' *=()") = ")(=* '&YX"
```

12.4. match

(match <s1> <s2>)

Returns NIL if the string <s1> does not appear as a substring of <s2>. Otherwise, the value is the position (an integer) in <s2> in which <s1> can first be found as a substring.

Examples.

```
(match "ABC" "AABABCABCD") = 4
(match "CAT" "DOG") = NIL
(match "ABC" "B") = NIL
(match "$" "$$$$$$$") = 1
```

12.5. cat

(cat <s1> <s2>)

This function returns as its result the string <s1> with the string <s2> concatenated on the end of it. If either <s1> or <s2> is NIL, the other is returned uncopied.

Examples.

```
(cat NIL x) = x for any string x.
(cat x NIL) = x for any string x.
(cat "ABC" "DEF") = "ABCDEF"
(cat "123456789" "0") = "1234567890"
```

12.6. substring

(substring <s> <c> <n>)
(substring <s> <c>)

This function will compute and return a substring of the string <s> in one of two different ways. If three arguments are given, as in the first form, a substring <n> characters long starting at position <c> in <s> will be returned. Both <c> and <s> should be fixed-point numbers. If two arguments are given, as in the second form, the substring of <s> which starts at position <c> and continues to the end of <s> will be returned. If <c> is out of its proper range in either case, the result will be either NIL or shorter than the expected length.

Examples.

```
(substring "ABCDEFGH" 2 3) = "BCD"
(substring "$'&()##" 4) = "()##"
(substring "12345" 3 7) = "345"
(substring "XYZPDQ" 8) = NIL
(substring x n c) = NIL if n <= 0.
```

12.7. readstr

(readstr <s>)

While performing string processing it sometimes becomes necessary to change the string representation of an entity into a list structure. The operation, somewhat analogous to COMPRESS, is accomplished by READSTR, a function of one string argument. READSTR submits its argument to the read routines for parsing and returns a list structure of equivalent print form. If the read routines find that the string is not a valid print form (e.g. unbalanced parenthesis) an End of File contingency is registered (see page 71).

EXAMPLE:

```
EVAL:  (EVAL (READSTR "(PLUS 2 3)"))
VALUE:  5

In the pathological case...

EVAL:  (READSTR "(((A))")
EOF ENCOUNTERED
EVAL:
```

12.8. string

(string <a>)

This function returns the print name of the atomic symbol <a>. The result will always be a string unless the symbol was created by GENSYM, in which case the result will be an integer.

Examples.

```
(string 'XYZ) = "XYZ"
(string NIL) = "NIL"
(string (gensym)) returns an integer
(string 'FUNCTIONS! LOADED) = "FUNCTIONS LOADED"
```

12.9. atsymb

(atsymb <s>)

This function hashes the string <s> into the oblist, and creates, if necessary, an atomic symbol with that name. If the symbol is already in the oblist, it is returned, otherwise the

newly created atomic symbol is returned.

Examples.

```
(atsymo "XYZZZ") = XYZZZ
(atsymb "NIL") = NIL
(string (atsymo NIL)) = NIL, not "NIL".
(atsymo "X Y") = X! Y
```

13. Executive Interface Functions

The designers of Maryland LISP have tried to make it a self-contained programming system, free from the details of the outside world. To allow the user to work directly with EXEC-8 as infrequently as possible, the functions documented in this chapter have been included in Maryland LISP to allow the user to do the necessary dealing with EXEC-8 from inside LISP. These functions are: EXEC, PCT, TWAIT, ED, DUMP, and LOAD.

13.1. exec

(exec <s>)

This function takes a string <s> which should be a legal 1100 EXEC CSF$ control card and submits it to CSF$, returning the octal status which the system returned from the request. The first character in the string should be an asterisk ("*") rather than an at-sign ("@"). For details on the legal CSF$ control cards, see [Uni78], Volume 2, page 4-67.

Examples.

```
(csetq assignfile (lambda (fn)
     (exec (cat "*ASG,A " (cat (string fn) ","")))))
```

This function makes sure that the file whose internal name it is given as an atomic symbol is assigned to the run.

```
(csetq usename (lambda (un fn)
     (exec (index (list "*USE " (string un) ","
               (string fn)) nil cat))))
```

This function establishes @USE names.

13.2. pct

(pct <n> <s>)

This function gets the user's Program Control Table and returns a list of <n> octal words starting at the <s>th word. The PCT contains interesting information about the run status, including the various user identifications, and the status of files. More specific information about the PCT and PCT$, the EXEC-8 PCT access request can be found in [Uni78], especially Volume 2, page 4-30. (The exact structure of the PCT changes often, so that the only way to get exact diagrams of it is by scrounging. The copy of the PRM (ie., [Uni78]) which resides in the Computer Science Center Program Library is a good starting point.)

Examples.

```
(pct 1 1) = user's runid in octal form
(pct 2 17) = user's project id in octal form
(pct 2 19) = user's account number in octal form
```

13.3. twait

(twait <n>)

This function takes a non-negative integer argument <n> and returns NIL after waiting <n> seconds of real time.

Examples.

```
(csetq idle (lambda nil
     (prog nil
      loop (print "I WILL BE BACK IN A SECOND")
           (twait 30) (go loop)))
```

This is a function which will print a message every thirty seconds until stopped by a BREAK-@@X sequence or something more forceful.

13.4. ed

(ed <s>)
(ed <s> <c>)

This function allows the user to use the University of Maryland Text Editor [Hag77] without leaving LISP. The first argument, <s>, should be a string which can be interpreted as an Editor control card, with the first character an asterisk ("*") rather than an at-sign ("@"). The string is parsed and interpreted as though typed at the Executive, and the user is placed in the Text Editor. If a second argument, <c>, is specified, it should be a string which will be given to the Text Editor as a "first command", which will be executed before giving a prompt. If a fatal error, such as an attempt to access a non-existent file, is encountered, the editing session will be aborted. The result of the ED function is an integer denoting the number of lines output by the Editor to the source output file or element.

Examples.

```
(csetq textcopy (lambda (f1 f2)
     (ed (cat "*ED " (cat f1 (cat "," f2)))
         "EXIT")))
```

This function copies the text in file/element F1 into file/element F2. (eg., (textcopy "INFILE." "LIBFILE.X-ELT/NUM3").)

13.5. dump

(dump <obj> <f/e>)

This function will cause the binary representation of the LISP object <obj> to be written out in a compact form to the file or element encoded as <f/e>. If <f/e> is an atomic symbol, it will be interpreted as the internal name of some data file which exists and can safely be written into. On the other hand, if <f/e> is a dotted pair of atomic symbols, the CAR is taken to be the internal name of an existing program file, and the CDR is taken to be the name of an element which will be created to hold the dumped output. The dumped objects can be read back in by (and only by) the LOAD function. The <obj> argument should be either an atomic symbol or a list of atomic symbols. The only atomic symbols' value cells which will be followed will be the ones given as arguments in this way. This is done so that if a function points at the definition of another function, that definition will not unintentially overwrite the one in effect when LOAD is called.

Examples.

One neat trick is to assign the object being dumped to an atomic symbol which describes what it is, e.g.,

```
(csetq PROJECT4! FUNCTIONS! LOADED '(SORT MERGE REV1
    SEARCHLIST UNION INTERSECTION))
(dump 'PROJECT4! FUNCTIONS! LOADED '(LIB-FILE . PROJ4))
```

The advantages of this become clear in the examples section for LOAD.

13.6. load

(load <f/e>)

This function recovers LISP objects which were sent to file/element <f/e> by the DUMP function.

Examples.

Suppose that some things had been dumped as in the example under DUMP. Then if

```
(load '(LIB-FILE . PROJ4))
```

is typed at LISP, the result printed on the terminal is:

```
PROJECT4 FUNCTIONS LOADED
```

14. Program Statistics Functions

The functions documented in this chapter are used to keep statistics on the operation of LISP during run-time. They are useful for testing the efficiency of programs, and guiding storage management. They are: TIME, GCTIME, DATE, DTIME, TRASH, MEMORY, GROW, SWAPS, and *PACK.

14.1. Timing Functions

14.1.1. time

(time)

This returns the amount of memory time used so far in the current LISP session, less the time taken by garbage collection. The result is an integer in units of milliseconds.

Examples.

```
(csetq timer (lambda (expr iter)
     (prog ((sum 0) (n iter))
      loop (cond ((zerop iter)
                  (return (quotient sum n))))
           (setq sum (plus sum
                (print (plus (minus (time))
                   (do (eval expr) (time))))))
           (setq iter (sub1 iter))
           (go loop))))
```

This defines a function which allows one to test the efficiency of another function. Given an s-expression and an integer, it EVALs the s-expression the indicated number of times, printing the amount of time each EVALuation took, and returning the average of the printed values.

14.1.2. gctime

(gctime)

This returns the amount of memory time which has been used in garbage collections during the current LISP session. The result is an integer in units of milliseconds.

14.2. Time/Date Functions

14.2.1. date

(date)

This returns a list with three integer elements. The first is the number of the current month, between 1 and 12, the second is the number of the current day, between 1 and 31, and the third is the last two digits of the current year, 78 at this writing. All three are integers.

14.2.2. dtime

(dtime)

This returns the current time of day, an integer in units of milliseconds past midnight.

14.3. Memory Management Functions

14.3.1. trash

(trash)

The TRASH function calls the garbage collector, and returns as its value the number of data pages the collector was able to return to the available page list.

14.3.2. memory

(memory)

This function returns a 2-member list of integers (<u> <a>), where <u> is the number of words of data area being used, and <a> is the total number of such words available.

14.3.3. grow

(grow <n>)
(grow)

(No virtual memory option): This function causes <n> more 512-word blocks of memory to be allocated for LISP. The result is the new highest address available to LISP data areas. If LISP attempts to grow beyond the current system core limit, an error message will result and control will revert to the latest ATTEMPT to trap on error 0, or, failing that, the latest level of LISP supervision. If <n> is zero, no growing is attempted, and the current upper limit is returned. The default value for <n> is zero.

(Virtual memory option): The GROW function causes <n> more 16K data banks to be made available to the storage allocation routines. There are six banks which GROW can add, and an attempt

to allocate more than six will cause just the six to be allocated. See the Appendix to this manual on the virtual memory option for further details on virtual memory.

14.3.4. swaps

(swaps)

If the virtual memory option is on, this function will return an integer representing the number of times the swapping algorithm has been called to make a data bank available. If the virtual memory option is not on or if no GROWing has yet occurred, SWAPS will return 0.

14.4. *pack

(*pack <n>)

This function sets a flag to trigger a garbage collection every time a new page for objects of type number <n> is allocated, thus keeping pages of that type packed. Because of the costs involved in repeated garbage collections, this should not be used except in most extreme circumstances.

15. Compiler Functions

The functions documented in this chapter were designed for use by the compiler in analyzing a function definition. For the sake of completeness, and because some are of use to the more sophisticated user, they are presented here. They are: *DEF, *SPEC, *MACRO, *CHAIN, *BEGIN, *EMIT, *ORG, *EPT, *EXAM, *DEPOSIT, MANIFEST, and BUFFER.

15.1. Function Definition Retrieval Functions

15.1.1. *def

(*def <a>)

This function, given an atomic symbol <a> which is constantly bound to a regular function defined through LAMBDA, will return a list of the arguments which were submitted to LAMBDA when the function was defined. If <a> is not constantly bound, or if <a>'s value is not a function, or if <a> is bound to a special form, macro, compiled or assembled function, or system function, *DEF returns NIL.

Examples.

If FOOBAR has been defined like so:

```
(csetq foobar (lambda (x y)
     (plus (times x y) (times (add1 x) (sub1 y)))))
```

then

```
(*def 'foobar) = '((x y) (plus (times x y)
     (times (add1 x) (sub1 y))))

(csetq substdef (lambda (fn new old)
     (cset fn (eval (cons 'lambda
          (subst new old (*def fn)))))))
```

This defines a function which can make substitutions in the definitions of user-defined regular functions. For example,

```
(substdef 'foobar 'quotient 'times)
```

causes:

```
(*def 'foobar) = ((x y) (plus (quotient x y)
     (quotient (add1 x) (sub1 y))))
```

15.1.2. *spec

(*spec <a>)

This function, given an atomic symbol <a> which has been defined to be a special form by DEFSPEC, will return the second argument which was submitted to DEFSPEC when <a> was defined. Note that this may be a linker node or a system function, but will never be a cons node. If <a> is not a user-defined special form, *SPEC returns NIL.

Examples.

```
(csetq specp (lambda (atm)
     (not (null (*spec atm)))))
```

Defines a function which, given an atomic symbol as its argument, returns T if the symbol defines a special form and NIL otherwise.

```
(csetq specdef (lambda (atm)
     (*def (rplaca (plus 0q 0q) (*spec atm)))))
```

Defines a function which takes an atomic symbol and returns its definition if it is a special form and NIL otherwise. For example, if FOO is defined:

```
(defspec foo (lambda (x y z)
     (plus (times x y) z)))
```

then doing

```
          (specdef 'foo)
```

returns

```
          ((x y z) (plus (times x y) z))
```

Note, however, that *SPEC and functions defined with it do not work with system-defined special forms such as CSETQ, QUOTE, and OR.

15.1.3. *macro

(*macro <a>)

This function is similar to *SPEC, except that <a> must have been defined a macro through DEFMAC.

Examples.

Functions similar to the ones given with *SPEC can be defined for *MACRO:

```
(csetq macrop (lambda (atm)
     (not (null (*macro atm)))))
```

```
(csetq macdef (lambda (atm)
     (*def (rplaca (plus 0q 0q) (*macro atm)))))
```

These are used in the same way as SPECP and SPECDEF, which were defined as examples above.

15.1.4. *chain

(*chain <cr>)

This function takes as its argument one of the special C<A\D>*R atomic symbols, and returns the octal code used by LISP to follow the indicated chain, where each bit stands for a direction, with 1=CDR and 0=CAR, and a leading 1 bit for counting purposes.

Note that the CAR and CDR functions are implemented directly rather than though the general bit array mechanism. *CHAIN will return 2Q and 3Q respectively for these, even though those bits will not normally exist.

Examples.

```
(*chain 'cr) = 1q
(*chain 'car) = 2q
(*chain 'cdr) = 3q
(*chain 'cadaaadaar) = 7059
(*chain 'cadddddddddddddddddddddddddddddddddddr)
      = 777777777777q
```

(Anyone who would use the last one deserves to have to count the D's himself.)

15.2. Code Generating Functions

15.2.1. *begin

(*begin)

This function is used by the compiler and assembler to set up a boundary in the compiled code area of core, usually to prepare for the generation of code for a new function. Its value is a pointer to the first location at which the new code will be placed.

15.2.2. *emit

(*emit <l>)

This function, given a list <l> of octal words, generates a word of compiled code by or'ing together the elements of <l>. The result is a pointer to the location where the new instruction

was placed.

15.2.3. *org

(*org <a>)

This function is used by the compiler and assembler to declare a position in core where *EMIT should start placing code. It is used mainly to plug in forward references.

15.3. Other Compiler-Oriented Functions

15.3.1. *ept

(*ept <n>)

This function, given a non-negative integer <n> less than 32 returns a pointer at the <n>th compiler entry point into LISP. The entry point table, which is described in detail in the LISP assembler documentation, is a set of pointers at LISP routines which are helpful to the compiler.

15.3.2. *exam

(*exam <p>)

This function, given a pointer <p> to an arbitrary location in memory, will return an octal word representing the contents of the word being pointed to.

Examples.

(csetq octal *exam)

This defines a function which, given an integer, returns its translation into octal.

(*exam NIL) = 250000250000Q (for the time being)
(*exam "ABC") = 607100250000Q (ditto)

15.3.3. *deposit

(*deposit <obj> <loc> <off>)
(*deposit <obj> <loc>)

This function is used to alter the contents of an arbitrary word of memory. The first argument, <obj>, should be an octal word containing the new contents of the location being changed. The second argument, <loc>, should be a pointer at the location to be changed (as opposed to an octal word containing the address to be changed). The last argument, <off>, should be an integer which will be added to the pointer given as the second argument to give the actual address to be changed. The default for <off>

is 0. The result is <obj>.

15.3.4. manifest

(manifest <sexp>)

This function informs the compiler that <sexp> can be evaluated at compile-time so as to avoid generating code to do it. When evaluated by the interpreter, MANIFEST just returns <sexp>. The argument to MANIFEST may not be a special form or macro.

15.3.5. buffer

(buffer)
(buffer <ind>)

This function will return a pointer to a new buffer, which is a blank 128-work chunk of memory of type 9. If <ind> is present and non-NIL, then whatever the buffer points to will be marked during garbage collection if the buffer itself is marked. Also in this case, the buffer will be zeroed to keep things from being unintentionally marked. If <ind> is missing or NIL, then the contents of the buffer will not be marked.

16. LISP Directives

This chapter describes the LISP directives, commands which can be typed at LISP to perform various tasks which cannot be (or are not for various reasons) implemented as standard functions. All the directives begin with colons (eg., ":LOAD"), and must begin in column 1. The directives are: :LOAD, :END, :LIST, :UNLIST, :CKPT, :RSTR, :LISP, :EXEC, :CODE, :TIME, :BACK, :PEEK :STOP, :OOPS, and :DATA.

16.1. :LOAD

This directive suppresses the "EVAL:" and "VALUE:" messages of a READ-EVAL-PRINT loop, allowing large numbers of s-expressions to be processed without producing large amounts of redundant or useless output. The suppression remains in effect until a matching :END directive is found. :LOAD/:END pairs may be nested to any depth. They are especially useful in files which are to be @ADD'ed. The normal scheme is to have a :LOAD in line 1 of the file, an :END in the next-to-last line, and some descriptive string such as "FUNCTIONS LOADED" in the last line, thus causing LISP to respond to an @ADD of the file with a friendly message indicating the loading has completed.

16.2. :END

This directive cancels the effect of its matching :LOAD directive. Also, the :END directive can be used to turn off data mode if a :DATA directive had been entered.

16.3. :LIST

This causes all input to LISP to be echoed back before being processed. The :LIST directive is useful when output is being redirected to a file so that outputs can be conveniently associated with their corresponding inputs. The echoing remains in effect until an :UNLIST directive is found.

16.4. :UNLIST

This directive turns off the echoing of input caused by the processing of a :LIST directive.

16.5. :CKPT

This directive, given the internal name of some assigned data file (with no trailing period), copies out all of LISP's data area to the indicated file for possible later recovery.

This facility is useful when one desires to run a large LISP program such as the compiler, MLISP, or PLNR repeatedly without having to @ADD or LOAD a fresh copy each time.

16.6. :RSTR

This directive, given the internal name of an assigned data file (again, without a trailing period), copies its contents in and uses them as LISP's data area. If the file was not created by :CKPT, havoc will naturally result.

Examples.

To use PLNR several times in a day, enter LISP, set up a file, load in PLNR, and :CKPT to the file:

```
@LISP*LIB.LISP
:EXEC @ASG,T PLNR-DUMP.
:EXEC @USE LISP.,LISP*LIB.
(LOAD '(LISP . PLNR))
:CKPT PLNR-DUMP
```

Now use PLNR. If it becomes necessary to leave LISP before using PLNR again, upon reentering LISP, PLNR can be loaded rather quickly by doing:

```
:RSTR PLNR-DUMP
```

16.7. :LISP

This directive causes control to return to the latest level of LISP supervision. It is useful as a response to LISP "HELP: " requests, and to make sure one is indeed talking to a LISP READ-EVAL-PRINT loop rather than one's own program or the Text Editor, for instance.

16.8. :EXEC

This directive causes the UNIVAC control card which succeeds it to be sent to CSF$. When this process completes, LISP responds with ":EXEC COMPLETED" or some error message and error code as returned by CSF$. Documentation on CSF$ and the control cards which it will process is found in [Uni78], Volume 2, page 4-67.

Examples.

```
:EXEC @ASG,T LISP-TEMP.
:EXEC @SYM,U PRETTYP-FILE.,T,PRULC
:EXEC @USE LISP.,LISP*LIB.
:EXEC @START MYQUAL*LIST-FILES.
```

16.9. :CODE

This directive should be used at the beginning of a LISP run to reserve a contiguous chunk of data area to be occupied by compiled and assembled code. Its format is:

:CODE <n>

where <n> is the number of 128-word pages being requested. This is a one-time request; if LISP runs out of this code area, no more can be allocated. Ample room is provided in LISP's static data area to handle all but the largest applications. (Custom versions of LISP can easily be put together for huge applications.)

16.10. :BANK

This directive, entered as :BANK <n>, specifies a virtual memory bank number from which all data area is to be allocated until the next :BANK or the end of the run, where <n> is T means any bank (effectively turning off the feature), NIL refers to the static data area, and a non-negative integer refers to bank number <n>, where <n> is, of course, no larger than the highest active bank number. If the v option is not in effect, the :BANK directive has no effect.

16.11. :TIME

This directive causes LISP to give a message indicating the amount of time used so far for computation and garbage collection, and the number of garbage collections performed so far in the current run of LISP.

16.12. :BACK

This causes LISP to unwind the stack back to the latest level of LISP supervision (or any -7 trap via ATTEMPT), possibly with the contents of the stack printed along the way. If the :BACK is followed by a T, everything unwound will be printed, if there is no argument nothing will be printed, and if there is an integer, that many entries will be printed.

Examples.

:BACK T

This prints out everything on the stack.

:BACK

This prints nothing, and is roughly equivalent to doing :LISP.

:BACK 12

This prints only the top 12 entries found on the stack during unwinding.

16.13. :PEEK

This directive is used like :BACK, except control is not transferred back to the latest level of LISP supervision, but is unaffected, since :PEEK is a transparent directive.

Examples.

```
:PEEK T
:PEEK
:PEEK 1
```

The first causes the entire contents of the stack to be printed, the second does nothing, and the third prints the top entry on the stack. The last one is useful as a response to a "HELP: " query from LISP, printing the object which provoked the request.

16.14. :STOP

This directive causes LISP to be exited normally. Any EXEC8 control card (except @ED, @ADD, and sometimes @EOF, will have the same effect).

16.15. :OOPS

This directive can be used to cause LISP to stop reading the input currently being typed, and to start reading again.

Examples.

```
(CSETQ FACT (LMBDA (N)
     (COND ((ZEROP N) 1)
:OOPS
(CSETQ FACT (LAMBDA (N)
     (COND ((ZEROP N) 1)
           (T (TIMES N (FACT (SUB1 N)))) )))
```

16.16. :DATA

The :DATA directive may be inserted in front of data cards to be read by a user program (via READ). This will prevent an attempt to evaluate any one of them in case something goes wrong and a premature return to the supervisor is made. Data mode is turned off when the next :END or :LISP directive is read.

17. Alphabetic Index

This is an alphabetic index of all the Maryland LISP functions defined in this section. Function names marked with an asterisk ("*") are special forms, those marked with a plus-sign ("+") are predicates (ie., they never return anything but T or NIL), and those marked with a pound sign ("#") set and access internal variables. A function of the latter kind usually has an optional argument which, if given, is set as the new value of the internal variable (or entry in the internal table) in question. In either case, the old value of the variable is returned.

*begin		102	bindcon	(#)	71
*car		31,42	break		72
*cdr		31,43	buffer		104
*chain		102	car		30
*def		100	carcon	(#)	71
*deposit		103	cat		91
*emit		102	cdr		30
*ept		103	clearbuff		54,61
*exam		103	compress		43
*macro		101	cond	(*)	77
*org		103	conlim	(#)	68
*pack		99	cons		31
*spec		101	copyin		66
:BACK		107	copyout		65
:BANK		107	cos		86
:CKPT		105	cset		46
:CODE		107	csetq	(*)	46
:DATA		109	curchar		54
:END		105	currcol		57
:EXEC		106	date		98
:LISP		106	define		46
:LIST		105	defmac	(*)	47
:LOAD		105	defspec	(*)	47
:OOPS		108	delim	(#)	53
:PEEK		108	difference		84
:RSTR		106	digits	(#)	57
:STOP		108	do	(*)	26
:TIME		107	dtime		98
:UNLIST		105	dump		96
add1		85	eq		95
alist		28	endchar	(#)	56
and	(*)	77	entier		89
append		38	eofcon	(#)	71
aread		60	eq	(+)	31
assoc		40	equal	(+)	31,82
atom	(+)	32,42	erase		45
atsymb		42,92	error		68
attempt	(*)	67	eval		26
axmit		59	exec		94
axmit1		60	explode		43
backsp		55	explode2		44
backtr	(#)	69	ficlose		61

fiprop		64
fierase		64
fiopen		61,63
fipack		65
fiprint		63
fiprintrec		65
firead		64
fireadrec		65
fitoc		65
fixp	(+)	83
flag		75
floatp	(+)	83
funcon	(#)	71
function		50
fuzz	(#)	88
gctime		97
gensym		42
get		74
go	(*)	79
greaterp	(+)	82
grow		98
ifflag		75
index		35
into		33
lambda	(*)	48
lamda	(*)	49
leftshift		88
length		38
lessp	(+)	82
lisp		26
list		38
load		96
log		86
logand		87
logor		87
logxor		88
manifest		104
map		34
mapc		33
match		91
member		32
memory		98
minus		86
minusp	(+)	83
misscon	(#)	71
nconc		37
not		78
nth		40
null	(+)	32
numberp	(+)	81
oblist		27
ondex		35
onto		34
or	(*)	78
pct		94
plength		58
plength2		59
plimit		58
plus		84
power		87
prin1		56
prin2		57
print		56
prog	(*)	79
prop		76
put		74
quote	(*)	26
quotient		84
rand		28
read		52
readch		56
readmac	(#)	53
readrec		56
readstr		92
remainder		85
remob		44
remprop		74
return		80
reverse		39
rplaca		36
rplacd		37
set		46
setcol		55
setq	(*)	46
sin		86
size		90
space		57
srev		90
stack		27
string		42,92
stringp	(+)	90
sub1		85
subst		39
substring		91
swaps		99
terpri		57,62
time		97
times		84
token		52
trash		98
twait		95
type		27
unbreak		73
unflag		75
zerop	(+)	81

Section 3

Maryland LISP Function Packages

Contents

1. Introduction, Explanation, and Acknowledgements 116
2. Arithmetic Package 117
   2.1. Basic Operations 117
   2.2. Basic Predicates 117
   2.3. Other Basic Functions 117
   2.4. Sorting Functions 118
   2.5. Numeric Output Formatting Functions 118
   2.6. Some Number Theory Functions 119
   2.7. The Expression Printer 119
3. Array Facility 121
   3.1. Defining Arrays 121
   3.2. Pointer Arrays 122
   3.3. Typed Arrays 122
   3.4. Internal Arrays 122
   3.5. Using Arrays 123
   3.6. Retrieving Array Specifications 123
   3.7. Implementation 124
4. Autoloader Functions 126
   4.1. Using the Autoloader 126
   4.2. Autoloader Restrictions 126
   4.3. Other Autoloader Applications 127
5. Bignum Package 128
   5.1. Creating Bignums 128
   5.2. Bignum Predicates 129
   5.3. Bignum Arithmetic Functions 129
   5.4. Other Bignum Functions 129
6. LISP Compiler 130
   6.1. Using the Compiler 130
   6.2. Free Variables 131
   6.3. When Free Variables Aren't Really Free 131
   6.4. Linking Between Compiled Code and LISP 132
   6.5. Compiling PROGs 132
   6.6. Compiling Special Forms 133
   6.7. Compiling Macros 133
   6.8. Using GROW 133
   6.9. Manifest 133
   6.10. Excise 133
   6.11. Listing 134
7. Debug Package 135
   7.1. $TRACE 135
   7.2. $BREAK 137
   7.3. $TRACEV 138
   7.4. $UNBUG 139
   7.5. $BRKPT 139
   7.6. DB-LIMIT 140
   7.7. DB-TLEV and DB-BLEV 140
   7.8. DB-ETEXT and DB-LTEXT 140
   7.9. Tracing System Functions and Special Forms 140
   7.10. Usage of DUMP and LOAD 141
8. Dynamic Dumping Package 142
   8.1. Formatted Dumping Functions 142
   8.1.1. Acore, Icore, Ocore, and Pcore 142

8.1.2. Core 143
8.2. Partial Word Examination Functions 143
8.3. The Core Altering Function - Change 144
8.4. General-Purpose Dumping Functions 144
8.4.1. Cdump 144
8.4.2. Addrof 144
8.4.3. Contents 144
8.4.4. Octal 144
8.4.5. Integer 145
8.4.6. Real 145
8.4.7. Cnode 145
8.4.8. Octstr 145
9. Function Editor 146
9.1. Calling the Function Editor 146
9.2. Using the Function Editor 146
9.3. Edit Mode Commands 146
9.4. An Example 148
10. Generator Package 150
10.1. Creating a Generator 150
10.2. Using Generators 151
10.3. The Possibilities List 151
10.4. Example 152
10.5. Compiling Generators 152
11. Library File Functions 153
11.1. Declaring a Library File 153
11.2. Retrieving a Library File Definition 153
11.3. Creating a Library File Definition 153
11.4. Library File Notes 154
12. Mail System 155
12.1. Plugging in to the Mail System 155
12.2. Sending Messages 155
12.3. Checking the Mailbox 156
12.4. Receiving a Message 156
12.4.1. Isolate Next Message - Rnext 156
12.4.2. Go to Nth Message - Rgo 157
12.4.3. List the Current Message - Rlist 157
12.4.4. Delete the Current Message - Rdel 157
12.4.5. Copy Out the Current Message - Rcopy 157
12.5. Finding Out About Other Users 157
13. Matrix Manipulation Package 159
13.1. Matrix Addition - Matadd 159
13.2. Matrix Multiplication - Matmult 159
13.3. Matrix Subtraction - Matsub 159
13.4. Matrix-Scalar Multiplication - Matcmult 159
13.5. Matrix-Scalar Addition - Matcadd 159
13.6. Print a Matrix - Matprt 160
13.7. Copy a Matrix - Matcopy 160
14. Stanford MLISP 161
14.1. Limitations of the Use of $ 161
14.2. Character Set Limitations 161
14.3. Operating Domain 162
14.4. Escape Character 162
14.5. Loading MLISP 162
14.6. Running MLISP 163

14.7. Sample Run 163
14.8. Compilation 164
15. Wisconsin MicroPlanner 166
15.1. Loading PLNR 166
15.2. PLNR Primitives 166
15.3. Abbreviations 172
15.4. PLNR Notes 172
15.5. PLNR Example 173
16. Prettyprinter 175
16.1. Using the Prettyprinter 175
16.2. Dumping 175
16.3. Prettyprinter Notes 176
17. Suspend/Resume Package 178

1. Introduction, Explanation, and Acknowledgements

The LISP programs described in this section were written by various hackers at the University of Maryland and elsewhere for use with Maryland and Wisconsin LISP. They reside in the LISP library file, LISP*LIB., and the specifics of loading and using each program are listed in the various chapters which follow. The procedures for loading the various programs are also detailed in the symbolic element LISP*LIB.LOAD-DOCS, which can be inspected using WED. The text in this element is kept up to date, so in cases where it differs from the accounts given here, the element's instructions are the correct ones.

The prettyprinter, MicroPlanner, compiler, the debugging package, and part of the arithmetic package and their documentation came to the University of Maryland with the original Wisconsin LISP, and the documentation of the first four is reproduced here in basically the original form. The Suspend-Resume functions were written and documented by Mache Creeger. Maryland LISP's version of Stanford MLISP was converted to Maryland LISP by Chuck Rieger and documented by Mache Creeger. The array, bignum, core dumping, generator, mail, and matrix manipulation packages were written and documented by Phil Agre.

2. Arithmetic Package

The Arithmetic Package is a collection of functions useful for playing with numbers in LISP. Maryland LISP itself supports several intrinsic arithmetic functions such as PLUS, POWER, and SIN, and these are documented seperately (see page 81). To use the functions described here, enter LISP and do:

```
@ADD LISP*LIB.ARITH
```

The functions documented here are "+", "-", "*", "/", "**", MOD, "=", GT, LT, GE, LE, NE, EVENP, FLOAT, ABS, OCTAL, MAX, MIN, FACTORIAL, LOGN, RANDOM, TAN, SORT, SORTEDP, FPRINT, IPRINT, OPRINT, XPRINT, PFACTS, PRIMEP, EXPPRT.

2.1. Basic Operations

The Arithmetic Package defines a set of short names for the basic arithmetic operations. These are: "+" for PLUS, "-" for MINUS, "*" for TIMES, "/" for QUOTIENT, "**" for POWER, and MOD for REMAINDER. Either these or the long names may be used once the Arithmetic Package has been loaded.

2.2. Basic Predicates

A set of basic predicates is defined in the Arithmetic Package, each with the obvious meaning: "=", GT, LT, GE, LE, NE, EVENP.

2.3. Other Basic Functions

A few other basic functions are defined by the Arithmetic Package.

The FLOAT function converts its argument to floating-point.

The ABS function returns the absolute value of its argument.

The OCTAL function returns the octal representation of its argument.

The MAX function takes any number of arguments and returns the largest of them.

The MIN function takes any number of arguments and returns the smallest of them.

The FACTORIAL function computes the factorial of its integer argument.

Tne LOGN function takes the logarithm of its first argument to the base of its second argument, returning a real number.

The RANDOM function of no arguments returns a randomly generated floating point number in the range [0,1).

The TAN function returns the tangent of its argument, which should be in radian measure. It uses the intrinsic LISP functions SIN and COS.

2.4. Sorting Functions

The Arithmetic Package provides two functions which are useful for sorting, SORT and SORTEDP. Each takes as its arguments a list L of LISP objects and an ordering predicate F. Before describing SORTEDP and SORT, it is necessary to place some restrictions on the nature of F. The function F must: 1) take two arguments; 2) accept as arguments any two members of the list L in either order; 3) always return either T or NIL; and 4) be transitive, ie., if F(x,y)=T and F(y,z)=T, then F(x,z)=T, if x, y, and z are any three members of L.

The function SORT returns a list (l1 l2 ... ln) where each li is in L, and F(li,lj)=T if i<j. If for some x and y in L, both F(x,y) and F(y,x) are NIL, then only one of them will be in the resulting list. Thus SORT can be made to remove duplicates by specifying F such that F(x,x)=NIL for all x in L. If F has the default value of LE, SORT will return a sorted permutation of L in non-decreasing order without removing duplicates. If, for example, F is GREATERP, SORT will return a version of L in decreasing order without duplicates, so that (SORT '(1 3 5 2 3 4 2) GREATERP) = (5 4 3 2 1).

The SORT function can accommodate arbitrarily complex sorting tasks, depending on the sorting predicate F. For example, if L is a list of atomic symbols, each of which has a number under property list indicator IND, (SORT L (LAMBDA (X Y) (LE (GET X 'IND) (GET Y 'IND)))) will return a version of L in which the atoms are reordered so that the values of (GET atom 'IND) are non-decreasing.

(SORTEDP L F) returns F if (SORT L F) would return a result EQUAL to L.

2.5. Numeric Output Formatting Functions

The four functions described here allow the user to edit numeric output into fields of specific sizes.

(FPRINT <real #> <field width> <# dec. places>) edits the floating-point number <real #> into <field width> output positions including <# dec. places> decimal places and a decimal

point. If the field specified is too small, the output will overflow the field. The user should remember to make room for the decimal point.

(IPRINT <int> <width>) edits the integer <int> into the next <width> columns on output, right-justified and blank-filled to the left.

(OPRINT <oct> <width>) edits the octal number <oct> into the next <width> columns on output, right-justified and zero-filled to the left.

(XPRINT <obj> <rep>) PRIN1's the object <obj> <rep> times.

2.6. Some Number Theory Functions

(PFACTS <n>) returns the prime factors of <n> in ascending order. For example, (PFACTS 24) = (2 2 2 3), (PFACTS 0) = (0), (PFACTS 1) = NIL, (PFACTS -3) = (-1 3), and (after a great while) (PFACTS 333667) = (333667). The algorithm used by PFACTS is necessarily very slow for large integers.

(PRIMEP <n>) returns T if <n> is a prime number and NIL otherwise. Note that no integer less than 2 can be prime.

2.7. The Expression Printer

For many purposes, LISP's Polish notation for arithmetic expressions is inconvenient. Because of this, there is the LISP Expression Printer which, because of its size, does not come with the Arithmetic Package but can be loaded by doing:

```
@ADD LISP*LIB.EXPPRT
```

To print out an expression <exp>, call EXPPRT, ie., (EXPPRT <exp>). It will PRIN1 out in infix notation the expression <exp> without (TERPRI)'ing. It comes with a set of pre-defined symbols which may be inspected by looking at the binding of the atomic symbol FALIST. This is an association list which associates to each function name an infix symbol in string form, eg., (PLUS . "+"). A list of the function names recognized is in FLIST. The Expression Printer also recognizes operator precedences, and these are stored as an association list bound to the atomic symbol PRECS. Also recognized are symbols which come before (eg., unary minus) and after (eg., factorial) their arguments. These function names are listed in UNOPS and POSTOPS, respectively, and are associated with their infix symbols in UNALIST and POSTALIST, respectively. The user may freely change these lists to have the Expression Printer recognize any set of symbols. Any function name which the Expression Printer does not recognize is printed in the standard "f(x,y,z)" format along with

its arguments.

For example, given the default function lists,

```
(EXPPRT '(PLUS (F (MINUS X) (FACTORIAL N)) (TIMES X Y)))
```

prints

```
F(-X,N!)+X*Y
```

and

```
(EXPPRT '(AND (GT X 4) (LE (+ X Y (- Z 1)) (/ 4 N))))
```

prints

```
X>4&X+Y+(Z-1)<=4/N
```

3. Array Facility

The Maryland LISP Array facility allows LISP users to define multi-dimensional, (pseudo-)sequential-access arrays of LISP data structures. Each array is defined as a function which will retrieve from or change words in the area of core assigned for it. Thus, features such as subscript checking can be assigned implicitly to each array when it is created rather than explicitly each time it is used. There are three kinds of arrays, pointer, typed, and internal, and these are all defined below.

To use the Array facility, enter LISP and do:

```
@ADD LISP*LIB.ARRAY
```

The functions documented here are ARRAY, ARRAY1, TARRAY, TARRAY1, IARRAY, IARRAY1, ARRALIST, NUMSUBS, SUBMAXS, MAXSUB, ARRTYPE, ARRFUNC, ARRCHECKP, ARRAREA, ARRNWDS, ARRGET, and ARRPUT.

3.1. Defining Arrays

The formats for defining the various kinds of arrays are:

```
pointer: (ARRAY[1] <dim list> [<init val>])
typed: (TARRAY[1] <dim list> <type num> [<init val>])
internal: (IARRAY[1] <dim list> <type num> [<init val>])
```

where [...] delimits optional constructs. In each case the returned value is the function which will be used to access the array. If ARRAY1 or TARRAY1 or IARRAY1 is used, this function will check the legality of its arguments. The token <dim list> stands for a list of non-negative integers which are to serve as the maxima of the various dimensions of the array (1 is always the lowest value of each dimension). If the array is to be one-dimensional, then <dim list> can be a single integer. In the cases where it is given, <type num> stands for an integer which is the type number for the type of the array being defined:

code	type
-1	pointer (ARRAY or ARRAY1 will be called)
0	cons node
1	integer
2	octal
3	real

Attempting to specify an illegal type number in a call on TARRAY or TARRAY1 results in an error message and an (ERROR 3) (see page 67). In each form, an optional last argument is <init val>,

which, if given, will cause each entry in the array space being reserved to be initialized to that value. Otherwise, the initial value will be "garbage".

3.2. Pointer Arrays

The pointer array is the most general and, consequently, the least space-efficient form of array. In the pointer format, each word in an array space contains a single pointer to another LISP object of any type. When the array is accessed, this pointer is returned as the value, and when a new value is installed, the pointer is replaced by a pointer to the new value. This way, one can have arrays of linker nodes or strings or atomic symbols or mixed arrays pointing at objects of different types.

3.3. Typed Arrays

Sometimes when all the elements of an array are of the same type (especially when dealing with numbers), it is somewhat inefficient to have the array entries stored as pointers into the regular data area, thus taking up twice as much space as necessary. For this reason, typed arrays are available. In the typed format, the array entries (presently limited to cons nodes and numbers) are physically stored in the array space in the conventional manner. When accessed, the array function creates a new node of the proper type, copying out the contents of the appropriate array space entry and returning a properly typed node. Similarly, when a value is being changed in a typed array, the contents of the node containing the new value are copied out into the proper array space entry. This way, as far as the user can tell, typed arrays are indistinguishable in use from pointer arrays.

3.4. Internal Arrays

Sometimes when using typed arrays, it is inefficient to create a new node every time the array is accessed. For this reason, there are internal arrays. Internal arrays are the same as typed arrays, except when accessing the array, what is returned is a pointer to the proper array space entry rather than at a node containing its contents. Internal arrays have several drawbacks which restrict the situations in which they may be used. First, the value returned from the array has no type (actually it is of type 9, which for most purposes is the same idea), so that any routines which type-check their arguments (most LISP internal routines do not) will reject them. Second, since the value returned by the array function is a pointer into array space, should a node in the array space be changed by the array function, it will have the side-effect of changing all pointers to that node. Also, since the value is just a pointer to a type-less object, there is no way of determining what the

type of an object is just by looking at it, so that most LISP routines will interpret pointers at internal real or cons node array space entries as fixed-point numbers. For these reasons, it is usually not safe to use internal arrays of real numbers or cons nodes. However, for many purposes involving numbers, internal arrays can be made to work if the user provides his own type-conversions when necessary.

3.5. Using Arrays

As an illustration of array usage, suppose that an array ARR has been defined by one of the six array-defining functions listed above, say,

```
(CSETQ ARR (ARRAY1 '(2 3) NIL))
```

Then, (ARR '(1 1)) would return NIL, the first entry in the first row of the array defined by ARR. The call (ARR '(2 1) '(A . B)) would cause a pointer to the structure (A . B) to be placed in the first entry in the second row of ARR, and a subsequent (ARR '(2 1)) would return (A . B) as its value. (Any call which changes an entry in an array returns the new value.) Since ARRAY1 has been used here to define ARR, subscript checking is performed on all uses of ARR. For example, (ARR '(1 4)), (ARR '(3 2) 'XX), (ARR '(1)), (ARR '(0 3)), (ARR '(1.0 1) (PLUS 4 X)), and (ARR (LIST 'A 2)) would all result in error messages and (ERROR 1)'s.

The array ARR can be passed as an argument to a function, and the local variable to which it becomes bound can be used to access and change the array. For example, ((LAMBDA (A) (A '(2 3) '(TWO THREE))) ARR) would have the same effect as (ARR '(2 3) '(TWO THREE)). Note that for a one-dimensional array, the first argument need not be in a list, eg., (ARR1 37 '(C D)).

The other kinds of arrays, typed and internal, are used the same way, the only differences being in what they return.

3.6. Retrieving Array Specifications

Each array function carries with it information about the way it was defined, and this information can be retrieved in cases where a routine works with several kinds of arrays and needs to discriminate among them. The information is stored in an association list in the function definition. This list and the specific items of information on it can be retrieved by the following functions:

1. (ARRALIST <array>) returns the entire list of information for array function <array>.

2. (NUMSUBS <array>) returns the number of dimensions that

were defined for <array>.

3. (SUBMAXS <array>) returns the <dim list> (dimension maxima list) which was specified when <array> was defined.

4. (MAXSUB <array> <dim num>) returns the <dim num>th entry in the <dim list> dimension maxima list for <array>.

5. (ARRTYPE <array>) returns the numeric type of the entries in <array>, where -1 is the code for pointers, 0 is for cons nodes, 1 is for integers, 2 is for octals, and 3 is for reals.

6. (ARRFUNC <array>) returns the name of the function which defined <array>, that is, one of ARRAY, ARRAY1, TARRAY, TARRAY1, IARRAY, or IARRAY1.

7. (ARRCHECKP <array>) returns T if subscript checking was specified for <array>, and NIL otherwise.

8. (ARRAREA <array>) returns a list of those (type 9) buffer pages where <array>'s entries are stored. See the notes on implementation details below.

9. (ARRNWDS <array>) returns the total number of entries in <array>, which can be found by multiplying together all the numbers in the <dim list> which was specified when <array> was defined.

Since the information listed above must always be accessible, the user should not put a trace on any array function. Since it often is necessary to trace array accesses and changes, though, there are two functions which provide alternate ways of accessing and changing arrays, and these can be traced. The function ARRGET takes two arguments, an array function and a list of subscripts. It simply passes the subscripts along to the array and returns the result of the accessing operation. The function ARRPUT takes three arguments, an array function, a list of subscripts, and a value of an appropriate type, and sends them along to the array, making the desired change.

3.7. Implementation

Maryland LISP provides as one of its data types a 128-word sequential area known as a buffer page. Arrays are laid out on these pages by a first-fit algorithm which allocates NWDS/128 full pages and NWDS.mod.128 words on a broken page for an array requiring NWDS words. Individual entries are located by converting an n-dimension subscript list into a single integer I, (see [Knu68], Section 2.2.6, Page 296 on "Sequential Allocation") and computing the address of the (((I-1).mod.128)+1)th word on

the (((I-1)/128)+1)th page which was allocated as described above.

Array space is garbage collected a page at a time when all arrays sharing a page have disappeared. The objects to which the pages themselves point are marked by the garbage collector for pointer and type 0 (cons node) array pages, but not for pages dedicated to numeric arrays.

Because of the difficulties involved in introducing buffers to the design of Maryland LISP, it is at present impossible to use LOAD and DUMP on arrays. Also, arrays may not be compiled.

4. Autoloader Functions

The Autoloader is a program which allows the user to delay loading the definition of a function, value of an atomic symbol, or entry on a property list until it is actually needed by EVAL, and then to do it automatically. To use the Autoloader, enter LISP and do:

```
@ADD LISP*LIB.AUTOLOAD
```

It is also necessary to load the Library File routines (see page 153).

Most of the LISP support routines are in the file LISP*AUTOLIB., so that to use them, one can do:

```
:EXEC @USE AUTO.,LISP*AUTOLIB.
(SETFILE 'AUTO)
```

The rules for using the Autoloader on each of the various function packages are contained in the element LISP*LIB.LOAD-DOCS, which may be inspected using @ED.

4.1. Using the Autoloader

Once a library file has been established, it is possible to proceed as though everything in the file has been loaded. Whenever an atomic symbol is found to be unbound or a property list indicator is not found when asked for, LISP contingency routines will give control to the Autoloader, which will search the table of contents of the library file for the appropriate entry. If it is found, the image in the appropriate record will be sent to EVAL, and control will return to the user program. Otherwise, the traditional "UNBOUND X, HELP:" message will appear.

In order to put an entire list of definitions into a library file, one may call (AUTOFILE <defns>), where <defns> is a list of s-expressions. Those which AUTOFILE recognizes as being legal value definitions will be put in the library file, and the rest will be returned as AUTOFILE's value. These definitions must be loaded manually, via @ADD or similar means. Given the result from AUTOFILE and a sequential access file descriptor (see page 60), the function GEN-LOADFILE can print out the extra definitions into the file. By using the Text Editor's BEGIN and COPY commands, one can transfer this text into a file or element which must be @ADD'ed before the Autoloader can load any other of the definitions for that collection of functions.

4.2. Autoloader Restrictions

Several kinds of things may not be Autoloaded:

1) Special forms or macros; a special form or macro linker node returned by LISP's contingency routines will be treated as a regular function,

2) Unprintable objects such as linker nodes, compiled or system code, or buffer pages which are not constantly bound to any atomic symbol other than the one being dumped,

3) Redefinitions of intrinsic LISP functions; an intrinsic LISP function's atomic symbol will not normally be unbound, so that it will never trigger the Autoloader mechanism,

4) Objects whose definitions must occur before or after those of other Autoloaded functions because of, say, redefinitions of functions.

There are some other restrictions on Autoloader use. An atomic symbol will only have its value Autoloaded if it is EVALuated and found to be unbound. Since the Prettyprinter, for example, uses LISP intrinsic functions such as *DEF to get at the value of an atomic symbol, an unloaded atom will appear unbound to the Prettyprinter. Fluid bindings (via SET or SETQ) may not be Autoloaded. Other LISP hacks not explicitly said to be legal in this manual will probably foul up the Autoloader also.

Note that the Autoloader redefines the GET function to capture non-existent property list entries and defines a BINDCON contingency to capture unbound atoms.

4.3. Other Autoloader Applications

Compiled code can be Autoloaded by using the PUTDEFN function, for example,

```
(PUTDEFN 'FOOBAR '(LOAD '(MYFILE . FOOBAR)))
```

where FOOBAR has been compiled and DUMPed to element FOOBAR in a program file with internal name MYFILE.

5. Bignum Package

The Bignum Package allows the user to do arithmetic with integers of arbitrary magnitudes (up through several thousand digits) and bases (up through 16). To load it into LISP, load the Arithmetic and Array packages and do:

```
@ADD LISP*LIB.BIGNUM
```

5.1. Creating Bignums

There are two ways to create a bignum, literally and through the BIGCONV function. A literal bignum has the following form, which is interpreted by the code assigned as a readmacro for the "^" character:

```
<bignum> -> <based-bignum> | <base10-bignum>
<based-bignum> -> ^$<base><sign><digits>
<base10-bignum> -> ^<opt-sign><digits>
<opt-sign> -> <sign> |
<sign> -> + | -
<base> -> an integer in {2,...,16}
<digits> -> 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |
            A | B | C | D | E | F
```

where, of course, each digit must be less than the base of the bignum. The following are examples of literal bignums:

```
^1098476 is a positive base 10 integer
^$2-1100101000 is a negative base 2 integer
^$13+A4099C18735 is a positive base 13 integer
^0 = ^+0 = ^-0 = ^$10+0 = ^$10-0 = zero in base 10
```

Note that there is no "negative zero" bignum; all zeros are given positive sign by the bignum creation routine and by the various functions which return bignum results.

The second way to create a bignum is through the BIGCONV function, which takes as its arguments a LISP integer and an integer base and bignum version of the first argument to the base of the second. The inverse of BIGCONV is BIGINT, which, given a bignum less than 2^35, returns an integer version of it (it returns the argument otherwise).

Generally speaking, bignums of larger bases are more efficient, because they take (slightly) less room, and because the algorithms used to manipulate bignums do digit-by-digit operations. The only way to convert between bases is to use BIGINT to convert to an integer, and then to use BIGCONV with the appropriate base to convert back. There is no easy way to change the base of a very large bignum.

Those bignum functions which take more than one bignum argument generally require that they be of the same base, except as noted.

5.2. Bignum Predicates

These functions take bignum arguments and return T or NIL.

BIGEQUAL(A,B)	Returns T iff A=B
BIGLESSP(A,B)	Returns T iff A<B
BIGZEROP(B)	Returns T iff B=0
BIGPOSP(B)	Returns T iff B>0 or B=0
BIGNEGP(B)	Returns T iff B<0

5.3. Bignum Arithmetic Functions

These functions each take two bignum arguments of the same base and return bignums of that base:

BIGPLUS(A,B)	Returns A+B
BIGDIFF(A,B)	Returns A-B
BIGMULT(A,B)	Returns A*B
BIGDIVIDE(A,B)	Returns (A/B . A.mod.B)

The algorithm for BIGDIVIDE is very slow for large arguments. For a discussion of the integer division algorithm, see [Knu69], pages 235-240.

There is also a function, BIGEXP, which takes two bignum arguments of any bases and returns the first raised to the power of the second. The base of BIGEXP's result is the base of its first argument. This cost of this operation grows rather rapidly with the magnitude of the first argument and the logarithm of the second argument.

5.4. Other Bignum Functions

The BIGPRINT function takes one bignum argument and prints it out in the literal notation without the "^" prefix, and without doing a (TERPRI). It returns its argument.

The BIGSIZE function takes a bignum argument and returns the number of digits in its representation (ie., in its base).

The BIGSIGNGET function takes a bignum argument and returns 1 if it is zero or positive, and -1 if it is negative.

The BIGBASE function takes a bignum argument and returns its base as a LISP integer.

6. LISP Compiler

The Maryland LISP system normally acts as an interpreter that reads expressions and evaluates them. However, there are often times when one has already defined and debugged some functions and would like them to run more efficiently. For this purpose the Maryland LISP Compiler is used. The compiler uses the definition of the function to generate machine code. This machine code is placed directly in core and thereafter used whenever the function is applied, just as if it were a hand-coded, system-defined function. It must be emphasized that the compiler is designed to be used only after a function has been debugged with the interpreter. The compiler gives almost no diagnostics and the code generated for a badly defined function will crap out in exotic and wonderful ways.

The functions documented here are COMPILE, FLUID, UNFLUID, and EXCISE.

6.1. Using the Compiler

The compiler is loaded by typing (in LISP):

```
@ADD LISP*LIB.COMPILER   or,

:EXEC @USE LISP.,LISP*LIB.
(LOAD '(LISP . COMPILER))
```

Before anything may be compiled, it is necessary to use the :CODE directive to allocate a contiguous chunk of memory for the compiled code to reside on. See page 107 for details.

The major function in the compiler is COMPILE. To cause a group of functions to be compiled, say:

```
(COMPILE L)
```

where L evaluates to a list of designators, each of which has one of the following forms:

FNN	Compile the function, special form or macro constantly bound to the atomic symbol FNN.
(I A1 .. An)	For each of the atomic symbols A1 .. An, compile the function on its property list indicated by I.

(Compare the form of the argument to PRETTYP.)

For example:

```
(COMPILE '(F1 F2 (IND AT1 AT2)))
```

or:

```
(CSETQ CLIST '(F1 F2 (IND AT1 AT2)))
(COMPILE CLIST)
```

COMPILE will then generate machine code for each function indicated and place the code directly into core.

6.2. Free Variables

Normally, this is all that needs to be done in order to use the compiler. However, there is one situation where the compiler needs some additional information - when free variables are used. A free variable is an atomic symbol used in a function which neither has a global binding nor appears as a LAMBDA or PROG variable in the function. The compiler must know whether these atomic symbols are actually constants (ie., globally bound atoms) which have not yet been defined, or are fluid variables which should be handled through the association list mechanism. By convention, the compiler assumes that all free atomic symbols will have a constant binding at run-time, unless they have been explicitly declared as being fluid via the function FLUID:

```
(FLUID '(V1 V2 ...))
```

Any variable which is used free in a function to be compiled, or which appears as a LAMBDA or PROG variable in a function to be compiled but is referenced as a free variable in some lower-level function, must be declared fluid before compilation. After compiling all functions which either bind or reference a free variable, it may be declared unfluid if desired:

```
(UNFLUID '(V1 V2 ...))
```

This allows subsequent functions in which the variable appears as a normal local variable to be compiled in an optimal manner.

As an aid to discovering free occurrences of variables, the value returned by COMPILE is a list of all atomic symbols which were used free and had not been declared fluid (the compiler assumes they are constants). If any are in fact not constants, you must declare them fluid, restore the original function definitions, and recompile.

6.3. When Free Variables Aren't Really Free

There are several cases where variables which appear free in a LAMBDA-expression needn't be declared fluid, i.e., the compiler is smart enough to reference them directly. The necessary conditions are:

(1) The LAMBDA-expression appears in the first position of an expression, i.e., ((LAMBDA ...) ...), or as an argument to one of the system functions MAPC, MAP, PROP or OBLIST; and

(2) The referenced variable is bound in the function being compiled (in a higher-level LAMBDA or PROG).

Thus when compiling:

```
(LAMBDA (X L) ... (MAPC L (LAMBDA (Y) ... X ...)))
```

X needn't be declared fluid even though it appears free in the second LAMBDA-expression.

6.4. Linking Between Compiled Code and LISP

There is another somewhat more subtle situation, involving calls from a compiled function to the LISP interpreter, in which variables must be declared fluid. Normal variables are assigned a location on the stack and addressed directly by compiled code. The interpreter, however, accesses variables by looking them up on the association list. Thus, any variables which are passed unevaluated to the interpreter must appear on the association list, i.e., they must have been declared fluid at compile-time.

There are two ways by which a compiled function can cause the interpreter to be entered (fortunately both rather rare). The first involves explicit calls to EVAL. For example, if V is a variable, (EVAL (LIST FNN 'V)) will cause V to be passed unevaluated to the interpreter, so that it must have been declared fluid. The second way to enter the interpreter is to do a SET (SETQ is handled automatically by the compiler). for example, if V is a variable and L gets bound to the list (V ...), the expression (SET (CAR L) ...) will cause V to be passed to the interpreter.

To reiterate, any usage of EVAL or SET within a function to be compiled should be examined carefully to see if any variables might be sent unevaluated to the interpreter.

6.5. Compiling PROGs

Special care must be taken when compiling functions involving calls on PROG. If X is a label in a PROG to be compiled, (GO X) may not be done outside of the PROG. Also, the s-expression (GO X) may not be sent to EVAL. In each case, compilation will have reduced X to a hard-coded address which cannot be found by the GO function, and GO will unwind the stack trying to find a PROG with a label X in it.

Also, at present, if a call on RETURN is to be compiled, it must have an argument.

6.6. Compiling Special Forms

Special forms may be compiled just as normal functions. Care must be taken, however, if the special form contains any calls to EVAL (see above). It is possible for function A to pass an unevaluated variable to special form B, which then sends it on to EVAL. In such a case, the variable must be declared fluid before compiling function A.

6.7. Compiling Macros

The compiler handles macros by expanding them (at compile-time) and compiling the resulting expression. Thus the macro itself need not be compiled (although if it is frequently used it may be). After compiling all functions which call a particular macro, the macro may be discarded - it is no longer needed at run-time. Note that this procedure works for normal macros - it does not work for macros which expand as a function of the run-time environment, or which have any run-time side-effects (e.g., changing a constant binding or printing anything). Such macros should be written as special forms.

6.8. Using GROW

When not using the V option, it is recommended (though not necessary) that functions to be compiled be loaded before executing a GROW of more than 55 (67Q). This allows the compiled code to access addresses directly rather than through index registers. After reading in all functions to be compiled, core may be expanded beyond this limit if desired. It is imperative, however, that compiled code which has been DUMPed be subsequently LOADed before executing a GROW of more than 55. (see also page 98)

6.9. Manifest

The expression (MANIFEST X) is equivalent to X when using the interpreter. However, using this expression in a function to be compiled tells the compiler that the value of X can be computed at compile-time, so that the compiler will evaluate it then instead of generating the code to evaluate it. [Note: Some expressions are automatically EVALed at compile-time. Thus (QUOTE X) is actually equivalent to writing (MANIFEST (QUOTE X))].

6.10. Excise

The normal use of the compiler is to compile some functions and then save them using DUMP. However, there are several flags

and things left hanging around by the compiler which will be DUMPed also, and hence will be with you forever after. To purge these unwanted items, type (EXCISE) when you are through compiling to remove the compiler from core.

6.11. Listing

The compiler works by generating a pseudo-machine code before generating the actual UNIVAC code. In certain cases it is helpful (and interesting) to look at this code to see just what the compiler thinks of your function (especially useful for debugging the compiler!). To have this code printed out on the terminal or line printer, include a second argument to COMPILE, e.g.:

```
(COMPILE CLIST T)
```

The intermediate code for each function will then be listed as it is compiled. This code is for the most part self-explanatory. If it's not, you probably shouldn't be printing it out anyway.

7. Debug Package

The LISP Debug Package provides the user with a powerful set of functions for monitoring the execution of his program. It is loaded by typing (to LISP):

```
@ADD LISP*LIB.DEBUG   or,

:EXEC @USE LISP.,LISP*LIB.
(LOAD '(LISP . DEBUG))
```

The major functions in the Debug Package are $TRACE, $BREAK, $TRACEV and $UNBUG. The auxillary function $BRKPT is also available for use if desired. Each of these functions is described in detail below.

7.1. $TRACE

$TRACE allows the user to trace conditionally the entry to and/or exit from functions. It is designed to be simple enough for use in most normal trace situations, while still providing a capability for complex conditional monitoring of function execution. This flexibility is attained by providing default values for the various arguments. The complete form of the calling sequence is:

```
($TRACE FNS E1 L1 E2 L2)
```

where FNS EVALs to either an atomic symbol to which a function, special form or macro is constantly bound (it may be either compiled or uncompiled), or to a list of such atomic symbols; E1 and E2 EVAL to LISP expressions; and L1 and L2 evaluate to lists of expressions.

Each time one of the functions indicated by FNS is entered, E1 will be EVALed. If it EVALs to NIL, EVALuation proceeds as usual. Otherwise each of the expressions in L1 is EVALed and printed before entering the function. When evaluation is complete, E2 is EVALed and, if non-NIL, the expressions in L2 are EVALed and printed. The function is then exited as usual.

E1, E2 and the expressions in L1 and L2 may reference the arguments passed to the function via the variable $ARGS and expressions ($ 1), ($ 2), ..., where $ARGS is bound to the entire argument list and ($ n) returns the nth argument (if n is too large, the atomic symbol **UNDEF** is returned). When tracing uncompiled functions the arguments may also be referenced by the names of the corresponding formal parameters. Thus you may reference the two arguments of (LAMBDA (L V) ...) by ($ 1) and ($ 2), or by L and V.

E2 and the expressions in L2 may reference the value

returned by the function via the variable $VAL. This gives the user the capability of conditionally tracing the exit from a function depending on its value.

Since in most cases one doesn't need all of the power of the complete trace routine, any of the arguments E1, L1, E2, or L2 may be omitted, with various default interpretations (of course, if an argument is present all arguments preceding it must also be present). The default interpretations are:

ARGUMENT	DEFAULT VALUE	
E1	T	(TRACE UNCONDITIONALLY)
L1	T	(PRINT ARGUMENTS TO FUNCTION)
E2	E1	(TRACE EXIT ONLY WHEN TRACING ENTRY)
L2	'($VAL)	(PRINT VALUE RETURNED BY FUNCTION)

Following are several examples of the use of $TRACE:

(1) (STRACE '(F1 F2 ...))
causes each of the functions F1, F2, ... to be traced unconditionally, with the arguments printed upon entry and the value upon exit.

(2) (STRACE 'FUNC '(NULL ($ 1)))
causes FUNC to be traced only when its first argument is NIL.

(3) (STRACE 'FUNC T '(($ 1) (LENGTH ($ 2))))
causes FUNC to be unconditionally traced, with its first argument and the length of its second argument printed upon entry, and its value printed upon exit.

(4) (STRACE 'FUNC T T F)
causes FUNC to be unconditionally traced upon entry only, with its arguments printed.

(5) (STRACE TRFNS T NIL T '(TREE))
causes each of the functions in the list TRFNS to be unconditionally traced, with no expressions printed upon entry and the value of the atomic symbol TREE printed upon exit.

As an example of the trace format, consider the following:

```
(CSETQ LENGTH
(LAMBDA (L)
        (COND [<NULL L> 0]
              [T <ADD1 (LENGTH (CDR L))>])))

(STRACE 'LENGTH)
```

```
(LENGTH '(A B))

>ENTERING LENGTH [0]
 L: (A B)
    >ENTERING LENGTH [1]
     L: (B)
        >ENTERING LENGTH [2]
         L: NIL
        <LEAVING LENGTH [2]
         $VAL: 0
    <LEAVING LENGTH [1]
     $VAL: 1
<LEAVING LENGTH [0]
 $VAL: 2
```

The level numbers (e.g., [0]) are printed to make it easier to match entering/leaving pairs in deeply nested functions.

When a function is traced or broken, the trace routine is attached to the atomic symbol rather than to the function definition itself. Therefore, in the following case, Y and Z will be traced, but X and W will not:

```
(CSETQ X (LAMBDA (A B C) . . .))
(CSETQ Y X)
($TRACEV '(Y))
(CSETQ Z Y)
(CSETQ W Z)
($UNBUG '(W))
```

7.2. $BREAK

$BREAK allows the user to conditionally interrupt the execution of a function as it is entered and/or exited. It is similar to $TRACE, except that instead of printing the values of a given set of expressions, a READ-EVAL-PRINT loop is entered to give the user an opportunity to interrogate the status of his program. Note that a function may be either traced or broken, but not both at the same time. The complete form of the calling sequence is:

```
($BREAK FNS E1 E2)
```

where FNS EVALs to either an atomic symbol to which a function, special form or macro is constantly bound (it may be either compiled or uncompiled), or to a list of such atomic symbols; and E1 and E2 EVAL to LISP expressions.

Each time one of the functions indicated by FNS is entered, E1 will be EVALed. If it EVALuates to NIL, evaluation proceeds as usual. Otherwise a READ-EVAL-PRINT loop is entered - the user may type in expressions which will be EVALed and printed. To exit from the loop, type T. Evaluation of the function will then

continue. When evaluation is complete, E2 is EVALed and, if non-NIL, another READ-EVAL-PRINT loop is entered. Again, type T to exit. The function will then be exited as usual.

As in $TRACE, the expressions E1 and E2 (and expressions typed in by the user) may reference the arguments passed to the function and its result via $ARGS, ($ 1), ($ 2), ... and $VAL. Formal parameter names may also be used when breaking uncompiled functions.

E1 and E2 may be omitted when calling $BREAK, with the following default interpretations:

ARGUMENT	DEFAULT VALUE	
E1	T	(BREAK UNCONDITIONALLY)
E2	E1	(BREAK EXIT ONLY WHEN BREAKING ENTRY)

Following are several examples of the use of $BREAK:

(1) ($BREAK '(F1 F2 ...))
causes each of the functions F1, F2, ... to be broken unconditionally on both entry and exit.

(2) ($BREAK 'FUNC '($ 2))
causes FUNC to be broken only if its second argument is non-NIL.

(3) ($BREAK BRFNS T F)
causes each of the functions in the list BRFNS to be broken unconcitionally upon entry, with no breaking upon exit.

7.3. $TRACEV

$TRACEV allows the user to monitor the values assigned to specified atomic symbols via SET, CSET, SETQ or CSETQ. There are two calling sequences to $TRACEV:

($TRACEV ATS) and ($TRACEV ATS E)

where ATS EVALs to either an atomic symbol or a list of atomic symbols, and E EVALs to an expression. When the second argument is not present, a message of the form "*CSETQ (name): (val)" is printed each time the binding of an atomic symbol indicated by ATS is changed via SET, CSET, SETQ or CSETQ. Note that this does not apply to other means of changing bindings (e.g., PROG variable initialization), nor to bindings changed by SETQ or CSETQ within compiled code.

When the second argument to $TRACEV is present, the above message is printed only when the expression E EVALuates to

non-NIL. In addition, a READ-EVAL-PRINT loop is entered to allow the user to examine the state of his program at a point just before the value of the atomic symbol is changed. Type T to exit from the loop. The new value about to be assigned to the atomic symbol is bound to $VAL, and may be referenced by E if desired. For example, the following will cause a READ-EVAL-PRINT loop to be entered each time one of the indicated atomic symbols is set to NIL:

```
($TRACEV '(A1 A2 ...) '(NULL $VAL))
```

Note also that ($TRACEV 'AT T) may be used to cause a READ-EVAL-PRINT loop to be entered each time the binding of AT is changed.

7.4. $UNBUG

Tracing, breaking or value-tracing may be turned off via $UNBUG. The calling sequence is:

```
($UNBUG ATS)
```

where ATS EVALs to either an atomic symbol or a list of atomic symbols. Each atomic symbol indicated by ATS will have all tracing, breaking or value-tracing removed. If an atomic symbol appears which is not currently being traced, broken or value-traced, it is ignored.

It is occasionally useful to turn off all debugging without having to type in a (possibly long) list of atomic symbols. This may be accomplished by typing:

```
($UNBUG)
```

Note that $UNBUG must be called before a traced or broken function may be edited or prettyprinted. Also, you may trace a broken function, break a traced function, or change the trace or break conditions of a function by calling $TRACE or $BREAK without bothering to call $UNBUG first.

Note that if a traced function is redefined, it will not still be traced.

7.5. $BRKPT

$BREAK and $TRACEV both use this function when a READ-EVAL-PRINT loop is required. It may also be used in user programs to establish breakpoints at interesting locations.

```
($BRKPT X)
```

will cause X to be printed and a READ-EVAL-PRINT loop to be

entered. Expressions will be read, EVALed and printed until a T is read, at which time $BRKPT will exit with a value of T. For example,

```
(COND [BRFLAG <$BRKPT "BRKPT 295:">])
```

will cause a READ-EVAL-PRINT loop to be entered whenever BRFLAG is non-NIL.

7.6. DB-LIMIT

The print depth and length limits used by the Debug Package are bound to the atomic symbol DB-LIMIT, and may be changed if desired. The initial value of DB-LIMIT is '(3 . 5) (i.e., a depth of 3 and a length of 5).

7.7. DB-TLEV and DB-BLEV

Both $TRACE and $BREAK maintain level counters which are printed to show the current depth of nested function calls. $TRACE also uses its counter to control the number of spaces to indent before printing. If a function exits abnormally, these counters may get screwed up. If this happens, CSETQ the atomic symbols DB-ELEV and DB-BLEV to 0.

7.8. DB-ETEXT and DB-LTEXT

When tracing on the teletype one might like to cut down on the length of the message printed out when a function is entered or exited. This may be accomplished by changing the constant bindings of the atomic symbols DB-ETEXT and DB-LTEXT. For example, (CSETQ DB-ETEXT "+") will cause ">ENTERING FUNC [0]" to be changed to "+FUNC [0]".

7.9. Tracing System Functions and Special Forms

Reason tells us that if we wish to trace or break a system function, and that function is itself used within the Debug Package, there is a distinct possibility of an infinite loop being entered (horrors!). Since it is on occasion desirable to trace certain system functions, the Debug Package has been compiled in such a way that any system function or special form may be traced or broken when using the compiled version of the Debug Package (This heroic feat was brought about through some rather sneaky uses of the MANIFEST function). Thus, if you wish to trace or break any system functions or special forms, you should use LOAD to load the Debug Package rather than @ADD.

It should also be noted that certain system functions (and all system special forms) are compiled in-line by the compiler,

so that they will not be traced or broken when they appear in compiled code.

Note further that $TRACEV works by BREAKing SET, CSET, SETQ and CSETQ, so that if you are doing any value-tracing you must not trace or break these functions yourself.

7.10. Usage of DUMP and LOAD

It is good practice to turn off all debugging (e.g. via (SUNBUG)) before doing a DUMP, since otherwise you may have problems when you later LOAD what you dumped if debugging routines are still attached to the function being loaded. Note also that if you do a LOAD while you are debugging, all functions loaded will automatically be $UNBUGed. This normally is of no serious consequence, although it may surprise you occasionally. Problems will arise, however, if you are doing any value-tracing (via $TRACEV) when you do a LOAD. It is therefore strongly recommended that all value-tracing be turned off before utilizing the LOAD function.

8. Dynamic Dumping Package

The Dynamic Dumping Package is a set of LISP functions which are useful for interactively examining and possibly altering the innards of LISP while it is running. It was designed primarily for the purpose of debugging functions written with the Maryland LISP Assembly Program, but has potential application for anyone who is doing bit-pushing with LISP. The functions outlined here are basic functions which can be used to write specialized core dumping and examining routines.

Before attempting to use these functions, it is a good idea to understand the distinction between a LISP pointer and the octal address of an object. Normally when referring to a LISP object, one is using a LISP pointer which is represented internally as the address in core of the object. For example, feeding the LISP pointer to the LISP integer 3 to PRINT (eg., (PRINT 3)), prints "3". The octal address of a LISP object, on the other hand, is a LISP octal node which contains the address of the object. This should be thought of as an indirect pointer to the object. For example, the octal address of the LISP integer 3 might be 74036Q, and it would print that way. The LISP garbage collector goes not follow the pointers implied by octal addresses, so that if a LISP object is only known by its octal address, a garbage collection will take it away.

The functions which are described below may be loaded into LISP by doing:

```
@ADD LISP*LIB.CORE
```

The functions documented here are ACORE, ICORE, OCORE, PCORE, CORE, H1, H2, XH1, XH2, S1, S2, S3, S4, S5, S6, T1, T2, T3, Q1, Q2, Q3, Q4, CHANGE, CDUMP, ADDROF, CONTENTS, OCTAL, INTEGER, REAL, CNODE, and OCTSTR.

8.1. Formatted Dumping Functions

Five functions are available in the dumping package for producing a CPMD-style formatted listing of selected core locations. There are four different formats available: octal, integer, alphanumeric (FIELDATA), and instruction. For the time being, instruction format dumping is available only when the LISP assembler is loaded, since several assembler routines and tables are used by the dumping functions.

8.1.1. Acore, Icore, Ocore, and Pcore

These four functions are similar in use and effect. Each dumps an arbitrary number of words starting at an arbitrary location in the indicated format, as given here:

function	format
ACORE	FIELDATA alphanumeric
ICORE	integer
OCORE	octal (Q suffix)
PCORE	instruction

Each of the four functions is called this way:

(<func> <starting address> <number of words>)

and provides a line of output for each address formatted this way:

<address> : <contents>

Again, note that instruction format dumping may not be done unless the Maryland LISP assembler is present in core.

8.1.2. Core

This function combines the facilities of the last four, providing a dump of each word in several formats. It takes the same first two arguments with the same meanings, as well as up to four extra arguments specifying the formats in which each location is to be dumped, A for alphanumeric, O for octal, I for integer, and P for instruction. Each output line will be of the same basic format as before, and colons will seperate the various formats of each word. The formats will be printed in the following order: alphanumeric, integer, octal, and instruction.

8.2. Partial Word Examination Functions

There are 17 functions which are used for examining only a specific part of an octal word:

function	bits returned
H1	35-18
H2	17-0
XH1	35-18, sign-extended
XH2	17-0, sign-extended
S1	35-30
S2	29-24
S3	23-18
S4	17-12
S5	11-6
S6	5-0
T1	35-24, sign-extended
T2	23-12, sign-extended

T3	11-0, sign-extended
Q1	35-27
Q2	26-18
Q3	17-9
Q4	8-0

Each takes a single argument which must be a a fixed-point number, and returns the specified part of it. All but the quarter-word functions were written through the LISP assembler and are loaded in "compiled" form.

8.3. The Core Altering Function - Change

There is a single function for altering the contents of a given location or part of a location. This is the CHANGE function, and it takes two required arguments and one optional argument. The calling format is:

(CHANGE <pointer> <new value> [<partial-word mnemonic>])

The first argument is a LISP pointer to the object to be changed. The second argument is an octal word which will be inserted in the desired part of the word indicated by the first argument. The third argument is one of these partial-word mnemonics: H1, H2, T1, T2, T3, S1, S2, S3, S4, S5, S6, Q1, Q2, Q3, Q4, or W, or the numeric code for the mnemonic. The default third argument is a whole-word transfer. All but the quarter-word transfers are handled through a function written through the assembler. Note that the XH1 and XH2 mnemonics have no meaning in this context.

8.4. General-Purpose Dumping Functions

8.4.1. Cdump

This function takes two arguments - an octal address and a number of words - and returns a list containing the octal values stored in those locations.

8.4.2. Addrof

This function takes one argument, a pointer at any LISP object, and returns an octal word containing its address in core.

8.4.3. Contents

This function takes one argument, an octal word containing the address of any LISP object in core, and returns a pointer at it. This is the inverse of the ADDROF function.

8.4.4. Octal

This function takes one argument, a pointer to any LISP object, and returns an octal word containing the contents of the word being pointed at.

8.4.5. Integer

This function is similar to OCTAL, but its result is an integer rather than an octal number.

8.4.6. Real

This is also similar to OCTAL, but the result will be a real number, assuming that the argument points at a word with a suitable normalized floating-point format.

8.4.7. Cnode

This function takes a pointer at an arbitrary word and makes a cons node out of it. It is expected that both halves of the word will be legitimate LISP pointers.

8.4.8. Octstr

This function takes as its single argument a list of octal numbers and makes those numbers into a string, assuming of course that each word is the octal representation of six characters except maybe the last, which, if it does not represent six characters, should be left-justified and zero-filled. Note that LISP does not allow "@" signs or 077 characters in its strings. The OCTSTR algorithm stops processing when it finds an entire half of a word zeroed. For example, (OCTSTR '((0607100607110 1010050000000Q)))="ABCABDCC ".

This function is useful for working with the user's Program Control Table. (see page 94) For example, (OCTSTR (PCT 1 1)) returns the user's runid in string form, (OCTSTR (PCT 2 19)) returns the user's account number in string form, and (OCTSTR (PCT 2 17)) returns the user's project id in string form, all left-justified and padded with blanks to lengths of 6, 12, and 12 characters respectively.

9. Function Editor

The Maryland LISP Function Editor is a program which allows the user selectively to edit function definitions. It is designed to be used with the Library File routines (see page 153) and the Autoloader (see page 126) to provide complete function development facilities which can be used as independently of features outside LISP as possible. The Function Editor can be loaded by typing:

```
@ADD LISP*LIB.FUNC-ED
```

9.1. Calling the Function Editor

The Function Editor is called through the function EDIT:

```
(EDIT 'FUNC)
```

where FUNC is some atomic symbol. If FUNC is constantly or fluidly bound, an s-expression which, when EVALuated, can re-establish that value is constructed and is made available for editing. The user is placed in edit mode (see next paragraph). If FUNC has no value and a library file has been established, the file's table of contents is searched for an entry under FUNC. If one is found, the binding is made and execution continues as though the binding had originally been there. If FUNC has no value and either no library file has been established or FUNC is not in the library file, the user is placed in the Text Editor [Hag77] in input mode to enter and edit the initial version of the function. When the Text Editor is exited, the s-expression which was entered there is read in to LISP and constantly bound to FUNC, and the user is placed in the Function Editor's edit mode.

9.2. Using the Function Editor

Once a binding for the atomic symbol being edited has been established, the user is placed in edit mode, under the control of the function editing routine. This routine gives "==---->" as a prompt, to which the user can respond with any of the commands described below. These commands can be used to follow the structure of the s-expression to the region which needs changing. Each time a command is given to descend into the structure in this way, the editor calls itself with the part to be edited and goes into a new loop. Once editing has finished at one level, the user can cause that level to return to the one above it or to the top level, with or without the changes which were made at intervening levels.

9.3. Edit Mode Commands

These are all the commands one can type in response to the Function Editor's prompt. Several may be typed on one line, but if one causes an error, the rest will not be processed. If the user asks the Function Editor to do something impossible, such as editing the CAR of an atom, the Editor will give an error message and offer the user another prompt without making any changes.

(1) P - use the Prettyprinter (see page 175) to print out what is being edited at the current level.

(2) A - recur and edit the CAR of the current structure.

(3) D - recur and edit the CDR of the current structure.

(4) L <n> - if the current structure is a list, recur and edit the <n>th member of the list.

(5) R <s> - replace the current structure with <s>, which should be an s-expression with balanced parentheses.

(6) U - return one level, implementing the changes made at the current level.

(7) T - return all the way to the top level, implementing the changes made at all levels.

(8) AU - return one level without implementing the changes made at the current level.

(9) E - end the editing session and EVALuate the edited structure, thus creating a new binding for the atomic symbol being edited.

(10) AB - abort the editing session without making any changes to the binding of the atomic symbol being edited.

(11) UF - if a library file has been established, write out the current structure to the file as the value of the atomic symbol being edited.

(12) SE - call the Text Editor with the current structure. If the Text Editor exits normally by its EXIT command, the structure resulting from the editing session will become the structure at the current level. If the Text Editor exits by its ABORT command, no change will be made at the current level.

(13) S <new> <old> - substitutes <new> for every occurrence of <old> in the current structure.

(14) C <s> - cons the s-expression <s> onto the front of the current structure, and edit the result.

(15) C* <s> - cons the current structure onto the front of the s-exression <s>, and edit the result.

(16) EV <s> - print the result of EVALuating the s-expression <s>. Both this and the M command may make use of the fluid variables STRUCT, which is bound to the current structure, and TOP, which is bound to T if the current level is the top level and NIL otherwise.

(17) M <a> - EVALuates (<a>), where <a> is some atomic symbol which is bound to a function of no arguments which, presumably, makes some changes to the binding of the variable STRUCT.

9.4. An Example

In this example, upper case text in quotes is that typed at the user by LISP, unquoted upper case text is that typed at LISP by the user, and lower case text is commentary by the author. In this example, we show the use of the Function Editor in the development of a function FACTORIAL, which should accept an integer argument and return its factorial.

```
:EXEC @ASG,A MYFILE.          assign the user's library file
":EXEC COMPLETED"
(EDIT 'FACTORIAL)             factorial need not have a binding
"INPUT"                       we are now in the Text Editor
(CSETQ FACTORIAL (LAMBDA (N)
     (COND ((EQUAL N 1) 1)
           (T (TIMES N (FACTORIAL (SUB1 M)))) )))
                              CR gets us in Edit mode
"EDIT"
"*"LA                         go to last line to fix that "m"
          "(T (TIMES N (FACTORIAL (SUB1 M)))) )))"
"*"C / M/ N/                  (we're still in the Text Editor)
          "(T (TIMES N (FACTORIAL (SUB1 N)))) )))"
"*"E                          leave the Text Editor
"=====>" P                    (now we're in the Function Editor)
"(CSETQ FACTORIAL
     (LAMBDA (N)              this is Prettyprinter output
          (COND [<EQUAL N 1> 1]
                [T <TIMES N (FACTORIAL (SUB1 N))>]))))"
      oops. <equal n 1> should be <equal n 0>.
"=====>" L 3 P                go to third element and print
     "(LAMBDA (N)
          (COND [<EQUAL N 1> 1]
                [T <TIMES N (FACTORIAL (SUB1 N))>]))"
"=====>" L 3 D A A L 3 P      go way in
"1"
"=====>" R 0                  change the 1 to 0
"=====>" T P                  go to the top level and print
"(CSETQ FACTORIAL
```

```
     (LAMBDA (N)
             (COND [<EQUAL N 1> 1]
                   [T <TIMES N (FACTORIAL (SUB1 N))>])))"
"=====>" UF E                write the def'n to the file and exit
"EVAL: " (FACTORIAL 4)
"VALUE: 24"
"EVAL: " (FACTORIAL 0)       always test special cases
"VALUE: 1"
"EVAL: " (FACTORIAL -1)      and error cases
"STACK OVERFLOW"             we don't want that
"EVAL: " (EDIT 'FACTORIAL)   let's go fix it
"=====>" S (LESSP N 2) (EQUAL N 0) P
"(LSETQ FACTORIAL
     (LAMBDA (N)
             (COND [<LESSP N 2> 1]
                   [T <TIMES N (FACTORIAL (SUB1 N))>])))"
"=====>" UF E                write it out and exit the editor
"EVAL: " (FACTORIAL -1)      try it again
"VALUE: 1"                   it works
```

Now, during another LISP session, if the Autoloader is hooked up to the library file, the new FACTORIAL function can be used as though it is part of LISP.

10. Generator Package

Maryland LISP has available a limited generator facility (a la Conniver) which allows the user to write functions which can return multiple (or no) results and which have the ability to suspend themselves and be restarted where they left off. This is basically a LISP implementation of coroutines. A somewhat deeper understanding of the intent of a generator facility can be had by seeing [McD74].

To load the generator package, do:

```
@ADD LISP*LIB.GEN
```

10.1. Creating a Generator

The special form CDEFUN is used to define generators. The format of the call is:

```
(CDEFUN <name> <args> (PROG <vars> <stmt>...<stmt>))
```

where <name> is the name of the generator, <args> is its argument list (any form allowed by LAMBDA is legal here), and <vars> and <stmt> are the standard arguments to PROG. The CDEFUN algorithm reformats the structure of the PROG and binds this redefinition to <name>.

There are three functions which are of use in a generator. The NOTE function returns its argument as one of the values of the generator. The AU-REVOIR function of no arguments suspends operation of the generator and allows the calling program to process values which have been NOTE'd and resume the generator later. The ADIEU function returns from the generator and lets the calling function know that it should not be resumed because it has no more values to NOTE. If ADIEU is given an argument, that argument will be the last value returned by the generator. This is useful for marking the end of values returned by that call to the generator.

Syntactically, the functions AU-REVOIR, ADIEU, and NOTE are used in the same manner as in Conniver, but the algorithm used to resume the generators requires some restrictions on the ways in which AU-REVOIR may be called. If a call on AU-REVOIR is dynamically constructed or made outside of the scope of the function body specified in the call on CDEFUN, then it must have an argument (unquoted, since AU-REVOIR is a special form), which is a label in the body of the generator at which execution will be resumed. If AU-REVOIR is called from inside the body of a generator, then the <stmt> in the PROG in which it appears must have the following form:

```
<stmt> -> (AU-REVOIR) | <call> | <sexpr>
<call> -> <cond> | <attempt> | <do>
<cond> -> (COND <condcl>*)
<condcl> -> (<sexpr> <stmt>*)
<attempt> -> (ATTEMPT <sexpr> <attcl>*)
<attcl> -> (<integer> <stmt>*)
<do> -> (DO <stmt>*)
```

where <sexpr> is any EVALable LISP s-expression not involving a call on AU-REVOIR. This grammar may seem restictive, but experience has shown that almost all generator algorithms can be comfortably stated in these terms.

10.2. Using Generators

To use a generator, it is necessary to establish a variable which is to be bound to the possibilities list for that instantiation of the generator. This can be done using the GENERATOR function:

```
(SETQ <var> (GENERATOR <call-on-generator>))
```

where <call-on-generator> is an s-expression whose CAR is the name of the generator and whose CDR is the argument list for this call.

The generator can then be made successively to return its values until failure by using the TRY-NEXT function:

```
(TRY-NEXT <var> <cont>)
```

where <var> is the variable established with GENERATOR, and <cont> is an s-expression to be sent to EVAL in case there are no possibilities left to be returned by this generator.

10.3. The Possibilities List

Two kinds of items reside on a possibilities list, items being returned and "*GENERATOR" items which are used to restart a generator which has been just created by GENERATOR or which has just suspended itself using AU-REVOIR. These items have the form:

```
(*GENERATOR <call> [<bindings>])
```

where <call> is the s-expression which can be EVALuated to call the generator and <bindings> is a list of those alist entries which were destroyed when the generator was last exited. If the generator has not yet been entered, <bindings> is omitted. GENERATOR and TRY-NEXT play with the arguments given in <call> so that 1) a new first argument is inserted which is the label inside the generator body where processing is to resume and 2) if

the generator has already been called, the other arguments have been replaced by NIL to avoid side-effects when they are re-EVALuated. In the latter case, the original values of the arguments will be on the <bindings> list. Of course, it is not advisable to return as a possibility some s-expression whose CAR is *GENERATOR unless it is intended to resume some generator.

It is possible for a possibilities list to contain several *GENERATOR entries. In each case, when a generator is resumed, the values it NOTE's are appended to the end of the possibilities list, that is, the possibilities list is a queue structure.

10.4. Example

Suppose we define a data-base retrieval generator $FETCH:

```
(defun $fetch (pattern)
     (prog ((matches (fetch pattern)))
      loop (cond ((null matches) (adieu)))
                 (t (note (car matches))
                    (setq matches (cdr matches))
                    (au-revoir)
                    (go loop)))
```

Then, assuming that the function FETCH returns a list of those data-base entries which match the given pattern, the entries which match, say, (LOVES -X -Y), can be retrieved one at a time and printed by doing:

```
(prog ((poss (generator '($fetch '(loves -x -y)))))
 loop (print (try-next 'poss '(return)))
      (go loop))
```

10.5. Compiling Generators

The games played by the generator package with the labels in the body of a generator make compiling a generator a tricky matter. To compile a generator, load the compiler, but call COMPILEGEN instead of COMPILE. This function will automatically declare all the arguments and PROG variables in the generator FLUID and slightly reformat the generator so as to avoid dynamically constructing a call on GO used to restart the generator. However, the user must still obey the rule against dynamically building calls on GO or feeding a call on GO to EVAL when the label is in a function to be compiled. This rule is stated and explained in the compiler documentation. For example, a generator containing the following call could not be compiled:

```
(TRY-NEXT 'POSS '(GO ALOOP))
```

11. Library File Functions

These functions use the random access file I/O routines to allow the user to maintain a library of LISP function definitions, each of which need not be loaded until it is needed. This package is designed to be used by the Autoloader (see page 126) and the Function Editor (see page 146) so that the user can work on a project completely within LISP, thus avoiding the grunge work associated with EXEC-8 file manipulation. To load the Library File functions, enter LISP and do:

```
@ADD LISP*LIB.LIBFILE
```

11.1. Declaring a Library File

To declare a file to be used as a library file, establish an internal name for it and send it to the SETLIB function. This initializes all the routines necessary to access the file.

As an example, here is how one might create a library file and initialize it for use by the Library File routines:

```
:EXEC @CAT,P QUAL*MYLIBRARY.
:EXEC @USE L.,QUAL*MYLIBRARY.
:EXEC @ASG,A L.
(SETLIB 'L)
```

11.2. Retrieving a Library File Definition

To retrieve the definition of an atomic symbol X in the established library file, do (GETDEFN 'X). This returns the s-expression which was written to the file as the definition of X if such an entry exists, and NIL otherwise. It is also possible to store and retrieve property list entries. To retrieve the value associated with the indicator I on the property list of atomic symbol A, do (GETDEFN '(A . I)). This returns the definition of the entry if one exists and NIL otherwise.

11.3. Creating a Library File Definition

To create an entry in the established library file which defines a value for the atomic symbol X, call (PUTDEFN 'X S), where S is some s-expression which, when EVALuated, will establish a binding for X. Similarly, to put a property list definition in the library file, call (PUTDEFN '(A . I) S).

Alternatively, the defining s-expression of some atomic symbol or property list entry may be automatically generated and written to the established library file by (PUTLIB 'X) or (PUTLIB

'(A . I)).

In either case, the definition written to the file must be made up exclusively of printable objects; buffers, linker nodes, and pointers to compiled or system code cannot be printed. If PUTDEFN is used, the routines do not care what the s-expression will do when it is EVALuated as long as one of its side-effects is to define a binding for the given atom or property list entry. This is especially important for the Autoloader, which goes into an infinite loop if the binding is not made.

11.4. Library File Notes

A heavily-used library file may often contain many obsolete copies of functions and property list entries. These can be removed from the current library file by doing (LIBPACK).

As a system convention, record 1 of a library file is an association list which associates atomic symbols or atom-indicator pairs with record numbers in the file (see page 62).

12. Mail System

The Maryland LISP Mail System is a humble attempt at allowing the various members of the Maryland LISP user community to communicate with one another. It has three basic capabilities: 1) the sending of messages, 2) the receiving of messages, and 3) the dispersal of information about the users of LISP at Maryland. Use of the Mail System is available to anyone using LISP on the University of Maryland UNIVAC 1100/40 computer. Users are distinguished by their run identifications (runid's), and each user of the Mail System should establish a unique runid for his own usage. The Mail System holds all its code and messages in a single global file. Therefore, the user need not pay for file space to use the system. On the other hand, the system is not burglar-proof (or idiot-proof for that matter), so that absolute privacy cannot be guaranteed. Also, users are asked to keep the amount of text they have on file to a minimum, and to report any bugs to runid SYSTEM's mail file.

The Mail System may be used by entering LISP via

```
@LISP*LIB.LISP
```

and loading in the System by doing

```
@ADD LISP*MAIL.MAIL
```

The functions documented here are MAILHELLO, MAIL, MAILBOX, and GETMAIL.

12.1. Plugging in to the Mail System

In order to use the Mail System, the user must tell the system something about himself. This serves to identify users, and to allow users of LISP to get in touch with one another. To do this, enter LISP and load the Mail System as described in the introduction. Then call the function MAILHELLO, which takes 7 arguments, all LISP strings. They are: 1) the runid you usually use, 2) your name, 3) your home and/or local street address, 4) your home and/or local city/state/zip code, 5) your home and/or local and/or office phone number, 6) your office number if you have a campus office (and the building its in if not the Computer Science Center), and 7) a short description of your project(s) and interest(s) relating to Computer Science in general and LISP in particular. The strings may be up to 60 characters in length, so you may go into considerable detail if you wish. If you do not wish to give out one or more of the seven items listed, just specify a string of blanks. The information you supply will subsequently be available to all other users of the Mail System. To change the information from a previous MAILHELLO, just call it again with the revised parameters. To remove yourself from the Mail System, send a message to SYSTEM, asking that this be done.

12.2. Sending Messages

To send a message to another user, call MAIL. This function takes one argument, which should be the runid of the receiver of the message in string form or as an atomic symbol. You will be put into the Text Editor (see [Hag77]), and an Editor macro (initiated by LISP), will ask you for the subject of your message. After you answer this, you may proceed to type your message, and edit that part of the file which contains the text of the current message. When you are satisfied with the message and want it sent, type "sexi" ("send exit"), and the message will be recorded. None of the lines in the message being sent should have a percent sign ("%") in column one. The message sending may be aborted by typing "abort" while in the editor. This will cancel the message, but a message will be put down as having been sent.

12.3. Checking the Mailbox

To see if you have any messages, enter LISP, load the Mail System, and call the function MAILBOX of no arguments. It will return a list of all those Mail System users which have sent you messages which you have not looked at yet.

12.4. Receiving a Message

To receive a message sent to you by another user, (enter LISP, etc., of course, and) call GETMAIL. This function takes one argument, which should be the runid of the sender in string or atomic form. You will be placed in the text editor, and a (LISP-initiated) editor macro will tell you how many messages from the sender are in the element being edited. To inspect these messages, you should use the system-supplied editor macros described in the following paragraphs.

Some points should be made before discussing these macros. First, the Mail System @USE's ED$PF to the file where the macros are stored, so if you want to use your own editor macros, you must use the editor call command and specify your filename. Second, the Mail System's macros need to keep values in editor variables g, h, i, and j, so your macros should not play with them.

12.4.1. Isolate Next Message - Rnext

This editor macro will find the message directly after the one you are currently looking at (or the first message if you are at the top), and isolate it. The subject of the message will be printed, and you will be prompted as to what to do next. If there are no more messages after the current one, a message will

be printed, and you will be placed at the top of the file.

12.4.2. Go to Nth Message - Rgo

An alternative to the rnext macro is rgo, which takes as its argument an integer which is both positive and no greater than the number of messages currently in the element being edited. That numbered message will be isolated for inspection. This is convenient for looking at the most recent message in the file, which will be the last message in the file.

12.4.3. List the Current Message - Rlist

Calling the rlist editor macro will cause the current message to be printed out in its entirety. Standard editor commands can be used to list out parts of a large message.

12.4.4. Delete the Current Message - Rdel

Once a message has been inspected, perused, and digested, it should be removed so that it will not be around the next time messages from the current sender are looked at. The rdel editor macro does this, deleting the entire text of the current message. The rnext editor macro must be used to move on to the next message if any.

12.4.5. Copy Out the Current Message - Rcopy

The current message can be copied out into an element by using the rcopy editor macro, and specifying as an argument the element to be copied to, for example,

```
rcopy myqual*lib-file.message17/from-joe
```

or some less complicated specification. Doing

```
rcopy msg-from-sam
```

will copy the current message into workspace element msg-from-sam.

12.5. Finding Out About Other Users

There is a facility available with the Mail System that allows any user of LISP to find out about anyone else who uses the Mail System. To find out about someone whose runid you know, use the function WHO. This function takes one argument, which should be the runid in string or atomic form, and looks that runid up in the user directory. If an entry for that runid is

found, the MAILHELLO information that user specified will be listed. Otherwise, a message to the affect that that person does not use the Mail System will be printed.

Obviously there is often a need to find out about a user given only a name (or interests or zip code). For this reason, the user directory may be retrieved like so: First, create an @USE name for LISP*MAIL., say, M. Then enter LISP and do

```
(LOAD '(M . INFOLIST))
```

After this is completed, the LISP atom $$INFOLIST will be bound to a list of the arguments to MAILHELLO which were specified by all the users of the system (that is, CAR=runid, CADR=name, etc.). This list may now be inspected. For example, this function, given a string which is someone's first or last name, will find all entries in $$INFOLIST whose name fields MATCH the argument:

```
(csetq namesearch (lambda (name)
   (prog ((result nil))
         (exec "*USE M.,LISP*MAIL. . ")
         (load '(m . infolist))
         (mapc $$infolist (lambda (entry)
             (cond ((match name (cadr entry))
                    (setq result (cons entry result)) ))))
         (return result) )))
```

Further refinements and changes could develop an algorithm sufficiently general for a specific user's purposes.

13. Matrix Manipulation Package

The Matrix Manipulation Package is a small set of basic functions for the dynamic manipulation of matrices, ie., two-dimensional arrays. Each of the functions takes one or more matrices as arguments and returns a newly-created matrix as its result. The Maryland LISP Array Package (see page 121) must be loaded into LISP before these functions can be used. To load these functions into LISP, do

```
@ADD LISP*LIB.MATRIX
```

The functions documented here are MATADD, MATMULT, MATSUB, MATCMULT, MATCADD, MATPPT, and MATCOPY.

13.1. Matrix Addition - Matadd

The MATADD function takes any non-zero number of arguments which should be two-dimensional numeric arrays of the same size, and returns their matrix sum, ie., (a1+a2+...+an)[i,j] = a1[i,j] + a2[i,j] +...+ an[i,j] for all i,j.

13.2. Matrix Multiplication - Matmult

The MATMULT function takes two arguments which should be two-dimensional numeric arrays such that the second subscript maximum on the first array is equal to the first subscript maximum on the second array. Violation of this restriction results in an error message and an (ERROR 4) (see page 67). Also, if both arrays are not numeric, an (ERROR 5) occurs. The result is a new matrix which is the matrix product of the two argument arrays, ie., (a*b)[i,j] = SUM(k=1 to n, a[i,k]*b[k,j]), for all i,j, where n is the number of rows in the first matrix (and the number of columns in the second).

13.3. Matrix Subtraction - Matsub

The MATSUB function takes as arguments two two-dimensional numeric arrays which are the same size and returns their difference, ie., (a-b)[i,j] = a[i,j] - b[i,j] for all i,j.

13.4. Matrix-Scalar Multiplication - Matcmult

The MATCMULT function takes two arguments, a number c and a two-dimensional numeric array a, and returns their Matrix-Scalar product, ie., a'[i,j] = c * a[i,j] for all i,j.

13.5. Matrix-Scalar Addition - Matcadd

The MATCADD function takes two arguments, a number c and a two-dimensional numeric array a and returns their matrix-scalar sum, ie., a'[i,j] = c + a[i,j] for all i,j.

13.6. Print a Matrix - Matprt

The MATPRT function takes as its argument any two-dimensional typed or pointer array and prints out its entries along-side the corresponding indices.

13.7. Copy a Matrix - Matcopy

The MATCOPY function takes as its argument any two-dimensional array and returns a typed range-checked array with the same type and entries.

14. Stanford MLISP

This section describes the changes made at the University of Maryland to make Stanford MLISP compatible with Maryland LISP. For a full tutorial on MLISP, the reader is referred to [Smi70], which is Stanford Artificial Intelligence Project Memo AIM-135, October 1970.

14.1. Limitations of the Use of $

An additional reserved symbol, "$", has been added. Occurrences of "$" in identifiers must be preceded by the escape character "!".

14.2. Character Set Limitations

Several operators in MLISP have been changed to accommodate the use of the FIELDATA character set used by Maryland LISP. These changes are:

Symbol	Function
$=	SETQ
$6	PRELIST
$5	SUFLIST
$N	NULL or NOT
$<	LEQUAL
$>	GEQUAL
$E	MEMBER
$0 or $/	OR

Predefined atoms in MLISP are limited to FIELDATA representations. Changes have been made to accommodate the character set listed on page 11 of the MLISP manual. They are

Atom	Value
LARROW	$=
CIRCLEX	$X

Deletions from this list because of missing FIELDATA correlates are

TAB
LF
VT
FF
CR
ALTMODE
UNDERBAR

14.3. Operating Domain

Note should be taken that MLISP at the University of Maryland is operated in the domain of Maryland LISP, NOT Stanford LISP. Programming should be restricted to the use of Maryland LISP functions since only part of Stanford LISP is defined (to help the translator work). One change has been made, however, in the function CAT. Since the function conflicts with Maryland LISP's CAT, MLISP's CAT is now !&CAT. Maryland LISP's CAT remains the same (see page 91).

14.4. Escape Character

The escape character remains "!" NOT "?" (see "literally next character," p. 11, MLISP manual). The "?" remains the LISP comment character, along with the "%" <comment> "%" commenting convention of MLISP. All identifiers containing special operator symbols (+, *, $, -, &, etc.) must have "!" preceding them.

14.5. Loading MLISP

There are two ways to load the system on 1100 LISP.

1. To get started, issue the following two commands to 1100 EXEC (once per session is sufficient):

```
@USE MLISPDUMP.,LISP*MLISPDUMP.

@ASG,A MLISPDUMP.
```

To get into LISP, type

```
@LISP*LIB.LISP
```

Load in MLISP by typing

```
:RSTR MLISPDUMP
```

2. If you do not have the dump file available to you, you have to go the more expensive route and load the relocatable form from disk. To do this, get into LISP by typing:

```
@LISP*LIB.LISP
```

First get a little breathing room.

```
(GROW 20)
```

Now load in the system by typing

```
(LOAD '(L . MLISP)) (L is an @USE name for LISP*LIB.)
```

14.6. Running MLISP

To run the MLISP translator after loading, simply type

```
(MLISP)
```

Now the translator is waiting for your code. If you feel especially hardy, you can type it in by hand. The rest of us mortals "@ADD" a file containing our code.

If you wish to see the code being read as it is being parsed (advisable for debugging), type

```
:LIST
```

To turn off listing mode, type

```
:UNLIST
```

In LIST mode, as the program is processed each line will be printed and errors pointed out. At the end of the translation you will be asked if you want to see the LISP produced. The system will query

```
LIST CODE? (Y OR N)
```

If you would like to see the LISP code produced initially, type

```
(MLISP L)
```

instead of

```
(MLISP)
```

14.7. Sample Run

```
@LISP*LIB.LISP
LISP/8.02-KN 07/01-17:43

EVAL:  (GROW 20)
VALUE:   11324Q
```

```
EVAL:  (LOAD '(L.MLISP)) ? L is an @USE name for LISP*LIB
VALUE:   MLISP LOADED, TYPE (MLISP)

EVAL:  (MLISP)


UOM-STANFORD MLISP
:LIST
@ADD MYPROG

BEGIN
     NEW VAR;
     VAR$=10;
     WHILE VAR$>0 DO
          BEGIN
               PRINT VAR;
               VAR$=VAR-1
          END;
     PRINT " ";
     PRINT "ALL DONE"
END.
RESTART
*****
LIST CODE? (Y OR N) Y
(CSETQ RESTART
(LAMBDA NIL
         (PROG <VAR>
               <SETQ VAR 10>
               <&WHILE PROG2 '(GEQUAL VAR 0) '
                (PROG NIL
                      <PRINT VAR>
                      <SETQ VAR (SUB1 VAR)>)>
               <PRINT " ">
               <PRINT "ALL DONE">)))
0 ERRORS DETECTED,     0 FUNCTIONS REDEFINED
END MLISP.  TIME:  106 MSEC.
10
9
8
7
6
5
4
3
2
1
0

ALL DONE

VALUE:     ***-END-OF-RUN-***
```

14.8. Compilation

After debugging your functions, you may wisn to compile them for greater efficiency. To do this, you must take the following actions:

(1) Call MLISP with the "C" option (MLISP C) or (MLISP L C). This action informs the translator that you wish to compile.

(2) On initial load, MLISP is not big enough to handle the compiler. If you intend to compile, before you call MLISP, submit a (GROW 20) to the LISP interpreter to increase its size. When you call MLISP with the "C" option, it will load the compiler for you (it may die trying if you don't increase its size).

(3) SPECIAL declarations should be used to define the Maryland LISP notion of FLUID (see page 131).

(4) To save compiled functions, see the documentation for the Maryland LISP compiler (page 130) and the LOAD and DUMP functions (page 96).

15. Wisconsin MicroPlanner

This document serves as a summary of the primitives available in the Maryland LISP implementation (which was slightly revised from a version written at the University of Wisconsin for Wisconsin LISP) of Micro-Planner (hereafter referred to as PLNR). It is intended to be used in conjunction with (1) below, and thus explains in detail only the differences between Wisconsin PLNR and MIT MicroPlanner. Useful (and relatively accessible) sources of PLNR information include the following:

For a complete description of the primitives:

(1) Sussman, G., T. Winograd and E. Charniak: Micro-Planner Reference Manual. MIT AI Lab Memo No. 203a, December 1971.

For an overview of the language:

(2) Winograd, T.: Understanding Natural Language (Sec. 6). Academic Press, 1972.

(3) Hewitt, C.: Procedural Embedding of Knowledge in Planner. Proceedings of the 2nd IJCAI, September 1971.

For details about the implementation:

(4) Baumgart, B.: Micro Planner Alternate Reference Manual. Stanford AI Lab Operating Note No. 67, April 1972.

15.1. Loading PLNR

PLNR is loaded by typing (to EXEC-8):

```
@ADD LISP*LIB.STRTPLNR
```

This will start LISP running, change the comment character from "?" to ";", load in the compiled versions of PLNR, PRETTYP and EDIT, do a (GROW 30) to get you some more core, and then do a (PLNR) to get PLNR running. You will then be communicating with a new level of LISP supervision which acts as a READ-THVAL-PRINT loop instead of the normal READ-EVAL-PRINT. All LISP control cards are recognized, except that a return to the supervisor returns you to the PLNR supervisor (e.g. if you get into trouble, :LISP will return you to PLNR). You may return to the normal LISP supervisor via (RETURN <expr>), and may re-enter PLNR via (PLNR).

15.2. PLNR Primitives

This section describes the PLNR primitives which are currently available. Comments are appended only when there are differences between this implementation and MIT MicroPlanner. Note that meta-linguistic variables are enclosed in angle brackets (<>), and arguments which are evaluated before the primitive is applied are indicated by preceding them with a quote-sign ('<>).

(PLNR)

This function enters a new level of PLNR supervision, just as (LISP) enters a new level of LISP supervision (see page 26). You may return from PLNR to LISP (via RETURN), and then re-enter PLNR without affecting the PLNR data base.

(THAMONG <PLNR-VAR> '<LIST>)

(THAND <E1> <E2> ...)

(THANTE <THM-NAME> <VAR-DECLARATIONS> <PATTERN> <BODY>)
:: ($TA ...)

(THAPPLY <THM-NAME> <DATUM>)

(THASSERT <SKELETON> <REC1> <REC2> ...) :: ($A ...)

<RECS> :: (THPSEUDO)
(THPROP '<EXPR>)
(THTBF <FILTER>) [$T :: (THTBF THTRUE)]
(THUSE <TH1> <TH2> ...)

(THASSERTLIST <<SKELETON> <LIST>> <REC1> <REC2> ...)
:: ($AL ...)

This new primitive allows one to generate a number of related assertions. For example,

($AL (ROSES ARE (RED PINK PRETTY)) <RECS>)

is equivalent to

(THDO ($A (ROSES ARE RED) <RECS>)
($A (ROSES ARE PINK) <RECS>)
($A (ROSES ARE PRETTY) <RECS>))

(THASVAL <PLNR-VAR>)

(THBKPT '<E1> '<E2> ...)

Each <E> will be EVALed and, if THBKPTS are being THTRACEd, printed separated by blanks, e.g.,,

(THBKPT 'X 'IS 'NOW $?X)

(THBREAK '<E1> '<E2> ...)

Each <E> will be EVALed and printed separated by blanks, and then a READ-EVAL-PRINT loop is entered. The loop is exited (with a value of T) when T is read.

(THCOND <PAIR1> <PAIR2> ...)

(THCONSE <THM-NAME> <VAR-DECLARATIONS> <PATTERN> <BODY>)
:: ($TC ...)

(THDATA)

The data read should be of the form ((AN ASSERTION)), ((AN ASSERTION) . PROP), or (THM-NAME). The loop ends when an atom is read, which is returned as the value of THDATA.

(THDO <E1> <E2> ...)

(THDUMP '<FILENAME>)

<FILENAME> must be a data-file which is currently assigned. THDUMP does a THPRETTYP of all current PLNR theorems, and then a (THSTATE) into the indicated file. An @ADD <FILENAME> at some later date will then restore everything to its current state. (THDUMP) will print out all the above on the terminal.

(THEDIT <THM-NAME>)

Allows one to edit a PLNR theorem.

(THERASE <SKELETON> <REC1> <REC2> ...) :: ($E ...)

(see THASSERT for available recommendations).

(THERASING <THM-NAME> <VAR-DECLARATIONS> <PATTERN> <BODY>)
:: ($TE ...)

(THERT <COMMENT>)

Prints out <COMMENT> (unevaluated) and, unlike MIT MicroPlanner, causes failure to propagate all the way back to the PLNR supervisor.

(THEV <EXPR>) :: $&<EXPR>

This is equivalent to (THVAL '<EXPR> THALIST). Note that it is abbreviated $&<EXPR> instead of $e<EXPR>. It may be used both in patterns or as a stand-alone function (e.g., if you want <E> THVALed instead of EVALed when in a THBREAK loop, type $&<E>).

(THFAIL <ARG1> <ARG2> <ARG3>)

(THFAIL THTAG <TAG> T)
(THFAIL THTAG <TAG>)
(THFAIL THPROG)
(THFAIL THEOREM) Causes the current theorem to fail, even if there are nested THPROGs.
(THFAIL THMESSAGE <MESSAGE>)
(THFAIL)

(THFAIL? '<PREDICATE> '<ACTION>)

(THFINALIZE <ARG1> <ARG2>)

(THFINALIZE THTAG <TAG>)
(THFINALIZE THPROG)

(THFINALIZE THEOREM) finalizes everything done in current theorem, even if there are nested THPROGs.

(THFIND <MODE> <SKELETON> <VAR-DECLARATIONS> <BODY>)

<MODE> :: ALL
<N>
(EXACTLY <N>)
(AT-LEAST <N>)
(AT-MOST <N>)
(AS-MANY-AS <N>)
(AT-LEAST <N1> AT-MOST <N2>)
(AT-LEAST <N1> AS-MANY-AS <N2>)

(THFLUSH <IND1> <IND2> ...)

(THFLUSH) :: (THFLUSH THASSERTION $TC $TA $TE)

(THGO <TAG>) :: (THSUCCEED THTAG <TAG>)

(THGOAL <PATTERN> <REC1> <REC2> ...) :: ($G ...)

<RECS> :: (THNODB)
(THDBF <FILTER>)
(THTBF <FILTER>) [$T :: (THTBF THTRUE)]
(THUSE <TH1> <TH2> ...)
(THANUM <N>) :: $N<N>
(THNUM <N>)

(THMATCH '<EXPR1> '<EXPR2>)

(THMESSAGE <VAR-DECLARATIONS> <PATTERN> <BODY>)

(THNOHASH <ATOM> <IND1> <IND2> ...)

(THNOHASH <ATOM>)
:: (THNOHASH <ATOM> THASSERTION $TC $TA $TE)

(THNOT <EXPR>) :: (THCOND [<EXPR> (THFAIL)] [(THSUCCEED)])

(THNV <VARIABLE>) :: $=<VARIABLE>

(THOR <E1> <E2> ...)

(THPRETTYP '<THM-NAMES> '<FILE>)

The list of PLNR theorems indicated by <THM-NAMES> will be prettyprinted to the data-file <FILE> (if <FILE> is absent they are printed on the user's terminal).

(THPROG <VAR-DECLARATIONS> <BODY>)

(THPUT '<ATR> '<IND> '<VALUE>)

(Notice the Wisconsin PLNR argument order)

(THPUTPROP '<ATR> '<VALUE> '<IND>)

(THREMPROP '<ATR> '<IND>)

(THRESTRICT <PLNR-VAR> <LISP-FN1> <LISP-FN2> ...) :: ($R ...)

(THRETURN '<EXPR>) :: (THSUCCEED THPROG '<EXPR>)

(THRPLACA '<DOTTED-PAIR> '<NEW-CAR>)

(THRPLACD '<DOTTED-PAIR> '<NEW-CDR>)

(THSETQ <VAR1> '<E1> <VAR2> '<E2> ...)

(THSTATE <IND1> <IND2> ...)

(THSTATE) :: (THSTATE THASSERTION $TC $TA $TE)

(THSUCCEED <ARG1> <ARG2>)

(THSUCCEED)
(THSUCCEED THTAG <TAG>) :: (THGO <TAG>)
(THSUCCEED THPROG '<E>) :: (THRETURN '<E>)
(THSUCCEED THPROG)
(THSUCCEED THEOREM '<E>)
(THSUCCEED THEOREM)

Causes the current theorem to succeed, even if there are nested THPROGs.

(THTRACE <IND1> <IND2> ...)

<INDS> :: <ATOM>
(<ATOM> <TRACE PREDICATE>)
(<ATOM> <TRACE PREDICATE> <BREAK PREDICATE>)
<ATOM> :: <THM-NAME>, THEOREM, $G, $A, $E, THBKPT

THBREAK is used if the <BREAK PREDICATE> evals to non-NIL. Type T to exit from the break loop.

(THUNTRACE <ATOM1> <ATOM2> ...)

(THUNTRACE) :: Untrace everything that's traced.

(THUNIQUE 'EXPR1> 'EXPR2> ...)

(THV <VARIABLE>) :: $?<VARIABLE>

(THVAL 'EXPR> 'ALIST>)

(THVDO <E1> <E2> ...)

This new primitive acts like THDO, except it is not undone on failure backup. It is equivalent to

(THPROG NIL (THDO <E1> <E2> ...) (THFINALIZE THPROG))

(THVSETQ <VAR1> 'E1> <VAR2> 'E2> ...)

15.3. Abbreviations

The following abbreviations may be used to save your thfingers from typing many long thnames:

```
$?<X>  :: (THV <X>)
$=<X>  :: (THNV <X>)
$$<X>  :: (THEV <X>)
$N<N>  :: (THANUM <N>)
$T     :: (THTBF THTRUE)
$G     :: THGOAL
$A     :: THASSERT
$AL    :: THASSERTLIST
$E     :: THERASE
$R     :: THRESTRICT
$TC    :: THCONSE
$TA    :: THANTE
$TE    :: THERASING
```

15.4. PLNR Notes

Most PLNR error messages are described in (1). Note, however, that Wisconsin PLNR simply THFAILs back to the PLNR supervisor after an error occurs.

All PLNR variables ($?<X> or $=<X>) must be bound before they are referenced, either by appearing in a <VAR-DECLARATIONS> list in a THEOREM, THPROG, THFIND or THMESSAGE, or by explicitly creating a global binding (e.g., (THSETQ $?X 'THUNASSIGNED)).

LISP expressions may be used with relative impunity within

PLNR expressions - the rule is that if an expression EVALs to NIL it fails (like (THFAIL)), otherwise it succeeds. To evaluate a LISP expression which returns a value of NIL without failing, you must therefore do something like (THDO <EXPR>) or (DO <EXPR> T).

Note, however, that complex PLNR primitives should never be placed inside of a LISP expression. For example, a COND may appear in a THPROG, but it may not contain any THGOALs (use THCOND instead). This is necessary since PLNR's control structures are not implemented through LISP's recursive stack mechanism. Simple PLNR primitives like THV and THASVAL may appear in LISP expressions (e.g., (PRINT $?X) is OK). When in doubt, you may always do an explicit $&<EXPR>.

Since PLNR makes rather extensive use of the question-mark character, the LISP comment character is changed to semi-colon when PLNR is loaded. You may temporarily change it back to "?" (e.g., to @ADD in some code that contains comments) by typing (see page 53):

```
(READMAC "?" (READMAC ";" F))
(DELIM "?" (DELIM ";" F))
```

15.5. PLNR Example

The best way to become familiar with an interactive programming system is to sit down and play with it. As a simple introduction to the use of PLNR, try the following conversation (described more fully in (1)):

```
@ADD LISP*LIB.STRTPLNR
        ;Wait for THVAL: to appear.

($A (HUMAN TURING))
        ;Turing is human.

($TC (X) (FALLIBLE $?X) ($G (HUMAN $?X)))
        ;All humans are fallible.

($G (FALLIBLE TURING))
        ;Is Turing fallible?
        ;No?  Aha - We forgot to tell PLNR to
        ;   use the available theorems.

($G (FALLIBLE TURING) $T)
        ;Poor Turing.

(THPROG (Y) ($G (FALLIBLE $?Y) $T) (THRETURN $?Y))
        ;Is anything fallible?

(THDO ($A (HUMAN SOCRATES)) ($A (GREEK SOCRATES))
      ($A (HUMAN NEWTON)))
```

```
        ;Add some more facts.

(THSTATE)
        ;Check the data base.

(THDUMP)
        ;A more complete picture of the data base.

(THTRACE THEOREM $G)
        ;Let's watch PLNR work.

(THFIND 1 $?X (X) ($G (FALLIBLE $?X) $T) ($G (GREEK $?X)))
        ;Find a fallible Greek - note the backtracking.

(THUNTRACE)
        ;Note that it tells you what was untraced.

(RETURN 'DONE)

:STOP
```

16. Prettyprinter

The Prettyprinter allows the user to print constant bindings and property lists in readable format on the printer or terminal, or to dump them to mass storage for permanent storage. It is loaded by typing (in LISP):

```
@ADD LISP*LIB.PRETTYP   or,

:EXEC @USE LISP.,LISP*LIB.
(LOAD '(LISP . PRETTYP))
```

16.1. Using the Prettyprinter

The major function of the Prettyprinter is PRETTYP. To cause constant bindings and property lists to be printed on the current output device (printer or terminal), say:

```
(PRETTYP L)
```

where L evaluates to a list of designators, each of which has one of the following forms:

A	Print the object constantly bound to the atomic symbol A (which may be a function, a special form, a macro, a composite (but not circular) object, a number, a string or another atomic symbol).
(I A1 .. An)	For each of the atomic symbols A1 .. An, print the attribute-value pair and/or flag indicated by I.
"C"	Print the readmacro and delimiter flag for the character "C".

For example:

```
(PRETTYP '(F1 F2 V1 (IND V1 V2) "!"))
```

or:

```
(CSETQ DUMPALL '(F1 F2 V1 (IND V1 V2) "!"))
(PRETTYP DUMPALL)
```

16.2. Dumping

To redirect the prettyprinter output to a sequential file, create a file desciptor for it using FIOPEN with NIL as its second argument (see page 60), and give that descriptor as the second argument to PRETTYP. This will cause the prettyprinter

output to be sent to that file, with a :LOAD/:END pair surrounding the text and list of the printed objects as the last line of the file. To specify some other last line for the file (especially a string identifying the functions should the file be @ADD'ed), give it as a third argument to PRETTYP. Several sets of Prettyprinted functions can be placed end-to-end in the file by giving the same file descriptor to several calls on PRETTYP. Once the user is done Prettyprinting text into a file, it is necessary to give the file desciptor to FICLOSE to wind up the output operation.

As an example of file dumping, consider the following:

```
:EXEC @ASG,T DUMPFILE.
(CSETQ DFILE (FIOPEN 'DUMPFILE NIL))
(PRETTYP DUMPALL DFILE "FUNCTIONS LOADED")
(FICLOSE DFILE)
```

Then, if DUMPALL had been bound to the list (X Y (FLG X Y)), then the DUMPFILE file might look like:

```
:LOAD
(CSETQ X 'X-VAL)
(CSETQ Y 'Y-VAL)
(PUT 'X 'FLG 'X-FLGVAL)
(PUT 'Y 'FLG 'Y-FLGVAL)
:END
"FUNCTIONS LOADED"
```

Once the file has been properly closed, the dumped information may be restored by typing:

```
@ADD DUMPFILE.
```

The "value" of the @ADD will be the last line of the file, in this case the string "FUNCTIONS LOADED". The file may be copied to a program file element with the Text Editor (see [Hag77]).

16.3. Prettyprinter Notes

Lines printed (or card images dumped) will not extend beyond column 72. To change this parameter (for example when prettyprinting to the line printer) CSETQ the atomic symbol PP-WIDTH to the desired maximum column.

When prettyprinting functions which have been placed on property lists, a search is made to see if any atomic symbol is constantly bound to the function. If so, its name is printed (e.g., (PUT 'ATM 'IND 'FUNC)). Otherwise the actual LAMBDA expression is printed.

If you attempt to prettyprint an object which can't be explicitly printed (e.g., compiled code or a traced function), a

search is made to see if any other atomic symbol is constantly bound to the object. If so, its name is printed (e.g., (CSETQ SUM PLUS)). Otherwise the object is considered unprintable, and the atomic symbol **UNDEF** will be printed as its value. To notify you of such mistakes, the value returned by PRETTYP is a list of all atomic symbols, attribute-atom pairs or readmacro characters which were bound to a nameless unprintable object.

17. Suspend/Resume Package

The Suspend/Resume package provides a facility to save output that would normally be sent to the terminal. To load the package one either @ADD's the source file LISP*LIB.SUSP-RESUME or creates a USE relation on LISP*LIB., by saying (for example) :EXEC @USE LISP,LISP*LIB (once per session) and then loading in the compiled version of the package by typing (LOAD '(LISP . SUSP-RESUME)). Once the system is defined you have access to two functions, SUSPEND and RESUME.

SUSPEND - A function of no arguments which directs all succeeding output to a file.

RESUME - A function with a single optional argument which directs the output back to the terminal and determines its disposition. If no argument is given then the system will prompt you for it by saying:

EXAMINE, HOLD, PRINT OR DELETE?

The proper responses to this question are listed below:

E - Edits the output file in the UOM Text Editor.

P - Prints the output file on a high speed printer and then deletes it.

H - Holds the file for a future decision.

HP or PH - Prints the output file and does not delete it.

D - Deletes the output file.

If P, PH or HP are given as responses the system will ask:

WHERE?

This allows on which you to select the printer you want your work to appear. At Maryland printers can go up and down with little notice, so it is advisable to use PRA for the 1108 and PRE for the 1100/41. These virtual printers will get your job printed on the first available physical printer.

EXAMPLE:

Send output to a file

```
(SUSPEND)

.

.

Perform intervening LISP computations.

.

.

Send file to system Editor.

(RESUME E)
```

The user may echo all inputs into LISP by using the :LIST feature of the system. To turn off echo mode submit :UNLIST to the system.

It may sometimes be desirable to find out the name of the file where your output was stashed. The atom containing the string file name is called SR-FILE.

Section 4

Maryland LISP Assembler

Contents

1. Introduction 184
2. Basic Features of the Assembler 185
2.1. Calling the LISP Assembler 185
2.2. The Code Format 186
2.2.1. Instructions 186
2.2.2. Labels 187
2.2.3. Directives 187
2.2.4. Macros 188
2.3. AXR$ 188
2.4. Expressions, Literals, and Forward References 189
2.5. The Assembler Algorithm 190
2.5.1. First Pass EQU Processing 190
2.5.2. Instruction Processing 191
2.5.3. Forward Reference Resolution Table 191
2.5.4. Symbol Table 192
2.6. Miscellaneous Directives 192
2.6.1. Conditional Assembly - ON and OFF 192
2.6.2. Storage Allocation - +, RES, and GET 192
3. Data Areas and Location Counters 193
3.1. Using Load and Dump 193
3.2. The Normal Form of Data Area Allocation 194
3.3. The V Option Form of Data Area Allocation 194
3.4. Location Counter Sharing 195
4. Utility and General Purpose Macros and Functions 197
4.1. System-Defined Macros 197
4.1.1. Equf 197
4.1.2. Provide a Pointer to an S-Expression - Sexp 197
4.1.3. Declare LISP Functions to be Called - Fnames 198
4.1.4. Form 198
4.1.5. Flostr 198
4.1.6. Provide a Pointer into LISP - Ept 199
4.1.7. Ept$ and Lspcon$ 199
4.1.8. Register Saving Macros 199
4.1.8.1. Generate Register Saving Area - Savepkt 200
4.1.8.2. Saving the Minor Set of Registers 200
4.1.8.3. Saving the Entire Set of Registers 200
4.1.9. Mimic UNIVACs DO Directive - Ado 200
4.2. General Purpose Functions 201
4.2.1. Address 201
4.2.2. Nf 201
4.2.3. Ljstr 202
4.2.4. Rjstr 202
4.2.5. $ 202
4.2.6. Nbits 202
4.3. Arithmetic Functions 203
4.4. Macros for Quarter-Word Manipulation 203
4.4.1. Ascii 203
4.4.2. Fieldata 204
5. Interfacing with LISP 205
5.1. LISP Naming Conventions 205
5.2. LISP's Data Structures 206
5.2.1. Cons Nodes 207

5.2.2. Numeric Types 207
5.2.3. Unallocated Pages and System Code 207
5.2.4. Assembled and Compiled Code 207
5.2.5. Linker Nodes 208
5.2.6. Atomic Symbols 210
5.2.7. Strings 211
5.2.8. Buffer Pages 211
5.3. The Entry Points 211
5.3.1. The Stacks - Cstak and Stack 212
5.3.2. Enter a Function - Entry 213
5.3.3. Enter a Routine - Entryr 214
5.3.4. Leave a Function - Exit 214
5.3.5. List the Objects on the Stack - Listem 214
5.3.6. Apply a Break Function to Arguments - Breaker 214
5.3.7. Error Return - Bacl 215
5.3.8. Look Up Value of Fluid Variable - Lookie 215
5.3.9. Create a Local Variable - Bind 215
5.3.10. Node Allocation - Typtab 216
5.3.11. Return Nil - Gfal 217
5.3.12. Closure of Function - Funarg 217
5.3.13. Routine to Run a Funarg - Funarg+3 217
5.3.14. Expand a Macro - Mexpand 217
5.3.15. The Page Table - Pagtab 218
5.3.16. Remove Several Variable Bindings - Unbind 218
5.3.17. Get the Type of an Object - Getyp 219
5.3.18. Put List Members on the Stack - Stakem 219
5.3.19. Expand a Special Form Call - Sexpand 219
5.3.20. Follow a Car-Cdr Chain - Follow 220
5.3.21. Apply S-Expression to Arguments - Apply 220
5.3.22. Apply Function Closure to Arguments - Apply1 221
5.3.23. Establish a Trap Point - Trap 221
5.3.24. Disconnect a Trap Point - Untrap 222
5.3.25. Trace Path Back Down Stack - Unwind 222
5.3.26. The Current Gensym Number - Genno 223
5.3.27. Create a New Special Form or Macro - Defspm 223
5.3.28. Temporary for Prog Result Storage - Pvsave 223
5.3.29. Make a String out of the Name Buffer - Makstr 223
5.3.30. Copy String into Name Buffer - Getnama 224
5.3.31. Blank-Fill Rest of Name Buffer - Blanks 224
5.3.32. Sequential String Buffer - Name 224
5.4. Argument Handling 224
5.4.1. Arguments to Regular Functions 224
5.4.2. Arguments for Special Forms 227
5.5. Returning a Value 227
5.6. Calling LISP Functions 227
5.7. Macros for Interfacing with LISP 229
5.7.1. Get Car of a Word - $upper 229
5.7.2. Get Cdr of a Word - $lower 230
5.7.3. Follow a Chain of Pointers - $chain 230
5.7.4. Allocate LISP Object Nodes - $node 230
6. Programming Advice and Notes 232
6.1. Concerning Large Integers 232
6.2. More Load/Dump Side-Effects 232
6.3. Space Restrictions 232

6.4. Support Routines 233
6.4.1. The LISP Dynamic Dumper 233
6.4.2. Asm-excise 233
6.4.3. Asm-prettyp 233

1. Introduction

This section serves two purposes. The first, as indicated by its title, is to document the Maryland LISP Assembler, a LISP assembler for use with Maryland LISP on the UNIVAC 1100/40 system at the University of Maryland. The purpose of the LISP assembler is to allow a user of Maryland LISP to do specialized work in LISP using UNIVAC capabilities for which functions are not provided without physically altering LISP.

The second purpose of this section is to provide some documentation of the implementation of Maryland LISP, a subject long relegated to the depths of the cryptic comments in the LISP source code. This material should be of help to any hardy soul who wishes to make improvements and alterations to Maryland LISP. The implementation notes have been given here rather than in some appendix or separate section because of a realization on the part of the author that the intersection of LISP users and UNIVAC assembler users at the University of Maryland is not substantial, consisting mainly of people involved with the upkeep of LISP.

Facilities are provided in the assembler for macros, listing options, interfacing with the LISP system, and so forth, and the use of these facilities is described in the sections which follow.

In this section, LISP functions and atoms to be evaluated will be underlined and in lower case. Assembler instruction and partial-word transfer mnemonics and register and directive names will be in upper case. Assembler symbols (labels and other EQU'ed constants) will be in plain lower case. In descriptions of syntax, words enclosed in angle brackets denote syntactic entities and constructs enclosed in square brackets are optional. Parentheses in syntax descriptions are always those of LISP s-expressions. Ellipses denote anything which can legally replace them in the expressions in which they appear.

Unless otherwise specified, the word "assembler" refers to the Maryland LISP Assembler.

Except for the section on LISP's data structures, it is assumed that the reader has some knowledge of LISP and/or the UNIVAC @ASM assembler, (see [Uni74]) upon which the LISP assembler's syntax is based.

2. Basic Features of the Assembler

2.1. Calling the LISP Assembler

To use the LISP assembler, one must do the following: After entering LISP, load in the assembler using:

```
@ADD AGRE*AGRE.ASM/LISP .
```

At present, no compiled version of the assembler is available.

Before using the assembler, it is necessary to have reserved an area of core for assembled code to be stored in. To do this, one uses the :CODE directive (see page 107).

Then, assuming that an assembler program as described in succeeding sections has been bound to the atom program, one may define a function, say foobar, by doing

```
(csetq foobar (assemble program))
```

or a special form by doing

```
(defspec foobar (assemble program)) .
```

Once this is done and the assembly is completed without encountering error or warning conditions, foobar is a function which can be used just like any normal function. Handling of arguments and so forth is explained in the chapter pertaining to interfacing with LISP.

The name assemble can also be shortened to asm, as both are defined. Optional extra arguments are the options, which are quoted single-letter atomic symbols. For example, the a, l, and o options could have been specified in either of the above calls:

```
(csetq foobar (assemble program 'a 'l 'o))
```

The options, which need not be in any order, are as follows:

a Load the arithmetic package if it is not already loaded. This is determined by finding out whether the atom ar-gump has a constant binding.

c Share the use of some location counters. The use of this option is complex, so a thorough discussion has been postponed to a later chapter.

d Used with the l option to specify a double-spaced instruction listing.

e Perform an (asm-excise) to remove the assembler from

core after the completion of the assembly.

f Don't clear the various flags and tables that the assembler sets up for its own use after the assembly has been completed. This option is mainly for use in debugging the assembler.

g In the formatted instruction listing, use the backslash prefix notation for labels in expressions. The default is to omit the backslash and the getsym and getsym1 calls in the listing.

l Provide a full listing. Conflicts with the n option. The d, g, o, and p options are meaningless unless the l option is also specified.

n Suppress all output except for messages resulting from really fatal error conditions.

o Print out octal numbers with a "0" prefix rather than a "Q" suffix.

p Page up after each section in the listing except the last. This should only be used on very large programs and usually in conjunction with the s option.

s Load the suspend/resume package and breakpoint the listing into a file. The assembler will give the standard "EXAMINE, PRINT, . . ." queries before returning its value. The suspend/resume package will not be loaded if the atom suspend is found to have a constant binding.

v Indicates that the function being defined uses the form of data area allocation (next chapter) which is suitable for dumping and re-loading. This entails some special restrications and modifications in algorithms involving static data areas, but is not often used.

The default listing gives simply a signon line, starting address, and signoff timing information. Other options may be added according to the wishes of the user community.

2.2. The Code Format

The (required) first argument to the assembler is a list of labels, instructions, directives, and macro calls. They are encoded as s-expressions, but retain the same basic form as the analogous constructs in the UNIVAC @ASM assembler.

2.2.1. Instructions

Instructions should be coded as lists of this form: (<op>

[<a>] <u> [<x> [<j>]]). Fields may be omitted by coding them either as the atom nil or as the octal value 0Q, and trailing omitted fields may be dropped. The a field should not be present in those instructions (for example, SZ, SLJ, J, JO, EX, ER, etc.) which do not use it.

Examples:

```
LISP                                  @ASM
(LA A0 0 X5)                          LA        A0,0,X5
(EX output (* X4))                    EX        output,*X4
(LNM A10 (* startloc) nil XH2)        LNM,XH2   A10,*startloc
(ER exit$)                            ER        exit$
(LMJ X11 corout (* X1))               LMJ       X11,corout,*X1
```

The constants, symbols, and expressions which may appear in the various fields of an instruction are explained in detail later in this chapter.

2.2.2. Labels

The user may attach labels to instructions in the source code by simply inserting the atomic label before the desired instruction. There is no limit on the number of labels which may appear in any one place, nor is there any limit on the number of labels which may appear in the program. Any LISP atom (except nil) is a permissible label. No label may appear more than once in an assembler program. No constant binding is associated with the labels by the assembler, so that a label may be the name of a LISP- or user-defined function or constant. However, the user should take care that no label name coincides with any which is defined through xr, that is, register, partial-word, and executive request mnemonics, IO$ function codes, the LISP naming conventions, and the entry points into LISP. All these are explained later.

Example: LISP assembler: (. . . lab1 object (L A0 3 nil U) . . . print (ER print$) . . . (J print) . . .)

```
UNIVAC assembler:
                    . . .
lab1
object    L,U       A0,3
          . . .
print     ER        print$
          . . .
          J         print
          . . .
```

2.2.3. Directives

For the time being, the LISP assembler's directives are as follows: EQU, +, RES, SUSPEND, RESUME, LIST, UNLIST, ON, OFF, END, $, and GET. All but the last have the same meanings as in

the UNIVAC assembler. The exact effects of many of these are discussed in various sections below. The last directive, GET, is involved with assembler storage allocation and is a subject of the next chapter. These terms will all be explained later.

Here are some examples of LISP assembler directives and their UNIVAC translations:

(RES 34)	RES	34
(END start)	END	start
(SUSPEND)	SUSPEND	
($ 9)	*(9)	
(ON (getsym abc))	ON	abc>0

The last example shows how a symbol table lookup function getsym is used to resolve ambiguities caused by assembler symbols appearing in expressions. For a more detailed discussion of what constants and expressions may appear as directive operands, read on in this chapter.

2.2.4. Macros

The LISP assembler has an easy-to-use and very powerful macro facility. Simply stated, when the assembler comes across an item in the source code list which is not an atom, and whose car is neither and instruction nor a directive name, it evaluates it. The result should be a list of instructions, labels, directives, and/or macro calls. Processing resumes at the start of this list. This allows, among other things, macro generation of labels and recursive macros.

Many macros come pre-defined with the LISP assembler and these are described in the appropriate sections below.

2.3. AXR$

A macro axr$ is provided for defining system symbols. With no arguments, axr$ (conceptually) returns a list of EQU directives which are processed by the assembler and which define the standard register and partial-word mnemonics, that is, X1-X15, A0-A15, R1-R15, S1-S6, T1-T3, W, U, H1-H2, XH1-XH2, and XU. Specifying an (unquoted) l option loads the LISP conventions for register naming and other EQU'ed constants. An e option loads in the entry points by their LISP names. These are explained in those sections which deal with interfacing with LISP. An r option loads the codes for the executive requests. Anyone doing ER's from an assembled program should specify the r option on the axr$ call. Finally, an i option loads the IO$ function codes, such as R$ and W$.

For those readers interested in implementation details, axr$ actually installs the desired symbols in the symbol table itself

and returns nil, this only to reduce the inefficiencies associated with building up and tearing apart hundreds of EQU directives.

The user should make it standard practice to make a call on axr$ the first thing in any assembler program. An example would be: (axr$ e l). This loads the standard register and partial-word mnemonics, the LISP entry points, and the LISP naming conventions, and is the most common form of the axr$ call.

2.4. Expressions, Literals, and Forward References

LISP arithmetic expressions like (add1 (myfunc 3)) may appear in several places in an assembler program, namely as arguments to macros, as operands to most directives, and in the a, u, x, and j fields of instructions. Several restraints, however, must be imposed because of the fact that the assembler's code generating functions are those which were tailored to the LISP compiler, which has a rather fixed and predictable style of programming, and because of the fact that macro expansion and processing of EQU directives is a separate pass in the assembly. For this latter reason, the places in which EQU directives appear in the code have no meaning except perhaps to the user.

Because the assembler handles its symbol table through property lists, there is a problem with assembler symbols appearing in expressions. Any assembler symbol which appears in an arithmetic expression which is to be evaluated must be an argument to a call on getsym. This is a special form which looks its argument up on the symbol table, returning its value if one exists. A readmacro for the character "\" has been defined so that (getsym mylab) can be shortened to \mylab, where mylab is any symbol that the user or axr$ defines.

When using getsym, special precautions must be taken to assure that a value for the symbol will indeed exist, lest garbage be returned. For the purposes of our discussion here, we will define three categories of assembler symbols, pre-defined symbols, post-defined symbols, and labels. Pre-defined symbols are 1) all those defined through axr$, and 2) any symbol which is EQU'ed to a numeric constant, or to the value of any expression, all of whose symbols' definitions through EQU qualify as being pre-defined and are defined before it. Post-defined symbols are all EQU'ed symbols which do not satisfy the definition of being pre-defined because the expressions defining them either had forward references or references to labels. Labels are those atomic symbols which appear alone in the code.

Example:

```
(EQU l1 30)
(EQU l2 (plus \l1 22))
(EQU l3 (leftshift \l4 2))
(EQU l4 \l1) start
```

```
(EQU l5 \start)
(EQU l6 (difference \l1 (logand \l5 77Q)))
($ (sub1 \l1))
(+ \l3)
(RES \l3)
```

Here, each line but the last is legal. The symbols l1, l2, and l4 are pre-defined. The symbol start is a label. The symbol l3 is post-defined because it has a forward reference to l4. The symbol l5 is post-defined because it depends upon a label, start, and the symbol l6 is post-defined because its value depends upon that of another post-defined symbol, l5. The $(l1) directive is legal because l1 is pre-defined. The + l3 directive is legal because evaluation of its operand can be put off until l3 becomes defined. And finally, the RES l3 directive is illegal, and will generate an error message, because the assembler must know when the RES is being processed how many words to reserve.

The assembler is smart enough not to evaluate any expression which appears in the u field of an instruction of the operand field of a + directive until it can be certain that all the assembler symbols used in it are defined; this eliminates any restrictions on the expressions which can appear in these places. On the other hand, any expression which appears in the a, x, or j field of an instruction or as the operand of a RES, ON, or GET directive must contain only 1) numeric constants, 2) pre-defined symbols, and 3) labels which appear at least one instruction (or RES or + directive) before the instruction or directive in question. This is only rarely a problem, and violations of this restriction are marked with error messages in the right-hand margin of an l-option listing. (Note that if an expression which appears in the a, u, x, or j field of an instruction consists of a single symbol, then the getsym or "\" is optional.)

Useful for simulating literals are three functions, form, nf, and address, which are provided with the assembler. Each is described generally in another chapter, but for the purpose of literals, form defines a function which returns an octal word split up in the specified way, and nf is a general-purpose function that partitions an octal word into equal parts, depending on the number of its arguments. The function address should be used to make the translation between a LISP object (such as an octal node) and the address of that object, for purposes where it is the address that is required.

2.5. The Assembler Algorithm

This section describes how the assembler works and sketches the format of the full (l-option) listing.

2.5.1. First Pass EQU Processing

Pass one of the assembler expands the macros and processes the EQU directives. All EQU directives which can be resolved at this time without forward references, references to labels, or to undefined symbols are put on the symbol table. The arguments to all remaining EQU directives are put on another table for processing after all the code has been generated.

Each of the symbols whose value is put on the symbol table in pass 1 is listed in the form <sym> = <expr>, where <sym> is the symbol, and <expr> is the expression or constant it was EQUated to.

2.5.2. Instruction Processing

As the code is generated, an instruction listing is provided in which the various instructions and directives are listed in the order in which they are found after macro expansion. Each line of the listing has four fields: 1) the address (octal) at which the instruction was placed; 2) the actual octal word which was placed at that location, if any; 3) the directive or instruction itself in the UNIVAC format (so as to get the whole thing on a 72-character line); and 4) any error message which the instruction or directive may have provoked. Some fatal error messages are printed on their own lines in case no instruction listing is being given.

The UNLIST directive of no operands will cause the listing to be turned off, and the LIST directive (also of no operands) will turn it back on. These are useful for preventing the listing of expanded macros. Some directives, such as ON, SUSPEND, and RESUME, generate no instructions, so that the listing records no address or emitted word for these. The RES directive, on the other hand, may generate many lines of listing (one per emitted word). The SUSPEND and RESUME directives (both of no operands) can be used just as in the UNIVAC assembler to avoid printing more than one line for a RES directive.

2.5.3. Forward Reference Resolution Table

Usually, a program will contain forward references in the code. When a forward reference occurs in the u field of an instruction or in the operand of a + directive, a record is kept containing the address and the expression which could not be resolved. Then after all the code has been generated, several passes through the list of unresolved symbols are made in an effort to put them in the symbol table. When this is complete, the forward reference list is processed. All expressions which are resolvable (all symbols defined) are evaluated and plugged into the proper addresses. All those which cannot be resolved

are assumed to be zero. The Forward Reference Resolution table records the outcome of this process. Each entry in the table contains the address where the result was plugged, and the value of the expression itself. The user can use this and the instruction listing to reconstruct exactly what code was generated.

2.5.4. Symbol Table

The last part of the listing is a list of all user-defined symbols (those not defined through xr). Each such symbol is listed here with its value. Symbols whose values could not be determined are flagged with an error message. This might happen because the symbol is EQU'ed to some expression containing an undefined symbol or if the symbol has a recursive definition.

2.6. Miscellaneous Directives

2.6.1. Conditional Assembly - ON and OFF

The LISP assembler provides two directives for the conditional assembly of blocks of assembler code. These are ON and OFF and they work in basically the same way as their counterparts in the UNIVAC assembler. ON takes one operand, an expression, which is evaluated. If the resulting value is positive, assembly continues normally. However, if the value is zero or negative, no code is generated until the matching OFF directive is reached. ON/OFF pairs may be nested to any depth. If the expression which is the operand of an ON directive cannot be evaluated (one of the symbols in it is undefined), the default is 1, the assembly continues normally, and an error message is generated if a listing is being produced. The OFF directive takes no operand. During any time that code is not being generated because of an ON command, the instruction listing continues normally, with the address and generated octal word fields left blank.

2.6.2. Storage Allocation - +, RES, and GET

The + and RES directives are just as used in the UNIVAC assembler. The GET directive is peculiar to the LISP assembler. It takes a single operand, which must be an evaluatable expression (evaluatable in the same way that operands to RES as described above must be). Once the value of the expression is obtained, the assembler attempts to guarantee that that many consecutive words of memory will be available for succeeding + and RES directives. The GET directive is only of use in data areas as described in succeeding sections, and it should never be used or needed with the v option.

3. Data Areas and Location Counters

The assembler, because of restrictions placed on it by the design of Maryland LISP, has a system of data area allocation and location counters which differs somewhat from that of the UNIVAC assembler. The location counter facility itself comes in two versions, a more restrictive one being necessary if the v option is in force.

The beginning user will probably want to skip this chapter because of the fact that data area manipulation under the LISP assembler is clumsy, confusing, and not totally debugged. The use of static data areas such as described here can often be circumvented by creative use of literals and special registers, and by using some form of dynamic allocation when large chunks of memory are needed for short-range storage. Also, Maryland LISP has an array facility which may be used instead for some applications.

Each location counter corresponds to (at least) one 128-word area known as a buffer page. Buffers have the advantage that they will never be split into parts by LISP, thus maintaining the consecutive nature of data areas. However, the disadvantages compared to the UNIVAC assembler's location counter facility are considerable, especially if the code is to be dumped (read on).

Location counter 0 is the compiled code area. It is not a buffer, rather the (type 5) compiled/assembled code area, so no data area allocations larger than one word can be done there. To declare a location counter, put a call to the $ directive (as distinguished from the $ function) in the desired location, just as in the UNIVAC assembler, for example, ($ 3). Initially, the location counter is $(0). All code between a $ directive and the next one (or the end of the program) goes on the buffer page corresponding to the number gotten by evaluating the operand to the directive.

There is a facility, the c option, for allowing several assemblies to use the same buffer pages for little-used location counters. This is called sharing, and is described near the end of this chapter.

3.1. Using Load and Dump

In many LISP Assembly Program (LAP) systems, the LAP is used both to write assembler programs and (mainly) to provide a meta-language for output from the compiler. This latter use is necessary because there is no general-purpose facility for dumping the binary core representations of s-expressions and compiled code out to a mass-storage file. In these systems, the LAP code must be re-assembled every time it is needed (for shame!). In Maryland LISP, however, a facility is available to

write these binary representations out to files directly and reload them when needed in another LISP session. This is implemented through the dump and load functions.

The function dump takes two arguments. The first is an object to dump, usually an atom to which is bound either a function or a list of LISP constructs. The second is either an atom which is the name of an assigned mass storage data file, or a dotted pair of such a file name and an atom name representing an element of that file. In the latter case, LISP dumps to an element of type omnibus with that name, and in the former case, LISP dumps to the file itself in a form not readable by any other processor.

The function load takes a single argument, which is a file or omnibus element name as just described. From that file is loaded the object which dump wrote out to it. The object loaded is returned as the result. The loaded object(s) will not be at the same locations they occupied when dumped. This fact, however obvious, will prove inconvenient in the discussion of the dumping of data areas in a later section.

These functions can be used to dump an assembled program and store it until it is needed in another LISP session. In order to dump an assembled function which uses data areas as described in this chapter, it must have been generated under the v option restrictions and modifications outlined in the section on that subject.

3.2. The Normal Form of Data Area Allocation

For a one-shot assembler program which is not going to be used again, or which is going to be re-assembled (yuck!) each time it is used, the normal form of data area allocation is used. In this, each label has as its symbol table value the absolute core address it refers to, so that it can be used like any label in the UNIVAC assembler. Words on buffer pages may point at anything they like in the normal form provided that whatever they point at is not subject to garbage collection.

Of use when assembled programs are not going to be dumped is the GET directive, especially when data area sharing is in effect. By putting a, say, (GET 50) in front of a number of RES's and +'s which must be physically together, one may assure that they indeed will be together. If necessary, the assembler will allocate a new buffer page to accomodate the GET. RES directives do their own GETting automatically.

3.3. The V Option Form of Data Area Allocation

Whereas the mechanics of data area allocation are rather straightforward if the user does not have the restrictions

regarding dumpable code to deal with, if dumpability is desired, then certain rules must be followed. This section outlines the rules which hold under the v option, which must be present if the result it to be dumped.

The first restriction is that the only place on a buffer which may be pointed at by the code is the first word. For this reason, the values of all labels under non-zero location counters are relative to the start of the buffer itself. The best way to operate under this restriction is to keep the base address of each buffer page handy, and have it in an index register to use as an offset when data from that page is being used. Unfortunately, LISP has permanent information stored in registers X1-X7 and X9, and registers A0 and A2 often have special meanings in interfacing with LISP, so that the number of available index registers is small. In any event, if the number of location counters is small, one buffer starting address could be stored in each of several available index registers, and EQUF's could be defined for the labels to be the relative address label indexed by the index register for that buffer. Since it is necessary to know on which buffer page any given label is, it is important that only 128 words be put under any one location counter, even through sharing. Therefore, the GET directive is not of use in v option situations.

A function $ exists which, given a number, will return the octal address of the beginning of the buffer corresponding to that number.

A final restriction under v-option conditions is that no word on any dumped buffer page (non-zero location counter) may point at any object which would be subject to relocation upon re-logging. Objects not subject to relocation are as follows: 1) all atoms which are initially created with the system (and are bound to system functions, eg, cons), their initial values and print names; 2) the atom nil; 3) all atoms whose print names are single characters and their initial values (if any) and print names; 4) all entry points into LISP; and 5) all other objects classified as type 4 (unallocated pages/system code) by LISP at dumping time.

3.4. Location Counter Sharing

Often, the 128-word size of location counters' buffers is much too large. For this reason, a facility is available for several functions to share location counter areas. This way, all of a user's assembler programs could use the same buffer or buffers instead of each one just using a small part of one and wasting the rest. This facility is implemented by the c option.

If the c option is specified, the assembler expects there to be an extra argument after all the options which is either a positive integer or a list of positive integers (the default is a

1). If the last argument is a positive integer, all location counters up through and including the integer's will be shared with the previous assembly (which may have shared them with others). If a list of integers is given, all those listed counters will be shared. Counter 0 is always shared. Note that if the v option is in effect on any of the sharing assemblies, the number of words reserved under any one non-zero location counter may not exceed 128.

If two or more functions share a buffer, they should always be dumped and loaded together. If not, then identical copies of the buffers will be made for each dump, so that there will be expensive duplication of buffers upon re-loading.

4. Utility and General Purpose Macros and Functions

Because there are several useful capabilities of the UNIVAC assembler which are not directly implemented in the LISP assembler, and because LISP uses many conventions and structures which are not compatible with assembly language programming, several special purpose macros and functions are provided along with the assembler to allow the assembler user to apply techniques familiar to him to deal with LISP. These are described in this section.

4.1. System-Defined Macros

4.1.1. Equf

The macro equf is designed to be the LISP counterpart of the UNIVAC EQUF directive. It takes 2, 3, or 4 arguments. The first is the symbol being defined. The second, third, and fourth arguments are any expressions representing u, x, and j fields respectively. The expressions may contain forward references, but any symbol appearing in them must have a "\" prefix. If the expressions can be evaluated immediately, they are. The result of the equf call is (a list containing) an EQU directive EQUating the symbol to a constant (if all 3 expressions can be evaluated) or another expression, which when finally evaluated can be logored with an existing instruction to give the intended instruction. Unwanted arguments should be coded as 0Q, and if the fourth argument is 0Q, it can be omitted, and if both the third and fourth arguments are 0Q, they can both be omitted. Do not code unwanted arguments as nil.

4.1.2. Provide a Pointer to an S-Expression - Sexp

Since this assembler is meant to interface with LISP, there is a need for symbols to have as their values the locations of LISP s-expressions. The macro sexp provides this capability. Sexp takes two arguments. The first is the symbol being defined. The second is an s-expression (atom, number, cons node, etc.) which will be evaluated. The address in memory of the resulting value will be the value of the symbol. The result will be (a list containing) an EQU directive EQUating the symbol to the octal address of the value of the s-expression.

Examples

(sexp datastr (cat "*ASG,A " "FILENAME. . ")) causes datastr to have as its symbol table value the core location of the string "*ASG,A FILENAME. . ".

(sexp paramnum 42717Q) causes paranum's symbol table value to be the location of an octal node containing the octal number

42727Q. Note that this is not equivalent to (EQU paranum 42717Q), which causes paranum's value to be the constant 42717Q.

(sexp cons cons) causes cons to have as its symbol table value the core location of the code for the function cons.

(sexp cons 'cons) causes cons to have as its value the location in core of the atom cons.

4.1.3. Declare LISP Functions to be Called - Fnames

The arguments to fnames should be atoms to which are bound those functions which are to be referred to in the assembly. This is a means of declaring them and putting them on the symbol table. All fnames does is to generate a list of sexp macro calls.

Example

(fnames cons fiopen) generates the list ((sexp cons cons)(sexp fiopen fiopen)).

4.1.4. Form

The form macro is the LISP counterpart of the UNIVAC directive of the same name. It takes any number of arguments, the first of which must be an unbound atom, and the rest of which must be a series of integers or expressions whose values must add to 36. The function form returns nil and defines the atom to be a function which, given a number of arguments equal to the number of fields specified in the form call, will return an octal word in which the arguments are placed in the desired locations and truncated from the left if necessary (ie., low-order bits are kept). Since the function defined by form returns an octal word rather than its address, the address function can be used to get the address. For example, suppose a (form pf 12 6 18) is executed. Then (pf 32Q 7Q 2014Q) = 32070020140.

4.1.5. Fldstr

Since LISP strings are stored not in sequential memory but as linked lists, some facility is necessary to make the translation. This is the purpose of the fldstr macro. This macro takes as its first argument a string of any length. It generates a series of + directives, each of whose operands is an octal number representing up to six characters, and a GET directive to keep them in sequential memory. If the string's length is not divisible by six, then the last word is blank-filled, unless a second argument is given, in which case the last word is zero-filled. Note that LISP does not allow the character "@" to appear in strings, nor is the 077 character allowed as it is the LISP end-of-string flag. If zeroes are

required in a sequential string, a function using logand could be written to mask out the undesired characters. The fldstr macro should never be used under $(0) unless the string is less than seven characters in length.

Examples: (fldstr "ABCD12 34") generates the list: ((GET 2) (+ 607080961620) (+ 563640505050))
(fldstr "ABC" t) generates the list: ((GET 1) (+ 607080000000))

4.1.6. Provide a Pointer into LISP - Ept

For use by the LISP compiler, LISP provides a list of addresses of useful stacks, tables, and pieces of code which can be accessed by an assembler program. The purposes of these various entry points into LISP are explained in a separate section under their conventional LISP names. This macro, ept, allows one to assign one's own names to these entry points. The macro ept takes two arguments, the symbol being defined, and an expression or numeric constant. It returns (a list containing) an EQU directive EQUating the symbol to the address of the entry point. If it is possible to evaluate the second argument (all symbols involved in it are defined), then the address of the entry point is computed and given as the second operand to EQU. Otherwise, an expression is constructed which will compute this address, and it is given as the EQU directive's second operand. The symbol thus defined is listed as being a user-defined symbol in the symbol table, and is printed out as such in any listing of the symbol table. Also, this symbol may appear anywhere in the code or in data areas, as it is not subject to relocation in any dump/load process. However, any assembler program using the entry points should be re-assembled each time that a new version of LISP is put on line.

4.1.7. Ept$ and Lspcon$

These macros are used in place of e and l options, respectively, to axr$ so as to make the conventional definitions of the entry points and the LISP conventions user-defined. Using these instead of axr$ options has only the effect of having the definitions show up on any symbol table listing. Neither ept$ nor lspcon$ takes any arguments, and each returns a list of EQU directives.

4.1.8. Register Saving Macros

In complex assembler programs, a user will often use many registers and call LISP functions which also use them. Because of this problem, a facility is provided to stash registers away in data areas and restore them later. The basic saving macros described here are probably not going to be sufficient for many

situations; but they are provided should they be convenient.

Most LISP functions can be expected to trash some or all of the scratch index registers (X8,X10,X11) and the scratch accumulators (A0-A6), as well as R1-R4. In addition, registers X1 through X7 are reserved by LISP for permanent storage, as are X9, A15, and R15. The latter registers should, in general, not be changed.

4.1.8.1. Generate Register Saving Area - Savepkt

To generate an area to store the registers, the function savepkt of one argument returns a list containing the label save$area and a RES directive with the one argument, which should be an expression or numeric constant, as its operand. This should be used to provide the proper amount of space for the registers being saved. The minor registers are 10 in number, The x registers are 11 in number, the a registers are 16 in number, and the r registers are 15 in number.

4.1.8.2. Saving the Minor Set of Registers

A set of macros is provided for saving the minor set of registers, that is, X11, A0-A5, and R1-R3. The macro savem of no arguments returns a sequence of Store instructions which store the registers X11-A5 in the first 7 locations under save$area, and the registers R1-R3 under the next 3 locations. The macro restorem of no arguments reloads the 10 minor registers from the area under save$area. A call to savem should preceed any assembler program's call to any function which disturbs the minor registers, and a call to restorem should come afterward.

4.1.8.3. Saving the Entire Set of Registers

A set of macros is also provided for saving and restoring the entire set of 42 registers under save$area. This is most useful for when a large number of index registers and so forth are necessary in a function so that LISP's permanent values must be stored upon function entry and restored upon exit. The function save generates a list of instructions to save all 42 registers under save$area, and the function restore generates instructions to load them all back. In order to use these safely, the user should have a savepkt call reserving at least 42 locations under some location counter other than $(0).

4.1.9. Mimic UNIVACs DO Directive - Ado

In the UNIVAC assembler, the directive DO is used to provide repetitive generation of similar instructions. The LISP counterpart of this facility is the macro ado. This macro takes three required arguments, a dummy symbol, an expression evaluating to a number, and a LISP assembler instruction or macro call which may or may not involve the dummy variable. If the expression evaluates to a number which is less than one, the null list nil is returned. Otherwise, a number of copies of the instruction or macro call corresponding to the number is made, with integers starting from 1 successively substituted for the dummy variable. Even if the variable is not involved in the third argument, it must be present as an argument. No constant binding is given to the variable, so anything will do.

Example: (ado i \X11 (S i (plus \save$area (sub1 i))))
generates a list of Store instructions which will store the first 11 x registers in the first 11 locations under save$area. This is how the register saving macros work.

4.2. General Purpose Functions

The functions described here are provided with the LISP assembler and are meant to be used in the operands of the various directives and instructions.

4.2.1. Address

This is a function which, given any LISP object (s-expression or pointer into compiled code) will return an octal word containing its core address. This is very useful for pointing at the various linked-list type objects LISP deals with, and also provides a way to allocate single-word literals without resorting to the mess and bother of location counters and data areas.

Example

```
        L,U     A0,address(quote((s . expre) ssi (o . n)))
        L,H1    A1,0,A0
        L,H2    A2,0,A0
```

causes A0 to contain the location of the s-expression ((s . expre) ssi (o . n)), A1 to point at its car, namely (s . expre), and A2 to point at its cdr, namely (ssi (o . n)).

4.2.2. Nf

This function provides the equivalent of the @ASM literal facility by partitioning an octal word into equal parts and

putting its arguments in those parts, returning the resulting octal word. The address function function can be used to get the address of this word for the purposes of u fields of instructions and operands of + directives.

Example. The UNIVAC assembler code lines:

```
L,U     A3,(3,4,abc,q4*con+4)
+       28,buffer3
```

would translate into these two LISP assembler instructions:

```
(L A3 (address (of 3 4 \abc (plus
          (times \q4 \con) 4))) nil U)
(+ (of 28 \buffer3))
```

4.2.3. Ljstr

The ljstr function takes a string whose length is no greater than 6 characters and returns an octal word which is the FIELDATA octal representation of the string left-justified. Any remaining part of the word is zero-filled unless a second argument is specified (either a one-character string or the octal coding of a single character) in which case it is used to fill the space. Remember that LISP does not allow "@" signs in its strings.

4.2.4. Rjstr

This function is similar to ljstr except that the characters in the resulting word are right-justified and any filling which occurs happens to the left of the characters.

4.2.5. $

The $ function takes as its single argument a positive integer and returns the location of the first word under the location counter with that number. This is useful in the v option form of data area allocation, in which all labels are relative to the beginnings of their respective buffer pages, and index registers must be used to provide the starting addresses of those pages. (Unfortunately, the LISP debug package also has a $ function, so for the time being, one must be careful when both the assembler and the debug package are being used. Note that no other assembler functions use the $ function, so the user is safe if there is no explicit call on it in the assembled program. If both the assembler and the debug package are to be used under this restriction, though, the debug package must be loaded last.)

4.2.6. Nbits

This function takes two arguments, the first an octal word and the second a positive number not greater than 36. The function masks the word so that only the number of low-order bits indicated by the second argument remains. For example, (nbits 777q 5)=37q and (nbits -0 9)=777q.

4.3. Arithmetic Functions

Those very fluent in UNIVAC assembler language may have noticed that there are several functions available in the UNIVAC assembler which do not have direct analogues in LISP. These functions are available with the LISP assembler and are listed here:

name	@ASM	args	value
10exp	/+	x,y	x*10^y
m10exp	/-	x,y	x/10^y
2exp	/*	x,y	x*2^y
asm-eq	=	a,b	if a=b then 1 else 0
asm-gt	>	a,b	if a>b then 1 else 0
asm-lt	<	a,b	if a<b then 1 else 0
covquo	//	x,y	if x.mod.y=0 then x/y else x/y+1
//	.//	x,y	ditto

It is true that 2exp is equivalent to leftshift, but only as long as it does not produce an overflow. Note that leftshift does a logical shift if its second argument is positive (denoting a left shift), and a circular shift if its second argument is negative (denoting a right shift). Also note that if the arithmetic package is loaded, the * function may not be used to denote multiplication in assembler expressions, as it conflicts with the notation used for indirection and autoincrementation.

4.4. Macros for Quarter-Word Manipulation

Since Maryland LISP presently operates in FIELDATA (third-word) mode, the quarter-word partial-word codes are not normally available to the assembler programmer. For this reason, the ascii and fieldata macros are provided. The user of these macros should note: 1) The third-word and XH1 partial-word codes may not be used while quarter-word mode is on, 2) No LISP function or routine should be called unless third-word mode is on, and 3) Although the quarter-word mnemonics can be used by the assembler user, any instruction-format core dump will use the third-word and XH1 mnemonics.

4.4.1. Ascii

This macro generates two instructions which trash A0 and set PSR bit 17 to 1. This causes the quarter-word mode to be turned on.

4.4.2. Fieldata

The fieldata macro generates two instructions which set PSR bit 17 to 0. The instructions use only A0. If the ascii mode has been turned on, this will turn it off.

5. Interfacing with LISP

The most crucial part of using the LISP assembler is interfacing with LISP itself. Since the assembled programs will be used as LISP functions, and may need the services of other LISP functions and parts of the LISP interpreter itself, this includes handling function arguments, setting up function calls, receiving values from called functions, returning a value as a function, and returning control back to LISP. Each of these topics will be covered in this chapter.

5.1. LISP Naming Conventions

Inside the LISP interpreter's source code, a large number of symbols are defined to ease understanding of the code. These consist mostly of special names for registers, and will be referred to as the LISP naming conventions. These symbols are loaded into the LISP assembler by specifying an l option on the call to axr$ or by using the lspcon$ macro.

LISP uses several of the registers for permanent storage of certain quantities and the LISP naming conventions assign mnemonic names to these registers. These are listed below:

AXR$	LISP	purpose in LISP
X1	XT	points at top of value stack
X2	XF	points at function frame in stack
X3	XC	points at top of control stack
X4	XL	h1=current alist, h2=return address
X5	XP	trap chain pointer for error recovery
X6	XI	pointer for input routine
X7	XR	used in node allocation
X8	XW	scratch register
X9	XO	pointer for output routine
X10	XW1	another scratch register
A0	XX	scratch
A1	XY	scratch
A2	XV	scratch, usually current LISP object
A3		(scratch) no LISP name, but used as XV+1
A4	XA	scratch
A5		(scratch) no LISP name, but used as XA+1
A6	XXA	scratch
A15	XMCNT	contains # of words allocated
R15	XFLAG	permanently contains -0

None of the registers listed as scratch has any permanent values, but they are often used to pass arguments to LISP routines and are subject to trashing by many functions.

Also in the LISP conventions are a series of constants which

have been given mnemonic names, and which are used by various macros. These are as follows:

name	defined as	purpose
consed	0	type # for cons nodes
integer	1	type # for integer nodes
octal	2	type # for octal nodes
real	3	type # for real nodes
system	4	type # for unused pages/system code
code	5	type # for compiled/assembled code
linker	6	type # for linker nodes
symbol	7	type # for atomic symbols
string	8	type # for string nodes
buffer	9	type # for 128-word buffers
maxtyp	9	the largest type number
numtyps	maxtyp+1	the number of types
pagbit	7	defined such that 2^pagbit=128
pagsiz	2^pagbit	# words in a page
pagmsk	377777q-177Q	mask to get page location
pagnum	2^(17-pagbit)	# of pages in system
nodsiz	3	size of each type table entry

The last five of these are used by the various system macros and functions and are added for flexibility. The type numbers are the ones returned by the $getype macro's instructions and by the type function.

During the discussions which follow, the LISP convention names for registers X1 through X10 and the axr$ conventions for all other registers will be used in discussion and examples.

5.2. LISP's Data Structures

Maryland LISP has ten data types. These are cons nodes, integers, octal numbers, real numbers, unallocated pages/system code (a general non-type), assembled/compiled code, linker nodes, atomic symbols, strings, and buffer pages. At the level at which the regular LISP user works, the details of these data types and their associated structures are hidden from view, so that the user may concentrate on details considered more important than the book-keeping associated with the manipulation of data structures. The power of the assembler, on the other hand, comes from its ability to interact with these data structures to provide specialized features and efficiency. For this reason, one not learned in how Maryland LISP's data structures are organized should look at this section. Little knowledge of UNIVAC assembler is required to read these descriptions.

For book-keeping purposes, LISP divides the core available to it into 128-word segments known as pages. Each page is either

a 128-word chunk that happens to fall in some of LISP's code, a page which has not been allocated, or a page devoted to a single type of node. The number of this type is recorded in the page table entry for that page. When all available nodes of some type have been allocated, and a new one is requested, LISP picks out a new page and dedicates it to the requested type, and when all the nodes on that page become unused, the garbage collector returns it to the list of free pages.

The garbage collector marks one- and two-word LISP objects by setting a bit in a mark bit table in the page the object is on. Specifically, if an object resides at octal address <a>, then bit number <a>.mod.32 in the word at address <a>-<a>.mod.32 is used to mark it. Thus every page of cons nodes, numeric nodes, linkers, strings, and atomic symbols has four mark bit tables. Two words are reserved for every such table to maintain uniformity and to provide room for future extensions of the garbage collector. Compiled code and buffer pages are marked differently, as is explained in the sections describing them.

5.2.1. Cons Nodes

A cons node (type 0) is a single 36-bit word broken down into two halves, each of which is a pointer at one of the 2^17 memory locations LISP can theoretically deal with.

5.2.2. Numeric Types

There are three numeric types in Maryland LISP, integers (type 1), octals (type 2), and reals (type 3). Each is a single word of memory which contains the number. While reals are stored in the standard exponent-mantissa format, octals and integers are stored and used exactly the same by all routines except for the output routines.

5.2.3. Unallocated Pages and System Code

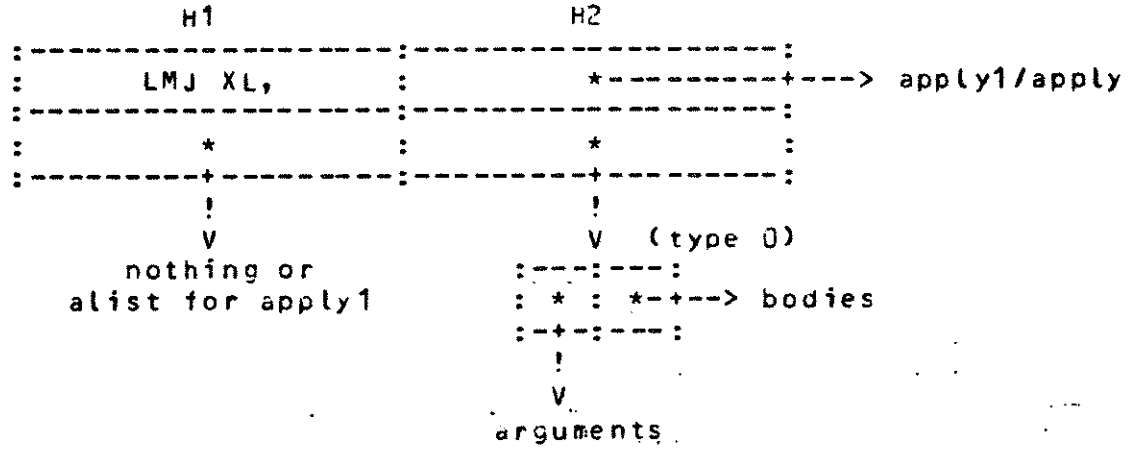
Unallocated pages (type 10) and system code (type 4) are the two "non-types" of Maryland LISP. It does not make sense to allocate or deallocate a page of either of these types, consequently the garbage collector ignores pointers at them. Type 10 is the state a page is in before it has been committed to some type, and type 4 is the type which is assigned to pages which are not available for data area allocation such as system code. It does not make sense to allocate or deallocate a page of either of these types, consequently the garbage collector ignores pointers at them.

5.2.4. Assembled and Compiled Code

Maryland LISP has a feature which allows programs to be compiled directly into machine language in memory and to be moved about on files and so forth without a symbolic intermediate form. Thus, the output from the compiler is a set of machine language instructions placed directly into core. These instructions are placed in the compiled code area which is reserved by the :CODE directive (see page 107) and which is of type 5. When a type 5 address is given as the value of an atom, that address will be taken as the starting address of the function to be called when that atom is used, just as though the function was one of the assembly-language functions loaded with LISP. The garbage collector will not de-allocate parts of the compiled code area.

5.2.5. Linker Nodes

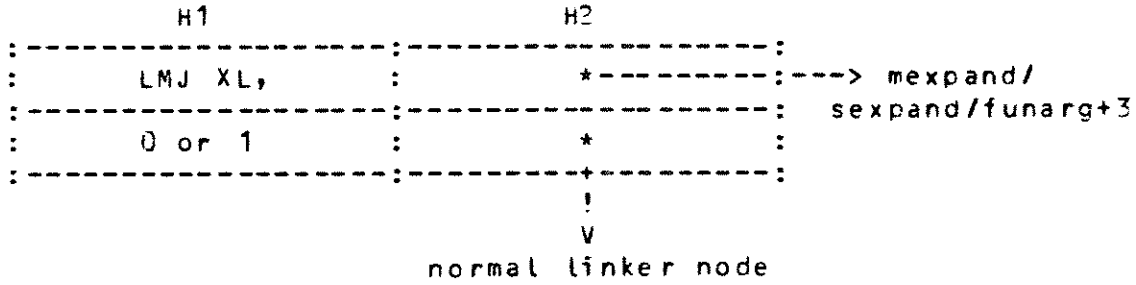
A linker node is a two-word chunk of core on a type 6 page. Linker nodes are used in several ways, all relating to the execution of functions. In its most common form, a linker node looks like so:



As the structure here suggests, to execute the function associated with this linker node, LISP just jumps to the first word of the node, which in turn branches to the appropriate processing routine and passes along a method of getting at the lambda expression associated with the function. Apply is the routine to apply a regular LISP function to a set of arguments and apply1 is the routine which guarantees that a function is only applied when a certain alist is in effect (funarg+3 has a similar effect, read on). When such is the case, the alist to be substituted for the real one is to be found in H1 of word 2 of the linker node. Otherwise this half-word is unused. Both of these routines are entry points. Anything about the number of arguments can be determined by retrieving the lambda expression which hangs off of H2 of word 2. The function lambda very simply

creates a linker node such as this one and puts its arguments on a list in H2 of word 2.

Linker nodes are used somewhat differently in the contexts of special forms, macros, and "funargs". Specifically, they look like so:



In this construction, H2 of word 2, rather than pointing at a set of arguments and function bodies, points at another linker node such as in the first diagram of this section, unless H1 of word 2 is 1, signalling that the special form was defined by the system and must be handled differently. The routines at sexpand, mexpand, and funarg+3 (entry points all) handle the execution of the normal linker as a special form, a macro, or a funarg, respectively. Very specific descriptions of what these routines mean and how they work can be found in the sections which deal with them as entry points.

There are two more special cases for linker nodes which bear discussing. The first is in the case of a function of the form C<A\D>*R which follow the pointers through an s-expression. (For example, caddadaddaaar.) These are handled by the read routine which, if it detects a symbol of the form C(A\D)*R, creates a linker node to bind to the atom which would perform the desired function. Although in theory only 1 through 35 a's and d's are accepted in the atom name, any number from 0 on up is accepted. (The 0 case, cr, is interesting, resulting in a function which just returns its single argument.) The linker node which is bound to the atom has in its second word a bit string representing the a's and d's (with 0=a and 1=d), with a leading 1. The first word's H2 field is a pointer to the routine called follow (which is also an entry point) which takes the bit string in the linker node and applies its operations (except for that of the leading 1, which is just there to signal the end of the bit string) to the argument on the stack.

The LISP function break is used for specifying a function which is to be applied instead of the one bound to a given atom. This is useful for function tracing and so forth. More specifically, break takes two arguments, like so:

(break <atom> <function>)

where the first argument is an atom which is bound to a function

(ie., a linker node), and the second argument is a function (again, a linker node), whose arguments are: 1) the <atom> it is breaking; 2) the function that was bound to the <atom>; and 3-n) the arguments specified to <atom> in the "broken" call. Break constructs a new linker node with

```
LMJ     XL,breaker
```

in word 1, a pointer to its second argument (<function>) in H2 of word 2, and a pointer to a cons node whose car points to the <atom> and whose cdr points to its value in H1 of word 2. Breaker is a routine (and an entry point) which moves each argument on the stack up two spaces and sticks the car and cdr of the cons node pointed at by H1 of the second word of the linker in argument positions 1 and 2 respectively, and jumps to the function pointed at by H2 of word 2 of the linker node. The function unbreak, if given as an argument a "broken" atom, will restore it to its original condition.

5.2.6. Atomic Symbols

Maryland LISP has a fairly novel design for atomic symbols which allows a considerable gain in efficiency over other LISPs. An atomic symbol is a two-word part of a page of type 7. Each atomic symbol has four fields: H1 of word 1 is the value field, H2 of word 1 is the property list, H1 of word 2 is a pointer in the hash list of atomic symbols (the oblist), and H2 of word 2 is a pointer to the print name in string form. When created, an atomic symbol has no value (ie., the value field is 0), the property list is null (ie., H2 of word 1 contains the address of the atomic symbol nil), the print name is a pointer at a string (exception: gensyms. read on.), and the hash link points at the last atom with the same hash code (exception: gensyms again). When an atomic symbol is given a value (through cset or csetq or more devious means) the value cell is changed to point at the structure which is to be its value. However, when its value is fluid, only the association list is changed. No atomic symbol can have both a constant binding and a fluid binding. In some other LISPs, the value is kept on the property list, badly slowing the evaluation process.

Gensymmed atomic symbols are constructed rather differently. First, a gensym's print name is an integer rather than a string (eg., G1's print name is 1), at least until it is interned by creating a string version of the print name. The print name is reconstructed by the print routines by checking the print name of the atomic symbol at which the gensym's hash link points, and printing it and the integer which is the gensym's print name. Gensymmed atomic symbols are not recognized by oblist, but they are not subject to garbage collection, either.

All atoms which are in the oblist and all atoms which are pointed at by other non-garbage pointers in the system are

marked. Since gensyms are not kept in the oblist, this means that they will be garbage-collected when no longer needed.

5.2.7. Strings

Maryland LISP has a (FIELDATA) string capability in which the strings are (conceptually) unlimited in length. Since each page has only a limited number of words, a linked structure is necessary to give this effect. Therefore, a string is made up of a chain of one-word structures, each containing three characters and a pointer to another string node (or the atom nil). String nodes are on pages of type 8. In case the string's length is not a multiple of three, the last node is padded out with zeros. For this reason, "@" ("at") symbols are not permitted in strings (by the various string manipulation routines which recognize the "@" character as meaning end-of-string). The atom nil, for the time being, is Maryland LISP's null string.

5.2.8. Buffer Pages

A buffer page is an entire page of type 9 which is used in situations where large blocks of contiguous memory are required. The name derives from their original use, as buffers for file I/O. They are also used by the assembler as data area, and will someday be used to implement an array package. Nothing they point at is marked (for the time being), and for the time being they may not be gumped (or loaded). Buffer pages are marked just like compiled code pages, in the page table.

5.3. The Entry Points

The entry point is a set of pointers into the LISP system which can be used by the assembler user. They were designed, as was much else the user must deal with here, for use by the compiler, which never makes mistakes. Some are sections of code which simulate or synthesize many of the actions taken when a normal s-expression is evaluated by LISP. Others are tables, stacks, and constants used by LISP to keep track of its inner workings.

These LISP facilities should only be used with the greatest possible caution, as they are quite fragile, and the LISP system's repertoire of error detection and reporting facilities is not extensive.

Much of the terminology and notation used in this section is regrettably but necessarily highly obscure. Much of it can be explained by other sections of this document, but a great deal of the remainder can only be explained by someone learned in the inner workings of LISP.

The entry points are explained below in subsections whose numbers are the numbers of the entry points themselves (except for the stacks, which are explained together). Examples of calling sequences will often be written in the UNIVAC instruction format, as the translations to LISP are easy to make. The entry points are referred to by their conventional LISP names, although the user can declare other names for them using the ept macro.

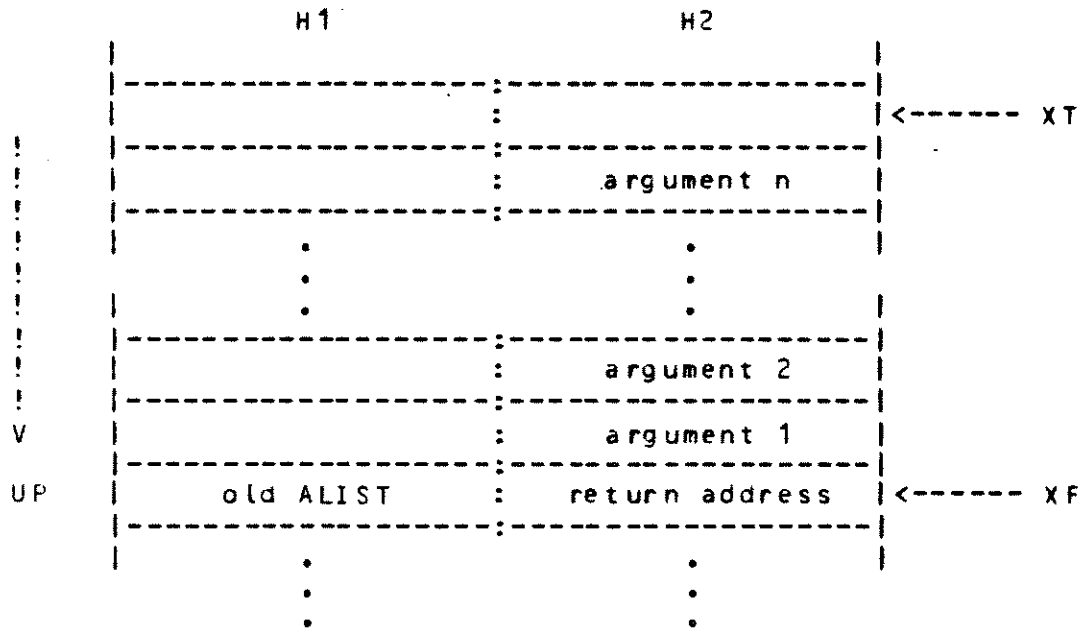
5.3.1. The Stacks - Cstak and Stack

Entry point 0 is cstak, the control stack. Each word in it corresponds to a single function call and looks like so:

frame pointer	-0 or old XT

where H1 points to the function frame of the calling function, that is, the value of XF just before the call was made, and H2 is -0 if the call was a function call (by the evaluation of an s-expression of by a call through macro $call or entry point 2, entry), and is the function frame for the called function, that is, XT when the call was made, if the call was as a routine (through $callr or entry point 3, entryr).

Entry point 1 is stack, the value stack. In addition to the confusion resulting from the fact that it grows down (ie., from higher addresses to lower), it is somewhat more complicated in structure than the control stack, which is basically just alot of pointers into the value stack. When an assembled function is called as either a function or routine, the stack looks like this:



Thus the first argument to a function can be loaded by:

```
L,H2        A2,stack-1,XF
```

and the number of arguments to a function is given by (XF)-(XT)-1. Note that XF and XT are addresses relative to the start of the stack.

The offset XF points at the old value of XL, since H1 of XL contains the association list, and H2 of XL contains the return address of the function call.

In the case that the value stack overflows its 2048 allocated words, a guard mode interrupt will occur and LISP's contingency routines will issue the appropriate messages and reset the interpreter.

The term "stack" used unmodified usually refers to the value stack, and this convention will be used in later sections.

It is not recommended that the inexperienced assembler user play with either stack except through the provided macros, which are explained in detail in a future section, as a messed-up stack will result in some of the most spectacular error aborts ever witnessed.

5.3.2. Enter a Function - Entry

To call a function once the stacks have been arranged, do a

```
LMJ         XL,entry
```

The macro $call will do this also, generating this single instruction. Upon returning, the value of the function will be sitting on top of the stack. Further discussion may be found in the section on function calling from assembled programs.

5.3.3. Enter a Routine - Entryr

This is the analog of entry for calling routines. Calling something as a routine means that the result will be returned in A2, rather than on top of the stack. The stack should be set up in just the same manner. Again, further discussion can be found in the section on function calling.

5.3.4. Leave a Function - Exit

Performing a

```
J          exit
```

simply returns the contents of A2 as the value of the function. The macro $return simply generates this single Jump instruction.

5.3.5. List the Objects on the Stack - Listem

This entry point simply makes a list of the top several entries on the stack. If entering at listem, A1 must contain the stack offset "above" which the listing is to start. Normally, (XF)≥(A1)>(XT), and (A1)≤(XT) is illegal. The list is returned in register A2. The calling sequence is as follows:

```
LMJ        XL,listem
```

If the entry is at listem-1,

```
LMJ        XL,listem-1
```

all stack entries will be listed (by first doing a LX A1,XF). Registers A0, A2, A3, and in the second method A1 are changed, and in either case, the result is returned in A2.

If the user simply wishes to list the stack entries and return the list as the value of the function in the same operation (as in the list function), the following will suffice:

```
LXM,U      XL,exit
J          listem-1
```

5.3.6. Apply a Break Function to Arguments - Breaker

This entry point is used by the break function to apply an alternate function to the arguments on the stack. Its mechanics are explained (as well as they can be) in the section on the structure of linker nodes.

5.3.7. Error Return - Badi

This is the abort routine for compiled and assembled functions. A simple

```
J       badi
```

causes an error message to be printed and control to be returned back to the latest level of LISP supervision or to any attempt call which handles an error type 0.

5.3.8. Look Up Value of Fluid Variable - Lookie

The entry point lookie allows an assembled program to inspect or alter the fluid binding (if any) of an atom on the current association list, which is stored in the upper half of XL. Its mechanics will not be discussed here, though its use will. Lookie is called this way:

```
LMJ       XL,lookie
<op>,U    <reg>,<atom>
```

where <op> is either a load or store instruction (load will get a value, store will change it), <reg> is the register where the value is to be placed (or the register where the new value is stored), and <atom> is the address of the atom whose fluid binding is to be changed. Although these two instructions should be coded consecutively, the user should not worry that LISP might place a jump instruction between them. Since this entry point is also used by the LISP compiler, there is a trick in lookie's code which follows the pointer of any jump instruction XL might point at. If the atom happens to have a constant binding, that binding will be retrieved (or changed). If the atom is fluidly bound, the (most recent) binding will be retrieved (or changed). If the atom is found on the current association list but has no value associated with it, an attempted value retrieval (load instruction) will cause LISP to resort to a user- or system-defined contingency routine to obtain that value from the user. And if the atom cannot be found on the current association list, a binding for it (with no initial value) will be entered and the whole process will be repeated. The lookie routine trashes registers A4 and R2, as well as any registers (outside of A0-A3) needed by any contingency handling routine.

5.3.9. Create a Local Variable - Bind

This is the routine which creates fluidly bound variables on the association list (alist). It is called by

```
LMJ     XL,bind
```

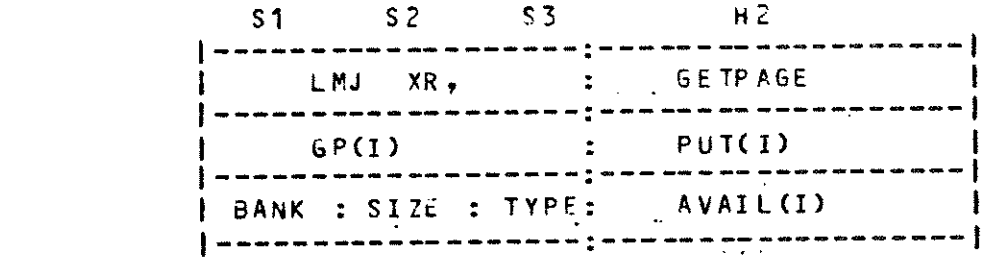
and it assumes that the atom's location is in A3 and its initial value is in A2. For example, to bind the first argument to the variable arg1,

```
sexp    arg1,arg1       . get loc of atom arg1
. . .
L,U     A3,arg1         . get arg1's loc in A3
$load   1               . macro to get 1st arg in A2
LMJ     XL,bind         . make binding
```

Both A2 and A3 are changed by the routine.

5.3.10. Node Allocation - Typtab

This is the type table, used to allocate storage. This table consists of 3 words for each of the 10 types. (In the LISP conventions, there are symbols EQUated to these two quantities, namely nodsiz and numtyps respectively, and the user should use these names rather than the numbers should extra types or words be added.) Each 3-word entry looks like this:



where:

GETPAGE is either the page extraction routine if the AVAILable nodes list is empty, or the routine to get the first AVAILable list node;
GP(I) is the page initialization routine for this type;
PUT(I) is the node allocation routine for this type;
BANK is the preferred bank for this type, 0 = must be static, 1 = prefers to be static; 2 = doesn't care; 3 = prefers to be dynamic; and 4 = must be dynamic;
SIZE is the size of each node of this type;
TYPE is the type, 0=consed, . . ., 9=buffer; and
AVAIL(I) is a pointer at the AVAILable node lists for this type, where avail(i)+n gets the available nodes of type i in bank n, where n varies from -1 to the maximum allowable bank number.

To allocate a node of type I, LMJ through XR to the location computed by typtab+nodsiz*I. The function $storit(type) computes this location, and the macro $node(type) will generate the instruction

```
LMJ     XL,$storit(type)
```

to allocate a node of the desired type. Note that this may trigger a garbage collection.

5.3.11. Return Nil - Gfal

Performing a

```
J       gfal
```

simply returns nil as the value of the function. It has the same effect as

```
sexp    nilat,nil       . can't use nil as asm symbol
L,U     A2,nilat        . get pointer to nil in A2
$return                 . J  exit
```

5.3.12. Closure of Function - Funarg

Funarg, as it happens, is the address of the code for the function function, which, given a function as its (last) argument, returns another linker which guarantees that its copy of the first is always executed with the association list in effect which was in effect when funarg was called. This new linker node is constructed by putting funarg+3 in H2 of word 1, the first linker in H2 of word 2, and the current alist in H1 of word 1. Funarg+3 (entry point 13) retrieves this stored alist and puts it into effect for purposes of the call to the original linker. This all has the same basic effect as apply1, which is used by lamda to do the same thing.

5.3.13. Routine to Run a Funarg - Funarg+3

This is the address used by funarg to make sure the appropriate alist is in effect when a given linker node is called. Its mechanics and use are explained in the discussion of the funarg entry point.

5.3.14. Expand a Macro - Mexpand

Mexpand is used in roughly the same way as apply, apply1, and sexpand in H2 of the first word of a linker node as the routine which handles macros. Since it is jumped to by the first

word of a linker node, it expects the stack to have on it a single "argument" which is a list of all the arguments which were specified to the macro, and it expects XL to point at the second word of some linker node which in turn points at another linker node which defines a normal function using apply or apply1. It then executes the macro, and ships the results off to eval to be evaluated and returned as the result. For the curious, the code looks like:

```
fnames    eval            . get eval's address
. . .
$mark                     . macro to set up first call
L,H2      A2,0,XL         . get function address
$store                    . use it as function to call
LMJ       XL,stakem       . put all args on stack
$callr                    . call function; value in A2
$store    1               . set up 1 arg for eval
LXM,U     XL,exit         . eval returns its value
J         eval            . do final evaluation
```

5.3.15. The Page Table - Pagtab

This is the page table, which records information about every page in the system. If, say, A0 contains the page number, doing

```
L,S3      A1,pagtab,A0
```

will get the type of the nodes on that page in A1. To compute the page number of any address in the system, mask-and-shift-out bits 16-7 of the address. The macro $getype will do just that by generating the instructions

```
AND,U     A2,pagmsk       . pagmsk is a LISP convention
SSL       A3,pagbit       . so is pagbit
LA,S3     A3,pagtab,A3    . type is in S3
```

assuming that A2 contains the node location.

At present each word in the page table contains only the type in S3. In addition, H2 of each word is used by some routines for marking purposes.

Note that in the LISP conventions, pagbit is EQUated to 7, which is the number of bits which are not significant in determining the page number, and the symbol pagmsk is EQUated to octal 037760=((2^17-1)/(2^pagbit))*(2^pagbit). The size of the page table is given by the symbol pagnum=2^(17-pagbit)=1024.

5.3.16. Remove Several Variable Bindings - unbind

This is a pointer into the code which undoes the effect of

bind (entry point 9) by removing variables from the association list. It is called by

```
LMJ     XL,unbind
```

and it assumes that the number of variables to be unbound is in A3. The unbind routine changes the values of registers A0 and A3.

Example. Remove the bindings of 4 variables which were bound earlier in the function's code. Note that each function is responsible for unbinding all the variables it has fluidly bound (using bind) before it leaves.

```
L,U     A3,4
LMJ     XL,unbind
```

5.3.17. Get the Type of an Object - Getyp

This is an entry point to a four-line piece of code which simply gets the type of the object A2 is pointing at and returns it in A3. Its only practical use for the assembler user is to save a few lines of source code to make functions doing much type-checking smaller in size. To use this entry point, assuming A2 points at the object in question, do:

```
LMJ     XL,getyp
```

The type of the object will be returned in A3 without changing the contents of any other register. Of course, the user could also do:

```
$getype
```

which would generate three instructions instead of the one.

5.3.18. Put List Members on the Stack - Stakem

The purpose of this entry point is to take the list which A2 is pointing at and put its members on that stack. The calling sequence is just:

```
LMJ     XL,stakem
```

Registers A2, A3, and A4 will be changed by the operation, and the stack top pointer XT will also have a new value upon returning. This will be useful for dealing with argument handling in special forms, which is explained in the next section.

5.3.19. Expand a Special Form Call - Sexpand

Sexpand is the routine which is specified in H2 of the first word of a linker node for a user-defined special form. It is much like mexpand (entry point 14) in that it expects the arguments which were specified to the special form to be on a list on the top of the stack, and it expects XL to point at the second word of the appropriate linker node. It is somewhat simpler in that the second evaluation is not made. The code looks like so:

```
$mark                          . set cstak for call
L,H2        A2,0,XL            . get addr of func to call
$store                         . put it in new function frame
$load       1                  . get list of arguments in A2
LMJ         XL,stakem          . and put them on the stack
LXM,U       XL,exit            . entryr returns its value
J           entryr             . evaluate, return final value
```

5.3.20. Follow a Car-Cdr Chain - Follow

Follow is the routine which the input scanner places in the linker nodes it creates for atoms of the form c<0-35 a/d's>r. It expects a single argument to be on the stack and it expects XL to point at a word containing a 0-to-35-long bit string, prefixed with a 1 for counting purposes. It moves through the bits one by one, interpreting a 1 as a cdr and a 0 as a car, applying these functions to the single argument until only the 1 prefix is left, at which point it returns the expression it has retrieved. An attempt to take the car or cdr of an atom (non-cons node) through this routine will result in an error message, and the execution of either a user-defined (through carcon) contingency routine or an (error 0) contingency action (which will result in an ER err$ in batch mode, and a return to either a trap of error code 0 or the latest level of LISP supervision in demand mode). More details on the workings of routines related to these special c<a/d*>r atoms can be gotten from the section on the structure of linker nodes.

5.3.21. Apply S-Expression to Arguments - Apply

This is the place to which the interpreter is routed when it is time to apply a lambda expression to a set of arguments. It expects XL to point at the second word of a linker node which in turn should point at a list containing the arguments to lambda when the function was defined (that is, a single atom or list or extended list of atoms consed to a list of s-expressions), and XF and XT to be arranged as would be expected when a function is called. The code for apply then makes the appropriate bindings on the association list, branches to a contingency routine if there are not enough arguments on the stack, and finally sends

the list of s-expressions to the code for the function do.

5.3.22. Apply Function Closure to Arguments - Apply1

This entry point is identical in use to apply except that it is the entry point used by functions defined through the LISP function lambda (as opposed to using function and lambda). The purpose of this is to see to it that the function being defined is always run using the association list that was present when the function was defined, not the one present when it is executed. This entry point simply makes sure that the old association list is substituted for the current one and then it jumps to apply.

5.3.23. Establish a Trap Point - Trap

This is the routine which is used to declare error contingencies. It is called like so:

```
        LMJ     XL,trap
```

and expects A2 to point at either an s-expression such as is given as the second argument to the attempt function (that is, a list of n-tuples each of which has as its car an integer and which has as its cdr a list of s-expressions to be evaluated in order should an error contingency be processed with their number) or a trap chain, a construct more suited to the assembler user. A trap chain is basically a linked list of n-word structures. The first word of each structure contains the trap number (ie., error code) in H1 and 0 or a link to the rest of the trap chain in H2. This word should be followed by a number of instructions which process the contingency, possibly registering a new one themselves. LISP will jump to the first instruction after the trap number/chain link word if that numbered contingency is processed. The user need not worry if LISP sticks a jump instruction after the first word of such a trap chain structure, because if LISP jumps to that word, it will be correctly routed to the location of the actual routine. An example of a trap call and its associated trap chain would be as follows:

```
doit    . . .
        L,U     A2,trapch       . trap chain in A2
        LMJ     XL,trap         . process the trap
        J       continue        . continue processing
        form    pf,18,18        . half words
trapch  +       pf(1,err14)     . error code 1
        J       gfal            . just return nil in this case
badcase +       pf(13,syserr)   . unlucky case
        L,U     A4,0            . fake an (error 0)
        J       unwind          . see section on unwind
err14   +       pf(14,badcase)  .
        LMJ     XL,recover      . user-defined routine
```

```
          J         goit          . try again (carefully!),
syserr    +         pf(-3,0)      . system-defined error num
          ER        err$          . really crap out
recover   . . .
          . . .
          J         0,XL          . return from recovery
continue  . . .
          . . .
```

The trap routine does not change the value of any scratch register. The section on the unwind entry point discusses the mechanics of the processing of an error contingency.

5.3.24. Disconnect a Trap Point - Untrap

This entry point has the effect of undoing all the traps that have been set through trap in this function frame. It is called like so:

```
          LMJ       XL,untrap
```

5.3.25. Trace Path Back Down Stack - Unwind

The unwind entry point is the routine which processes an error contingency, as through the function error. If entry is at unwind, A4 should contain the number of the error. LISP has reserved meanings for various of the non-positive error numbers, but the user is free to work with the positive ones, as well as unwinding on a non-positive number from an assembled program. The call is just a simple jump to unwind:

```
          J         unwind        .
```

If entry is at unwind+1, the register A2 should contain the length of any printed backtrace (0 is no backtrace, 0777776 is full backtrace, and anywhere in between specifies the number of stack levels to be described on output.

What the unwind routine does is to follow a chain of pointers down the two stacks, looking for traps that have been set through entry point trap. When it finds an s-expression trap chain defined through attempt, it looks through the various trap numbers until it finds the number it is looking for. If it is found, the associated s-expressions are retrieved, the traps are removed, and the s-expressions are handed to the do function. The result of that evaluation is returned as the value of the function in whose function frame the traps were declared. If unwind finds an assembled trap chain, on the other hand, it follows the chain looking for the error number in H1 of one of the chain words. If none is found, the unwinding process continues. If a matching number is found, LISP just jumps to the

first word after the trap chain entry that matched (after removing all traps at that level), and the assembled code there processes the contingency, possibly re-establishing some traps or re-calling unwind.

5.3.26. The Current Gensym Number - Genno

This entry point is a pointer to a single memory location where the last integer that was used in creating a gensymmed atom is located. These atoms are implemented by having a print name that is an integer, and by being linked in the hash table just before the atom they are named after.

5.3.27. Create a New Special Form or Macro - Defspm

Defspm is a routine in LISP which allows the user to define special forms without going through defspec. It should be called as a routine (entry point entryr or macro $callr) with three arguments on the stack: 1) the atom to which the special form is to be bound; 2) the special form expansion routine, sexpand (or the macro expansion routine, mexpand, to define a macro); and 3) a pointer at the normal linker node which would have been returned by lambda during a normal definition through defspec or defmac. Registers XW (X8), A3, and A6 are trashed by the routine, which returns the atom's address in A2.

5.3.28. Temporary for Prog Result Storage - Pvsave

This one-word cell is the location used by the prog feature to store temporary results, especially go labels and returned values. The go and return functions are implemented by processing an error contingency which causes the stack to unwind to the most recent prog call (actually the most recent -1 and -2 code traps), which then retrieves the object being referred to from the pvsave word and, in the case of the return call, returns it as the value of the prog, and in the case of the go call, searches through the prog's arguments for the proper label, resuming evaluation at that point. The user of the assembler might want to use this entry point to intercept go and return calls to inspect and/or alter their results. More information on error contingencies and traps may be found in the discussions of the trap, untrap, and unwind entry points.

5.3.29. Make a String out of the Name Buffer - Makstr

The purpose of makstr is to make a string out of the contents of the name buffer (entry point 32) and return it in register A2. The calling sequence is:

```
LMJ     AC,makstr
```

It changes the values of the following registers: A0, A1, A2, A3, and A6.

5.3.30. Copy String into Name Buffer - Getnama

This is a routine which takes a string in A0 and translates it into sequential form in the buffer name (entry point 32). The calling sequence is:

```
LMJ XL,getnama
```

If A2 points at an atom, its print name can be translated by entering at getnama-1:

```
LMJ XL,getnama-1
```

In either case, registers A0, A1, and A4 are changed by the routine.

5.3.31. Blank-Fill Rest of Name Buffer - Blanks

After one has done a getnama call, the remainder of the name buffer may be full of unpredictable "trash". To change these all to blanks, one uses the blanks routine. It is called by:

```
LMJ XL,blanks
```

and it changes registers A0, A1, A3, and A4.

5.3.32. Sequential String Buffer - Name

The name buffer is used by routines like makstr, getnama, and blanks for translating strings from sequential to LISP format and vice versa. Its first word contains a pointer (absolute address, not relative) to the last word being used in the name buffer. The next 23 words are the buffer area itself. It is generally not safe to store material in the name buffer between function calls because many routines and functions use it. Anything which must be saved can be block-transfered out to a user's own buffer area.

5.4. Argument Handling

Once a function has been successfully set up using the LISP assembler, it can be used just like any other of LISP's functions. One thing this means is that a user of the new function can supply it with various arguments. This section describes how the function can handle these arguments.

5.4.1. Arguments to Regular Functions

If a new assembler function (say foobar) is defined using setq or csetq like so:

```
(csetq foobar (assemble program <opts>))
```

then it is a regular function, that is, LISP will evaluate its arguments for it. What the latter statement means to the assembler user is that if a regular function is called (say) like so:

```
(foobar 3 "ABCDEFG" '(a . b))
```

then the three numbers on the stack are: 1) the address of a location in integer space which contains the number 3, 2) the address of the first word in string space of the string "ABCDEFG", and 3) the address of a cons node whose left pointer is the address of the atom a and whose right pointer is the address of the atom B. The number 3 will not appear on the stack because nothing meaningful resides at location 3. So when the first argument is loaded into, say, A2, A2 will contain the address of a word containing 3 and not the number 3 itself. To get this number, index off of A2 like so:

```
LA      A2,0,A2
```

When such a function is entered, the stack is arranged as shown in the section on the stack entry points. All the arguments which were supplied to the function will have been evaluated, and their values placed on the stack. The arguments can be picked off the stack by indexing off the stack using the XF index register. For example, to load the third argument in A4, do:

```
L,H2    A4,stack-3,XF
```

and to load the last argument in AC, do:

```
L,H2    AC,stack+1,XT
```

Remember that the value stack grows down.

A macro, $load, is available with the assembler which generates a single instruction to load an arbitrary argument into A2, which is the standard register for such things. For example, to load the first argument into A2, do

```
($load 1)
```

which will generate the instruction

```
L,H2    A2,stack-1,XF
```

Of course, loading into A2 is not mandatory, if the user wishes to load an argument into another register, an instruction akin to the last one can be written out. The argument to load may be an expression involving EQU'ed symbols and so forth. Since $load is a special form, its argument will not be evaluated, so if some of the symbols are undefined when the macro is expanded, the expression will be passed on to the generated instruction. For example,

```
($load (myf \17n))
```

will return

```
(L A2 (difference \stack (myf \17n)) XF H2).
```

Often, the stack is used for more than receiving arguments. It is a good place to store temporary values, set up for various entry points and other routines, and provide special capabilities to the user of the function, such as optional arguments and indeterminate numbers of arguments. The mechanics of the first can be gotten from the discussions above of how the stack is laid out. Examples of the second are explained in the section on the entry points into the LISP interpreter. The third capability will be explained here.

Suppose one wants to define an assembler function which has one meaning when a second argumert is provided, and another meaning otherwise (such a real-life function is fiopen). If no second argument is specified by a calling function, only one entry will rest on the top of the stack. One way to see how many arguments are on the stack is to compute (XF)-(XT)-1. An easier way is this: put an extra "zero" argument on the stack by doing:

```
SZ      stack,*XT
```

and then load the second argument by doing:

```
L,H2    A2,stack-2,XF
```

If A2 then contains zero, then no second argument was specified. The "dummy" zero on the stack can then be removed if desired by incrementing the stack top pointer like so:

```
AX,U    XT,1            . remember stack grows down
```

The macro $pop of no arguments will generate this single instruction.

Many (especially arithmetic) functions allow the user to specify as many arguments as he wishes. When such a function is entered, all the arguments which were specified are on the stack. To process them all, the function can assign an index register with (XF)-1 in the modifier portion and -1 in the increment portion and use autoincrementation to move through the arguments

until the modifier is one greater than the contents of the modifier of the stack top register (XT). Another way is to use the listem entry point to put all the arguments in a list and then cdr down the list, processing each member of the list. As it happens, the list function just uses listem and returns.

5.4.2. Arguments for Special Forms

If a function is defined using defspec instead of cseig, then it is a special form and its arguments are not evaluated. The arguments to such functions are handled differently from those of regular functions in that when the function's code is entered, the stack has a single "argument" on it which is a list of all the arguments which were specified on the function call. If the user wants to have these arguments arranged on the stack one at a time like a regular function's arguments, the list can be loaded, $pop'ed, and the elements of the list can be put on the stack using the stakem entry point, all this before normal processing begins. Often, though, it is useful for a special form to have its arguments on a list rather than on a stack because many functions (such as and and or) like to cdr down the list of arguments until something of interest is found, possibly evaluating as they go. To evalutate an expression which is an argument to an assembled special form, the user can call eval using the method described in the section on calling LISP functions.

5.5. Returning a value

When an assembled function returns to LISP by jumping to entry point exit, the value of the function is taken to be the contents of register A2. Note that if the result is, say, 3, A2 should not contain 3 but a pointer to an integer node containing 3.

5.6. Calling LISP Functions

Often an assembler function may require the services of another LISP function, whether user-defined, system-defined, or created through the assembler. There are four steps involved in such a call: 1) initializing the stack for a call, 2) putting the arguments on the stack, 3) calling the function, and 4) retrieving the value. These are covered in roughly that order here.

To initialize the stack for the call, use the $lets macro, whose argument should be either a label which has been EQUated (directly or through sexp or fnames) to the address of the desired function. For example, to set up for a call to fitoc, one can do:

```
        fnames    fitoc          . somewhere in the program
        $lets     fitoc          . read "let's fitoc"!
```

This sets up the control stack and assigns new values to the function frame and stack top pointers (XF and XT), as well as setting up a new function frame. The mechanics of the $lets macro are rather complicated and will not be explained here. However, the expanded instructions will appear in any listing, and a clever assembler user can decode them if he wishes. As a general rule, though, one should do one's stack manipulating through macros and not attempt "optimizing" variations on them.

To place values on the stack, use the $store macro. It generates this instruction:

```
        SA        A2,stack,*XT
```

which puts the contents of A2 on top of the stack and bumps the stack top pointer. Of course, other registers can be used by writing out this instruction with other a fields.

The actual call on the function can be made in either of two ways, depending on how the result is to be returned, so we will discuss the call and the retrieval of the value together.

If the result of the function call is to be placed on the top of the stack like an extra argument to the calling function, the call should be done through the macro $call of no arguments which generates a single instruction,

```
        LMJ       XL,entry
```

as we will demonstrate in an example.

Example: To add 1 to the integer node A2 points at:

```
        fnames    add1           . get the address of add1
        $lets     add1           . macro to set up stack
        $store                   . macro to push A2 on stack
        $call                    . generates LMJ to entry
        L,H2      A2,stack+1,XT  . get top thing on stack
        $pop                     . pop that result
```

In reality, it would probably have been easier to do the adding without calling add1. A further explanation of the macros used here will have to await a future section.

If the call is made through the $callr macro, on the other hand, the result will be returned in A2. Technically, the function is being called as a "routine". This method of calling is useful when the value is going to be immediately used in computations, or when the function is being called just for its effect rather than for its value. The $callr macro generates the single instruction

```
        LMJ     XL,entryr
```

as the example will demonstrate. We will use the same example as before, adding 1 to the integer node which A2 points at.

Example. Add 1 to the object A2 points at.

```
        fnames  add1            . get the address of add1
        $lets   add1            . get stack ready
        $store                  . put A2 on stack
        $callr                  . do it, result is in A2
```

A couple of notes should be made concerning the calling of LISP functions from assembled code. First, the called function will usually use a number of registers to do its work, and if the called function is a lambda expression or compiled function, there is usually no telling which of the index and "scratch" registers will be trashed. For this reason, some saving of registers may be helpful and necessary when calling functions. Second, it is usually not a good idea to be calling many LISP functions from assembled code because of the overhead in terms of time, instructions, and confusion. Usually, a lot of function calling can be avoided by designing assembler functions to work as helpers of larger functions defined through lambda. The purpose of each assembler function should be restricted enough so that any calling of other LISP functions can be done through the main function rather than through the assembled function. Of course, all this goes out the window in large, costly LISP programs in which efficiency of time and memory use is at a premium.

5.7. Macros for Interfacing with LISP

Several macros are provided with the assembler for the use of interfacing with LISP which are LISP versions of assembler procs used by the LISP interpreter's source code. These macros are those prefixed by a dollar sign ("$"). Several of these were described in the various sections above where they were relevant. These were $load, $store, $move, $call, $callr, $return, $mark, $lets, $getype, and $pop. The rest are more general-purpose and are included for the convenience of the programmer.

5.7.1. Get Car of a Word - $upper

The $upper macro takes one argument, the name of a register, and generates an instruction to load the magnitude of the (sign-extended) upper 18 bits of the word A2 points at into that register. This is useful for working with cons nodes, atoms, and the like. For example,

```
        ($upper A5)
```

generates

(L A5 0 A2 H1) .

5.7.2. Get Cdr of a Word - $lower

The $lower macro is similar to the $upper macro, only it loads the lower 18 bits of the word A2 points at in a sign-extended magnitude fashion into the desired register. For example,

($lower A5)

generates

(L A5 0 A2 H2) .

5.7.3. Follow a Chain of Pointers - $chain

Often an assembler function will need to follow an arbitrary sequence of pointers to find a desired object. The $chain macro will help do this. It takes an arbitrary number of arguments, each hopefully H1 or H2, and generates an instruction for each loading that part of the word A2 points at (sign-extended magnitude, in case the pointer happens to be negated) into A2. For example,

($chain H1 H2 H1 H2 H2)

generates five instructions as follows:

```
L,H1    A2,0,A2
L,H2    A2,0,A2
L,H1    A2,0,A2
L,H2    A2,0,A2
L,H2    A2,0,A2
```

which will take the cdadadar of whatever A2 points at. Of course, if the object A2 points at does not have a cdadadar, a guard mode violation will most likely occur.

5.7.4. Allocate LISP Object Nodes - $node

A common purpose of a LISP function is to create a LISP object such as a string, s-expression, or integer node, and return it or an s-expression involving it as its value. The macro $node can be used to allocate a single node of the desired type for this purpose. This macro takes a single argument which is either the number of the type (eg., 0=consed node and 8=string) or a symbol whose symbol table value is the number of the type (eg., consed, integer, or linker). The macro generates a single instruction

```
LMJ     XL,$storit(<type>)
```

whose effect is to allocate a node of the desired type, initialize it, and return its address in A2. Although the information to be stored in the node need not be specified in advance, the values in the various fields of the new nodes will be initialized as follows: For cons nodes, A2 should contain the car and A3 the cdr. For string nodes, A2 should contain the three characters which are to be in the new node and A3 should contain the address of the next string node in the chain if its location is known. For all numeric types (integers, octals, and reals), A3 should contain the number to be stored in the node. For linker nodes, A2 should contain whatever is to be in the second word of the node and A3 should contain the address to be put in H2 of the first word (apply, sexpand, etc.). The details of allocating code nodes and atomic symbols are much more complex and can be gotten from a real LISP hacker. No initialization is necessary for buffer pages, and it makes no sense to allocate a type 4 (unallocated node page or system code) node.

In addition to the fact that the allocation routines return their results in A2, several other registers have their values trashed by the routines as follows: Allocation of integers, octals, and reals trashes A6, allocation of cons nodes, string nodes, and buffers trashes A3 and A6, allocation of compiled code nodes trashes A0, and allocation of linker nodes and atomic symbols trashes XW (X8), A3, and A6.

6. Programming Advice and Notes

6.1. Concerning Large Integers

If an assembled program is to be dumped, the u field of most every instruction is treated as a pointer at some LISP object. Any such pointer is subject to modification when the function is loaded into LISP after being dumped. Numbers which happen to be the locations of unallocated nodes or system code (system-defined functions, entry points, low addresses, etc.) are treated as non-relocatable. The trouble occurs when an integer in the u field of some instruction happens to be the location of some (possibly huge) LISP object such as an s-expression. An example would be:

```
L,U     A3,516030      . or its integer equivalent
```

at a time when the object residing at location octal 051603 is a tremendously large s-expression (or even worse, part of the available nodes linked list for some node type). When the function containing that instruction is dumped, the number is relocated and the object it points at is dumped also. Then when the function is re-loaded, not only will the instruction be different, but a number of atoms may have their values changed and other bad things may happen. The solution to this problem is to use address to (effectively) allocate an octal node for the number like so:

```
L       A3,address(516030)
```

If it is done this way, what will be dumped will be an octal word containing 516030. Its address will be relocated, but the instruction will work the same as before dumping.

6.2. More Load/Dump Side-Effects

When dump is used to copy out an assembler program, all atomic symbols which the program's code points at are also dumped, along with their values and property lists. Thus, when the program is loaded, the old values and property lists of these atoms are restored, erasing any values which may have been current before the load. For this reason, the user should make sure that all such atoms have the desired values at the time dumping occurs so as to avoid both "phantom side effects" and the hauling around of (possibly large) unwanted s-expressions.

6.3. Space Restrictions

Since a UNIVAC 1100 series machine instruction has only 16 bits in its address field (in general), only 2^16 or 0200000

octal words may be addressed by the instructions. For this reason, no instruction may point at any address above this upper limit. The assembler has two ways to protect itself from the possibility of this limit being violated. First, before any code is generated, it does a (grow) to see how much core is available to it. If the maximum address returned is at or above this 02000000 mark, it aborts and requests that the user not take up so much core. Also, when instructions are being generated, the addresses at which the new instructions are being created at checked against the maximum and a warning message is printed if the address is too high.

6.4. Support Routines

6.4.1. The LISP Dynamic Dumper

This is a set of functions originally designed to aid in debugging assembled programs. They allow the user to selectively print out (in any format) the contents of chunks of core (and modify them, too), and allow non-standard type conversions. They are described in a document whose name corresponds to the name of this section.

6.4.2. Asm-excise

The sheer size of the assembler will often cause problems for the user. Because of this there is a function, asm-excise, of no arguments which removes all trace of the assembler from core. It is recommended that this be done before any dumping of newly-assembled functions is attempted (why this should be necessary has yet to be determined).

6.4.3. Asm-prettyp

There is a function loaded with the assembler called asm-prettyp which takes as its single argument an assembler program (list of directives, macro calls, instructions, and labels) and produces a formatted listing without attempting any interpretation or processing (eg., macro calls are not expanded). This is much faster than getting a listing from the assembler itself.

Appendices

Contents

1. Virtual Memory Option 236
1.1. Using Virtual Memory 236
1.2. Allocation Scheme 236
2. Character Sets 237

1. Virtual Memory Option

When the V option is specified on the call to Maryland LISP on the UNIVAC 1100/40 at the University of Maryland, LISP will use a form of virtual memory which is the subject of this appendix. A useful reference for this discussion is [Uni78], especially Volume 2, page 3-21.

1.1. Using Virtual Memory

When LISP is loaded, it extends itself to 32K words of memory, of which approximately 20K is data area. LISP's core map is broken into several 16K areas known as banks, where the n-th bank starts at location 16K x (n+1), the zeroeth bank is about 9K, and the -1st bank, containing LISP's static instruction and data areas, as well as compiled code and other atomic items, occupies the first 23K (approximately).

To allow more banks to be used as data area, the GROW function is called with an argument equal to the number of banks to be added. Up to six total banks may be added in the course of a run. The banks are automatically swapped in when they are referenced by the user program or by the LISP system, in the same way that a paging system works. The swapping algorithm requires approximately 300 instructions of UNIVAC microcode to perform each swap. Therefore, since an average run requiring more than 32K might execute one million swaps, the virtual memory option is only convenient during light load hours such as evenings and weekends. (The memory time used by a program using virtual memory is greater than one not using it by a factor of between 1.5 and 5.) The number of bank swaps executed so far in the LISP session is returned by a call on (SWAPS). This is useful for keeping statistics of the form swaps per second of run time.

1.2. Allocation Scheme

When extra banks have been added to LISP, the storage allocation mechanism tries to place cons nodes in the dynamic (swapped) areas, and atoms in the static (-1 bank) area. Compiled code must go in the static area, and the :CODE directive is used to reserve space for it. The algorithm also is biased in favor of the low-numbered banks, always trying to allocate storage in bank n-1 before bank n. This serves to keep the average density of used nodes in the lower banks higher and thus to keep the banks packed.

A compactifying garbage collector and a scheme to trigger a garbage collection before trying to allocate from a new bank would be useful and are planned, but have not yet been implemented.

2. Character Sets

This is a table of UNIVAC's FIELDATA character set. This character set has no lower-case letters or control codes. All these characters translate into the same characters in ASCII. The character codes in this table are octal numbers.

00	@	13	F	26	Q	41	-	54	?	66	6
01	[	14	G	27	R	42	+	55	!	67	7
02	]	15	H	30	S	43	<	56	,	70	8
03	#	16	I	31	T	44	=	57	\	71	9
04	^	17	J	32	U	45	>	60	0	72	'
05		20	K	33	V	46	&	61	1	73	;
06	A	21	L	34	W	47	$	62	2	74	/
07	B	22	M	35	X	50	*	63	3	75	.
10	C	23	N	36	Y	51	(	64	4	76	"
11	D	24	O	37	Z	52	%	65	5	77	_
12	E	25	P	40	)	53	:				

This is the ASCII character set, which is used by the AXMIT, AXMIT1, and AREAD functions, and by many other processors on the UNIVAC system. The codes 000 through 037 are control codes, the meanings of which are usually device-dependent.

000	NUL	032	SUB	064	4	116	N	147	g		
001	SOH	033	ESC	065	5	117	O	150	h		
002	STX	034	FS	066	6	120	P	151	i		
003	ETX	035	GS	067	7	121	Q	152	j		
004	EOT	036	RS	070	8	122	R	153	k		
005	ENQ	037	US	071	9	123	S	154	l		
006	ACK	040		072	:	124	T	155	m		
007	BEL	041	!	073	;	125	U	156	n		
010	BS	042	"	074	<	126	V	157	o		
011	HT	043	#	075	=	127	W	160	p		
012	LF	044	$	076	>	130	X	161	q		
013	VT	045	%	077	?	131	Y	162	r		
014	FF	046	&	100	@	132	Z	163	s		
015	CR	047	'	101	A	133	[	164	t		
016	SO	050	(	102	B	134	\	165	u		
017	SI	051	)	103	C	135	]	166	v		
020	DLE	052	*	104	D	136	^	167	w		
021	DC1	053	+	105	E	137		170	x		
022	DC2	054	,	106	F	140	~	171	y		
023	DC3	055	-	107	G	141	a	172	z		
024	DC4	056	.	110	H	142	b	173	{		
025	NAK	057	/	111	I	143	c	174	|		
026	SYN	060	0	112	J	144	d	175	}		
027	ETB	061	1	113	K	145	e	176	~		
030	CAN	062	2	114	L	146	f	177	DEL		
031	EM	063	3	115	M						