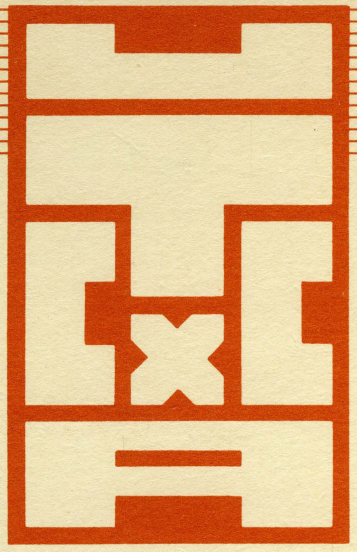
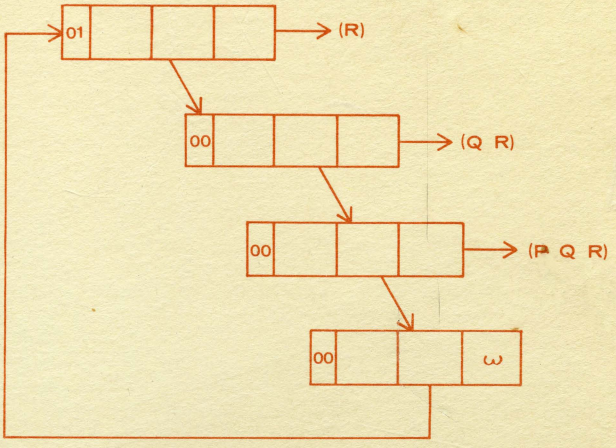


THE UNIVERSITY OF TEXAS
AT AUSTIN

*Computation Center*

# LISP Reference Manual
# CDC - 6000




CCUM 2
DEC 1975

# LISP Reference Manual

## CDC - 6000

Computation Center
University of Texas at Austin
Austin, Texas 78712

The Computation Center will appreciate any comments about this manual which may be used to improve its usefulness in later editions. In comments concerning errors, additions, deletions, or omissions, please include the date of the manual and, for each item, the number of the page involved. Comments should be addressed to: Editor, Computation Center, University of Texas at Austin, Austin, Texas 78712.

## Acknowledgements

This manual is extensively based on the manual written by Dr. E. M. Greenawalt for the earlier version of the UT LISP system. The Computation Center expresses its appreciation to Robert A. Amsler, Jonathon Slocum, and Mabry Tyson for their assistance in revising the manual to reflect the current LISP system and for producing a machine-readable preliminary form of the the manual.

TABLE OF CONTENTS

## FIGURES



## TABLES

# 1. INTRODUCTION

This manual is a reference document for UT LISP, the implementation of the LISP programming language developed at The University of Texas at Austin for Control Data Corporation 6000, 7000, and CYBER 70 series computers. UT LISP is available at many installations of these classes of machines. UT LISP is not necessarily compatible with LISP implementations existing on other types of computers. UT LISP is, however, a very rich implementation and provides very powerful tools for the LISP programmer.

This implementation is an interpretive system. A compiler is separately available which can improve the speed of production-type LISP programs (see chapter 6). UT LISP is available to users in both batch and conversational modes of operation.

The LISP programming language originated about 1960 when its formalism and first implementation were developed at M.I.T. by John McCarthy and others [1,2]. Its fundamental construct is the recursive function applied to data structures organized as lists. Although highly formal in structure, LISP is a very effective programming language for applications involving structured non-numeric data bases and processes employing non-algorithmic, heuristic methods. It is by no means restricted to these application areas; it is a fully general programming language. The most widespread use of LISP has been in artificial intelligence research, symbolic algebraic manipulation systems; and computer-assisted instruction.

This manual is not a primer on the LISP language. It documents the actual behavior of the various facilities available in UT LISP. It is recommended that the beginning LISP programmer study one of references 3, 4, 5, or 6 (or any other suitable introductory text) before reading beyond chapter 2 of this manual. Reference 7, though containing more advanced material, also has some elementary LISP exercises. Chapter 2 gives information about using the UT LISP system. Final definition of the behavior of any LISP function mentioned in the referenced introductory texts resides in the descriptions given in chapters 4 and 5.

Beginners should not be concerned over the large number of functions available in this implementation, since most are conveniences rather than essentials for LISP programming. The novice user can write complete programs using only the following functions:

a. the 4 general functions: CAR, CDR, COND, and CONS,

b. the 3 arithmetic functions: PLUS (+), TIMES (*), and DIFFERENCE (-),

c. the 3 predicates: ATOM, NUMBERP and EQ,

d. the 2 debugging aids: TRACE and UNTRACE, and

e. the 2 I/O routines: READ and PRINT.

Together with a knowledge of the system atoms FIN, F, LAMBDA, NIL, and T, these 14 functions enable one to re-create such additional system functions as GREATERP, EQUAL, LESSP, LENGTH,

APPEND, MEMBER, NULL, REVERSE, MAPCAR, MAPC, MAPLIST, MAP, and SUBST (a task which would be an excellent practice exercise).

Beginners should first master the above functions and then study the slightly more advanced routines: SETQ, SET, PROG, RETURN, GO, PUT, GET, and the additional debugging aids TRACESET and UNTRACESET.

Learning the functions in approximately this order will provide sufficient mastery of the basics of LISP to support one's confidence while digesting the other 160 functions and 100 atoms listed in the appendixes.

References:

1. McCarthy, J., "Recursive Functions of Symbolic Expressions and Their Computation by Machine, Part 1", Comm. of the ACM 3, April 1960, pp. 184-195.

2. McCarthy, J., Abrahams, P. W., Edwards, D. J., Hart, T. P., and Levin, M. I., LISP 1.5 Programmer's Manual (M.I.T. Press, Cambridge, Massachusetts) 1962.

3. Weissman, C., LISP 1.5 Primer (Dickenson Publishing Company, Inc., Belmont, California) 1967.

4. Friedman, D. P., The Little LISPer (Science Research Associates, Menlo Park, California) 1974.

5. Siklossy, L., Let's Talk LISP (Prentice-Hall, Inc., Englewood Cliffs, New Jersey) 1975.

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

7. Berkeley, E. C., Bobrow, D. C., The Programming Language LISP: Its Operation and Applications (M.I.T. Press, Cambridge, Massachusetts) 1966.

# 2. USING THE LISP SYSTEM

The UT LISP system is accessible by means of a control command. This chapter describes the LISP control command and the formats of input to and output from the UT LISP system. It is assumed in the following that the reader is somewhat familiar with LISP and its form.

## 2.1 THE LISP CONTROL COMMAND

The LISP control command is of the form:

```
LISP,A=<n1>,B=<n2>,C,E,F,G,I=<filename>,K=<n3>,L,N,
     O=<filename2>,P,S=<filename3>,T,X=<n4>,Y,Z,
     <subsystem1>,<subsystem2>,...,<subsystemn>.
```

where the parameters are described below.

| Parameter | Value | Meaning |
|---|---|---|
| A | A=<n1> | Allocate <n1> percent of available space to free space and the remainder to full-word space. <n1> must be in the range from 10 to 96. Available space is defined as the total memory size minus the size of the LISP interpreter minus the size of the stack minus initial binary program space. If the S keyword parameter is used to initialize LISP by loading an overlay previously created by the LISP function DEFSYS, the A keyword parameter is overridden by the free space allocation present when the overlay was created. |
| | A | Not legal. |
| | missing | Allocate 92 percent of available space to free space and 8 percent to full-word space. |
| B | B=<n2> | Set the stack size to <n2> words long. If the S keyword parameter is used to initialize LISP by loading an overlay previously created by the LISP function DEFSYS, the B keyword parameter is overridden by the stack size present when the overlay was created. |
| | B | Not legal. |
| | missing | Set the stack size to 1000 words long. |
| C | C | Set conversational mode of operation for LISP. LISP normal input (SYSIN) is read from file TTY. User-originated output (WRS) and LISP system output (SYSOUT) are listed on file TTY using an output |

| | | |
|---|---|---|
| | | line length of 70 characters. If the C keyword parameter is specified, the I, L, O, and P keyword parameters should not be specified. |
| | missing | Set batch mode of operation for LISP. LISP normal input (SYSIN) is read from the file specified by the I keyword parameter. User-originated output (WRS) and LISP system output (SYSOUT) are listed on the file specified by the O keyword parameter with an output line length of 132 characters. |
| E | E | Set expert mode. Expert mode allows primitive operations on atoms.<br>WARNING: Only users who really know what they are doing should set expert mode since this mode disables error detection of primitive operations on atoms. |
| | missing | Do not allow primitive operations on atoms, and enable all LISP error detection facilities. |
| F | F | Terminate the LISP run when any error is encountered. |
| | missing | Try to recover and continue running at the top level after an error is encountered. |
| G | G | Print a message each time a garbage collection occurs. The G keyword parameter does not affect the time when the garbage collection occurs. |
| | missing | Do not print a message each time a garbage collection occurs. |
| I | I=<filename1> | Read LISP normal input (SYSIN) from file <filename1>. If the I keyword parameter is specified, the C keyword parameter should not be specified. |
| | I=0 (zero) | Not legal. |
| | I | Not legal. |
| | missing | Read LISP normal input (SYSIN) from file INPUT, or, if the C keyword parameter is specified, from file TTY. |
| K | K=<n3> | Request that the job field length be changed to <n3> words. LISP requires a field length of at least 35000 (octal). If the S keyword parameter is used to initialize LISP by loading an overlay previously created by the LISP function DEFSYS, the K keyword parameter is overridden by the job field length present when the overlay was created. |

| | | |
|---|---|---|
| | K | Not legal. |
| | missing | Do not change the job field length. LISP uses the current job field length. |
| L | L | List LISP source statements on the LISP system output (SYSOUT) file determined by the O keyword parameter. If the L keyword parameter is specified, the C, N, and P keyword parameters should not be specified. |
| | missing | Suppress listing of LISP source statements unless the P keyword parameter is specified. If none of the keyword parameters L, N, or P is specified, the LISP source statements are printed only in internal format as S-expressions. |
| N | N | List only user-originated output (WRS) and LISP error messages (SYSOUT) on the file determined by the C or O keyword parameters. All LISP system-generated output except for error messages is suppressed. |
| | missing | Do not suppress listing of LISP system-generated output. |
| O | O=<filename2> | List user-generated output (WRS) and LISP system output (SYSOUT) on file <filename2> using an output line length of 132 characters. If the O keyword parameter is specified, the C keyword parameter should not be specified. |
| | O=0 (zero) | Not legal. |
| | O | Not legal. |
| | missing | List user-generated output (WRS) LISP system output (SYSOUT) on file OUTPUT using an output line length of 132 characters, or, if the C keyword parameter is specified, on file TTY using an output line length of 70 characters. |
| P | P | List LISP source statements and a parenthesis level count under each parenthesis on the LISP system output (SYSOUT) file determined by the O keyword parameter. If the P keyword parameter is specified, the C, L, and N keyword parameters should not be specified. |
| | missing | Suppress listing of LISP source statements unless the L keyword parameter is specified. If none of the keyword parameters L, N, or P is specified, the LISP source statements are printed only in internal format as S-expressions. |
| S | S=<filename3> | Initialize LISP by loading from file <filename3> the overlay previously created by the LISP |

| | | |
|---|---|---|
| | | function DEFSYS. <filename3> must be a disk-resident local file. If the S keyword parameter is used to initialize LISP by loading an overlay, the A, B, K, and X keyword parameters should not be specified since all storage allocation controls (free space allocation, stack size, job field length, and binary program space) are reset to the values present when the overlay was created. In addition, if the S keyword parameter is specified, no LISP subsystems should be specified. |
| | S=0 (zero) | Not legal. |
| | S | Not legal. |
| | missing | Initialize LISP with the standard functions and atom values given in Appendix A. |
| T | T | Print a timing message of the form *TIME: <number> after each evaluation of a top-level expression, where <number> is the time in milliseconds required to evaluate the expression. |
| | missing | Do not print a timing message after each evaluation of a top-level expression. |
| X | X=<n4> | Allocate <n4> words at initialization to binary program space. If the S keyword parameter is used to initialize LISP by loading an overlay previously created by the LISP function DEFSYS, the X keyword parameter is overridden by the size of the binary program space present when the overlay was created. |
| | X | Not legal. |
| | missing | Allocate no words at initialization to binary program space. |
| Y | Y | Insert a blank line before each *EVAL:,*VALUE:, *TIME: or trace output for a more readable listing. |
| | missing | Do not insert a blank line before each *EVAL:, *VALUE:, *TIME:, or trace output. |
| Z | Z | Simulate an interrupt when any LISP-detected error is encountered. The C keyword parameter must also be specified. |
| | missing | Do not simulate an interrupt when any LISP-detected error is encountered. |
| #i | <subsystemi> | Read the LISP subsystem on local file <subsystemi> and evaluate the expressions in it. No more than 10 LISP subsystems may be specified. The name of the LISP subsystem may not be a single letter. If any |

| | |
|---|---|
| | subsystem is specified, the S keyword parameter should not be specified. |
| null | Not legal. |
| missing | Do not read a LISP subsystem. |

The following list summarizes these parameters alphabetically by keyword letter:

| Parameter | Function |
|---|---|
| A=<number> | Allocation control |
| B=<number> | Stack size specification |
| C | Conversational mode control |
| E | Expert mode control |
| F | Error fatality control |
| G | Garbage collector message control |
| I=<filename> | Input file control |
| K=<number> | Field length control |
| L | Listing control |
| N | Listing control |
| O=<filename> | Output file control |
| P | Listing control |
| S=<filename> | Initialization control |
| T | Timing message control |
| X=<number> | Binary program space allocation |
| Y | Output spacing control |
| Z | Error interrupt control |

## Notes on LISP Parameters

The parameters described above may appear in any order, but no parameter should appear more than once. If an invalid parameter or combination of parameters is specified, LISP aborts after issuing the DAYFILE message

PARAMETER ERROR <letter>

where <letter> designates the first invalid parameter.

All parameters which require a numerical value allow either an unsigned decimal integer or an unsigned octal integer. Octal numbers are written as a string of octal digits followed by Q, B, or K. Octal numbers suffixed by K are multiplied by 1000 (octal); for example, 5K is the same as 5000Q.

The allocation control (A) keyword parameter is used to vary the relative sizes of free space and full-word space in order to fit the working storage of LISP to the program at hand. For instance, a program which deals with many numbers needs relatively more full-word space than a program whose main purpose is the manipulation of lists. If one relies on the default 8 percent of total available space allocated to full-word space, then the size of total memory necessary to get enough full-word space may be quite large and there may be much unused free space. An A parameter specification less than 92 allocates relatively more full-word space to the program, which may then be able to run in a smaller total memory than otherwise. To determine whether an adjustment of the allocation control parameter is advisable, the user should consult the output from a previous run. At the end of every run LISP prints a summary of how many

times the garbage collector was called when free space was exhausted and how many times it was called when full-word space was exhausted. If the number of times it was called because of full-word space exhaustion is significantly larger than the number of times it was called because of free space exhaustion, then the allocation control parameter should be decreased, otherwise the parameter should be increased.

The stack size specification (B) keyword parameter is used to vary the size of the stack in the LISP system. Increasing the stack size increases the number of levels of recursion which are allowed in the LISP program. Increasing the stack size decreases the amount of free and full-word space available in the same field length.

The binary program space allocation (X) keyword parameter is used to allocate memory space for the storage of compiled LISP code (see chapter 6) and for arrays (see section 4.13). If no binary program space has been allocated, LISP automatically requests more memory from the operating system. If the user knows, however, approximately how many words of binary program space are needed, he can preallocate the space using this parameter and thereby save the overhead incurred when LISP requests more space from the operating system.

Both the initialization control (S) and subsystem load control parameters offer mechanisms for the user to initialize the LISP system to some set of functions and atom values other than the standard ones given in appendix A. An overlay loaded by the initialization control parameter is the result of a previous LISP run and consists of all free space and full-word space that existed at the time in that run when the DEFSYS function was called. The initialization is performed very quickly via direct reading of the file into memory. A LISP subsystem has a name which is the name of the file on which it resides. The subsystem consists of a series of expressions which are automatically read by LISP and evaluated before any of the user's expressions are executed. Appendix B discusses subsystems. The overlay initialization procedure is very fast, but it is inflexible in the sense that the memory allocation and total memory size cannot be varied from that which was in effect at the time the overlay was defined. On the other hand, a LISP subsystem can be loaded into any size of memory under any allocation parameter set that will accommodate it. The disadvantage of the subsystem is that it requires processing (i.e., extra time) by the LISP system each time it is loaded.

When expert mode control (E) is specified, the user can perform CAR, CDR, CSR, RPLACA, RPLACD, and RPLACS operations on atoms. Normally, these operations are prohibited by LISP and cause an ILLEGAL ARGUMENT error. Improper use of these operations on literal atoms may cause changes in information vital to the operation of the LISP system, and may cause the program to fail catastrophically. Therefore, only the really expert user who has some valid reason for performing these operations should ever specify this parameter.

The actions specified by several of the control command parameters may be turned on and off dynamically under user control during the course of a run. The listing control is described in section 4.14. The garbage collector message control, timing message control, and expert mode control parameters may all be effected by means described in section 4.12. Input file control and output file control functions are

described in section 5.1.1. Error interrupt control may be handled by means described in section 9.2.

LISP requires a field length of at least 35000 (octal). All of the memory available to the job at the time LISP is requested is used by the LISP system. Therefore, no other programs (such as compiled FORTRAN code) may be coresident with the LISP system. Also, the total amount of working storage available to the LISP system can be controlled by changing the field length of the job. A field length of 50000 (octal) is recommended as a minimum practical size of memory in which to run the UT LISP system.

### Examples of LISP Commands

```
LISP.  is equivalent to
   LISP,A=92,B=1000,I=INPUT,O=OUTPUT,X=0.

LISP,F,G,P,T.  is equivalent to
   LISP,A=92,B=1000,F,G,I=INPUT,O=OUTPUT,P,T,X=0.

LISP,C,G.  is equivalent to
   LISP,A=92,B=1000,C,G,X=0.

LISP,A=80,B=1800,E,K=77000B,X=10100B,LAP,LCOMP.  is equivalent to
   LISP,A=80,B=1800,E,I=INPUT,K=77000B,O=OUTPUT,X=10100B,
        LAP,LCOMP.
```

## 2.2 INPUT FORMAT FOR UT LISP

A LISP program is usually composed of a simple sequence of LISP expressions, or forms, to be evaluated. Each form consists of a function name or lambda expression followed by a sequence of argument expressions, entirely enclosed in parentheses. At most one expression is allowed on a line. Program execution is handled by the function EVAL, the usual "top-level function". EVAL is called to evaluate successive input expressions in the order they appear. For example:

```
(EQ (QUOTE A)  (QUOTE B)
```

is a proper expression for EVAL. Note that constant items in the expression must be quoted to prevent their evaluation.

EVALQUOTE may be selected as the top-level function only by setting the //MODE variable to (EVALQUOTE . 2) (see section 4.14).

Execution terminates when an end-of-file occurs on the normal input file (SYSIN) or when a line whose first atom is FIN is read. Logical record marks on any file read by LISP are ignored.

LISP functions and data may be placed in free format on input lines in columns 1-72. The only restriction is that an atom may not be split between two successive lines. For details on the syntax of literal and numeric atoms the user is referred to section 3.1.

Any data that is to be read during evaluation of an expression must immediately follow that expression on the normal input file unless it is to be read from a different file. Figure 2.1 shows a representative input deck ready to be processed by LISP.

Figure 2.1 - LISP Input Deck

```
ABCD123,JOHN DOE.
XXX=PASSWORD.
JOB,TM=10,PR=10.
LISP,P,T,F,
7/8/9 - <end-of-record card>
% THIS FUNCTION RETURNS A LIST WHICH CONTAINS ALL
   THE ELEMENTS PRESENT IN BOTH OF ITS ARGUMENT LISTS%
(DEFINE "(
(INTERSECTION(LAMBDA(X Y)
   (COND((OR(NULL X)(NULL Y))NIL)
         ((MEMBER (CAR X)Y)
          (CONS (CAR X)(INTERSECTION(CDR X)Y)))
         (T(INTERSECTION(CDR X)Y)) )))
))
(INTERSECTION "(A B C) "(A D E))
(INTERSECTION "(X Z T Z) "(Z Z))
FIN
6/7/8/9  - <end-of-file card>
```

Certain characters have special meaning when encountered in LISP input. They are intended to make typing input data less tedious.

| Character (display code) | Meaning |
|---|---|
| " (60B) | This character causes the S-expression immediately following it to be quoted, e.g., "(A) is represented internally as (QUOTE (A)) and prints as (QUOTE (A)). |
| # (64B) | This character causes LISP to accept the immediately next character as part of a literal atom even if it is not normally allowed, e.g., A#.B is the atom A.B . |
| [ (61B) | This character is equivalent to a left parenthesis, but marks a closure point for a following ] character, e.g., [A) is equivalent to (A). |
| ] (62B) | This character is equivalent to an arbitrary number of right parentheses sufficient to match all left parentheses up to and including the most recent unmatched [ character. If no [ character precedes the ], all left parentheses remaining unmatched in the S-expression are matched by ], e.g., ((((A] is equivalent to ((((A)))). |
| % ... % (71B) | Any characters between a pair of % symbols will be treated as comments by the UT LISP system. These include all the special characters above. Comments may extend over an indefinite number of lines, but the closing % , must be within columns 1-72. A character may not precede the opening % symbol, e.g., #%.....% is not a legal comment, but %.....#% is. The LISP input routines discard any comments delimited by % |

symbols. To retain comments within defined LISP code the COMMENT function should be used (see section 4.3).

%% While not a single character, two immediately adjacent % characters have the meaning that the S-expression currently being input should be discarded. LISP will request input again. This capability is of primary benefit to the conversational user who may need to discard an erroneous S-expression which extends over several lines.

Also, if EVALQUOTE is being used as the top-level function, the occurrence of the atom STOP as the second half of a doublet cancels the current evaluation. This feature is of greatest value to conversational users who need to recover from a typing error in the first half of the doublet.

## 2.3 LISP SYSTEM OUTPUT

When LISP begins execution, it first sends a version message of the form:

UT LISP - VER. <number> (<date>)

to the job's DAYFILE. Here <number> is the version number of the LISP system and <date> is the date it was assembled. These items allow the user to determine which version of LISP he is using.

Following the printing of the version message, LISP begins execution of the user program.

For each expression executed, LISP produces output to inform the user what expression was evaluated and what result was obtained. At the beginning of each evaluation LISP prints a line containing

*EVAL:

or whatever is appropriate for the top-level function in use. This line is a signal that the top-level function has begun operation. (In conversational mode this is a signal that LISP is ready for the user to enter new information.) Following this line, the expression which is currently being evaluated is printed. If a listing control parameter is present on the control card, then an exact image of what was read on the source lines is printed. If listing control is not on, then the expression is printed by the normal LISP printing function. Expressions whose function is DEFINE are normally supressed if listing control is not on, since they are frequent and voluminous, and the user should be acquainted with their content already. Next LISP prints a line containing

*VALUE:

This line signals the completion of evaluation and is followed by the result of the evaluation. If the T control command parameter was specified, the timing message is then printed. Then LISP begins the sequence of output for the next evaluation. Any printed output produced by the evaluation itself precedes the *VALUE: message. Figure 2.2 shows as an example the output produced by the input deck shown in figure 2.1.

If the output suppression parameter (N) had been supplied on the control command, none of the output of figure 2.2 would appear. Only user-originated output and/or system error messages would appear in that case.

Figure 2.2 Example of LISP Output

```
 *EVAL:
% THIS FUNCTION RETURNS A LIST WHICH CONTAINS ALL
   THE ELEMENTS PRESENT IN BOTH OF ITS ARGUMENT LISTS%

 (DEFINE "(
 0        1
 (INTERSECTION(LAMBDA(X Y)
 2             3      4   4
    (COND((OR(NULL X)(NULL Y))NIL)
    4     56  7       77      76   5
          ((MEMBER (CAR X)Y)
          56       7      7 6
           (CONS (CAR X)(INTERSECTION(CDR X)Y)))
           6     7      77              8     8 765
          (T(INTERSECTION(CDR X)Y)) )))
          5 6              7      7 65 432
 ))
 10

 *VALUE:
 (INTERSECTION)

 *TIME: 11

 *EVAL:
 (INTERSECTION "(A B C) "(A D E))
 0              1     1  1     10

 *VALUE:
 (A)

 *TIME: 8

 *EVAL:
 (INTERSECTION "(X Z T Z) "(Z Z))
 0              1       1  1   10

 *VALUE:
 (Z Z)

 *TIME: 8

 *EVAL:

 GARBAGE COLLECTIONS: 0 0
```

# 3. DATA FORMATS

The LISP programmer needs to be aware of the formats of data treated by LISP. This chapter describes the external and internal representations of LISP data.

## 3.1 INPUT FORMATS

Programs and data for LISP have the same format, that of the S-expression. Input to LISP must be presented in the form of line images. Lines may be of any length, but UT LISP does not read more than 72 columns of a line. Reading of each line is terminated at column 73 or the first odd-numbered column after the last non-blank character on the line, whichever occurs first.

### 3.1.1 Lexical Classes

The significance of a character read by LISP is determined by the lexical class to which that character belongs, and each character belongs to some lexical class. The standard lexical classes are shown in table 3.1.

Function CHLEX (see section 5.7) may be used to change the lexical class of any character. By so doing, one can change the meaning of a character when it appears in input data. For instance, if the lexical class of the character : were changed to 0, then the : would be regarded as the end-of-line signal by LISP, and no information beyond the : would be read.

Table 3.1 Lexical Class Assignments

| Lexical Class | Display Code | Members |
|---|---|---|
| 0 | 00 | end-of-line |
| 1 | | letters except E and Q |
| 2 | 33-44 | digits 0 - 9 |
| 3 | | any characters not in some other class |
| 4 | 05 | E |
| 5 | 21 | Q |
| 6 | 45 | + |
| 7 | 46 | - |
| 8 | 53 | $ |
| 9 | 71 | % (percent sign or down arrow) |
| 10 | 65 | # (pound symbol or right arrow) |
| 11 | 61 | [ |
| 12 | 62 | ] |
| 13 | 51 | ( |
| 14 | 52 | ) |
| 15 | 57 | . (period) |
| 16 | 55-56 | , (blank and comma) |
| 17 | 60 | " (equivalence sign or double-quote) |
| 18 | | special (no members - see section 3.1.2.3, item 2) |

### 3.1.2 Literal Atoms

A literal atom is one of the basic types of S-expression. Literal atoms are represented in input and output data by strings of contiguous characters. For processing, literal atoms are objects which may be the constituents of larger S-expressions. The character string which represents the literal atom on external media is called the print name of the literal atom and serves to distinguish one literal atom from another. The particular constituent characters of the print name normally have no significance to the LISP processor. A literal atom may contain up to 30 characters in its print name.

#### 3.1.2.1 Standard Literal Atoms

A standard literal atom is one which can be read by the normal LISP READ function and whose print name will be an exact image of the character string used to represent it. The syntax of a standard literal atom is given by the rules:

1. The first character may be chosen from lexical classes 1-8 or 18.

2. The remaining characters, if any, may be chosen from lexical classes 1-9 or 17 except that if the first character is in class 8, the second character may not be.

3. The character string selected by rules 1 and 2 must not be interpretable as a numeric atom (see section 3.1.3).

4. The character string must be preceded by a member of lexical class 0, 9, or 11-17.

5. The character string must be followed by a member of lexical class 0, 11-16, or 18.

If we assume the standard lexical class assignments given in table 3.1, these rules plus the input format rules given earlier state that a standard literal atom is any string of 1-30 non-blank characters wholly contained on one line; does not start with $$, %, #, [, ], (, ), period, comma, or "; does not contain a #, [, ], (, ), period, comma, or blank; and is not interpretable as a number.

Examples:

| Valid | Invalid |
|---|---|
| A | 1234 |
| +B | A.B |
| 1234Z | X#Y |
| $12 | "P |
| ALONGATOM | (A) |
| BCD%XYZ | A LONG ATOM |

### 3.1.2.2 Special Literal Atoms

Special literal atoms have print names which do not obey the syntax rules for standard literal atoms. A syntactic mechanism is provided for specifying literal atoms with completely arbitrary print names. There are two categories of special literal atoms.

Category A:

Category A special literal atoms are denoted by;

1. The first two characters are from lexical class 8.
2. The third character is a delimiter and is chosen from lexical classes 1-18.
3. Following the third character is a string of 0-30 characters chosen from lexical classes 1-18, but not including the delimiter defined by rule 2.
4. The last character is a copy of the delimiter defined by rule 2.
5. The atom must be preceded by a member of lexical classes 0, 9, or 11-17.

The character string defined in rule 3 is the print name of the atom.

Examples: (Assuming class assignments of table 3.1)

| | | |
|---|---|---|
| $$$(.)$ | has the print name | (.) |
| $$A123A | has the print name | 123 |
| $$(XYZ( | has the print name | XYZ |

Category B:

Category B special literal atoms are formed according to the rules for standard literal atoms. However, any character not allowed in an atom under the rules of section 3.1.2.1 may be included if that character is prefixed by a character from lexical class 10. The effect of a class 10 character is to cause the immediately following character to be treated as class 1, regardless of its actual class membership. The characters from class 10 so used are not themselves counted as members of the print name of the atom. If the class 10 character occurs in column 72, it is considered to be followed by a blank.

Examples: (Assuming class assignments of table 3.1)

| | | |
|---|---|---|
| #(#.#) | has the print name | (.) |
| A#.B | has the print name | A.B |
| #123 | has the print name | 123 |

### 3.1.2.3 Additional Notes on Literal Atoms

1. Equivalence of print names is the mechanism for determining whether two external representations are the same atom. This means that the same atom can be represented in several ways in the input to LISP. For example:

   ```
   XYZ
   $$$XYZ$
   #X#Y#Z
   ```

   are all the same atom.

2. Lexical class 18 has a special significance. A member of class 18 serves as a terminator of atoms and may also be a first character. This class is intended to facilitate input of such things as punctuation in normal English text. For example, suppose the period were placed in class 18. Then the sequence

   ```
   DOG.
   ```

   in the input to LISP would be read as the atom DOG and the atom whose print name is the period. To achieve this effect without class 18 we would have to input

   ```
   DOG #.
   ```

3. If a string of characters intended to be a literal atom exceeds 30 characters, it will be truncated to the first 30 characters with no indication to the user.

4. Literal atoms are represented uniquely in the memory of a LISP program. All references to a given literal atom in LISP data structures refer to the same representation.

5. Appendix C lists those atoms initialized by the LISP system. These should be used only for their intended purposes. For example, T should not be reset by the user or used as a PROG variable.

## 3.1.3 Numeric Atoms

A numeric atom is another atomic type of S-expression. Numeric atoms are used in much the same way as literal atoms, but have a numeric value associated with them instead of a print name. There are two categories of numeric atoms: fixed-point and floating-point.

### 3.1.3.1 Fixed-point Numeric Atoms

The two categories of fixed-point numeric atoms are decimal integers and octal numbers. These two types are distinguished syntactically both on input and output, but may be used interchangeably as arguments of functions operating on fixed-point numbers.

Decimal integers are constructed according to the rules:

1. The first character may be an optional sign chosen from lexical class 6 or 7. Unsigned numeric atoms are assumed to be positive.

2. After the optional sign is a string of not more than 15 members of lexical class 2 which form an integer not exceeding 281,474,976,710,655 (2**48 - 1).

3. The numeric atom must be preceded by a member of a lexical class 0, 9, or 11-17.

4. The numeric atom must be followed by a member of lexical class 0, 9-14, or 16-18.

Examples: (Assuming class assignments as in table 3.1)

| Valid | Invalid |
|---|---|
| 1235 | 12Z3 |
| -6095 | -756340981005723 |
| +300001 | |

Octal numbers are constructed according to the rules:

1. The first character may be an optional sign chosen from lexical class 6 or 7. Unsigned numeric atoms are assumed to be positive.

2. After the optional sign is a string of not more than 20 members of lexical class 2 restricted to the digits 0-7.

3. Following the digit string there must be an occurrence of a character from lexical class 5 (normally a Q).

4. Following the class 5 character there may be an optional unsigned decimal integer. This integer is the number of bits by which the bit string represented by the octal digits is to be shifted left (end around) to form the final number.

5. The numeric atom must be preceded by a member of lexical class 0, 9, or 11-17.

6. The numeric atom must be followed by a member of lexical class 0, 9-14, or 16-18.

Examples: (Assuming class assignments as in table 3.1)

| Valid | Invalid |
|---|---|
| -7Q | 8Q |
| 14Q3(or 140Q) | 12345670123456701234567Q |
| -77777777777Q (or 7777777777Q30) | |

### 3.1.3.2 Floating-point Numeric Atoms

Floating-point atoms are constructed according to the rules:

1. The first character may be an optional sign chosen from lexical class 6 or 7. Unsigned numeric atoms are assumed to be positive.

2. After the optional sign there must be one or more characters from lexical class 2. These form the integer part of the number.

3. Following the integer part there may be an optional fraction part consisting of one character from lexical class 15 and one or more characters from lexical class 2.

4. An exponent part may follow the fraction part and must follow the integer part if there is no fraction part. The exponent part consists of a character from lexical class 4 followed by a signed or unsigned integer. The exponent part represents the power of 10 by which the number is multiplied.

5. The numeric atom must be preceded by a member of lexical class 0, 9, or 11-17.

6. The numeric atom must be followed by a member of lexical class 0, 9-14, or 16-18.

Examples: (Assuming class assignments as in table 3.1)

| Valid | Invalid |
|---|---|
| 1.23 | 8. |
| 4E5 | .2 |
| -6.2E-3 | 456 |
| +0.7E+6 | 2E |

### 3.1.3.3 Additional Notes on Numeric Atoms

1. A numeric atom is associated with a 60-bit memory word which holds the binary representation of its value. Character strings representing numeric values outside the normal range for numbers in the CDC 6000 machines will be converted into binary incorrectly without warning to the user.

2. Numeric atoms are stored separately in the computer memory. Each occurrence of a numeric atom creates a new data structure in memory, even if the value is the same as one encountered previously.

3. Although lexical class assignments can be made different from those in table 3.1, only the characters 0-9 can be correctly interpreted as digits in class 2. If other characters are used in class 2 they will convert to numeric values in a strange way.

4. If a character string whose initial elements resemble a numeric atom is encountered, LISP processes the string as a number until the syntax rules for numeric atoms are violated. At that point, the character string will be interpreted as a literal atom. For instance, the string

6.24A

would be read as a standard literal atom. This is the only way in which standard literal atoms can be created with a class 15 character embedded in the print name.

### 3.1.4 S-expressions

Atoms are basic symbolic data units of LISP. They may be combined into larger data structures called S-expressions. An atom by itself is the most primitive type of S-expression. S-expressions may appear in input text in a completely free-form manner except that atoms may not be split between 2 lines. Two types of composite S-expression are distinguished, one being a shorthand for a common form of the other.

#### 3.1.4.1 The Dotted Pair

Dotted pairs are the most primitive form of composite S-expression. They are represented in input text according to the rules:

1. The first character is a member of lexical class 13.
2. Next comes any S-expression. This is called the CAR of the dotted pair.
3. Then there must appear a character from lexical class 15.
4. Next comes any S-expression. This is called the CDR of the dotted pair.
5. Finally there must appear a character from lexical class 14.
6. The components may be separated optionally by any number of characters from lexical class 16.

Examples: (Assuming class assignments as in table 3.1)

| Valid | Invalid |
|---|---|
| (A.B) | (.A) |
| ((X.NIL).NIL) | (B.) |
| ((A.B) . (C.(D.E))) | (C.D.E) |
| (1 . 2) | (1.2) |

#### 3.1.4.2 The List

Dotted pairs whose only atomic CDR part is the atom NIL occur very frequently in LISP - so frequently, in fact, that a shorthand notation has been developed. These dotted pairs are called lists. By definition, the atom NIL is the empty list and can be represented by (). If we replace by () all occurrences of NIL in a dotted pair and then delete all matching sets of parentheses which are immediately preceded by a dot (deleting the dot also), we get the corresponding list notation for the dotted pair. For example,

(A.((B.(C.NIL)).(D.(E.NIL))))

is equivalent to

( A. (( B. ( C. ())). ( D. ( E. ()))))

which is equivalent to

(A (B C) D E)

More formally, a list is defined by:

1. The first character is a member of the class 13.

2. Next come zero or more separate S-expressions, each separated by an arbitrary number of characters from lexical class 16.

3. The last character must be a member of lexical class 14.

According to the class assignments of table 3.1, these rules mean that a list is simply a sequence of S-expressions separated by blanks or commas and surrounded by parentheses.

### 3.1.4.3 Additional Comments on Composite S-expressions

1. Each composite S-expression causes a new data structure to be built in memory, even if the same S-expression has been encountered previously.

2. Lists must be enclosed by balanced pairs of symbols consisting of one character from class 13 and one from class 14. The elements of a list may themselves be lists, which often results in a large number of class 14 characters appearing together at the end of a list. As a convenience to the user, the class 13 character of a list may be replaced by a character from class 11 ([) and any or all of the class 14 characters by one character from class 12 (]). Semantically, class 11 is equivalent to class 13. Also, a class 12 character is equivalent to any number of class 14 characters in a manner such that all class 13 characters between the class 11 and the class 12 characters will be properly matched. For example,

[(A(B(C(D]

is equivalent to

((A(B(C(D)))))

### 3.1.5 Additional Input Constructs

1. Comment information may appear any place in the input text that an S-expression may appear. A comment is any string of characters bracketed on both sides by a character from lexical class 9. Such comments appear on any listing of the LISP input text, but are never translated into any kind of data structures in memory.

2. The semantics of LISP functions requires that lists of the form:

(QUOTE S-expression)

appear frequently. As a shorthand notation, the LISP input procedures convert any S-expression preceded by a character from lexical class 17 into the above form internally. That is, using the class assignments of table 3.1,

"S-expression

is equivalent to

(QUOTE S-expression)

3. A mixed notation that is not purely dotted pairs or lists is possible. If one applies the process for converting dotted pairs to lists given in section 3.1.4.2 to a dotted pair whose atomic CDR is not NIL, the result is mixed notation. For example,

(A.(B.(C.D)))

is equivalent to

(A B C.D)

LISP can also read this form of notation.

## 3.2 OUTPUT FORMATS

LISP can output only S-expressions. All LISP output is in the form of line images. The user can specify the maximum length of the output lines for any file (see section 5.2) and he can force the first n columns of the lines to be blank. As characters are formatted into a line image, an internal pointer is kept to the position where the next character will be placed on the line. A new line is started whenever the current line is filled or the next atom cannot fit on the current line.

By using the character manipulating functions of section 4.10, the formatting functions of section 5.8, and the printing functions of section 5.3, the user can construct virtually any kind of formatted output he desires. The LISP printing functions each print an entire S-expression under the following formatting:

1. Each S-expression begins at the position specified by the internal pointer at the time printing starts.

2. A literal atom is represented in output by the character string that is its print image. Under user control the syntactic marks controlling special atom recognition may be reinserted (see section 5.3).

3. Fixed point numeric atoms are converted using an appropriate technique from binary to character representation and are printed without leading zeros. The letter Q is always used as the octal number indicator. A number following the Q indicates a binary power of two multiplier (a binary left shift); for example, 1Q6=100Q.

4. Floating point numeric atoms are converted from binary to character form according to the format parameters most recently set by function NFORMAT (see section 5.8). These formatting parameters specify the number of digits to be printed before and after the decimal point, and unless otherwise set are both 5. If the number falls in the range established by those parameters it is printed without an exponent part. If it does not fall in that range it is printed with only one digit preceding the decimal point and followed by an E and the appropriate exponent.

5. Composite S-expressions are always printed in list format with parentheses as delimiters and blanks as separators. Mixed notation is used if the S-expression has a non-NIL rightmost CDR (see section 3.1.5, item 3).

6. New lines are started only when a line is completely filled or when an atom will not fit on the current line.

See section 4.14 for some additional controls which may affect output formats.

## 3.3 INTERNAL FORMATS

All data manipulated by LISP ultimately resides in the computer memory. In this section, we discuss the various internal forms in which data may occur.

### 3.3.1 Storage Allocation

UT LISP uses all of the memory available to it. This total storage area is allocated into several different portions. Figure 3.1 shows the storage allocation.

The I/O buffers and program code areas are the fixed portions of LISP. These areas include all the code for the execution of LISP programs. Free space is the main area in which the internal forms of S-expressions are stored. Full-word space is used to hold character strings for atom names and the binary values of numeric atoms. The stack is an area used for holding the LISP program stack which enables recursive execution. Binary program space is used to contain compiled LISP functions and to contain LISP arrays. The size of each area is allocated at the beginning of LISP execution and, except for binary program space, remains fixed for the duration of execution. The field length, and consequently binary program space, is increased to accommodate as much code or as many arrays as desired, up to the limit imposed by the operating system.

The initial sizes of the areas shown in figure 3.1 depend on the following values:

FL - Field length

B - Value from B parameter, LISP control command (default value = 1000)

A - Value from A parameter, LISP control command (default value = 92)

X - Value from X parameter, LISP control command (default value = 0)

For these values, the initial sizes of the various areas are defined to be:

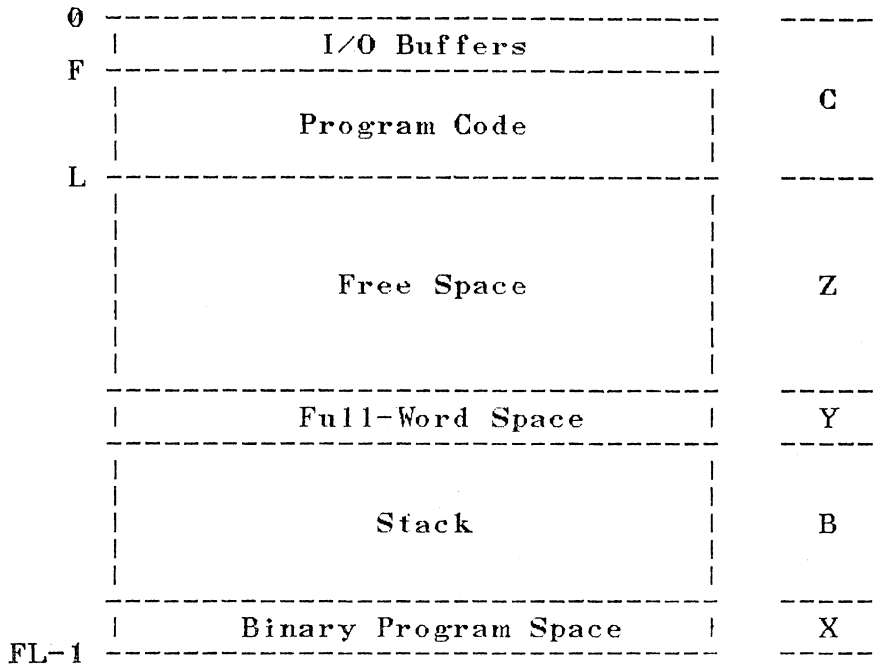
| | |
|---|---|
| Program code and I/O buffers: | C = 34000 (octal) changing with version |
| Binary program space: | X = X |
| Stack: | B = B |

Free space: Z = (FL - C - X - B) * A/100

Full-word space: Y = FL - C - X - B - Z

All of the area below the point marked F in figure 3.1 is overlaid by FORTRAN overlays and the area below the point marked L is overlaid by LISP overlays (see chapter 7).

Figure 3.1 UT LISP Storage Allocation

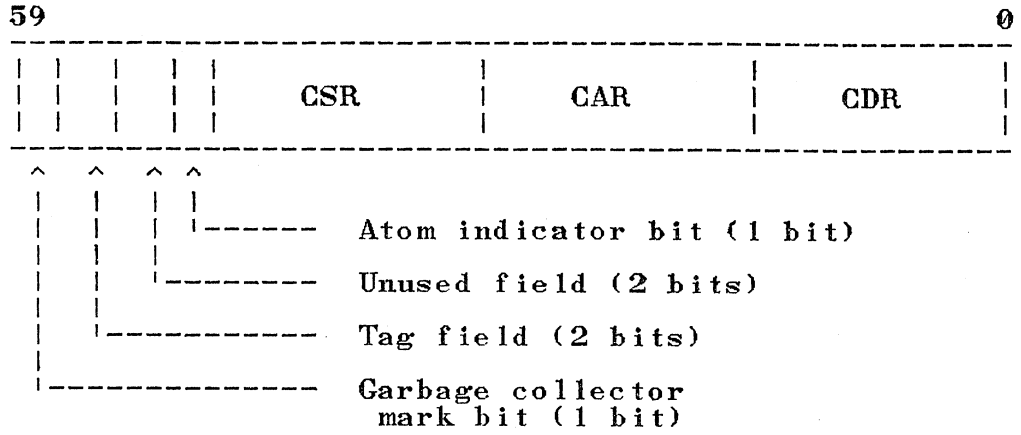



### 3.3.2 Free Space Data Formats

Every word in free space is used according to the format shown in figure 3.2. In each 60-bit word the three rightmost fields each occupy 18 bits and are used almost exclusively to hold pointers (addresses) to other words in free space or full-word space. These fields are named CSR, CAR, and CDR, from left to right. The leftmost bit of the word (bit 59) is used as a mark bit to indicate an active element of a data structure during the process of garbage collection. When garbage collection is not in progress this bit is always 0. A one-bit field (bit 54) indicates that the word is the header of an atom if the bit is set to 1. A two-bit tag field (bits 58 and 57) is used to distinguish among the various types of atoms. Bits 56 and 55 are unused and may contain anything.

S-expressions are represented in memory by sets of free space words linked together by pointers in the CSR, CAR, and CDR fields of the words. Internally, any S-expression is identified by the address of the first memory word in its structure. It is these addresses which are actually manipulated by the programs. In subsequent sections, more detailed pictures will be given of the various types of S-expression data structures.

Figure 3.2 Free Space Data Format




### 3.3.3 Full-Word Space Data Formats

Each 60-bit word in full-word space is used to hold a 60-bit object. An object is either a binary number or a string of up to 10 six-bit characters. Binary numbers are either one's complement integers or they are floating point numbers in the standard CDC 6000 representation. Character strings of fewer than 10 characters are stored left-justified with zero bit fill.

### 3.3.4 Dotted Pairs and Lists

Dotted pairs and lists are represented in storage by free space words which have zero in the tag field and in the atom indicator. A dotted pair is represented by a single such word whose CAR field contains a pointer to the CAR part of the dotted pair and whose CDR field contains a pointer to the CDR part of the dotted pair; the CSR field is not used.

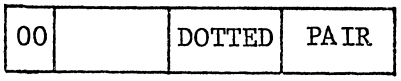
A list is represented by a set of words in free space, one for each element of the list. The CAR field of each word contains a pointer to an element of the list, and the CDR field of each word but the last contains a pointer to the next free space word in the list. The last word of a list contains a pointer to the atom NIL in its CDR field.

Diagrams showing examples of these structures are shown in figure 3.3. In these figures the convention of representing a pointer to an atom by writing the name of the atom in the field is used.

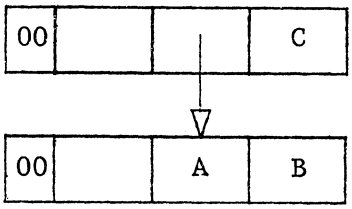
> Note: In figure 3.3 and the following figures the tag field, mark bit, atom indicator, and unused bits are shown as one 2-digit octal number.

Figure 3.3 Storage of Dotted Pairs and Lists

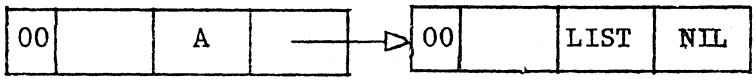
1. (DOTTED . PAIR)



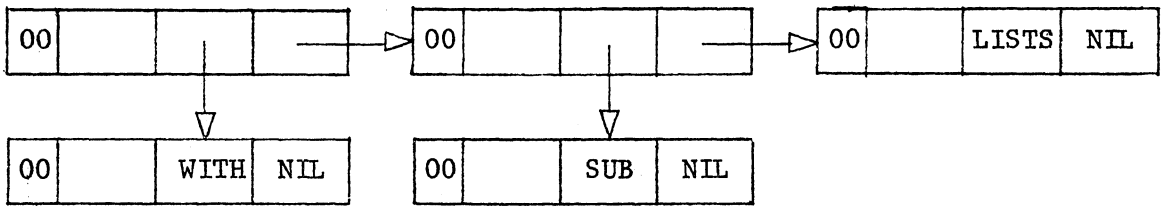

2. ((A . B) . C)




3. (A LIST) ≡ (A . (LIST . NIL))




4. ((WITH) (SUB) LISTS) ≡ ((WITH . NIL) . ((SUB . NIL) . (LISTS . NIL)))



### 3.3.5 Literal Atoms

A literal atom is a fairly complex structure which has components in both free space and full-word space. The first word of a literal atom, called the atom header, has the atom indicator bit set to 1 and the tag field set to 0. The CSR field of the atom header points to the property list of the atom. Each atom has at least two attributes: PNAME, with a value which is a structure representing the print image of the atom, and INFO, with a value which qualifies the print image. (Note: UT LISP atoms usually share INFO property-list cells. Therefore the "expert" user who is directly manipulating property lists must take care never to concatenate property lists with other structures without making a copy of the INFO cell.) Each element of the property list contains a pointer to the indicator (or attribute) in its CSR field, a pointer to the associated value in its CAR field, and a pointer to the next property list element in its CDR field.

The CDR field of the atom header contains a pointer to the S-expression which is the current value bound to that atom when it is accessed by LISP as a variable. A special pointer to an atom with an empty print image is stored here when the atom is not actually bound to any value.

The CAR field of the atom header contains a pointer to a list of other values bound to the atom in higher-level contexts. In an unbound atom the CAR field of the header points to the header itself. Figure 3.4 shows the changes which occur in an atom header as successive values are bound to it.

Figure 3.5 shows the different structures which occur for representing an atom's print image. Figure 3.6 shows the interpretation of the value associated with the INFO property. This particular value is not a pointer to some S-expression.

Finally, figure 3.8 shows a complete literal atom structure.

Figure 3.4 Binding Values to an Atom

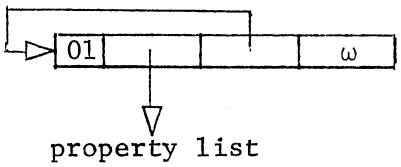
Given the function definitions:

```
function A:  (LAMBDA (X) (B (CDR X)))
function B:  (LAMBDA (X) (C (CDR X)))
function C:  (LAMBDA (X) X)
```

the figures below show the header for atom X during evaluation of the expression:

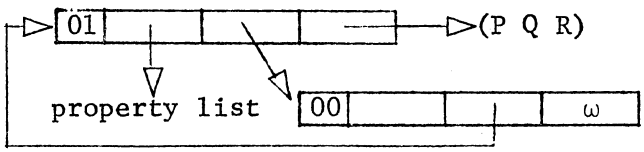
```
(A (QUOTE (P Q R)))
```

1. Before Entry to A

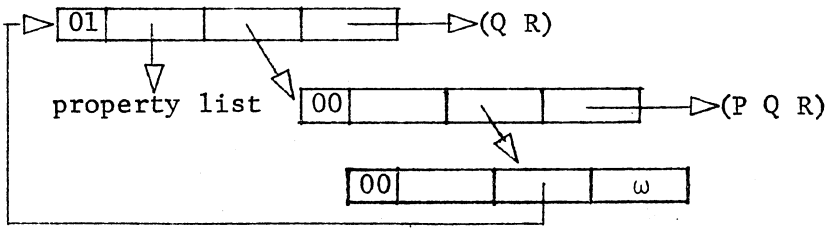



(ω = atom with empty name)

2. Before Entry to B




3. Before Entry to C




4. After Entry into C

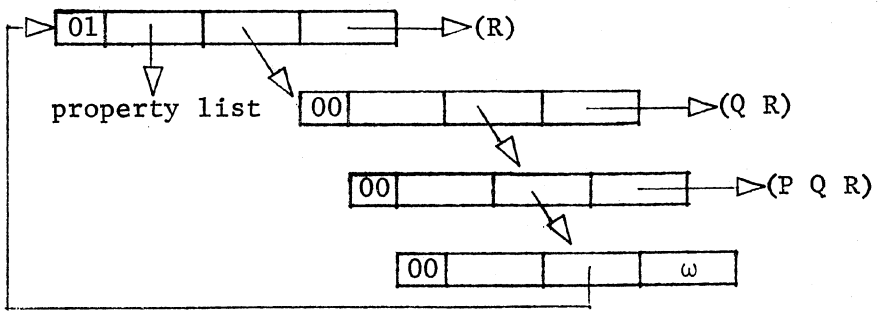
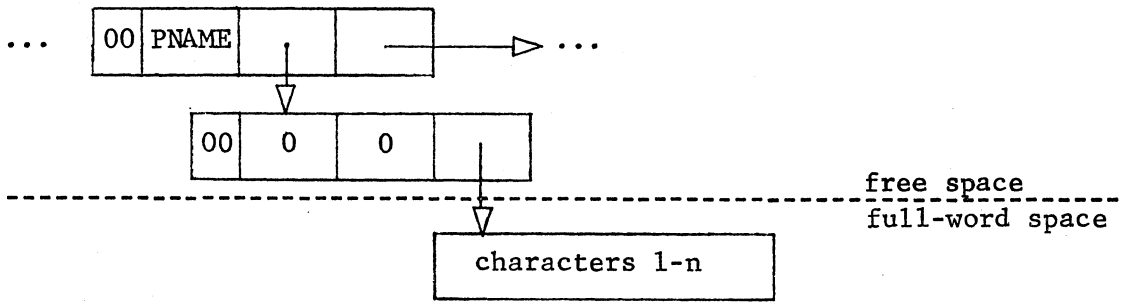
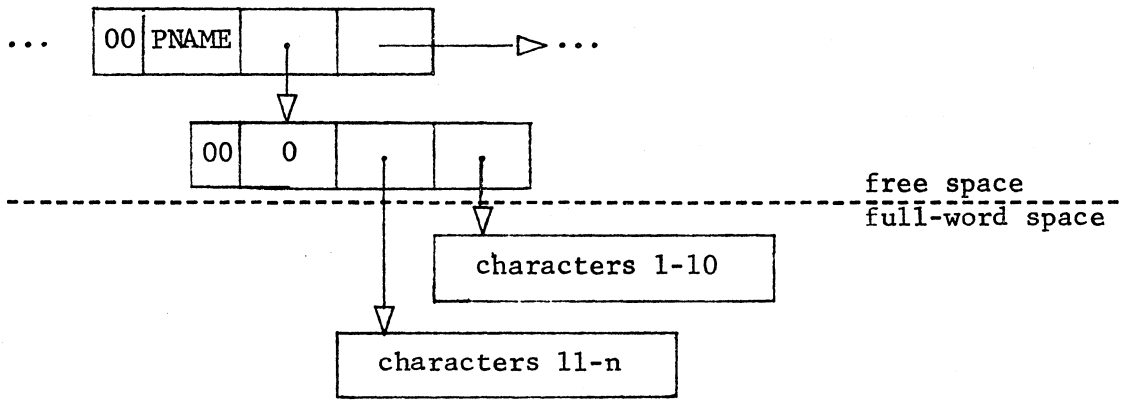


Figure 3.5 Print Image Structures

1. Atom With Print Image of 1-10 Characters




2. Atom With Print Image of 11-20 Characters




3. Atom With Print Image of 21-30 Characters

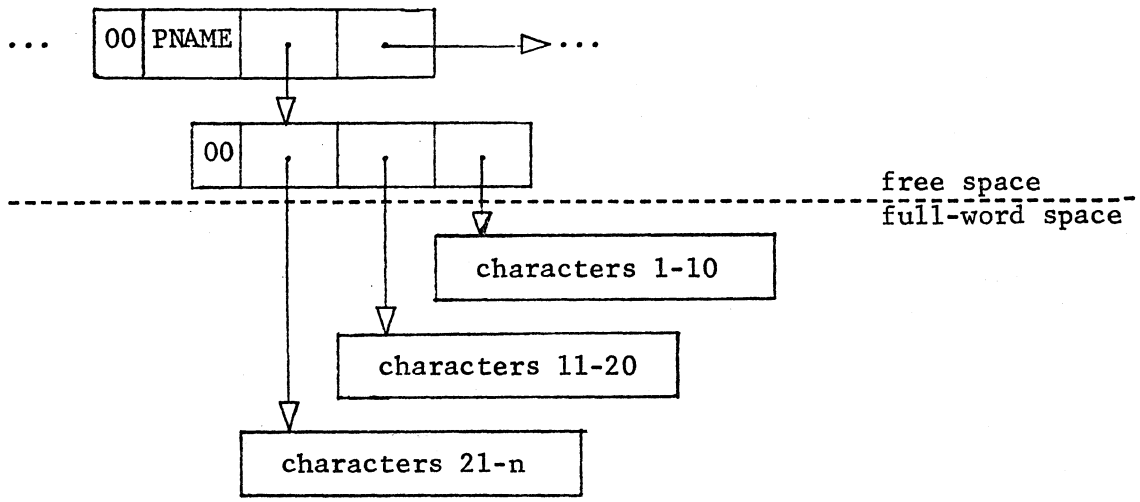


Figure 3.6 Interpretation of INFO Property Value

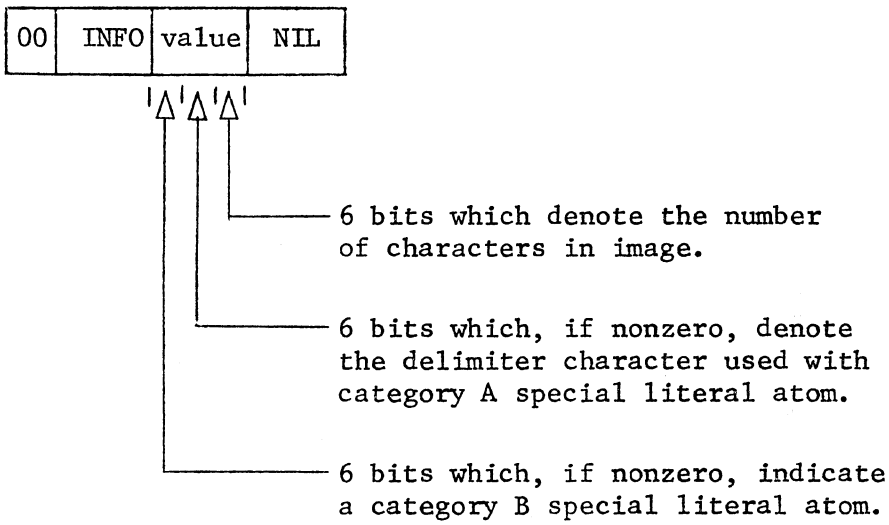
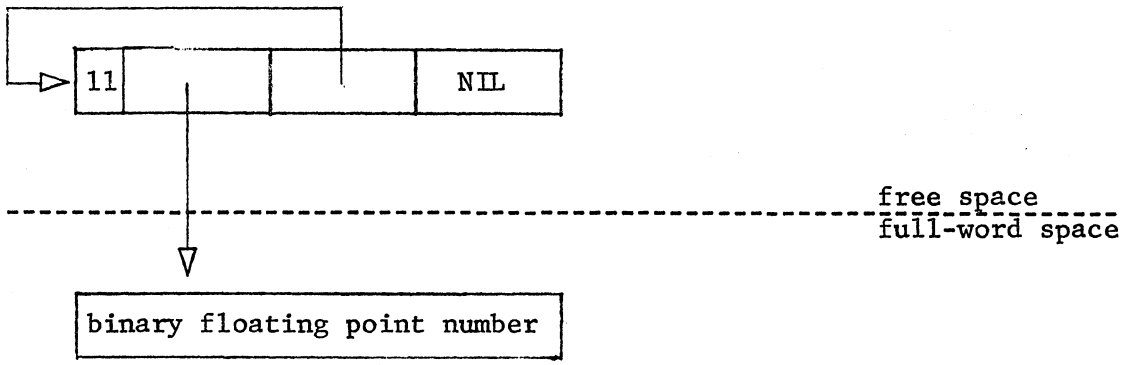
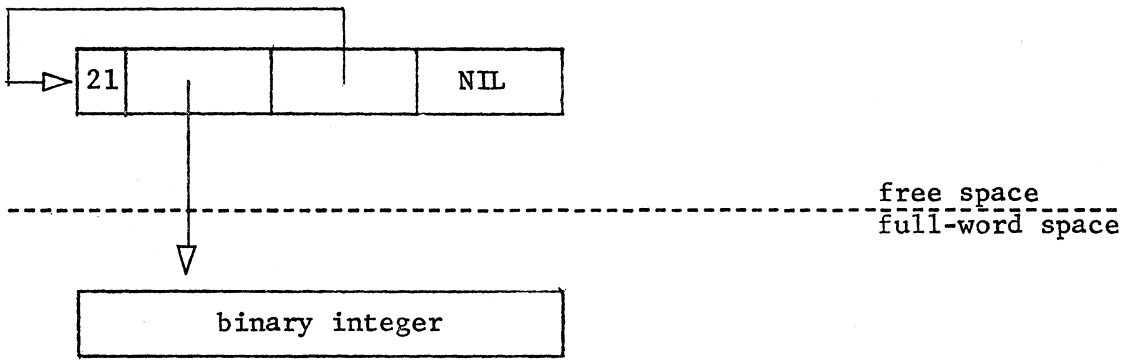


Figure 3.7 Numeric Atom Structures

1. Floating Point Numeric Atom




2. Integer Numeric Atom




3. Octal Numeric Atom

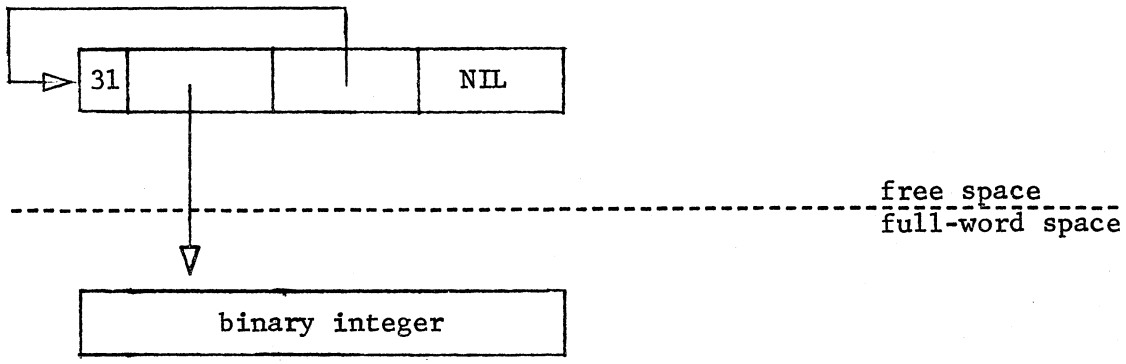
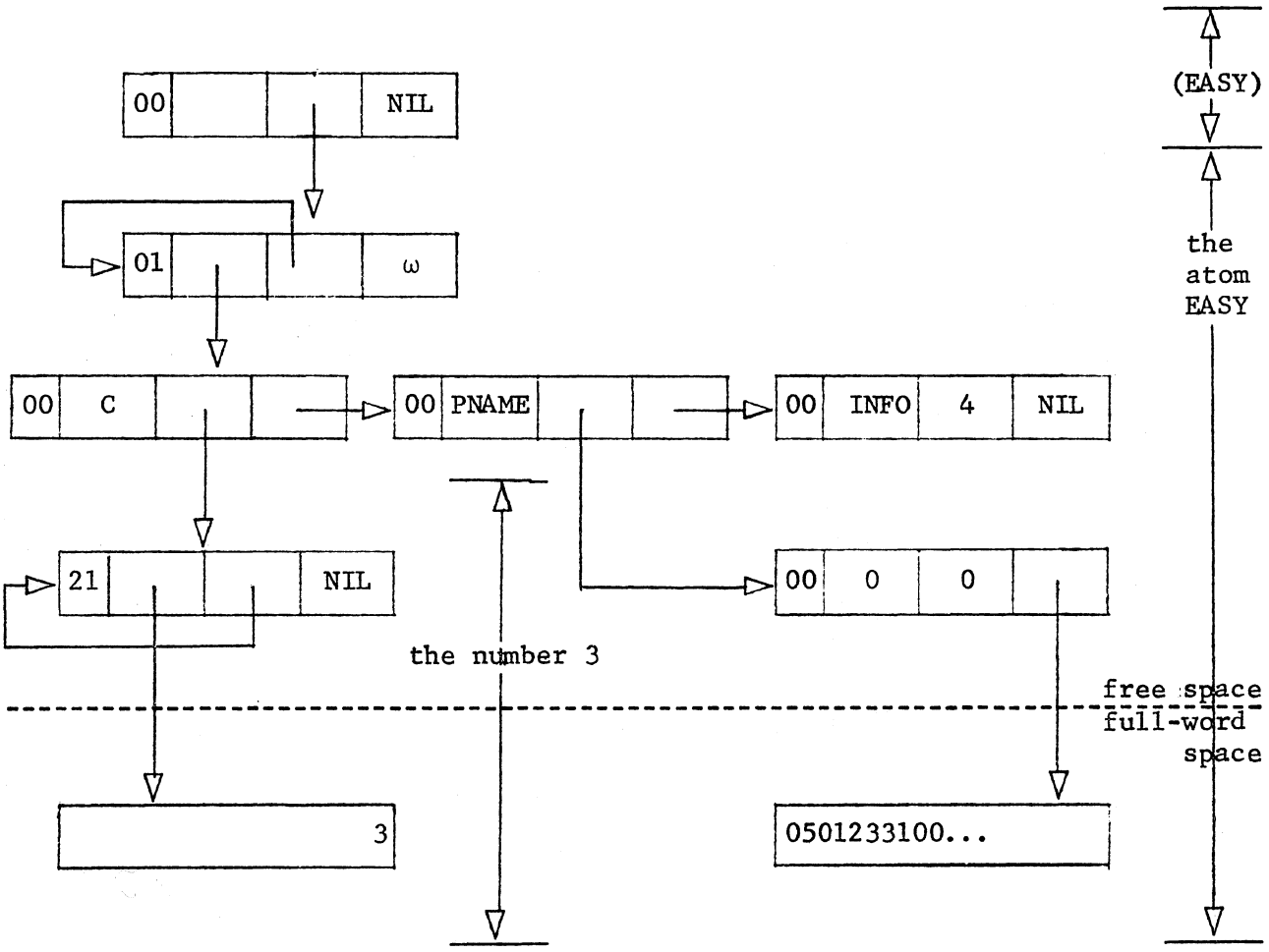


Figure 3.8 A Full Example

The list (EASY) is shown with atom EASY also drawn in detail. EASY is shown as having no value, but having attribute C with value 3.



### 3.3.6 Numeric Atoms

All numeric atoms are two-word structures. The numeric atom header is a free space word with the atom indicator bit set to 1 and the tag field containing 1, 2, or 3 to denote floating point, integer, or octal, respectively. The CSR field of the numeric atom header points to a word in full-word space which contains the binary value of the number. The CAR field always points to the header, and the CDR field always points to NIL. Figure 3.7 shows the structures in detail. The only difference between integers and octal numbers is that the print routines print one as a decimal number and the other as an octal number.

### 3.3.7 The Oblist

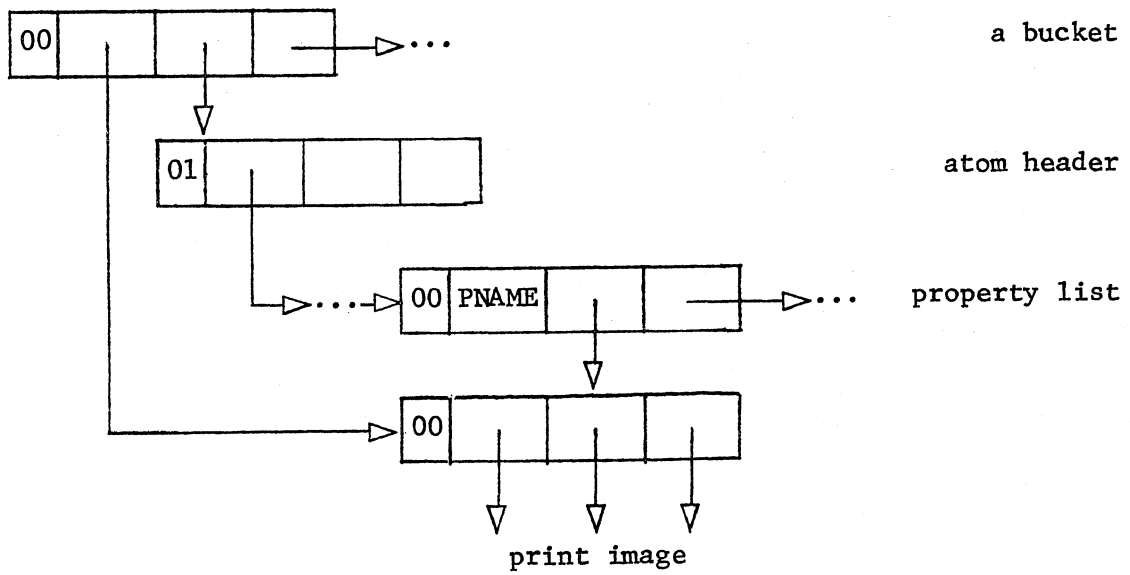
The "oblist" is a special list maintained by the LISP input routines and is deserving of special mention. Each literal atom in the LISP system appears somewhere in the oblist, and as a result the input routines can always determine if an atom being read already exists. This mechanism enables LISP to maintain the unique storage of literal atoms.

The oblist is a list of 128 sublists called "buckets". As each literal atom is read, its print image is used to generate a number between 0 and 127, thus selecting one of the buckets of the oblist. The atom will be in only the bucket so selected.

To facilitate searching by print images, the elements of the buckets have a special structure as shown in figure 3.9.

The oblist is normally available to the LISP program as the value of variable OBLIST.

Figure 3.9 An Oblist Element



# 4. FUNCTION DEFINITIONS

All LISP programs are built by function composition; that is, the user defines functions which in turn call on other functions, either user-defined or defined in the LISP system. This chapter describes precisely most of the functions defined in UT LISP. Some functions are described in chapters 5, 6, 7, and 9, as appropriate. These function descriptions should serve the user as his definitive guide to their use.

## 4.1 FUNCTION TYPES

There are four function types in UT LISP, distinguished by the names EXPR, SUBR, FEXPR, and FSUBR. Functions of type EXPR and FEXPR are stored in the machine as lambda expressions to be executed in an interpretive manner. Functions of type SUBR and FSUBR are stored in the system as machine code which is executed directly by the computer when requested. User functions are generally of type EXPR or FEXPR and the functions defined within the LISP system are always of type SUBR or FSUBR. Functions of type EXPR and SUBR are functions of a fixed number of arguments. Furthermore, when such a function is called, the argument expressions appearing in the calling expression are evaluated before they are given to the called function. Arguments of a SUBR may be omitted from the right, and the missing arguments are supplied as NIL. Functions of type FEXPR and FSUBR, on the other hand, can have an arbitrary number of arguments in the calling expression and those arguments are not evaluated before they are given to the called function. FEXPR and FSUBR functions are also sometimes called "special forms".

LISP functions may be further classified as normal functions, pseudofunctions, or predicates. Normal functions receive some arguments, perform some manipulation on those arguments, and return a value which depends on the manipulation performed. Pseudofunctions receive some arguments and perform some side-effect operation (i.e., an operation that changes some of the internal data structures of the LISP program) rather than some manipulation upon the arguments. Since all LISP functions must return some value, a pseudofunction also returns a value, but the value may bear no relationship to the arguments which were given it. A predicate is a function which receives some arguments and returns a value of either true or false, depending on some relationship that holds among the arguments. In LISP, falsity is always represented by the atom NIL and truth is represented by anything other than NIL. Some predicates do in fact return the atom *T* as the value for truth, but the user should endeavor not to use this fact.

## 4.2 NOTATION USED IN FUNCTION DEFINITIONS

The function descriptions in this manual are given using EVAL notation. For example:

(CONS <s1> <s2>)

which indicates that CONS is a function of two arguments, either of which may be any arbitrary S-expression. In the text of this manual LISP function names and system variables are always capitalized. In descriptions of the arguments expected by the function, < > brackets are used to delimit argument codes (see

table 4.1), with [ ] brackets denoting argument subscripts which are not simple integers. When a function can accept an arbitrary number of arguments, the ellipsis (...) is used to indicate this fact.

In general, a function has certain restrictions on the types of arguments it may receive. The descriptors given below incorporate these restrictions through a shorthand notation. Where one of the descriptors given in the table below appears in the list of arguments of a function definition, the argument actually supplied to the function must obey the characteristics defined for that descriptor. Notice that if the arguments to a function are evaluated (as in all SUBRs and in certain indicated FSUBRs), the descriptor defines what the result of that evaluation must be, whereas if a function does not evaluate its arguments, the descriptor defines what must literally appear in the code. If the restrictions are not met, the LISP system will probably detect an ILLEGAL ARGUMENT or ILL-FORMED ARGUMENT error.

Each function description given in the remainder of this chapter includes the name of the function, the types of arguments the function expects, the classifications of the function, and a precise description of the action of the function.

The functions are grouped into subsections according to overall purpose. Within each group, functions are described first and predicates last, with the most commonly used of each type given first. An alphabetical index of functions is given in appendix A.

## 4.3 ELEMENTARY FUNCTIONS AND PREDICATES

The elementary functions and predicates comprise the LISP primitive functions and predicates on which the language is based and the most commonly-used nonprimitive functions and predicates.

(CAR <nats>)

normal; SUBR

CAR returns as its value the left part of the dotted pair which is the result of evaluating its argument. In list terms, it returns the first element of the list. It retrieves the value stored in the CAR field of the non-atomic S-expression which is its argument.

(CDR <nats>)

normal; SUBR

CDR returns as its value the right part of the dotted pair which is the result of evaluating its argument. In list terms it returns the rest of the list after the first element is deleted. It retrieves the value stored in the CDR field of the non-atomic S-expression which is its argument.

Table 4.1 UT LISP Argument Descriptors

| Type Descriptor | Meaning |
|---|---|
| <atom> | The argument must be either a literal atom or a numeric atom. |
| <boolean> | The argument may be any S-expression, but it will be interpreted as a truth value: NIL is equivalent to false; anything else is equivalent to true. |
| <character> | The argument must be a literal atom whose name is a single character. |
| <exp> | The argument must be some LISP expression which can be evaluated by EVAL. |
| <filename> | The argument must be a literal atom containing at most seven letters and digits, starting with a letter. |
| <flnumber> | The argument must be a floating-point numeric atom. |
| <fixnumber> | The argument must be a fixed-point numeric atom (either integer or octal). |
| <fnexp> | The argument must be a functional expression. Either it must be the name of a function preceded by QUOTE or FUNCTION, or it must be a lambda expression preceded by QUOTE, FQUOTE, or FUNCTION. |
| <function> | The argument must be a function name, lambda expression or label expression. |
| <fw> | The argument must be a single full word, that is, a word in full-word space. |
| <fwl> | The argument must be a list of full words. |
| <lat> | The argument must be a list of literal atoms. |
| <letter> | The argument must be a literal atom whose name is a single letter (A...Z). |
| <list> | The argument must be a list or NIL, the empty list. By list is meant a non-atomic S-expression whose rightmost CDR is NIL, i.e., whose top-level printed representation contains no periods. |
| <litatom> | The argument must be a single literal atom. |
| <nats> | The argument must be some non-atomic S-expression. |
| <number> | The argument must be a numeric atom of any type. |
| <s> | The argument may be any arbitrary S-expression. |

(CSR <nats>)

normal; SUBR

CSR retrieves the value stored in the CSR field of the non-atomic S-expression which is its argument.

(CAAR <nats>)
(CADR <nats>)
(CASR <nats>)
(CDAR <nats>)
(CDDR <nats>)
(CDSR <nats>)
(CSAR <nats>)
(CSDR <nats>)
(CSSR <nats>)
(CAAAR <nats>)
.
.
.
(CSSSSSSSSR <nats>)

normal; EXPR

Multiple CAR-CDR-CSR functions are allowed and may contain up to 8 A's, D's, or S's between the C and R to denote a sequence of CAR, CDR, or CSR operations. The sequence of operations is applied in right-to-left order, i.e., (CDAR <nats>) is equivalent (CDR (CAR <nats>)). Users may find the NTH function more useful for combined forms composed of a CAD...DR sequence.

(CONS <s1> <s2>)

normal; SUBR

CONS builds a new S-expression. It constructs the dotted pair (<s1> . <s2>) by obtaining a new word from free space and storing its first argument into the CAR field of the word and its second argument into the CDR field of the word. CONS is the fundamental function for building new S-expressions.

(LIST <s1> <s2> ...<s[n]>)

normal; FSUBR

LIST takes an arbitrary number of arguments and constructs a new list such that <s1> is the first element, <s2> is the second element, and so on. If no arguments are given the value is NIL.

(COMMENT <s1> <s2> ... <s[n]>)

pseudofunction; FSUBR

COMMENT is a do-nothing function which provides a convenient means of retaining comments within a defined LISP program. COMMENT evaluates none of its arguments, and always returns NIL.

(QUOTE <s>)

normal; FSUBR

QUOTE is essentially a do-nothing function. It receives its argument unchanged and returns as its value that same argument. QUOTE is used to prevent evaluation of S-expressions which serve as data internal to a LISP function definition. Within a LISP function, most literal atoms are evaluated as variables and lists are evaluated as function calls. When QUOTE precedes an atom or an S-expression, it is a signal to the interpreter that these things are not to be evaluated as variables and function calls, but instead are actual constant data of the function.

(FQUOTE <s>)

normal; FSUBR

FQUOTE is identical to QUOTE with the exception that when compilation is occurring, the compiler compiles the argument of FQUOTE as a function and does not compile the argument of QUOTE. Therefore, in functions which are to be compiled, FQUOTE should appear only before lambda expressions.

(SET <atom> <exp>)

pseudofunction; SUBR

SET is the principal value-assignment function. Its first argument must evaluate to an atom and the value of the second argument is made the value of the atom. The value of SET is its second argument, or the new value which was assigned.

(SETQ <atom> <exp>)

pseudofunction; FSUBR

SETQ is similar to SET except that the first argument of SETQ is quoted. That is, the first argument is the actual atom which will have its value assigned. The value returned by SETQ is its second argument, or the new value which was assigned to the variable. For example,

(SETQ X Y)

is equivalent to

(SET (QUOTE X) Y)

(RPLACA <nats> <s>)

pseudofunction; SUBR

RPLACA replaces the CAR field of its first argument with a pointer to its second argument. This function modifies existing list structure in memory, and should be used with caution. 257 The value returned by RPLACA is its first argument, which has been modified by the action of RPLACA.

(RPLACD <nats> <s>)

pseudofunction; SUBR

RPLACD replaces the CDR field of its first argument with a pointer to its second argument. This function modifies existing list structure in memory. RPLACD should be used with caution as it may be used to create circular lists, which may cause difficulty with other processes. The value returned by RPLACD is its first argument, which has been modified by the action of the function.

(RPLACS <nats> <s>)

pseudofunction; SUBR

RPLACS replaces the CSR field of its first argument with a pointer to the second argument. This function is the only function which allows the user to place information into the CSR field. It modifies existing list structure in memory and therefore should be used with caution. The value returned by RPLACS is its first argument, which has been modified by the action of the function.

(ATOM <s>)

predicate; SUBR

ATOM returns true if its argument is either a literal atom or a numeric atom. It returns false if its argument is any other expression.

(EQ <s1> <s2>)

predicate; SUBR

EQ returns true if its two arguments share the same memory location. It returns false under any other circumstances. Literal atoms are stored uniquely in the LISP system so that their equality may be determined by this simple comparison of machine addresses. Since no other type of S-expression is stored uniquely, an EQ comparison of other types usually fails.

(EQN <s1> <s2>)

predicate; SUBR

EQN is similar to EQ except that it also works for numeric atoms. It returns true either if its two arguments are EQ or if they are two numeric atoms which have the same numeric value. The two numbers may be of different types. EQN returns false under all other circumstances.

(EQUAL <s1> <s2>)
(= <s1> <s2>)

predicate; SUBR

EQUAL returns true if its two arguments are equivalent S-expressions. Two S-expressions are equivalent if they are both the same literal atom, if they are two numbers with the same value, or if they are two non-atomic S-expressions composed of the same atoms in corresponding positions. EQUAL returns false if its two arguments are not equivalent S-expressions. = is a synonym for EQUAL.

(NULL <s>)

predicate; SUBR

NULL returns true if its argument is the atom NIL. It returns false under all other circumstances. NULL is a logical negation operator.

(NUMBERP <s>)

predicate; SUBR

NUMBERP returns its argument if the argument is a number of any type. It returns false if its argument is any other kind of S-expression.

(MEMBER <s> <list>)

predicate; SUBR

MEMBER searches the top level of <list> for an element EQUAL to <s>. If such an element is not in the <list>, the value of MEMBER is NIL. If it is present, the value of MEMBER is the remaining portion of the list beginning with the element sought.

(MEMQ <s> <list>)

predicate; SUBR

MEMQ is identical to MEMBER in all respects except that MEMQ uses EQ instead of EQUAL when testing for the equality of <s> with a member of <list>.

(ALPHAP <litatom1> <litatom2>)

predicate; SUBR

ALPHAP compares the print names of its two arguments and returns true if the first argument alphabetically precedes the second argument. It returns false if the first argument alphabetically follows the second or if they are the same atom. The collating sequence of the CDC 6000 series display code is used to determine the alphabetization.

(GRADP <s1> <s2>)

predicate; SUBR

GRADP returns true if its first argument resides at a lower memory address than its second argument. It returns false if its first argument resides at the same or higher memory address than the second argument. This function can be used as an arbitrary but consistent ordering predicate for literal atoms since a given literal atom will always reside at the same address during the course of one LISP run unless removed from the oblist (see REMOB, section 4.12). Non-atomic S-expressions and numbers may have many copies present in memory during the course of one run and each copy will reside at a different address. Thus, users should be certain GRADP is comparing the addresses of the same desired copies if it is used to order such expressions. The function EQN will detect different copies.

(A+ <s1> <s2>)
(A- <s1> <s2>)

normal; SUBR

A+ (or A-) returns an S-expression whose address is the sum (or difference) of the addresses of <s1> and <s2>. An error occurs if the resulting expression would cause a mode error.

## 4.4 LOGICAL CONNECTIVE FUNCTIONS

The logical connective functions are used to form complex Boolean expressions from simple predicate expressions.

(NOT <s>)

predicate; SUBR

NOT is in every way equivalent to the predicate NULL. It is the logical inversion operator. It returns true if its argument is false or returns false if its argument is true.

(AND <exp1> <exp2> ... <exp[n]>)

normal; FSUBR

AND evaluates each of its arguments in turn until it encounters the first argument which is false or until it evaluates all of its arguments. If all of its arguments evaluate to true the value of AND is the value of <exp[n]> (which is true). If one argument is false the value of AND is false. Expressions following the first false expression are not evaluated. If there are no arguments the value of AND is true.

(OR <exp1> <exp2> ... <exp[n]>)

normal; FSUBR

OR evaluates each of its arguments in turn until it finds the first argument which has a true value. If one of its arguments is true, then the value of OR is the value of that argument. If none of its arguments is true, then the value of OR is false. Arguments following the first true argument are not evaluated. If no arguments are present the value of OR is false.

## 4.5 SEQUENCE CONTROL AND FUNCTION EVALUATION

The functions of this section are used to control the sequence of execution of expressions in a LISP function by providing conditional and iterative control structures. Also included are functions which enable the user to construct and evaluate his own functions and expressions at run-time.

```
(COND (<boolean1> <exp> ... <exp>)
      (<boolean2> <exp> ... <exp>) ...
      (<boolean[n]> <exp> ... <exp>) )
```

normal; FSUBR

COND is one of the major functions of LISP. Most user-defined functions are defined in terms of a COND-expression (conditional expression). COND provides a conditional control structure similar to that provided by the IF statement in ALGOL. The arguments of COND are lists containing at least one S-expression each. COND proceeds as follows: <boolean1> is evaluated and if it is true then the remaining S-expressions (if any) in the list containing <boolean1> are evaluated in left-to-right order. If <boolean1> is false, then <boolean2> is evaluated and if <boolean2> is true the S-expressions following it (if any) are evaluated. This process continues for <boolean3>, <boolean4>, etc., until either some <boolean[i]> is true or all <boolean[i]>'s have been evaluated as false. In any case the value of COND will always be the value of the last S-expression evaluated under its control. The value of COND with no arguments given is NIL.

```
(SELECT <exp>  (<exp[1,1]> <exp[1,2]> ...  <exp[1,n1]>)
               (<exp[2,1]> <exp[2,2]> ...  <exp[2,n2]>) ...
               (<exp[m,1]> <exp[m,2]> ...  <exp[m,n[m]]>)
               <exp[m+1]>)
```

normal; FSUBR

SELECT allows the selection of a particular list of expressions to be evaluated depending on the value of the first argument of SELECT. It proceeds as follows. The first argument, <exp>, is evaluated. Its value is then compared successively to the values of <exp[1,1]>, <exp[2,1]>, <exp[3,1]>, ..., until the value of <exp> equals the value of <exp[i,1]. For that i, each <exp[i,2]>, <exp[i,3]>, ..., <exp[i,n[i]> is evaluated and the value of <exp[i,n[i]]> is returned as the value of SELECT. SELECT is very similar to COND in this respect, in that once a test is successfully completed, SELECT

evaluates an arbitrarily large number of associated expressions, returning the value of the last expression so evaluated. If the value of <exp> is not equal to the value of any <exp[i,1]>, i=1,...,m, then the value of <exp[m+1]> is returned as the value of SELECT.

(PROG2 <exp1> <exp2>)

pseudofunction; SUBR

The value of PROG2 is the value of <exp2>. This function is used to perform two actions where LISP normally allows only one.

(PROGN <exp1> <exp2> ... <exp[n]>)

pseudofunction; FSUBR

PROGN evaluates each of its arguments in turn and the value of the last argument is returned as the value of PROGN. This function is used to perform a simple sequence of actions where LISP normally allows only one.

(PROG <lat> <s1> <s2> ... <s[n]>)

pseudofunction; FSUBR

The PROG expression is used to form iterative functions. The first argument is a list of atoms which serve as temporary variables within the PROG expression. The remaining arguments are either literal atoms or else they are expressions to be evaluated. Literal atoms serve as labels and the expressions serve as statements of a sequential programming language. PROG proceeds by first assigning the value NIL to each of the temporary variables. It then evaluates each <s[i]> unless it is an atom. An atom is skipped and the expressions following it are evaluated, one after the other. The use of the GO function allows the sequence of execution to be altered (see below). The value of a PROG expression is NIL if all the <s[i]>'s are executed without encountering the RETURN function. If RETURN is encountered, the value of the argument of RETURN is the value of the entire PROG expression. GO and RETURN may be executed in other functions which are called by the function containing a PROG expression, provided none of these functions has been compiled (see chapter 6).

(GO <litatom>)

pseudofunction; FSUBR

GO is used to control execution sequence within a PROG expression. Its argument is not evaluated and must be one of the labels which appears in the PROG expression. Execution of GO causes the execution of the PROG expression to be continued with the statement immediately following the label which is the argument of the GO. GO may not be used to branch out of a PROG expression, but may transfer control only within the most recently occurring PROG expression. GO has no value as such.

(RETURN <s>)

pseudofunction; SUBR

RETURN causes LISP to exit a PROG expression. The argument of RETURN is made the value of the entire PROG expression. RETURN may be executed within a non-PROG expression which is evaluated under the control of a PROG expression, in which case LISP exits from the PROG expression.

(EXIT <litatom> <exp>)

pseudofunction; SUBR

EXIT causes LISP to exit the most recent call to the user-called function <litatom>. EXIT evaluates <exp> a second time. This second evaluation takes place in the context before the call to <litatom> rather than in the context of the call to EXIT. EXIT returns the resulting value as the value of the function <litatom>. EXIT differs from RETURN in that EXIT returns from <litatom> to the function which called <litatom> independent of whatever PROG expressions have been entered. EXIT is equivalent to:

(RETFROM (NTHFNBK <litatom> 1) <exp>)

(EVAL <exp>)

normal; SUBR

EVAL evaluates its argument a second time. The value of this second evaluation of <exp> is returned as the value of EVAL. Variables within the expression will be evaluated in the context in which the call to EVAL is made.

Example:

(EVAL (QUOTE COMMA)) = ,

(APPLY <function> <list>)

normal; SUBR

The <list> is a list of arguments for the <function> which is the first argument of APPLY. The value of APPLY is the value obtained by applying <function> to <list>. The first argument must be either a function name or a lambda expression and if it is a function name, the function must be of type EXPR or SUBR. If a <function> of type FEXPR or FSUBR is given to APPLY then an UNDEFINED FUNCTION error will occur even though the function is in fact defined.

(EVALQUOTE <function> <list>)

normal; SUBR

EVALQUOTE is similar to APPLY in that it applies <function> to <list>. However, EVALQUOTE allows the function to be of type FEXPR or FSUBR as well as EXPR or SUBR. The result of the application of <function> to <list> is the value of EVALQUOTE.

(EVLIS <list>)

normal; SUBR

EVLIS expects its argument to be a list of expressions. It evaluates these expressions one at a time from left to right and returns a list whose elements are the respective values of the expressions.

(FUNCTION <function>)

pseudofunction; FSUBR

The argument of FUNCTION is expected to be a function name or a lambda expression. This function is required in certain very special cases to prepare an environment for a functional argument being passed to another function. Its value is a so-called "funarg" expression.

(LABEL <litatom> <lambda expression>)

form

LABEL is a form which binds the <lambda expression> to <litatom> in a manner such that during execution of the <lambda expression>, if <litatom> is applied as a function and has no standard definition (EXPR, etc.), then <lambda expression> is used as the function definition. LABEL is used to permit a name to be given to a temporary, recursive expression.

## 4.6 LIST MANIPULATION FUNCTIONS

List manipulation functions are the principal means for dealing with S-expressions. Operations of construction, copying, reversal, and combination are possible.

(LENGTH <s>)

normal; SUBR

The argument of LENGTH may be either an atom or a list. If the argument is a list, then the value of LENGTH is the number of elements in the list. If the argument is an atom, the value of LENGTH is zero.

(NTH <list> <fixnumber>)

normal; SUBR

NTH returns the <fixnumber>th top-level element in <list>. <fixnumber> must be strictly positive and less than or equal to the number of elements in <list>. (NTH <list> 1) is equivalent to (CAR <list>).

(COPY <s>)

normal; SUBR

COPY copies its argument. That is, it returns an entirely new list structure occupying different memory words and equivalent to the original S-expression.

(REVERSE <list>)

normal; SUBR

REVERSE returns a new list whose top-level elements are the same as the elements of its argument but are reversed in order. Sublists of the list are not themselves reversed.

(REVERSIP <list>)

pseudofunction; SUBR

REVERSIP performs an in-place reversal of a list. The value of REVERSIP is the same <list> as its argument, but that list has been internally modified so that the top-level elements appear in reverse order.

(APPEND <list> <s>)

normal; SUBR

The first argument of APPEND must be a list. The second argument normally is a list, but does not have to be. APPEND joins its two arguments together such that the resulting new list contains the elements of both arguments with the elements of <list> appearing before the elements of <s>. The first argument is copied with the terminal NIL replaced by a pointer to <s>.

(NCONC <list> <s>)

normal; SUBR

NCONC is similar to APPEND in that it joins its two arguments into a single S-expression. NCONC actually modifies <list> to replace the terminal NIL with a pointer to <s>. The value of NCONC is its first argument, as modified by the action of the function. This can be contrasted with the value of APPEND which creates a new list structure and does not modify either of its arguments.

(CONC <list1> <list2> ... <list[n]>)

normal; FSUBR

CONC is similar to NCONC in that it concatenates its arguments into a single list. However, CONC accepts an arbitrary number of lists and returns <list1>, internally modified to be the concatenation of all of the lists in the order that they were presented. CONC modifies the existing list structure of all its arguments except the last.

(PAIR <list1> <list2>)

normal; SUBR

PAIR matches its two argument lists together, returning a resulting list whose length is equal to the length of its shorter argument. Each element of the resulting list is a dotted pair whose CAR is the corresponding element of <list1> and whose CDR is the corresponding element of <list2>. Thus, the first elements are paired, and the second elements are paired, and so forth.

(EFFACE <s> <list>)

normal; SUBR

EFFACE removes the first occurrence of the item <s> from <list>. EFFACE modifies the existing structure of the list.

(SUBLIS <list> <s>)

where <list> is an expression of the form

((<s1> . <s2>) (<s3> . <s4>) ... (<s[n-1]> . <s[n]>))

normal; SUBR

SUBLIS receives a list of dotted pairs, as indicated, for its first argument and an arbitrary S-expression as its second argument. The value of SUBLIS is the result of substituting the right part of a dotted pair for every occurrence of the left part of the dotted pair in <s>, the second argument of SUBLIS. Thus, SUBLIS allows the simultaneous substitution of a number of items in an arbitrary S-expression. SUBLIS does not modify existing memory structure. SUBLIS creates a new structure containing the various substitutions, but does not copy any unchanged substructure.

Example:

```
(SUBLIS "((A . 1)(B . 2)(C . 3)) "(A (B C B) A))
        = (1 (2 3 2) 1)
```

(SUBST <s1> <s2> <s3>)

normal; SUBR

The value of SUBST is the result of substituting <s1> for all occurrences of <s2> in <s3>. SUBST does not modify existing list structure. SUBST creates an entirely new list structure containing the substitutions, but does not copy any unchanged substructures.

Example:

```
(SUBST "A "B "(A (B C B) A)) = (A (A C A) A)
```

## 4.7 PROPERTY LIST MANIPULATION FUNCTIONS

Every literal atom in the system has a property list associated with it. The property list contains items of information about, or properties of, the atom. Users may assign properties of their own choice to atoms. Each property is characterized by an indicator, which is itself a literal atom and which defines what property is being mentioned, and by a value for that property, which may be any S-expression. The indicators INFO, PNAME, EXPR, SUBR, FEXPR, FSUBR, SYM, CMACRO, SMACRO, CSUBR, and CFSUBR have special meaning to the LISP system itself and should be avoided or used with care. Property lists do not have the same structure as ordinary lists (see chapter 3) and should be manipulated only by the following functions.

(PUT <litatom1> <litatom2> <s>)

pseudofunction; SUBR

If <s> is non-NIL, PUT searches the property list of <litatom1> for an occurrence of the indicator <litatom2>. If <litatom2> is found then PUT replaces the old associated value with <s>. If the indicator is not found PUT places a new element on the property list with indicator <litatom2> and value <s>. If <s> is NIL, the associated indicator is removed from the property list, if present; i.e., PUT calls (REMPROP <litatom1> <litatom2>). The value of PUT is its first argument.

(GET <litatom1> <litatom2>)

normal; SUBR

GET is the inverse of PUT. GET searches the property list of <litatom1> for a property list element whose indicator is <litatom2>. The value of GET is the value associated with that indicator if the indicator is present. Otherwise the value of GET is NIL. GET should not be used to obtain the PNAME property of an atom if the PNAME is going to be used with the character manipulation functions (see section 4.10). Instead, the function GETPN (see below) should be used.

(DEFLIST <s> <litatom>)

where <s> is an expression of the form

```
((<litatom1> <s1>) (<litatom2> <s>) ...
  (<litatom[n]> <s[n]>))
```

pseudofunction; SUBR

DEFLIST is used to simultaneously assign a new property to a number of different atoms. The first argument is a list of sublists, each of two elements as shown above. The second element of each of the sublists is placed on the property list of the first element of the sublist as the value of a property whose indicator is the second argument of DEFLIST, that is, <litatom>. The value of DEFLIST is a list of the <litatom[i]>'s which appear in the first argument; that is, a list of all of the atoms to which the new property has been assigned by DEFLIST.

(DEFINE <s>)

where <s> is an expression of the form

```
((<litatom1> <s1>) (<litatom2> <s2>) ...
   (<litatom[n]> <s[n]>) )
```

pseudofunction; SUBR

The argument of DEFINE is equivalent in form to the first argument of DEFLIST, with the exception that the second element of each of the sublists should be either the name of a function or a lambda expression. DEFINE makes <s[i]> a property of <litatom[i]> with the indicator EXPR. The primary purpose of DEFINE is the definition of LISP functions. DEFINE is equivalent to a call on the function DEFLIST with the second argument being the atom EXPR. The value of DEFINE is a list of the atoms which have received the function definitions.

(DEF ((<litatom1> <lat1> <s1>) (<litatom2> <lat2> <s>) ...
   (<litatom[n]> <lat[n]> <s[n]>) )

pseudofunction; FSUBR

DEF acts exactly as;

```
(DEFINE "((<litatom1> (LAMBDA <lat1> <s1>))
   (<litatom2> (LAMBDA <lat2> <s2>)) ...
   (<litatom[n]> (LAMBDA <lat[n]> <s[n]>)) ))
```

It thus is a convenient abbreviated form of DEFINE which eliminates the need to explicitly state the LAMBDA flag for each <litatom[i]> function defined.

(DEFF ((<litatom1> <lat1> <s1>) (<litatom2> <lat2> <s2>) ...
   (<litatom[n]> <lat[n]> <s[n]>) )

pseudofunction; FSUBR

DEFF is similar to DEF with respect to its abbreviated call upon DEFINE. However, DEFF defines <litatom1> through <litatom[n]> as FEXPRs rather than EXPRs (see section 4.1).

(GETD <litatom>)

pseudofunction; SUBR

GETD returns the functional definition of <litatom> by GETting the value associated with the EXPR or FEXPR indicator on <litatom>'s property list. GETD also forces a retrieval of the function <litatom> from the disk if <litatom> has previously been the argument of a call to DISKOUT (see section 7.6 on the LISP virtual memory facility for functions). If <litatom> does not have an EXPR or FEXPR indicator on its property list, GETD returns NIL. Thus system functions with SUBR or FSUBR definitions are not retrieved by GETD.

(PUTD <litatom> <s>)

pseudofunction; SUBR

PUTD replaces the definition of <litatom> associated with its EXPR or FEXPR indicator by <s>, using the same indicator already on <litatom>. If <litatom> does not have an EXPR or FEXPR definition, PUTD does nothing. The value of PUTD is always its first argument.

(QKEDIT <litatom> <s1> <s2>)

pseudofunction; SUBR

QKEDIT substitutes <s1> for <s2> within the definition of any EXPR or FEXPR function <litatom>. QKEDIT is equivalent to

(PUTD <litatom> (SUBST <s1> <s2> (GETD <litat>))

(GETPN <litatom>)

normal; SUBR

GETPN returns the PNAME property of the <litatom> structured as a list of full words; that is, a list whose CAR fields point directly to the full words which contain the characters of the print name of the <litatom>. Such a list structure is necessary for manipulation by some of the functions defined in section 4.10.

(REMPROP <litatom1> <litatom2>)

pseudofunction; SUBR

REMPROP removes the property whose indicator is <litatom2> from <litatom1>. The value of REMPROP is <litatom1>.

(FLAG <lat> <litatom>)

pseudofunction; SUBR

FLAG PUTs the property indicator <litatom>, with the associated value *T*, on the property list of each of the atoms in <lat>. The value of FLAG is NIL.

(REMFLAG <lat> <litatom>)

pseudofunction; SUBR

REMFLAG removes all occurrences of the indicator <litatom> from the property list of each of the atoms in <lat>. The value of REMFLAG is always NIL.

(PROP <litatom1> <litatom2> <fnexp>)

normal; SUBR

PROP is similar to GET in action. The property list of <litatom1> is searched for the indicator <litatom2>. If such a property is found, the entire property list of <litatom1> beginning with property <litatom2> is returned by PROP. If <litatom2> is not an indicator on the property list of <litatom1>, then <fnexp>, which must be a function of no arguments, is applied and the value returned by PROP is the value of <fnexp>.

## 4.8 FUNCTIONS WITH FUNCTIONAL ARGUMENTS

These functions all sequentially scan a list, performing an operation on the list with each scanning operation. The operation to be performed is specified by a functional argument. Thus these functions have variable semantics. Although FUNCTION must be used with functional arguments for some user-defined functions, it is never necessary with these functions.

(MAP <list> <fnexp>)

pseudofunction; SUBR

MAP applies the functional expression <fnexp> to <list>, then to (CDR <list>), (CDDR <list>), and so on, until <fnexp> has been applied to all non-NIL CDRs of <list>. <fnexp> must describe a function of only one argument. The value of MAP is always NIL. MAP is always used to perform some kind of side effect computation.

(MAPC <list> <fnexp>)

pseudofunction; SUBR

MAPC is similar to MAP except that it applies <fnexp> to successive CARs of <list> instead of to the entire <list>. Thus MAPC first applies <fnexp> to (CAR <list>), then to (CADR <list>) and so on until some CD...DR of <list> is NIL. MAPC always returns NIL as its value (compare this with the definition of MAPCAR).

(MAPLIST <list> <fnexp>)

normal; SUBR

MAPLIST is similar to MAP except that its value is a list of each of the values produced when <fnexp> is applied to successive CDRs of <list>. These elements are CONSed together as they are generated. <fnexp> must be a function of only one argument.

Example:

```
(MAPLIST '(A B C) (FQUOTE (LAMBDA (X) (CONS X NIL))))
        = (((A B C))((B C))((C)))
```

(MAPCON <list> <fnexp>)

normal; SUBR

MAPCON is similar to MAPLIST except that the value is the list which results from concatenating the individual results of applying <fnexp> to successive CDRs of <list>. This implies that each value returned by the <fnexp> must itself be a list or else erroneous results may occur when the elements are concatenated together. <fnexp> must be a function of only one argument.

Example:

```
(MAPCON "(A B C) (FQUOTE (LAMBDA (X) (CONS X NIL))))
          = ((A B C)(B C)(C))
```

(MAPCAR <list> <fnexp>)

normal; SUBR

MAPCAR is similar to MAPLIST in that it returns a list of the results of the individual applications of <fnexp>. However, <fnexp> is applied at each step to CAR of the successive CDRs of <list> instead of to the CDR itself. <fnexp> must be a function of only one argument.

Example:

```
(MAPCAR "(A B C) (FQUOTE (LAMBDA (X) (CONS X NIL))))
          = ((A)(B)(C))
```

(SEARCH <list> <fnexp1> <fnexp2> <fnexp3>)

normal; SUBR

SEARCH applies <fnexp1> to <list> and successive CDRs of <list> until it happens that the value of <fnexp1> is true. If this condition occurs, then SEARCH applies <fnexp2> to the remainder of <list> at that time. If the entire list is exhausted without any application of <fnexp1> giving a true value, then <fnexp3> is applied to NIL. SEARCH is therefore used to apply some function to a portion of the list depending on a condition which must be met by some element of the list. Each <fnexp> must be a function of only one argument.

Example:

```
(SEARCH "((A . 1)(B . 2)(C . 3))
        (FQUOTE(LAMBDA(X)(EQ (CAAR X) "B)))
        (FQUOTE(LAMBDA(X)(CDAR X)))
        (FQUOTE(LAMBDA(X)(ERROR "(SEARCH FAILURE)))))
          = 2
```

(SASSOC <s> <list> <fnexp>)

normal; SUBR

SASSOC searches its second argument (which must be composed from non-atomic elements) for an element whose CAR is EQUAL to <s>. If such an element is found, the value of SASSOC is that element. Otherwise, the function <fnexp> of no

arguments is applied and its value returned as the value of SASSOC.

Example:

```
(SASSOC "B "((A . 1)(B . 2)(C . 3)) NIL) = (B . 2)
```

## 4.9 ARITHMETIC FUNCTIONS AND PREDICATES

All of the functions described in this section must receive numeric atoms as their arguments. There are three types of numbers in LISP: octal numbers, integers, and floating-point numbers. Octal numbers and integers are not distinguished for arithmetic operations; both are considered to be fixed point. They differ only in their printed representation. All of the arithmetic functions may accept either fixed-point or floating point inputs or a mixture of the two. The result is floating point if any of the inputs are floating point and is fixed point otherwise.

(ADD1 <number>)

normal; SUBR

ADD1 increments <number> by 1 and returns the incremented value as its result. A fixed-point result is of integer type.

(SUB1 <number>)

normal; SUBR

SUB1 decrements <number> by 1 and returns the decremented quantity as its value. A fixed-point result is of integer type.

(PLUS <number> <number> ... <number>)
(+ <number> <number> ... <number>)

normal; FSUBR

PLUS evaluates all its arguments and returns as its value their sum. A fixed-point result is of integer type. + is a synonym for PLUS.

(DIFFERENCE <number1> <number2>)
(- <number1> <number2>)

normal; SUBR

DIFFERENCE returns as its value the difference <number1> - <number2>. A fixed-point result is of integer type. - is a synonym for DIFFERENCE.

(MINUS <number>)

normal; SUBR

MINUS returns the negative of its argument. A fixed-point result is of integer type.

(TIMES <number> <number> ... <number>)
(* <number> <number> ... <number>)

normal; FSUBR

TIMES evaluates all its arguments and returns as its value their product. A fixed-point result is of integer type. * is a synonym for TIMES.

(QUOTIENT <number1> <number2>)
(/ <number1> <number2>)

normal; SUBR

QUOTIENT returns as its value the quotient of <number1>/<number2>. The quotient of two fixed-point numbers is the integer part of the quotient and is of integer type. Thus, QUOTIENT(1 2) returns the value 0. / is a synonym for QUOTIENT.

(REMAINDER <number1> <number2>)

normal; SUBR

REMAINDER returns as its value the remainder of the division <number1>/<number2>. The remainder is calculated by the usual formula:

remainder = dividend - ( /quotient/ * divisor)

where /quotient/ is the greatest integer contained within the quotient. A fixed-point result is of integer type.

(DIVIDE <number1> <number2>)

normal; SUBR

DIVIDE returns as its value a list of two numbers wherein the first element is (QUOTIENT <number1> <number2>) and the second element is (REMAINDER <number1> <number2>).

(RECIP <number>)

normal; SUBR

RECIP returns as its value the reciprocal of its argument. That is, it is equivalent to (QUOTIENT 1 <number>). The reciprocal of any fixed-point quantity larger than one is an integer zero.

(MAX <number> <number> ... <number>)

normal; FSUBR

MAX evaluates all its arguments and returns as its value the algebraically largest <number> in the set.

(MIN <number> <number> ... <number>)

normal; FSUBR

MIN evaluates all its arguments and returns as its value the algebraically smallest <number> of the set.

(LOGAND <number> <number> ... <number>)

normal; FSUBR

LOGAND evaluates all its arguments and returns their bit-by-bit logical product (logical AND). Each <number> is treated as a 60-bit quantity. The value of LOGAND is always of octal type.

(LOGOR <number> <number> ... <number>)

normal; FSUBR

LOGOR evaluates all its arguments and returns their bit-by-bit logical sum (inclusive OR). Each <number> is treated as a 60-bit quantity. The value of LOGOR is always of octal type.

(LOGXOR <number> <number> ... <number>)

normal; FSUBR

LOGXOR evaluates all its arguments and returns their bit-by-bit logical difference (exclusive OR) with association to the left. Each <number> is treated as a 60-bit quantity. The value of LOGXOR is always of octal type.

(LEFTSHIFT <number> <fixnumber>)

normal; SUBR

LEFTSHIFT performs a shifting operation on its 60-bit first argument. <fixnumber> is a shift count of the number of bits <number> is to be shifted. If the shift count is positive the shift is left, end-around circular. If negative the shift is right, end-off with sign extension. The result of the LEFTSHIFT function is always of octal type.

(FIX <flnumber>)

normal; SUBR

The value of FIX is the largest integer contained in the floating point number. The value is of integer type.

(FLOAT <fixnumber>)

normal; SUBR

FLOAT returns a floating-point number whose value is the same as that of the fixed-point argument. <fixnumber> must be less than 2**48 in magnitude for this function to give the proper result.

(OCTAL <number>)

pseudofunction; SUBR

OCTAL converts its argument into a number of octal type. The print image of the resulting number is the octal representation of the original value. No conversion from floating point to fixed point is made. OCTAL actually modifies its argument directly. It does not make a copy of the argument.

(RANDOM <number>)

normal; SUBR

RANDOM returns a new random number in the range 0-1 each time it is called with a zero argument. If called with an argument between 0 and 1, it returns that argument, as its value, and on subsequent calls with a 0 argument, it will return new random numbers belonging to a new sequence begun by the call with a nonzero argument.

(ZEROP <number>)

predicate; SUBR

ZEROP returns <number> as its value if its argument is a numeric zero. It returns false otherwise.

(ONEP <number>)

predicate; SUBR

ONEP returns <number> as its value if its argument has the value 1 (either fixed-point or floating-point). It returns false if its argument is not equal to 1.

(MINUSP <number>)

predicate; SUBR

MINUSP returns <number> as its value if its argument is a negative number. It returns false if its argument is a positive number. Only the sign of the number is tested. Therefore, MINUSP returns true if <number> is -0.

(FIXP <number>)

predicate; SUBR

FIXP returns <number> as its value if its argument is a fixed-point number (either an integer or an octal number). It returns false otherwise.

(FLOATP <number>)

predicate; SUBR

FLOATP returns <number> as its value if its argument is a floating-point number, or false otherwise.

(LESSP <number1> <number2>)
(< <number1> <number2>)

predicate; SUBR

LESSP returns true if <number1> is strictly less than <number2>, or false otherwise. If either of the arguments is floating point, the comparison uses floating-point arithmetic. < is a synonym for LESSP.

(GREATERP <number1> <number2>)
(> <number1> <number2>)

predicate; SUBR

GREATERP returns true if <number1> is strictly greater than <number2>, or false otherwise. If either of the arguments is floating point, the comparison uses floating-point arithmetic. > is a synonym for GREATERP.

## 4.10 CHARACTER MANIPULATION FUNCTIONS

Normally in LISP, one is not concerned with the constituents of the atomic symbols, as atoms are treated as indivisible units. However, it is occasionally necessary to decompose an atom into its constituent characters or to build an atomic symbol from a group of individual characters. The functions described in this section assist in these operations.

A character in LISP is represented by a literal atom whose print name is the single character. Thus, the characters may be manipulated by any of the functions which may be used to manipulate atomic symbols. This means also that there is a difference between single digit numbers and the literal atoms whose names are the digit characters. The user should keep this difference in mind. There exists an internal buffer into which up to 120 characters may be placed. This buffer is useful for accumulating characters which are being used to construct atoms, and this buffer is manipulated by some of the functions described in this section.

(CLEARBUFF)

pseudofunction; SUBR

This function of no arguments clears the internal buffer and resets the counter of its contents to zero. The value of CLEARBUFF is NIL. This function should be executed before any of the other functions which affect the buffer are used.

(PACK <character>)

pseudofunction; SUBR

PACK places the character which is its argument into the next sequential position in the internal buffer, following all characters which have previously been placed there. The value of PACK is NIL.

(MKNAM)

pseudofunction; SUBR

MKNAM returns as its value a list of full words created from the contents of the internal buffer. The characters are taken from the buffer ten at a time and placed into full words containing up to ten display character codes. The last full word is filled with zero bits if necessary. The resulting list is one whose CAR fields point directly to the full words containing the character codes. This type of list structure is not a normal LISP list. The internal buffer is cleared after this function is completed.

(INTERN <fwl>) or (INTERN <litatom>)

pseudofunction; SUBR

The argument of INTERN can be either a <fwl> or a <litatom>. INTERN searches the oblist for an atom whose print image matches <fwl> or <litatom>. If such an atom is found, INTERN returns that atom. Otherwise, INTERN puts <litatom> or an atom it creates from the <fwl> on the oblist and returns this atom as its value.

(NUMOB)

pseudofunction; SUBR

NUMOB expects the internal buffer to contain a sequence of characters defining a number. The syntax of the numbers is the same as if the characters had been punched on a data card. The value of this function is the LISP number which corresponds to that character representation. If the characters in the buffer do not form a legal LISP number, then NUMOB returns as its value a literal atom whose print name contains those characters but is not INTERNed. The buffer is empty after this function is completed.

(UNPACK <fw>)

pseudofunction; SUBR

The argument to UNPACK must itself be a full word; that is, a memory word containing up to ten display code characters, filled with zero bits if fewer than ten characters are present. The value of UNPACK is a list of literal atoms, one atom for each character contained in the argument of UNPACK. The print names of the literal atoms in the list correspond to the character codes in the full word.

(IMAGEL <atom> <boolean>)

pseudofunction; SUBR

IMAGEL returns the integer length of the printed representation of <atom>. If <boolean> is false, the image used is the normal printed image. If <boolean> is true, the image used includes any delimiter or escape symbols necessary to reproduce a readable form of the atom. This function facilitates formatting printed output.

(NUMTOATOM <number>)

pseudofunction; SUBR

NUMTOATOM creates and returns a literal atom whose print name is equivalent to the representation <number> would have if printed. The literal atom is not placed on the oblist. The format used for floating-point numbers is under the control of NFORMAT (see section 5.8).

(COMPRESS <lat> <boolean>)

pseudofunction; SUBR

<lat> must be a list of single-character atoms. If <boolean> is false or omitted, COMPRESS creates and returns an atom identical to the result of a READ of the same characters as from an input file. That is, if the characters are a legal numeric representation, a numeric atom is returned; otherwise, a standard literal atom of the first 30 characters, created according to the syntax of section 3.1.2.1, is returned. If <boolean> is true, then special characters which would ordinarily terminate the reading of an atom from the input file are included in the first 30 characters of the atom returned. If <boolean> is true, the atom is INTERNed; otherwise it is not.

Examples:

```
(COMPRESS "( A B #.  #, #.)) = AB
(COMPRESS "(A B #.  #, #.)  T) = AB.,.
```

(EXPLODE <atom>)

pseudofunction; SUBR

EXPLODE returns a list of single-character atoms which if concatenated are equivalent to the print image of <atom>.

<atom> may be either numeric or literal. EXPLODE is the inverse of COMPRESS, i.e.,

(COMPRESS (EXPLODE <atom>) T) = <atom>

(LITER <s>)

predicate; SUBR

LITER returns <s> as its value if <s> is a literal atom whose print name is a single alphabetic character (A-Z). The value of LITER is false in all other circumstances.

(DIGIT <s>)

predicate; SUBR

DIGIT returns <s> as its value if <s> is a literal atom whose print name is a single numeric character (0-9). The value of DIGIT is false under all other circumstances. Notice that DIGIT is false for the single-digit numeric atoms 0-9.

(OPCHAR <s>)

predicate; SUBR

OPCHAR returns <s> as its value if <s> is a literal atom whose print name is one of the single characters +, -, /, or *. OPCHAR returns false under all other circumstances.

## 4.11 DEBUGGING AND ERROR PROCESSING FUNCTIONS

These functions are used to give the user some control over the operation of the LISP system or to enable him to obtain more information about the operation of his functions.

(ERROR <s>)

pseudofunction; SUBR

ERROR causes a recoverable error to occur. The argument to ERROR appears as part of an error message printed in response to this function. The message is:

***** ERROR: <s>

Continued execution of the current top-level expression is suspended unless the ERROR function is executed under the control of an ERRORSET (see below). This function has no value as such.

(DIE <s>)

pseudofunction; SUBR

DIE behaves much like ERROR (see above), except that it is irrevocably fatal and causes the LISP run to terminate. <s> is printed as part of a message:

!!!!! KILLED: <s>

(ERRORSET <exp> <boolean1> <boolean2>)

pseudofunction; SUBR

ERRORSET is a function which allows the LISP system to recover from a recoverable error without terminating the execution of the current top-level expression. The first argument of ERRORSET is evaluated a second time. If no error occurs during the evaluation, then the result of ERRORSET is a list of the value of the expression. If an error does occur during the evaluation the result of ERRORSET is NIL. If a computation causes a recoverable error to occur, the system returns to the point at which it entered the last ERRORSET function. <boolean1> controls the printing of the error message in response to the error detected. If <boolean1> is true the error message is printed; if false, the error message is not printed. <boolean2> controls printing of the backtrace with the error message. If both <boolean1> and <boolean2> are true, the backtrace is also printed. Note that ERRORSET is a SUBR; therefore the result of evaluating <exp> is passed to ERRORSET as its first argument. If ERRORSET is to be passed an expression to be directly evaluated, <exp> must be the QUOTEd form of this expression. This case is particularly important since ERRORSET does not cause recovery from an error produced before it receives the value of <exp>.

Example: (use of ERRORSET to detect unbound variables)

If the variable BETA is bound to the variable ALPHA, but ALPHA has no value, then

```
(ERRORSET "BETA) = (ALPHA)
(ERRORSET BETA) = NIL
(ERRORSET "ALPHA) = NIL
```

and (ERRORSET ALPHA) would produce an unbound variable error message.

(TRACE <lat>)

pseudofunction; SUBR

TRACE FLAGs the property list of each of the atoms in the <lat> with the indicator TRACE. Subsequently, for a function of type EXPR and SUBR whose name was so flagged, the actual arguments supplied to that function are printed every time that function is entered, and the actual value resulting from that function is printed every time the function is exited. This feature enables the user to get a complete history of the activity of a selected set of functions. To be effective, the TRACE indicator must be

placed on the property list of functions after those functions have been defined. The value of TRACE is always NIL.

The format of the printing is as follows:

[<xx>] ARGUMENTS OF <name>

This line is printed each time a traced function is entered. <name> represents the name of the function. <xx> is a two-character code which is changed every time a new "ARGUMENTS OF" message appears and is designed to enable the user to tell which occurrence of a call on the function is being described. The <xx> begins with A and sequences through Z, etc., up to ;;. Following this line, each argument received by the indicated function is stated on a separate line. The line

[<xx>] VALUE OF <name>

is printed each time a traced function is exited. <name> represents the name of the function which is being exited. The actual value returned by the function at this point is printed beginning on the next line. These two messages appear in pairs. Every function that is entered must ultimately be exited. The indicator <xx> is used to match the corresponding function entry and exit lines, since these lines may be separated by many pages of output if extensive tracing is used. Although tracing of functions can be very useful in the debugging process, the user is cautioned to be economical in his use of tracing as the extra printing consumes both time and paper. If too much tracing is done the user may receive a large amount of output which has too much detail.

(UNTRACE <lat>)

pseudofunction; SUBR

UNTRACE is the inverse of TRACE. It removes the flag TRACE from the property list of each atom in the <lat> . Any of those atoms which are function names will subsequently not be traced when applied as a function. The value of UNTRACE is always NIL.

(TRACESET <lat>)

pseudofunction; SUBR

The atoms in the <lat> should be the names of functions which are composed of a PROG expression. These functions are marked in such a way that while they are in control, every execution of a SET or SETQ causes a message to be printed, showing the value assigned to the variable at that point in time. The message consists of the variable name followed by an equal sign followed by the value which was assigned to that variable. This process is useful for tracing the internal operation of functions written using PROG expressions. If any atom in <lat> is not the name of a function with a top-level PROG, then that atom is not processed. An error message is generated after the entire <lat> is processed. The value of TRACESET is its unchanged argument. SET and SETQ tracing continues downward through

all called functions until another PROG expression which is not TRACESET is entered.

(UNTRACESET <lat>)

pseudofunction; SUBR

UNTRACESET is the reverse of TRACESET. The atoms in the <lat> should be the names of functions composed of PROG expressions. UNTRACESET removes the marks set by the TRACESET function so that subsequent execution of those functions will no longer cause the printing produced by the TRACESET. By the value of UNTRACESET is its unchanged argument.

(LOOK <fixnumber>)

normal; SUBR

<fixnumber> is treated as a machine address. LOOK returns an octal number whose value is equal to the contents of the designated location. There is no restriction on what address may be looked at except that it must be within the bounds determined by the current field length.

## 4.12 MISCELLANEOUS FUNCTIONS

These functions provide the user with various capabilities for affecting or gaining information about the system operation.

(TEMPUS)

pseudofunction; SUBR

This function of no arguments returns an integer value which is the TM time in milliseconds since the LISP run was begun. TEMPUS is initialized to zero whenever a new control command call to LISP is entered. TEMPUS is not reinitialized by the use of overlays.

(TM)

pseudofunction; SUBR

TM returns an integer equal to the number of milliseconds of TM time the job has used so far. TM time differs from TEMPUS time in that TM time includes all work done by the user during the current batch job or since logging in, regardless of whether this time was spent running LISP. TM time thus reflects the operating system's view of the user's computer use, rather than LISP's view.

(DATE)

pseudofunction; SUBR

DATE returns a special literal atom containing 10 characters of display code formed (left to right) from: a blank, the first two digits of the current date, a second

blank, the first three letters of the current month, a third blank, and the last two digits of the current year (e.g., an atom equivalent to $$/ 04 JUL 76/ for the date July 4, 1976).

(TIME)

pseudofunction; SUBR

TIME returns the current time of day as a special literal atom of 10 characters, composed from left to right as follows: a blank, two digits representing the current hour (24 hour basis), a period, two digits representing the number of minutes after the hour, a second period, two digits representing the number of seconds after the minute, and a final period (e.g., 2:31 PM would be equivalent to $$/ 14.31.00./).

(DEADSTART)

pseudofunction; SUBR

DEADSTART returns an integer equal to the number of milliseconds elapsed since the last machine deadstart.

(TMLEFT)

pseudofunction; SUBR

TMLEFT returns an integer equal to the number of milliseconds left for the current job. TMLEFT is useful only to batch programs; it returns 0 whenever called from an interactive program.

(SECTORS)

pseudofunction; SUBR

SECTORS returns an integer equal to the number of disk sector transfers performed by the current job.

(PP)

pseudofunction; SUBR

PP returns an integer equal to the number of PP seconds used by the current job. This value is determined by the operating system and is not specifically related to LISP use of PP time.

(CP)

pseudofunction; SUBR

CP returns an integer equal to the cumulative number of milliseconds of CP time used by this job. CP is determined by the operating system and is not initialized to zero at the beginning of the LISP run.

(RECLAIM)

pseudofunction; SUBR

This function of no arguments causes a garbage collection to occur whether it is needed or not. The value of RECLAIM is always NIL. If the Garbage Collector Message Control (//GC) is in effect (see sections 4.14 and 2.1), then a line is printed showing the number of words collected by the garbage collector.

(FULL)

pseudofunction; SUBR

This function of no arguments returns an integer result which is the number of full words currently avialable. This function consumes one full word and one free word each time it is called.

(FREE)

pseudofunction; SUBR

This function of no arguments returns an integer value which is the number of free words currently available. This function consumes one free word and one full-word each time it is called.

(REMOB <litatom> <boolean>)

pseudofunction; SUBR

This function removes its argument <litatom> from the oblist provided (1) <litatom> is not a special LISP system atom, or (2) <boolean> is true and <litatom> is not internally referenced by the LISP system. The net effect is that if the atom is not a member of any other list structure in memory, then the words comprising that atom will be collected during the next garbage collection. Furthermore, if subsequently an atom with the same name is read, an entirely new atom will be created and the old one will not be assumed. The value of REMOB is <litatom> if the <litatom> was removed from the oblist, or NIL otherwise.

(ADDR <s>)

pseudofunction; SUBR

This function returns as its value an octal number which is a representation of the address of <s>. ADDR enables one to determine the actual machine address of any given S-expression.

(ADDRP <s>)

predicate; SUBR

ADDRP returns <s> if <s> is actually an address outside of the free space and full-word space memory areas. This could be the case if <s> points into the LISP interpreter, the push-down stack, or binary program space. If <s> is an address within free space or full-word space, ADDRP returns false. Thus, ADDRP returns false if <s> is an S-expression.

(GENSYM <letter>)

pseudofunction; FSUBR

GENSYM creates an entirely new literal atom each time it is called. The literal atom has a name of the form <xnnnnn> where <x> is a single letter and <nnnnn> is an integer. The letter used to compose the name is the argument of GENSYM. Each time GENSYM is executed with an argument <letter>, it returns the next atom in the sequence for that letter. If GENSYM is executed with no argument, the atom generated is the next in the sequence started by the last execution of GENSYM with a <letter> argument. Each atom in the sequence differs from the previous one by having the integer part of the name incremented by one. If no argument is ever specified for GENSYM the letter G is used. The atoms created by GENSYM are not placed on the oblist. Therefore, if an atom with the same name is subsequently read, it will not correspond to the atom created by GENSYM. Users must therefore save the names of all GENSYM created atoms in order to reference them. These names can be saved easily by binding some variable to the value returned by GENSYM, or by keeping such values in a list or accessible via known property list indicators.

Examples: (assuming the calls are made in the order given below)

```
(GENSYM) = G00001      (first call)
(GENSYM T) = T00001
(GENSYM) = T00002
(GENSYM G) = G00002
```

(ALIST)

pseudofunction; SUBR

This function of no arguments creates an association list which shows all variable bindings at the time the function is called. The function proceeds by scanning the oblist and creating a dotted pair for every atom which currently has a binding. Each pair consists of the atom dotted with its binding. Thus a call on this function returns a dotted pair list reflecting the current evaluation environment of the LISP system.

(BACKTRACE)

pseudofunction; SUBR

This function of no arguments returns a list of the current indicators on the system stack. See sections 8.2 and 8.4 for a description of these items.

## 4.13 ARRAYS

Although the primary data structure of LISP is the linked list, an array structure is useful in many problems. UT LISP provides a primitive array capability. Array storage is allocated in binary program space (see section 3.3.1) with two array elements per word. Each array element can be a pointer to an arbitrary S-expression and is preserved during garbage collection. Memory allocated to an array cannot be reused by LISP for other purposes.

(MKARRAY <litatom> <list>)

where <list> has as its value an expression of the form

(<fixnumber1> <fixnumber2> ... <fixnumber[n]>)

pseudofunction; SUBR

MKARRAY defines and allocates an n-dimensional array whose "name" is <litatom>. Each <fixnumber[k]> is a dimension of the array such that elements of the array are indexed in the [k]th dimension by values of 0 through (<fixnumber[k]> - 1). A pointer to the array space is placed on the property list of <litatom> with the indicator ARRAY. Each array element is initialized to NIL. The value of MKARRAY is NIL. The total number of elements in the array is the arithmetic product of all the <fixnumber[k]> for k=1,n.

(SETEL <litatom> <list> <s>)

where <list> has as its value an expression of the form

(<fixnumber1> <fixnumber2> ... <fixnumber[n]>)

pseudofunction; SUBR

If the <litatom> has an ARRAY property, SETEL stores the S-expression <s> into the element indexed by the list of subscripts in the second argument. Each subscript may range from 0 to (i[k] - 1), where i[k] is the value of the [k]th dimension used in the call to MKARRAY which allocated the array. If fewer subscripts are used than are defined by the dimensionality of the array, the effect is as though they were omitted from the right and values of 0 are used in their places. For example, if A is a two-dimensional array, (SETEL "A "(1) NIL) is equivalent to (SETEL "A "(1 0) NIL). The value of SETEL is <s>, the third argument. In the interest of speed of access no validity checking is performed on the subscript list.

(GETEL <litatom> <list>)

where <list> has as its value an expression of the form

(<fixnumber1> <fixnumber2> ... <fixnumber[n]>)

pseudofunction; SUBR

If <litatom> has an ARRAY property, GETEL returns the value stored in the element indexed by the list of subscripts in the second argument. All conventions regarding treatment of subscripts are the same as defined for SETEL above.

(CLARRAY <litatom>)

pseudofunction; SUBR

If <litatom> has an ARRAY property, CLARRAY resets all elements of the array (regardless of dimensions) to NIL. The value of CLARRAY is always NIL. If <litatom> does not have an ARRAY property, an illegal argument error results.

## 4.14 SYSTEM CONTROL

It is often desirable to be able to change certain aspects of LISP's behavior while a job is running, particularly those set by control command parameters. In UT LISP these controls are effected by means of a set of system-defined variables and user-callable functions. The function LISTING effects control of the L, P, and N control command parameters. The variables //EXPERT, //FATAL, //GC, //TIMING and //ZAP control the E, F, G, T, and Z parameters, respectively. Several other variables of the system may also be set or interrogated by user programs at any time. Users are hereby warned to avoid using the names of these variables within their own functions except for the purposes described.

The descriptions of the LISTING function and these system variables are given below.

(LISTING <s>)

pseudofunction; SUBR

This function simulates the L, P, and N parameters of the LISP control command. If the argument is the atom P, then subsequent top-level expressions or doublets are listed with parenthesis counting. If the argument is the atom L, then subsequent top-level expressions or doublets are listed without a parenthesis count. If the argument is the atom N, then only user-originated output and error messages appear on SYSOUT. If the argument is anything else, then subsequent top-level expressions or doublets are printed by LISP, but not in source image form. The value of LISTING is its argument. LISTING with an argument of P has no effect if SYSIN and SYSOUT are the same file, e.g., when LISP is used interactively.

//MODE - toplevel function

Value: (<function> . <fixnumber>)

Default value: (EVAL . 1)

<function> is the function which is evaluated at each step of the main loop of LISP. <fixnumber> is the number of arguments expected by that function. The <fixnumber> is used to control the main loop input mechanism so that the correct number of S-expressions are read from SYSIN before evaluating <function> with these arguments. If one of the S-expressions read is the atom STOP, the main loop ignores any others already read and starts over. If the first S-expression is the atom FIN or if the end-of-file on SYSIN is read, the main loop terminates. When the LISP system is first called by control command, //MODE is initialized to (EVAL . 1). If //MODE is set to some value of incorrect format, LISP prints an appropriate error message. In batch mode this error aborts the program.

//INPUT - interpreter input function

Value: <expression>

Default value: (INPUT (QUOTE SYSIN))

<expression> should be some LISP expression which when evaluated reads information from SYSIN and returns an S-expression corresponding to the information read. <expression> is evaluated as many times as given by (CDR //MODE) at each step of the main loop. If <expression> returns the atom STOP, FIN, or $EOF$, the main loop interprets those atoms as described under //MODE above.

//OUTPUTA - interpreter echo output function

Value: <function>

Default value:

```
(LAMBDA (=====/////)
   (OR (AND (NOT (ATOM =====/////))
            (EQ (CAR =====/////) (QUOTE DEFINE)))
       (OUTPUT //SYSOUT =====/////  NIL)))
```

The <function> must be a function of one argument. This function is used to print each of the S-expressions read by the interpreter input function when the listing control parameters specify printing.

//OUTPUTB - interpreter result output function

Value: <function>

Default value:

```
(LAMBDA (=====/////) (OUTPUT //SYSOUT =====/////))
```

The <function> must be a function of one argument. This function is used to print the result of each top-level evaluation.

//PLEVEL - print level control

Value: <fixnumber>

Default value: 65536 (i.e., 2**16)

<fixnumber> defines the number of levels of parenthesis nesting which is printed by the system output routines. Information deeper in the structure is not printed, but is represented by ** in the output. Variation of the print level can be very useful for cases in which the structure of a list is desired, but its detailed contents may be uninteresting.

Examples:

For //PLEVEL = 0,

| S-expression | Printed Representation |
|---|---|
| ATOM | ** |
| (X) | ** |
| (A B C) | ** |
| (COND ((NULL X) Y)) | ** |

For //PLEVEL = 1,

| S-expression | Printed Representation |
|---|---|
| ATOM | ATOM |
| (X) | (**) |
| (A B C) | (** ** **) |
| (COND ((NULL X) Y)) | (** **) |

For //PLEVEL = 2,

| S-expression | Printed Representation |
|---|---|
| ATOM | ATOM |
| (X) | (X) |
| (A B C) | (A B C) |
| (COND ((NULL X) 7)) | (COND (** **)) |

//PLIMIT - list length print control

Value: <fixnumber>

Default value: 65536 (i.e., 2**16)

<fixnumber> defines the number of elements of a list which will be printed by the system output routines. If //PLIMIT has value <n>, the first <n> elements of each list are printed and if there are more than <n>, an ellipsis "..." will be printed to indicate their absence. This control is useful when it is desired to reduce debugging output.

Examples:

For //PLIMIT = 0,

| S-expression | Printed Representation |
|---|---|
| ATOM | ATOM |
| (A B C) | ( ... ) |

For //PLIMIT = 1,

| S-expression | Printed Representation |
|---|---|
| (A B C) | (A ... ) |

For //PLIMIT = 2,

| S-expression | Printed Representation |
|---|---|
| ((A B C)(X Y Z) P D Q)) | ((A B ...)(X Y ...) ...) |

//TPLEVEL - trace print level control
//TPLIMIT - trace list length print control

Value: <fixnumber>

Default value: 4

These controls function in the same way as //PLEVEL and //PLIMIT, respectively, but are effective only when trace information is being printed.

//PCSR - CSR field print control

Value: <boolean>

Default value: NIL

When //PCSR is true, the system output routines print the contents of the CSR fields of non-atomic S-expressions. If the CSR field of a word contains an S-expression, it is printed between % characters, and immediately preceding the contents of the CAR field of that word.

Example:

Structure


Printed

(%X% A %Y% B %Z% C)

//TIMING - timing message control

Value: <boolean>

Default value: NIL

When //TIMING is true, the message

*TIME: <number>

Is printed after each top-level evaluation. <number> is the number of elapsed milliseconds for that evaluation. //TIMING is set true when the T parameter appears on the LISP control command.

//SAVING - result saving control

Value: <boolean>

Default value: NIL

When //SAVING is true, LISP automatically binds the result of each top-level evaluation to the atom PREVIOUS. This allows the result to be used conveniently in the subsequent evaluation.

//GC - garbage collector message control

Value: <boolean>

Default value: NIL

When //GC is true, each time a garbage collection occurs a message is printed:

///// GARBAGE COLLECTED: <n1> <n2>

<n1> is the number of free-space words recovered and <n2> is the number of full words recovered. //GC is set true when the G parameter appears on the LISP control command.

//ZAP - error interrupt control

Value: <boolean> or <fixnumber>

Default value: NIL

If //ZAP is non-NIL, then an interrupt is simulated whenever an error occurs (see chapter 9 for a discussion of interrupts). If the value of //ZAP is a <fixnumber> in the range 1 to 12, the corresponding interrupt function is used. If //ZAP is non-NIL but not a <fixnumber>, interrupt 1 is simulated. When //ZAP has a non-NIL value the system performs as though the user had specified the Z control command parameter upon calling LISP (see section 2.1). The //ZAP variable has no effect unless SYSIN and SYSOUT are the same file, e.g., in conversational mode.

//EXPERT - expert mode control

Value: <boolean>

Default value: NIL

//EXPERT allows the user to change the expert mode control parameter from inside LISP (see section 2.1). Setting //EXPERT to true allows primitive operations to be performed on atoms and is equivalent to specifying the E parameter on the LISP control command. Setting //EXPERT to false disables expert mode.

//FATAL - error fatality control

Value: <boolean>

Default value: NIL

Setting //FATAL to true causes any subsequent error in the user's program to terminate execution. Setting the //FATAL variable to true achieves the same result as specifying the F parameter on the LISP control command (see section 2.1). If //FATAL is set to false, normal error recovery procedures will be in effect.

//SYSIN - system input file control

See function SYSIN (section 5.1.1)

//SYSOUT - system output file control

See function SYSOUT (section 5.1.1)

//CODEMIN - minimum in-core code control
//CODEMAX - maximum in-core code control

See function DISKOUT (section 7.6)

The LISTING function and system variables described above actually effect control of UT LISP. That is, a call to LISTING or resetting some variable's value has an immediate effect on the operation of LISP. Below are described several system variables which can be used to gain information about UT LISP. Their values are useful, but changing them does not affect the operation of LISP.

| Variable | Value |
|---|---|
| //RDS | name of currently-selected read file |
| //WRS | name of currently-selected write file |
| //FL | current field length |
| //GFR | number of free-space words recovered in last garbage collection |
| //GFU | number of full words recovered in last garbage collection |
| //NFR | number of garbage collections which have occured due to free-space exhaustion |
| //NFU | number of garbage collections which have occurred due to full-word space exhaustion |
| //FRS | number of words in free space |
| //FUS | number of words in full-word space |
| //STS | number of words allocated to the stack |

# 5. INPUT/OUTPUT

LISP provides an extensive set of functions which give the user facilities for input and output of LISP S-expressions and general data on any disk or tape file within the operating system. With these functions the user can perform rather complicated file manipulation operations. Functions are described in this chapter using the same notation as in chapter 4.

## 5.1 FILES

All input/output in the operating system is done via files of information. LISP regards a file as a contiguous string of lines terminating with an end-of-file and having a position pointer. Although the operating system can structure files into logical records, LISP completely ignores this structure; thus ends-of-records are invisible to the LISP input/output system. Files are named and their names are represented within LISP by literal atoms whose print representations are character strings obeying the usual file-name syntax of the operating system; i.e., one to seven letters or digits, beginning with a letter. Within the operating system, certain file names are specially designated to have particular sources or destinations. Usage of these files within LISP conforms to these system designations. These files are:

| File Name | Normal Source/Destination |
|---|---|
| INPUT | Cards from card reader |
| OUTPUT | Line printer |
| PUNCH | 80-column Hollerith punched cards |
| PUNCHB | Column binary punched cards |

All data read or written by LISP input/output functions is in the form of display coded line images obeying the usual system conventions. The input routines assume line images of 72 columns or fewer. If a line being read is longer than 72 columns, all information beyond column 72 is ignored by the LISP system. Under program control, the line length allowed for output lines is changeable.

The operating system defines an attempt to read a file immediately after writing on that file to be an error. LISP keeps a record of the last type of operation performed on a file and always performs a rewind operation any time a read is requested when the last operation was a write. Thus the user can write information on a file and immediately read that file from its beginning.

### 5.1.1 Standard System Input/Output Files

At any time, several files may be known and usable by LISP. Two of these files are designated as the standard system input/output files. The standard input file is the one from which the LISP main interpreter loop reads the program expressions or doublets. The standard output file is the one on which the LISP main interpreter loop writes the results of expression or doublet evaluations and also all error messages.

These standard input and output files are known by the pseudonyms SYSIN and SYSOUT, respectively. SYSIN and SYSOUT are each equivalenced to some other file which is the file actually read or written. This equivalencing is under program control. No matter what files are being used for the standard system input/output, they may be referenced by the names SYSIN and SYSOUT.

If not otherwise specified, SYSIN is initially equivalenced to INPUT and SYSOUT is initially equivalenced to OUTPUT. The input file control and output file control parameters on the LISP control command can be used to initialize SYSIN and SYSOUT to other file names. The conversational mode LISP control command parameter, C, equivalences both SYSIN and SYSOUT to file TTY.

SYSIN and SYSOUT can be equivalenced to other files by means of the SYSIN and SYSOUT functions described below.

(SYSIN <filename> <character>)

pseudofunction; SUBR

SYSIN makes <filename> the new standard input file and enters the main loop of LISP. The next expression will be read from <filename>. The SYSIN function makes a stack of filenames given in successive calls to SYSIN, so that when the end of the new SYSIN file is read, the stack is popped and the SYSIN file reverts to the last previous value. Thus the SYSIN function may be used to initiate the input of a chain of files, with the SYSIN file eventually returning to its first definition. (Note: Such a chain must not require opening more than six files altogether. See section 5.2.) The second argument of SYSIN provides listing control for the new SYSIN file. If the <character> is L, P, or N, the new SYSIN file will be listed or not as though the L, P, or N parameters appeared on the LISP control command (see section 2.1). If the <character> is S, the listing controls in effect at the time SYSIN is called will be in effect for the new SYSIN file. If the <character> is anything else, all listing control flags are cleared. SYSIN binds the atom <filename> to the LISP variable //SYSIN. SYSIN returns *T* as its value.

(SYSOUT <filename>)

pseudofunction; SUBR

SYSOUT makes <filename> the new standard output file. It takes effect immediately, and the value of SYSOUT will be written on <filename>. The function SYSOUT returns as its value the name of the old SYSOUT-equivalenced file. SYSOUT binds the atom <filename> to the LISP variable //SYSOUT.

### 5.1.2 Selected Read and Write Files

At all times, one file is designated the selected read file and one the selected write file. In general, those LISP functions which read information read from the current selected read file, and those functions which write information write on the selected write file.

Initially, the selected read and write files are the same as SYSIN and SYSOUT, respectively. Execution of functions SYSIN and/or SYSOUT has no effect on the selected read and write files, however. These files may be changed by functions RDS and WRS as described below.

(RDS <filename>)

pseudofunction; SUBR

RDS makes <filename> the new selected read file. It returns the previous selected read file name as its value. RDS binds the atom <filename> to the LISP variable //RDS.

(WRS <filename>)

pseudofunction; SUBR

WRS makes <filename> the new selected write file. It returns the previous selected write file as its value. WRS binds the atom <filename> to the LISP variable //WRS.

### 5.1.3 User Access to Selected Files

The user may at any time determine which files are currently selected by evaluating the system variables //SYSIN, //SYSOUT, //RDS, //WRS. These variables are always bound to the filenames currently selected (also see section 4.14).

| Variable | Default | Value |
|---|---|---|
| //SYSIN | SYSIN | filename currently selected as SYSIN |
| //SYSOUT | SYSOUT | filename currently selected as SYSOUT |
| //RDS | SYSIN | currently selected read file |
| //WRS | SYSOUT | currently selected write file |

## 5.2 FILE AND BUFFER ASSOCIATIONS

There must be a buffer area associated with each file currently in use by LISP. The standard LISP system reserves six file buffer areas to accommodate up to six active files at one time. It is not possible for the user to access more than six files at any given time. Under program control, however, the file buffers may be detached from files and attached to other files so that a total of more than six files may in fact be used during the course of a run.

The act of associating a buffer with a file and defining certain characteristics of the file is called "opening" the file. Unless stated otherwise, all functions described in other sections of this chapter "open" the files they manipulate using default values for the file characteristics. If the user wishes to explicitly "open" a file or define its characteristics, the function OPEN is used.

(OPEN <filename> <list>)

where <list> has as its value an expression of the form

(((<c1> . <v1>) (<c2> . <v2>) ... (<c[n]> . <v[n]>))

pseudofunction; SUBR

OPEN finds an available buffer and associates it with <filename>. OPEN returns <filename> as its value. The second argument of OPEN is a list of file characteristics to be set. They are chosen from among the following:

| | | |
|---|---|---|
| ECHO | LENGTH | SCR |
| ECHOP | MARGIN | RANDOM |

Descriptions of these file characteristics are given below:

ECHO

Value: NIL or non-NIL

Default value: NIL

If value is non-NIL, then each line of the file is echoed on SYSOUT as the file is read. NIL turns off the echo-printing.

ECHOP

Value: NIL or non-NIL

Default value: NIL

If value is non-NIL, each line of the file is echoed on SYSOUT as the file is read, and a parenthesis count is also printed for each line. NIL turns off this option.

LENGTH

Value: <fixnumber>

Default value: 70 (interactive) or 132 (batch)

The maximum output line length for this file is <fixnumber> characters. <fixnumber> should be less than 136.

MARGIN

Value: <fixnumber>

Default value: 0 (interactive) or 1 (batch)

Each line output to this file will have <fixnumber> blank characters appended to the left. Since the blanks occupy the leftmost <fixnumber> positions of the line, only LENGTH - MARGIN characters can be output on a line.

SCR

Value: NIL or non-NIL

Default value: NIL

Defines <filename> as scratch mode if the value is non-NIL. Scratch-mode files are never written to disk so long as the information written on them does not exceed the capacity of the buffer. Scratch mode is cancelled if the buffer overflows or if the function ENDFILE is executed. For short files which do not need to be preserved at the end of a run, scratch mode significantly speeds access.

RANDOM

Value: NIL or non-NIL

Default value: NIL

If the value is non-NIL, the file is defined to be accessible by the random-access I/O functions described in section 5.6. //RDS, //WRS, //SSYSIN, and //SYSOUT may not be RANDOM.

Any characteristics not otherwise specified when the file is first opened (either by calling OPEN or by calling some I/O function) are set to the default value. Once a characteristic is defined, it can be changed only by explicitly specifying the change in a subsequent call to OPEN. Any number of calls may be made to OPEN to change a file's characteristics dynamically.

For example:

```
(OPEN "ATOM "((SCR . T)))
```

makes the file ATOM a scratch-mode file. Later execution of

```
(OPEN "ATOM)
```

does not cancel the scratch mode. To cancel scratch mode, it is necessary to execute

```
(OPEN "ATOM "((SCR . NIL)))
```

(CLOSE <filename>)

pseudofunction; SUBR

CLOSE dissociates the specified file from its buffer. <filename> is returned by CLOSE as its value. After the buffer has been dissociated from its file, it is available for re-use for some other file. If the file being "closed" was last used for output purposes, an end-of-file mark is written on the file. Any "closed" file still exists on the disk and may be "opened" again later. //SYSIN and //SYSOUT may not be closed.

The user may determine what files are "open" at any time by executing function OPENFILES.

(OPENFILES)

pseudofunction; SUBR

OPENFILES returns a list with one element for each file presently "open" to LISP. Each element is itself a list of the form:

(<filename> ((<c1> . <v1>)(<c2> . <v2>) ... (c[n] . v[n])))

where each of the (<c[i]> . <v[i]>) pairs indicates the current value of one of the characteristics of a <filename>.

## 5.3 OUTPUT OF S-EXPRESSIONS

The functions PRINT, WRITE, PRIN1, and TERPRI are used to output LISP S-expressions on the current selected write file. Function OUTPUT can be used to output an S-expression to an arbitrary file. The S-expression is written on as many lines as are required, governed by the margin and line length of the file being written. Formatting of numeric atoms is completely automatic, but may be controlled to some extent by use of the function NFORMAT (see section 5.8). If the output functions are given something to write which is not an S-expression, they print in its place:

#<nnnnnn>

where <nnnnnn> is the octal value of the actual pointer received by the output function.

(PRINT <s> <boolean>)
(WRITE <s> <boolean>)

pseudofunction; SUBR

The value of PRINT is <s>. PRINT writes the S-expression <s> on the current selected write file. Writing commences on the current line of that file and extends over as many lines as necessary. PRINT always terminates the line so that the next S-expression written on the file will begin on a new line. If <boolean> is false, the S-expression is printed using the print names of its component atoms. If <boolean> is true, any atoms with nonstandard print names are written with the appropriate delimiters inserted so that the atoms may subsequently be read by the LISP input functions. WRITE is a synonym for PRINT.

(PRIN1 <s> <boolean>)

pseudofunction; SUBR

PRIN1 behaves exactly as PRINT, except that it does not terminate the line after writing the S-expression. Successive calls on PRIN1 cause several S-expressions to (possibly) be printed on the same line.

(PPRINT <s> <boolean>)

pseudofunction; SUBR

PPRINT "pretty-prints" the S-expression <s> on the current selected write file. The value of PPRINT is <s>. If <boolean> is true, any atoms with nonstandard print names are written with the appropriate delimiters inserted so that the atoms may subsequently be read by the LISP input functions. PPRINT operates exactly as PRINT, except that the output is neatly formatted for easy readability by the use of appropriate indenting for nested sub-expressions and special conventions for the printing of LAMBDA, COND and PROG expressions. PPRINT cannot print non-S-expressions.

Example:

(PPRINT //OUTPUTA T) would print:

```
(LAMBDA (=====/////)
 (OR
  (AND (NOT (ATOM =====/////))
       (EQ (CAR =====/////) "DEFINE))
  (OUTPUT //SYSOUT =====/////)))
```

(TERPRI)

pseudofunction; SUBR

The value of TERPRI is NIL. TERPRI terminates the current line on the current selected write file, so that the next S-expression written on the file will begin a new line. A call on PRIN1 followed by a call on TERPRI is equivalent to a call on PRINT. If nothing has been written on the current line, the effect of TERPRI is to skip a blank line.

(OUTPUT <filename> <s> <boolean>)

pseudofunction; SUBR

The value of OUTPUT is <s>. OUTPUT writes <s> on file <filename>. With respect to that file, the action of OUTPUT is the same as PRINT. OUTPUT does not change the selected write file. The definition of OUTPUT is equivalent to:

```
(OUTPUT <filename> <s> <boolean>) =
(PROG(A) (SETQ A (WRS <filename>))
  (PRINT <s> <boolean>)(WRS A)(RETURN <s>))
```

(OUTPUT1 <filename> <s> <boolean>)

pseudofunction; SUBR

OUTPUT1 behaves exactly as OUTPUT, except that it does not terminate the line after writing the S-expression. Successive calls on OUTPUT1 cause several S-expressions to (possibly) be printed on the same line.

(TTYCOPY <filename>)

pseudofunction; SUBR

TTYCOPY allows the interactive CRT user to have a copy of all subsequent terminal interaction written to <filename>. This is especially useful if one has a conversational program and desires to selectively save samples of its output for later use. A call to TTYCOPY with any legal filename as an argument turns on the dual output mode such that all terminal interaction is also printed on <filename>. TTYCOPY may be turned off by calling (TTYCOPY NIL). Turning off the dual output mode also performs a CLOSE of <filename> (see section 5.2).

## 5.4 INPUT OF S-EXPRESSIONS

The functions READ and INPUT are used to read LISP S-expressions from files. Each call of one of these functions reads the next available complete S-expression on that file, consuming as many lines of the file as necessary. When the complete S-expression has been read, the file is left positioned so that the next READ request will continue scanning the line on which the previous S-expression ended. Thus there may be more than one S-expression on a line. No more than 72 columns of each line are scanned.

(READ)

pseudofunction; SUBR

READ reads the next S-expression from the current selected read file and returns that S-expression as its value. If there is no next S-expression on the file, the atom $EOF$ is returned as the value of READ. If the end-of-file is detected while READing a list, an "unmatched left parenthesis" error occurs.

(INPUT <filename>)

pseudofunction; SUBR

INPUT operates in the same manner as READ except it reads from file <filename>. INPUT does not change the selected read file. The definition of INPUT is equivalent to:

```
(INPUT <filename>) =
(PROG (A X) (SETQ A RDS <filename>))
      (SETQ X (READ)) (RDS A) (RETURN X))
```

## 5.5 INPUT OF NON-S-EXPRESSIONS

It is often necessary to read information which is not in the form of S-expressions. LISP provides very primitive facilities for reading such data one character at a time. It is up to the user to employ the various character-manipulating functions described in section 4.10 to reform the data into meaningful objects. A character which is read by these functions is returned as the literal atom which has that single character as its print image. Users should carefully note that digits (0, 1, 2, ..., 9) read as characters are returned as literal atoms

(equivalent to $$$0$, $$$1$,..., $$$9, respectively) and are consequently not equivalent to the single-digit numbers. The function NUMOB must be used if it is desired to utilize the numeric values of numbers read as characters.

Users should also note that line images as stored on files by the operating system are variable in length. The operating system truncates all lines by removing trailing blanks, leaving an even number of characters followed by an end-of-line marker. LISP considers the end-of-line to occur at column 73 or when the end-of-line marker is encountered, whichever occurs first. Therefore users should not use column counts to determine when lines are terminated.

(READCH <boolean>)
(ADVANCE <boolean>)

pseudofunction; SUBR

READCH (ADVANCE is a synonym) returns the next character from the current selected read file if its argument is NIL; with each such call the input position is advanced one column. If the argument of READCH is non-NIL, the input position is backed up two columns before reading, to allow the previous character to be read again. This backing-up function of READCH cannot go beyond the beginning of the current line and any attempt to do so causes the atom $EOR$ to be returned as the value of READCH. $EOR$ is also returned by READCH when it detects the end-of-line. If the end-of-line is reached, the next call on READCH automatically sequences the input position to the beginning of the next line and the first character on that line is returned by READCH. The atom $EOF$ is returned by READCH if it advances the input position beyond the last line of the file.

Example:

Given the input line,

ABCD

then, in sequence,

```
(READCH NIL) = A
(READCH NIL) = B
(READCH NIL) = C
(READCH T)   = B
(READCH T)   = A
(READCH NIL) = B
(READCH NIL) = C
(READCH NIL) = D
```

(STARTREAD)

pseudofunction; SUBR

STARTREAD moves the input position to the beginning of the next line of the current selected read file whether or not the current line has been entirely scanned. The value of STARTREAD is the first character on the new line. Subsequent calls on READCH will continue scanning the new line. Contrary to some LISP implementations, UT LISP does

not require STARTREAD to be called before the first call on READCH. STARTREAD returns the atom $EOF$ if there is no next line on the file.

(TEREAD)
(ENDREAD)

pseudofunction; SUBR

TEREAD (ENDREAD is a synonym) immediately advances the input position to the end-of-line of the current line of the current selected read file. The value of TEREAD is always the atom $EOR$. After a call on TEREAD, the next call on READCH will read the first character on the next line.

## 5.6 RANDOM ACCESS OF DISK FILES

Ordinarily, LISP I/O is done in a purely sequential manner. It is sometimes useful, however, to be able to reference information stored at known positions on a disk file. UT LISP provides two functions which allow S-expressions to be referenced in random sequence from a disk file.

Any file which is going to be used for random access must be "opened" with the random option specified. Any file so designated may also be read sequentially using the ordinary LISP input functions. Each such read operation commences at the current position of the file.

Random access operations require the use of a "disk address" to specify the desired position within the file. The LISP random access functions use a word address, and regard the file as a linear sequence of character-filled words. A file written sequentially may be referenced randomly by specifying a word address.

The addresses employed by these functions are not LISP objects themselves. A suggested way to handle them is to assign an atomic key to each S-expression to be randomly written. The address returned by RANOUT can then be saved on the property list of the key, for subsequent use in a RANIN call. For maximum efficiency on any given file all RANOUT operations should be performed before any RANIN operations.

(RANOUT <filename> <s> <boolean>)

pseudofunction; SUBR

RANOUT positions <filename> to its extreme endpoint and then if <boolean> is true, no further action occurs. If <boolean> is false, RANOUT then performs an (OUTPUT <filename> <s> <boolean>) operation. The S-expression <s> is written on the file. The value returned by RANOUT is the address within the file after the positioning operation. Note that this address is not a LISP number and is not necessarily a pointer to a valid S-expression.

(RANIN <filename> <address> <boolean>)

pseudofunction; SUBR

The second argument of RANIN is a disk address of the same form as that returned by RANOUT (i.e., it is not a LISP number). The file <filename> is positioned to that address and then, if <boolean> is true, no further action occurs. If <boolean> is false, RANIN performs an (INPUT <filename>) operation. The value of RANIN is the S-expression thus read. A subsequent call on INPUT or READ for this file without an intervening RANIN or RANOUT will commence at the position in the file immediately following the S-expression read by RANIN.

(ROUT <filename> <s> <boolean>)

pseudofunction; SUBR

ROUT operates in the same way as RANOUT except that ROUT returns a LISP octal number for the address within the file of the beginning of the S-expression. The use of ROUT thus requires more storage (for numbers) than RANOUT, but may be more convenient for the user.

(RIN <filename> <fixnumber><boolean>)

pseudofunction; SUBR

RIN operates in the same way as RANIN except that RIN requires an octal number (as returned by ROUT) as its second argument.

## 5.7 INPUT CONTROL FUNCTIONS

UT LISP provides a limited amount of control over the input process. Functions are provided for skipping input columns or for tabbing to a particular column. Also, a function is provided for changing the lexical significance of individual characters so that a nonstandard syntax can be employed.

(ISPACE <number>)

pseudofunction; SUBR

ISPACE positions the input pointer of the current selected read file forward by <number> columns. ISPACE does not skip beyond the actual end or before the actual beginning of the current line. The value of ISPACE is the column number of the next column to be read after the skipping has been done. If the argument to ISPACE is less than or equal to 0 no skipping is done. Thus ISPACE(0) gives the present column position on the input line.

(ITAB <number>)

pseudofunction; SUBR

ITAB positions the input pointer of the current selected read file to the column specified by its argument. If the

present column number is greater than or equal to the argument of ITAB no positioning is done. The value of ITAB is the number of the next column to be read after the positioning has been done. ITAB does not move the input pointer beyond the actual end of the present line.

(CHLEX <character> <fixnumber>)

pseudofunction; SUBR

Each character in the system character set belongs to some lexical class (see section 3.1.1). The lexical class of a character determines how the character is interpreted by the input functions, as discussed in chapter 3. In certain very special cases it may be desirable to change the lexical class of a particular character. This is done by the function CHLEX. The first argument must be a single-character literal atom. The second argument must be a positive integer in the range 1-18, designating the number of the desired new lexical class of the character. The change takes effect immediately for the next input operation. Any character may have its lexical class changed any number of times. Unpredictable results will be obtained, however, if non-digit characters are assigned to the class of digits. The value of CHLEX is the number of the previous lexical class of the character.

Lexical Class Assignment Codes

| Lexical Class | Display Code | Members |
|---|---|---|
| 0 | 00 | end-of-line |
| 1 | | letters except E and Q |
| 2 | 33-44 | digits 0-9 |
| 3 | | any characters not in some other class |
| 4 | 05 | E |
| 5 | 21 | Q |
| 6 | 45 | + |
| 7 | 46 | - |
| 8 | 53 | $ |
| 9 | 71 | % (percent sign or down arrow) |
| 10 | 65 | # (pound symbol or right arrow) |
| 11 | 61 | [ |
| 12 | 62 | ] |
| 13 | 51 | ( |
| 14 | 52 | ) |
| 15 | 57 | . (period) |
| 16 | 55-56 | , (blank and comma) |
| 17 | 60 | " (equivalence sign or double-quote) |
| 18 | | special (no members - see section 3.1.2.3, item 2) |

## 5.8 OUTPUT CONTROL FUNCTIONS

UT LISP provides a limited amount of control over the output process in addition to that previously described. Functions are provided for intraline spacing and for formatting floating point numbers.

(OSPACE <fixnumber>)

pseudofunction; SUBR

OSPACE issues <fixnumber> blanks to the current line of the current selected write file, starting at the current column position. OSPACE does not space beyond the defined line length of the file. The value of OSPACE is the number of the output column in which the next output item will begin. Arguments of OSPACE which are less than or equal to 0 will not cause spacing to be performed. Thus, OSPACE(0) will give the current column number as its value.

(OTAB <fixnumber>)

pseudofunction; SUBR

OTAB positions the current selected write file so that the next item output will begin in column <fixnumber>. OTAB does not space beyond the defined line length of the file. Arguments of OTAB which are less than the current column position do not cause spacing to occur. The value of OTAB is the number of the output column in which the next item output will begin. Column numbering begins with 1 in the column immediately following any margin of blanks specified for the file. Thus, the tabbing functions independently of the margin.

(NFORMAT <fixnumber1> <fixnumber2>)

pseudofunction; SUBR

The first argument to NFORMAT specifies the number of digits which are to precede the decimal point when floating point numbers are subsequently output; the second argument specifies the number of digits to follow the decimal point. Both arguments must be integral and their sum cannot exceed 16. If their sum does exceed 16, the second argument is reduced if possible. The details of how number formatting is determined are given in section 3.1.3. Default values of 5 and 5 are assumed for the numbers prior to the first call to NFORMAT. The value of NFORMAT is NIL.

Example:

The number 23.123 is printed as

23.12300

under the standard format. After (NFORMAT 2 2) is executed, the number is printed as

23.12

## 5.9 FILE MANIPULATION

The remaining input/output functions of UT LISP are those for file manipulation. These provide for the usual cases.

(ABOLISH <filename>)

pseudofunction; SUBR

ABOLISH closes the file given as its argument and then returns the file to the operating system, releasing the disk space or tape unit assigned to the file. The file can no longer be referenced. The value of ABOLISH is its argument.

(ENDFILE <filename>)

pseudofunction; SUBR

ENDFILE writes an end-of-file on <filename>. If the file was OPENed as a scratch-mode file, ENDFILE cancels the scratch-mode and forces the file to be written on disk. If subsequent writes are made on a disk file, the end-of-file is changed to a system end-of-record. The value of ENDFILE is its argument.

(REWIND <filename>)

pseudofunction; SUBR

REWIND causes the named file to be repositioned at its beginning. If the file is already rewound, it is not disturbed. The value of REWIND is its argument.

## 5.10 BINARY I/O

Normally all input and output performed by LISP is done in CDC 6000 series display code. Occasionally a user may wish to read or write multiplexor (MUX) images consisting of five 12-bit characters packed per word, or to perform I/O using binary vectors. The function LEFTSHIFT provides the capabilty for manipulating such binary information within LISP. The following two functions provide capabilities for an interactive program to perform I/O operations with such binary information on the file TTY.

(INBIN)

pseudofunction; SUBR

A call to INBIN initiates the reading of binary information from the terminal. All information read by LISP in this mode is stored five characters per word, as 12-bit multiplexor images. INBIN terminates when the user inputs a carriage return or other line terminator. The total number of characters cannot exceed 48. The value of INBIN is a list containing the packed binary words which were read. These binary words can be printed properly at the terminal as characters only if the binary output function OUTBIN is used.

(OUTBIN <fwl>)

pseudofunction; SUBR

OUTBIN outputs its argument to the interactive terminal file TTY as 12-bit multiplexor images. ØUTBIN assumes the <fwl> is composed of binary information, packed in the form described under INBIN. The value of OUTBIN is its argument.

Example:

The following sequence initiates a read on the same line on which X is printed at the interactive terminal.

```
(PRIN1 X)(OUTBIN NIL)(READ)
```

# 6. THE LISP COMPILER/ASSEMBLER

The previous chapters have described the structure of UT LISP and the functions it provides. These facilities are most often employed by the user through interpretive execution of his functions. The UT LISP interpreter provides easy and effective use with good error control and diagnostic facilities. Interpretation runs slower, however, than the same functions would execute when coded directly in machine language. Also in the interpretive execution mode, the user cannot use the underlying machine in ways not defined by the built-in functions of the LISP system.

The LISP compiler is available to those users who wish to increase execution speed to as much as 2-7 times that of interpretive execution. The compiler translates LISP function definitions into an intermediate form which can be assembled into machine code by the LISP assembler.

The LISP assembler is normally used in conjunction with the LISP compiler, but it is also available for separate use. The LISP assembler enables users to write assembly language programs which make full utilization of the underlying machine.

## 6.1 ACCESS TO THE LISP COMPILER AND ASSEMBLER

The LISP compiler and assembler are very large programs and are relatively infrequently used. For these reasons, they are not provided as part of the standard LISP system. Instead, they are maintained as LISP subsystems (see appendix B) on the operating system library.

Two subsystems are required:

LAP - contains the LISP assembler program and necessary initialization code.

LCOMP - contains the LISP compiler and its initialization code.

These programs were themselves written in LISP and then self-compiled. They are loaded into binary program space in exactly the same way as user-compiled code.

To obtain the compiler, the following control command is recommended:

```
LISP,E,A=80,B=1800,LCOMP,LAP,X=10100B.
```

with a field length of at least 77000 octal. Other parameters of the defined parameter set may also be used if desired. The values of the A and B parameters are guidelines. The X parameter indicates the amount of compiled code loaded by LAP and LCOMP. Loading other compiled code may require increasing the field length of the job.

The assembler can be used by itself. To obtain only the assembler, the following control command is recommended:

```
LISP,E,A=80,B=1800,LAP,X=2600B.
```

with a field length of 74000 octal. The parameters shown on the

control command above are required, and others of the defined parameter set may also be used if desired.

The control commands given here define minimal configurations for these programs. As such, they may be used in either interactive or batch mode for the compilation/assembly of small functions (approximately 15 lines of LISP or fewer, depending on function complexity). The compilation/assembly process is a heavy user of memory, particularly full-word space, and frequent garbage collection is necessary. Users of the compiler/assembler should carefully observe the functioning of the system on their particular programs and adjust the field length and/or allocation control parameter to improve performance in their individual cases. Because of restrictions on the maximum field length of interactive jobs, not all functions can be compiled in interactive mode. Batch use of the compiler/assembler is therefore recommended for medium or large functions.

The control commands shown above initiate the process of loading the compiler and/or assembler. This loading process takes approximately 8 seconds of TM time before any processing of user input takes place. Two messages are printed during the loading process:

(**** LISP ASSEMBLER - VERSION x.x *****)

(**** LISP COMPILER - VERSION x.x *****)

These are printed as each subsystem starts loading, and x.x is the number of the current version. If these two messages fail to appear, assistance should be sought from a consultant.

## 6.2 LCOMP - THE LISP COMPILER

Once loaded, the LISP compiler is activated by executing the function COMPILE in the form:

(COMPILE <lat>)

The atoms given in the <lat> are names of functions which were defined before the activation of COMPILE. COMPILE expects these atoms to have lambda-expressions stored on their property lists with either the EXPR or the FEXPR indicator.

COMPILE transforms each lambda-expression into an internally represented assembly-language program, constructed from a set of macros representing the machine language of an idealized LISP machine. After this translation, the compiler automatically activates the assembler, which then expands the macros into machine language of the CDC 6000 machines. As will be described later, the user has the options of immediately loading the resulting machine code into memory, saving the machine code on a file, or simply discarding it (see section 6.3.6).

The result of compilation must be held entirely in memory. It may be a very large list structure, so the compiler must be provided with at least enough memory to compile the largest function in the list given to it.

The compiler itself detects no errors. Any function presented to it will be compiled. The compiler does automatically call the assembler, however, and compiled code for an illegal function almost always produces errors detected by the assembler. The errors are described in section 6.3.7. The value returned by the

function COMPILE is the list of function names given to it as its argument. If any of the names in the list was not defined as a function, its entry in the list is replaced by the message:

(<name> IS NEITHER FEXPR NOR EXPR)

See section 6.2.2 below for further discussion of the compiler's operation.

### 6.2.1 Output of the Compiler

In addition to the value it returns and the compiled code it produces, the compiler also produces some printed output for each function it compiles. The compiler always produces the following:

| | |
|---|---|
| (<name> <number>) | <name> is the function name. <number> is the number of words of free space occupied by the lambda-expression. |
| (COMPILE-TIME = <t1>) | <t1> = time (milliseconds) in compiler. |
| (PASS1-TIME = <t2>) | <t2> = time (milliseconds) in first pass of assembler |
| (LENGTH <number>) | <number> = length (in words) of compiled code |
| (LAP-TIME = <t3>) | <t3> = time (milliseconds) in assembler |
| (ORIGIN <number>) | <number> = address of start of code in memory |
| (TOTAL-TIME = <t4>) | <t4> = time (milliseconds) in compiler and assembler |

Each of these items is on a separate line. The first two and the last are produced by the compiler. The other four are actually printed by the assembler.

Optionally, a listing of the compiled code may be obtained, controlled by the variable PRNTFLAG. If PRNTFLAG has a non-NIL value, a neat listing of the compiled code is produced. Fig. 6.1 shows an example. If PRNTFLAG has the value NIL, no listing is produced. PRNTFLAG is initialized to NIL during the loading of the compiler.

### 6.2.2 Theory of Operation of the Compiler

The UT LISP compiler has been designed so that there are no restrictions on its use. Any function that executes properly interpretively and which on any function call includes all arguments whether required or not will compile and execute properly. There are no restrictions on variable references or on interaction between compiled and interpreted functions.

As the compiler processes a function, it generates a list of macro calls in the form of a program to be assembled by LAP. Each of the macros represents a specific operation of a "LISP machine", and is predefined when the compiler is loaded into the

Figure 6.1

Output of LCOMP Based On
INTERSECTION Function of Figure 2.1

```
(***** LISP ASSEMBLER - VERSION 1.5 *****)
(***** LISP COMPILER - VERSION 1.5 *****)

*EVAL:
(SETQ PRNTFLAG T)

*VALUE: *T*

*EVAL:
(COMPILE (QUOTE (INTERSECTION)))

(INTERSECTION 52Q)

        (MAIN INTERSECTION SUBR)
        (SAVE76 INTERSECTION 1)
        (BINDVARS (X Y) 2)
        (VALUE 1 X)
        (NULL 1 1)
        (OR 1 R00001)
        (VALUE 1 Y)
        (NULL 1 1)
R00001  (COND 1 C00001)
        (SETAK 1 NIL)
        (JUMP E00001)
C00001  (VALUE 1 X)
        (CAR 1 1)
        (VALUE 2 Y)
        (CALL MEMBER 2)
        (COND 1 C00002)
        (VALUE 1 X)
        (CAR 1 1)
        (SAVE 1 3)
        (VALUE 1 X)
        (CDR 1 1)
        (VALUE 2 Y)
        (CALL INTERSECTION 2)
        (SETA 2 1)
        (UNSAVE 1 3)
        (CONS 1 1 2)
        (JUMP E00001)
C00002  (VALUE 1 X)
        (CDR 1 1)
        (VALUE 2 Y)
        (CALL INTERSECTION 2)
E00001  (UNBINDVARS 1)
        (UNSAVE76)
        (JUMPB6)

(COMPILE-TIME =516)
(PASS1-TIME =138)
(LENGTH 43Q)
(LAP-TIME =240)
(ORIGIN . 74073Q)
(TOTAL-TIME =756)

*VALUE:
(INTERSECTION)
```

system. Some examples of these macro calls are shown in figure 6.1. The meanings of most of them are given in table 6.1 below.

The compiler is initially given a lambda-expression to process. For the lambda-expression the compiler first generates the LAP header element, and then code for binding arguments to variables (if any) and for reserving space on the stack. Then code is generated to evaluate the expression forming the third element of the lambda-expression. Following this code the compiler generates code to unbind variables (if any) and to return to the calling function.

The bulk of the compilation process is involved in generating code to evaluate expressions. If an expression is a number, code is generated to place a pointer to the number in a register. If an expression is a variable (<litatom>), code is generated to place the value of the variable in a register. If an expressin is of the form

(<functionname> <arg1> <arg2> ... <arg[n]>)

where each <arg> may itself be an expression, the compiler recursively compiles each <arg> in left-to-right order and then generates code to apply <functionname> to those values.

The most commonly used LISP functions whose machine-code realizations are short are known to the compiler and are compiled in line in the resulting program. These functions are represented in the code by macros whose names are the same as the functions they realize (e.g., CAR, CDR, etc.). Each such function is identified by having the property CMACRO on its property list. The value associated with this property is a function which performs the necessary code generation. Table 6.2 gives a list of the functions known in the standard compiler.

All other functions referenced in the expression cause a CALL macro to be generated. This macro creates a calling sequence to the named function, which is assumed to be defined elsewhere. The execution of an actual function call is considerably slower than execution of an in-line function.

Whenever the compiler encounters an expression whose <functionname> is FUNCTION or FQUOTE, it assumes that the expression contains a lambda-expression which is a sub-function of the one being compiled. This sub-function is compiled separately given an internally-generated name, and its code is appended to the end of the code is generated for the entire function. Such sub-functions can be identified by the occurrence of an initial ENTRY macro call.

This brief and simplified description of the operation of the compiler accounts for most of its activity. The description is offered here in the belief that some understanding of the compiler's operation will facilitate its use and the preparation of good programs. For further details the reader is referred to reference [1] at the end of this chapter.

### 6.2.3 Compiling Many Functions

When compiling many functions it is better to define just a few of them at a time, immediately compiling them after defining them. After the compilation, the function definitions should be deleted. For a given field length specification, this process reduces the storage required for function definitions and gives

Table 6.1

LAP Macros Defining the "LISP Machine"

| Macro name | Meaning |
|---|---|
| (AND <i> <loc>) | Controls sequencing of the AND function based on the value in register <i>. If register <i> contains false, control transfers to <loc>. |
| (BINDVARS <lat> <n>) | Controls binding of values to the <n> variables in the <lat>. |
| (CALL <name> <n>) | Sets up calling linkage to function <name> with <n> parameters. |
| (COND <i> <loc>) | Controls sequencing of the COND function based on value in register <i>. If register <i> is false control transfers to <loc>. |
| (FCALL <name> <list>) | Sets up calling linkage to an FSUBR or FEXPR <name>, with the <list> of arguments. |
| (FREE <i>) | Obtains a word from free space, leaving a pointer to it in A<i>. |
| (FUNARG <i> <name>) | Establishes function <name> as a functional argument in register <i>. |
| (GO <loc>) | Implements the GO function. |
| (JUMPB6) | Exits from a function. |
| (OR <i> <loc>) | Controls sequencing of the OR function depending on the value in register <i>. If register <i> contains something not false, control transfers to <loc>. |
| (PROGINIT <loc> <lat>) | Initializes a PROG expression where <lat> is the list of PROG variables and <loc> is the exit address. |
| (SAVE <i> <j>) | Saves register <i> in the <j>th position of the stack relative to its top. |
| (SAVE76 <name> <n>) | Saves the function <name> and returns information on the stack and reserves <n> words of temporary storage. |
| (SETA <i> <j>) | Sets register <i> equal to register <j>. |
| (SETAK <i> <constant>) | Sets register <i> to a <constant>. |

| | |
|---|---|
| (SETQ <name> <i>) | Sets the value of variable <name> to the content of register <i>. |
| (UNBINDVARS <i>) | Unbinds the most-recently bound list of variables. Register <i> is preserved during the operation. |
| (UNSAVE <i> <j>) | Retrieves the value from the <j>th position on the stack and puts it in register <i>. |
| (UNSAVE76) | Prepares stack for function exit. |
| (VALUE <i> <var>) | Retrieves the value bound to variable <var> and places it in register <i>. |
| (<fnname> <i> <j> <k> ...) | Implements the LISP system function <fnname>. Register <i> is always the result register and registers <j>, <k>, ..., are the arguments of the function. |

Table 6.2

Functions with CMACRO Properties

| | |
|---|---|
| ADVANCE | NOT |
| AND | NULL |
| ATOM | NUMBERP |
| CALLSYS | OR |
| CAR | OUTPUT |
| CDR | PRINT |
| CSR | PRIN1 |
| CONC | PROG |
| COND | PROGN |
| CONS | PROG2 |
| DEFSYS | QUOTE |
| EQ | READCH |
| ERRORSET | RETURN |
| FIXP | RPLACA |
| FLOATP | RPLACD |
| FQUOTE | RPLACS |
| FUNCTION | SET |
| GO | SETQ |
| LIST | WRITE |
| MINUSP | ZEROP |

more working space for the compiler. If this process is not used, a larger field length is required.

### 6.2.4 Compiling Large Functions

It is entirely possible that some large functions may not be compilable in any reasonably-sized field length. Such functions must be split into smaller independent pieces.

### 6.2.5 Compiling Functional Arguments

Functional arguments in expressions may be designated in three ways:

```
(QUOTE <function>)
(FQUOTE <function>)
(FUNCTION <function>)
```

The forms using QUOTE and FQUOTE behave identically except that when the compiler sees FQUOTE, the <function> expression following is compiled as a sub-function. A functional argument designated by QUOTE cannot reliably be distinguished from any other quoted expression so it is not compiled. The advantages of compilation are lost in this case. Whenever possible, FQUOTE should be used to designate functional arguments.

The form using FUNCTION also causes the <function> expression to be compiled as a sub-function. This form, however, generates a large amount of code for environmental preservation and hence is expensive in both space and time. FUNCTION serves a very special purpose and is needed in only a very few cases (see section 4.5).

### 6.2.6 Compiling References to FEXPR-FSUBR Functions

When the compiler generates a calling sequence to a function, the calling sequence is of EXPR-SUBR type or of FEXPR-FSUBR type, depending on the type of the referenced function. If the referenced function is as yet undefined, an EXPR-SUBR calling sequence is assumed. Therefore, all user-defined FEXPR-FSUBR functions must have those property indicators at the time when references to them are compiled.

### 6.2.7 Tracing Compiled Functions

Compiled functions can be traced in the usual manner, provided the function TRACE is called after the functions are loaded but before they are used. Once compiled functions are executed, internal linkages have been constructed, and tracing cannot be initiated.

Since SETQs are compiled in line, the function TRACESET has no effect on compiled functions.

### 6.2.8 Avoiding Name Conflicts

The compiler is itself a LISP program containing a number of LISP functions. If the user defines a function whose name is the same as one of the compiler functions (see table 6.3), then the user's function may well be used instead of the compiler function when the compilation is attempted. This situation usually causes

highly chaotic results. Compiler function names for the most part have a period (.) as their second character and are not too likely to cause conflicts.

Table 6.3

Compiler Function Names

| | | |
|---|---|---|
| COMPILE | IF | OPDEFINE |
| DATA | LAP | PUNCHMAC |
| ENTRY | MACRO | VFD |
| | | |
| A.DJOBJ1 | C.LIST | O.PTIMIZE |
| A.SER | C.COMPRESS* | P.ADCNT |
| A.SMAC2 | C.COMPRESS | P.AD |
| A.SMAC | C.ONCER | P.ASS1 |
| C.ANDOR | C.PRINT | P.C |
| C.ARGS* | C.SPEC | P.MAP |
| C.ARGS | I.NCR | R.EVERSIP |
| C.COND | L.ISTER | S.PACE2 |
| C.FARGS | M.ACPROC | S.VREG |
| C.FNCALL | M.ATCH2 | S.YSMAC |
| C.FNFORM | M.ATCH | T.IMER |
| C.FN | O.BJ1ADDP | T.RANS |
| C.FORM | O.BJ1ADD | T.ZCNT |
| C.LABEL | O.BJ1FIX | U.NSVRG |
| C.LAMBDA | O.PCODE | |

### 6.2.9 Redefining Standard Functions

Occasionally there is a need for a user to define his own versions of functions already defined in LISP. If the redefined function is one of those in the list of table 6.2, the compiler still generates code for the built-in definition, and does not generate a call to the user's new version. To circumvent this difficulty, it is merely necesary to remove the CMACRO property from those functions affected. The compiler then generates a call to the function instead of expanding it in line.

The CMACRO property may be removed from any of the functions in table 6.2 except:

COND FUNCTION GO PROG RETURN

If the CMACRO property is removed from these functions, the resulting compiled code will not be correct.

### 6.2.10 Using SMACRO for In-line Compilation

The in-line compilation of short functions such as CADR, CDDR, etc., may be forced by using SMACRO. If a function name has an SMACRO property whose value is a lambda-expression, the lambda-expression is compiled in line. Functions such as CADDR, etc., are not compiled in line unless the user generates appropriate SMACRO expressions for them.

## 6.3 LAP - THE LISP ASSEMBLER

The LISP assembler is a general two-pass assembler with macro and conditional assembly capabilities. It is most frequently called from within the LISP compiler, but may be called directly.

The assembler is called by:

(LAP <s1> <s2>)

where <s1> is the program to be assembled and <s2> is an initial symbol table. Program formats are described below in section 6.3.1. The initial symbol table is usually NIL, but in special cases may be a list of dotted pairs, each pair being a literal atom and a numeric value to be associated with that symbol.

The value returned by function LAP is the final symbol table of the assembly in the form of a list of dotted pairs.

Pass 1 of the assembler produces an intermediate form of the program with all operation codes and register numbers decoded. This intermediate form may be saved on a file for subsequent use and/or it may be further processed by pass 2 of the assembler and loaded into memory for execution. Pass 2 evaluates address expressions and finishes processing the function.

The intermediate form produced by pass 1 may be a very large expression, so sufficient memory must be made available, especially in full-word space. The user is cautioned to observe very carefully the use of memory during assembly of his programs and to adjust the LISP parameters accordingly.

### 6.3.1 Program Format

A program to be assembled by LAP is always in the form of a list. Machine instructions and assembler pseudo-operations are themselves sublists in the program. Atomic elements in the program list serve as labels in the program. If several atoms appear consecutively in the list, they all label the same instruction. Every program must begin with a MAIN pseudo-operation. An example assembly is shown in figure 6.2.

### 6.3.2 Symbols

Any literal atom may be used as a symbol in a program so long as it is not confused with some part of an operation.

The special symbol + when used as a label means to force the next instruction to be in the left-most bits of the next word to be assembled. If the previous word is not full, it is filled with pass instructions.

The special symbol * when used as an operand in an address expression stands for the address of the word containing the instruction being assembled.

Figure 6.2

Contents of LAPUNCH File Produced by Pass 1 of LAP
Based On INTERSECTION Function of Figure 6.1

```
(MAIN 0 ((INTERSECTION SUBR 0Q)) ((R00001 . 12Q) (C00001 . 14Q)
( C00002 . 34Q) (E00001 . 37Q)) 43Q ((611Q51 ((QUOTE
INTERSECTION))) (612Q51 (1)) (1Q54 (SAVE76ER)) + (611Q51 ((QUOTE
(X Y)))) (1Q54 (BINDIT )) (511Q51 ((QUOTE X))) 5311Q48 + (6021Q48
(- (QUOTE NIL))) (511Q51 (( QUOTE NIL))) (52Q51 ((+ * 1)))
(5011Q48 (+ 1)) (6011Q48 (- (QUOTE NIL) )) (51Q51 (R00001))
(511Q51 ((QUOTE Y))) 5311Q48 + (6021Q48 (- (QUOTE NIL))) (511Q51
((QUOTE NIL))) (52Q51 ((+ * 1))) (5011Q48 (+ 1)) (6011Q48 (-
(QUOTE NIL))) (41Q51 (C00001)) (511Q51 ((QUOTE NIL))) (4Q54
(E00001)) (511Q51 ((QUOTE X))) 5311021122Q30 5311Q48 (512Q51
((QUOTE Y))) 5322Q48 (VFD 12 (1Q6) 18 (LINKIT) 1 (0) 11 (2) 18
((QUOTE MEMBER) )) (6011Q48 (- (QUOTE NIL))) (41Q51 (C00002))
(511Q51 ((QUOTE X)))5311021122Q30 531107461Q33 (5167Q48 (+ (+
PDSTACK 3))) (511Q51 ((QUOTE X))) 53110531 1Q33 (512Q51 ((QUOTE
Y))) 5322Q48 + (VFD 12 (1Q6) 18 (LINKIT) 1 (0) 11 (2) 18 ((QUOTE
INTERSECTION))) 5421Q48 (5117Q48 (+ (+ PDSTACK 3))) 5311Q48
7461074720206221266 7Q 541Q51 (6211Q48 (- (QUOTE NIL))) + (7211Q48
(- (QUOTE NIL))) + (311Q48 ((+ * 1))) (1Q54 ))
5410054600530101016Q3 (4Q54 (E00001)) + (511Q51 ((QUOTE X)))
(GARBAGE 531105311Q33 (512Q51 ((QUOTE Y))) 5322Q48 + (VFD 12
(1Q6) 18 (LINKIT) 1 (0) 11 (2) 18 ((QUOTE INTERSECTION))) 6411Q48
+ (615Q51 ((+ * 1))) (4Q54 ((APPLLG)) 5611Q48 (5157Q48 (+
PDSTACK)) 6365Q48 215226375Q33 (26Q51 (0))))
```

### 6.3.3 Address Expressions

Some machine instructions require the specification of an 18-bit quantity in the address part. LAP allows these quantities to be specified in the form of expressions. A LAP address expression may have any of four forms:

a) a number
b) a symbol (literal atom)
c) a QUOTEd S-expression (e.g. "(A B C))
d) (<operator> <expression1> <expression2>)

If the expression is a number, the value placed in the instruction is the value of the right-most 18 bits of that number. The number may be positive or negative.

If the expression is a symbol, the value is determined by:

1) The value associated with the symbol on the LAP symbol table, if any. Otherwise

2) The value associated with one of the indicators SYM, SUBR, FSUBR on the property list of the symbol. If the symbol has more than one of these properties, the one most recently put on the atom is used.

The values associated with SUBR and FSUBR properties are the addresses of the entry points of the named machine-coded subroutine. Linkage to such subroutines can thus be effected merely by naming them in the address field of an appropriate jump instruction. Values associated with a SYM property are generally addresses of important locations within the LISP system which need to be accessed by the assembly-language routines. Table 6.4 gives a list of symbols with SYM properties defined in the

standard LISP system. The symbol * is treated specially in the evaluation of an expression and always evaluates to the current location within the program.

Table 6.4

Symbols with SYM Property in Standard LISP

| Symbol | Use/meaning |
|---|---|
| APPLLC | Unbinds a list of variables previously bound and saved on the stack. Can be used during exit from a function. |
| BINDIT | Binds a list of variables to the argument values sent by the calling function. The list of variables is saved on the stack with indicator VMARK. |
| BPROG | Location of cell containing the address of the next available word in binary program space. |
| CIO | Entry point of a routine for calling CIO for input/output. Enter via RJ, with FET location in B2 and CIO code in X0. |
| FGARBAG | Entry point to the garbage collector used when full space is used up. |
| FULLLIS | Location of the pointer to the next available full-space word. |
| GARBAGE | Entry point to the garbage collector used when free space is used up. |
| LINKIT | Entry point to a dynamic linkage routine which will link a machine-coded routine to another regardless of whether the called routine is machine-coded or interpreted. |
| PDERR | Location of error routine for stack overflow. |
| PDSTACK | Location of the bottom of the stack. |
| PROGINIT | Makes a standard entry to a PROG. |
| SAVE76ER | Entry point to a routine which constructs a stack header element and stores it on the stack, optionally reserving extra space on the stack. |
| SYSTEM | Entry point to a routine for making system requests. Enter via RJ, with formatted request in X6. |

If the expression is a QUOTEd S-expression, the value used in the instruction is the actual address of the S-expression, and the S-expression is not evaluated further. For instance, the expression CAR evaluates to the address of the machine subroutine for the CAR function, but the expression "CAR evaluates to the address of the atom CAR itself.

If the expression is of the form (<operator> <expression1> <expression2>), the values of <expression1> and <expression2> are combined according to the specified operator. The permissible operators are + and - . The expression is evaluated as a full-word value and then finally truncated to 18 bits.

The entire expression may be optionally preceded by a + or - sign.

### 6.3.4 Instructions Recognized by the Assembler

A complete set of CDC 6000 series central processor instructions is recognized by LAP, with mnemonic codes of the instructions being the same as those recognized by the COMPASS assembler [2].

Each instruction is represented by a list whose first element is the mnemonic operation code. Remaining elements are register designators, register numbers, and operator symbols. Register numbers are always enclosed in parentheses, and operator symbols must be separated from adjacent atomic symbols by at least one blank. Table 6.5 gives a list of the instructions and shows the forms that they may take.

It is assumed that any user of the LISP assembler is already familiar with the machine operations, and no further discussion of them will be given here.

### 6.3.5 Pseudo Instructions of the Assembler

The LISP assembler, like most assemblers, employs pseudo instructions to supply it with certain information about the program and to handle storage allocation. There are six pseudo instructions recognized by LAP. They are described below by giving the form of the pseudo instruction followed by a description of its function.

(MAIN <name> <type>)

The MAIN pseudo instruction must appear at the beginning of each program to be assembled. <name> is a literal atom which is the name by which this program will be known. <name> is also treated as though it were a label on the first instruction of the program. <type> specifies the type of property the code is to be given. <type> may be one of: SUBR, FSUBR, or SYM. If it is SUBR or FSUBR, then the program is a LISP-callable function. If type is SYM, then it is not LISP-callable but may contain code or data which may be referenced by other hand-coded programs. The address of the first word of the program is placed on the property list of <name> with indicator <type>.

Table 6.5

The Instruction Set Recognized by the LISP Assembler

In the following:

| | |
|---|---|
| i,j,k | represent register numbers in the range 0-7 |
| jk | represents a constant in the range 0-63 |
| <k> | represents an address expression |

In all cases where an alternate form has B(j)or B(k) omitted, B(0) is assumed.

| Operation Code | Standard Form(s) | Alternate Form(s) |
|---|---|---|
| 01 | (RJ <k>) | |
| 02 | (JP B(i) <k>) | (JP <k>) |
| 030 | (ZR X(j) <k>) | |
| 031 | (NZ X(j) <k>) | |
| 032 | (PL X(j) <k>) | |
| 033 | (NG X(j) <k>) | |
| 034 | (IR X(j) <k>) | |
| 035 | (OR X(j) <k>) | |
| 036 | (DF X(j) <k>) | |
| 037 | (ID X(j) <k>) | |
| 04 | (EQ B(i) B(j) <k>) | (EQ <k>)<br>(ZR B(i) <k>)<br>(ZR <k>) |
| 05 | (NE B(i) B(j) <k>) | (NZ B(i) <k>) |
| 06 | (GE B(i) B(j) <k>) | (GE <k>)<br>(LE B(j) B(i) <k>)<br>(LE B(j) <k>) |
| 07 | (LT B(i) B(j) <k>) | (LT B(i) <k>)<br>(GT B(j) B(i) <k>)<br>(GT B(j) <k>) |
| 10 | (BX(i) X(j)) | |
| 11 | (BX(i) X(j) * X(k)) | |
| 12 | (BX(i) X(j) + X(k)) | |
| 13 | (BX(i) X(j) - X(k)) | |
| 14 | (BX(i) - X(k)) | |
| 15 | (BX(i) - X(k) * X(j)) | |
| 16 | (BX(i) - X(k) + X(j)) | |
| 17 | (BX(i) - X(k) - X(j)) | |
| 20 | (LX(i) (jk)) | |
| 21 | (AX(i) (jk)) | |
| 22 | (LX(i) B(j) X(k)) | (LX(i) X(k) B(j)) |
| 23 | (AX(i) B(j) X(k)) | (AX(i) X(k) B(j)) |
| 24 | (NX(i) B(j) X(k)) | (NX(i) X(k)) |
| 25 | (ZX(i) B(j) X(k)) | (ZX(i) X(k)) |
| 26 | (UX(i) B(j) X(k)) | (UX(i) X(k)) |
| 27 | (PX(i) B(j) X(k)) | (PX(i) X(k)) |
| 30 | (FX(i) X(j) + X(k)) | |
| 31 | (FX(i) X(j) - X(k)) | |
| 32 | (DX(i) X(j) + X(k)) | |
| 33 | (DX(i) X(j) - X(k)) | |

Table 6.5 (contd.)

| Operation Code | Standard Form(s) | Alternate Form(s) |
|---|---|---|
| 34 | (RX(i) X(j) + X(k)) | |
| 35 | (RX(i) X(j) - X(k)) | |
| 36 | (IX(i) X(j) + X(k)) | |
| 37 | (IX(i) X(j) - X(k)) | |
| 40 | (FX(i) X(j) * X(k)) | |
| 41 | (RX(i) X(j) * X(k)) | |
| 42 | (DX(i) X(j) * X(k)) | |
| 43 | (MX(i) (jk)) | |
| 44 | (FX(i) X(j) / X(k)) | |
| 45 | (RX(i) X(j) / X(k)) | |
| 46 | (NO) | |
| 47 | (CX(i) X(k)) | |
| 50 | (SA(i) A(j) <k>) | |
| 51 | (SA(i) B(j) <k>) | (SA(i) <k>) |
| 52 | (SA(i) X(j) <k>) | |
| 53 | (S0A(i) X(j) + B(k)) | (S0A(i) X(j))<br>(SA(i) B(k) + X(j)) |
| 54 | (SA(i) A(j) + B(k)) | (SA(i) A(j))<br>(SA(i) B(k) + A(j)) |
| 55 | (SA(i) A(j) - B(k)) | (SA(i) - B(k) + A(j)) |
| 56 | (SA(i) B(j) + B(k)) | (SA(i) B(j)) |
| 57 | (SA(i) B(j) - B(k)) | (SA(i) - B(k))<br>(SA(i) - B(k) + B(j)) |
| 60 | (SB(i) A(j) <k>) | |
| 61 | (SB(i) B(j) <k>) | (SB(i) <k>) |
| 62 | (SB(i) X(j) <k>) | |
| 63 | (SB(i) X(j) + B(k)) | (SB(i) X(j))<br>(SA(i) B(k) + X(j)) |
| 64 | (SB(i) A(j) + B(k)) | (SB(i) A(j))<br>(SB(i) B(k) + A(j)) |
| 65 | (SB(i) A(j) - B(k)) | (SB(i) - B(k) + A(j)) |
| 66 | (SB(i) B(j) + B(k)) | (SB(i) B(j)) |
| 67 | (SB(i) B(j) - B(k)) | (SB(i) - B(k))<br>(SB(i) - B(k) + B(j)) |
| 70 | (SX(i) A(j) <k>) | |
| 71 | (SX(i) B(j) <k>) | (SX(i) <k>) |
| 72 | (SX(i) X(j) <k>) | |
| 73 | (SX(i) X(j) + B(k)) | (SX(i) X(j))<br>(SX(i) B(k) + X(j)) |
| 74 | (SX(i) A(j) + B(k)) | (SX(i) A(j))<br>(SB(i) B(k) + A(j)) |
| 75 | (SX(i) A(j) - B(k)) | (SX(i) - B(k) + A(j)) |
| 76 | (SX(i) B(j) + B(k)) | (SX(i) B(j)) |
| 77 | (SX(i) B(j) - B(k)) | (SX(i) - B(k))<br>(SX(i) - B(k) + B(j)) |

(ENTRY <name> <type>)

The ENTRY pseudo instruction is similar to MAIN. It appears, however, within a block of code instead of at the beginning. It defines a secondary entry point of type <type> and name <name>. The code following ENTRY may be referenced by <name> and may in turn reference instructions or data common to it and the program named in the MAIN pseudo instruction.

(MACRO <name> <lat> <inst1> <inst2> ... <inst[n]>)

Pseudo instruction MACRO defines a macro instruction for use in assembling the current program. <name> is the name of the macro; <lat> is a list of literal atoms which are the symbols to be replaced when the macro is expanded. <inst1> ... <inst[n]> are machine instructions, pseudo instructions, and/or macro instructions which define the body of the macro. Each substitutable parameter must be a distinct element of the S-expression defining the macro body.

A macro is used by placing the form:

(<name> <s1> <s2> ... <s[n]>)

anywhere an instruction might appear. <s1> ... <s[n]> are S-expressions which are substituted into a copy of the macro body for the substitutable parameters in one-to-one correspondence. After substitution, the macro body is inserted into the program instead of the macro instruction and then assembled.

A MACRO pseudo instruction may appear anywhere in the program so long as it precedes the first use of the macro. Macros defined by the MACRO pseudo instruction are purely local to the program being assembled and must be redefined for use in other programs.

Example:

When the macro definition:

```
(MACRO XX (I V))
(SA(I) (QUOTE V))
(SA(I) X(I)))
```

is referenced by:

```
(XX 3 ANATOM)
```

the macro definition is expanded to the code:

```
(SA(3) (QUOTE ANATOM))
(SA(3) X(3))
```

(IF (<p[1]> <inst[1,1]> ... <inst[1,m]>) ...
(<p[n]> <inst[n,1]> ... <inst[n,m]>))

The IF pseudo instruction provides a conditional assembly capability. Each <p[i]> is an expression, and the <inst[i,j]> are LAP instructions. The <p[i]> are evaluated using standard LISP evaluation rules until one is found to be true. The set of instructions following the first true <p[i]> is assembled at this point. If no <p[i]> is true, then no instructions are assembled.

IF is most often used inside macro definitions, and the <p[i]> generally are functions of the arguments substituted into the macro when it is expanded.

(DATA <expression>)

The DATA pseudo instruction defines a full-word data constant. <expression> is an expression of the form defined in section 6.3.3, and is evaluated according to the rules given there. It is left as a full-word value, however, and not truncated to 18 bits. Each DATA pseudo instruction allocates and defines one word of storage.

(VFD <n[1]> (<exp[1]>) <n[2]> (<exp[2]>) ... <n[m]> (<exp[m]>))

The VFD pseudo instruction defines the contents of one computer word, placing several values into specifiable fields within the word. Each <n[i]> is a simple integer which defines a field width. The <exp[i]> following each <n[i]> is an expression of the form described in section 6.3.3. Each <exp[i] is evaluated and truncated to <n[i]> bits. The resulting values are then concatenated in order and placed in the allocated word. The sum of all <n[i]> may not exceed 60. If the sum of the <n[i]> is less than 60, the packed values are stored in the high order bits of the word and the lower order bits are filled with zeros.

### 6.3.6 Operation and Control of the Assembler

The assembler performs a first pass over the program, and optionally a second pass. During the first pass, instructions are decoded and the operation codes and register numbers are reduced to numeric quantities. Also during the first pass, macros are expanded and the symbol table is constructed.

The second pass, if requested, evaluates address expressions and actually places the program code into memory.

The second pass is performed only if the variable LOADFLAG is not NIL. LOADFLAG is initialized to NIL when the assembler is loaded. If LOADFLAG is NIL, then assembly and error detection only through pass 1 takes place.

The intermediate code form produced by the action of the first pass may be saved on a file for subsequent processing by pass 2 (see description of READLAP, section 6.4). If variable LAPUNCH has a non-NIL value then that value is assumed to be the name of a file on which the intermediate code is to be written. If the value of LAPUNCH is NIL, then the intermediate code is not saved on a file. If the value of LAPUNCH is PUNCHB, the intermediate code is punched on binary cards with an end-of-record separator between functions. LAPUNCH is initialized to OUTLAP.

### 6.3.7 Errors Detected by the Assembler

The assembler is capable of detecting certain errors in the program being assembled. Each error causes a message to be printed. The various messages are given below with an explanation of their meaning.

## Errors Detected in Pass 1

<instr> HAS UNRESOLVED REGISTER NUMBER

<instr> is the instruction being processed. The instruction specifies a non-numeric value for a register number.

<instr> INCORRECT HEADER ELEMENT

<instr> is not a MAIN pseudo instruction and it should be.

<symbol> IS NOT A LEGAL OPERATION

<symbol> has been used as the operation code in an instruction, but it is not defined as a machine instruction, a macro instruction, or a pseudo instruction.

<symbol> IS MULTIPLY DEFINED

<symbol> has been used more than once as a label in the program.

<instr> HAS SYNTAX ERROR

<instr> has a legal operation code, but its form is unrecognizable.

<instr> ILLEGAL ENTRY

<instr> is an ENTRY pseudo instruction which has occurred without a MAIN pseudo instruction preceding it.

<instr> HAS NON-NUMERIC ARG

<instr> is a VFD pseudo instruction which has a non-numeric value where a field width is expected.

<instr> HAS MORE THAN 60 BITS DESIGNATED

<instr> is a VFD pseudo instruction in which the field widths total more than 60 bits.

## Errors Detected in Pass 2

ILLEGAL ADDRESS EXPRESSION (EVALK):<exp>
ILLEGAL ADDRESS EXPRESSION (EVALJ):<exp>

<exp> is an address expression which is somehow not permissible. In the first case, <exp> is an expression preceded by something other than a plus or minus sign. In the second case, <exp> either contains an invalid operator or is an undefined symbol.

If any errors are detected during pass 1, the intermediate code is not written onto a file, if such action was requested. If any errors are detected by either pass 1 or pass 2, the code is not loaded into memory. The message:

<n> ERRORS IN LAP ASSEMBLY

is printed if any errors were detected, where <n> is the number of errors.

### 6.3.8 Output of the Assembler

Aside from the intermediate form of the code which may be written to a file, and the symbol table which is returned as the value of LAP, there is very little printed output from the assembler. The four lines of output produced by the assembler have already been shown in section 6.2.1. They give statistical information about the assembly. They are:

| | |
|---|---|
| (PASS1-TIME = <t1>) | <t1> = time (milliseconds) spent in assembler first pass. |
| (LENGTH <number>) | <number> = length (in words) of the assembled code. |
| (LAP-TIME = <t2>) | <t2> = total time (milliseconds) spent assembling. |
| (ORIGIN <number>) | <number> = address of first word of code in memory. |

<t2> includes loading time if loading was performed. The ORIGIN is zero if the code was not actually loaded.

### 6.3.9 Coding Conventions

The user who wishes to write an assembly language function which can communicate with the rest of LISP needs to know certain things about the internal workings of LISP. A brief survey of important topics is given here. Users who need more knowledge should consult with the Computation Center personnel responsible for LISP.

#### 6.3.9.1 Register Conventions

Two registers have predetermined and fixed meanings within LISP. They are:

| Register | Use |
|---|---|
| A0 | Always contains a pointer to the next available free-space word. |
| B7 | Always contains a pointer to the current top of the stack. B7 + PDSTACK is the location of the top, and B7 decreases as new items are added to the stack. |

Assembly language programmers should always make sure these meanings are preserved over any programs they write.

### 6.3.9.2 Calling Sequences

All LISP functions are entered by direct jumps, not by the return-jump instruction. On entry to the function, register B6 contains the appropriate return address for the function, so that function exit is accomplished by a

```
(JP B(6))
```

instruction. If another LISP function is called during the course of this one, then the value in B6 must be saved and restored.

Arguments are passed to LISP functions in registers A1-A5. For a SUBR function, A1-A4 contain the first four arguments, while A5 contains a pointer to a list of any remaining arguments; for an FSUBR, A1 contains a pointer to a list of all arguments.

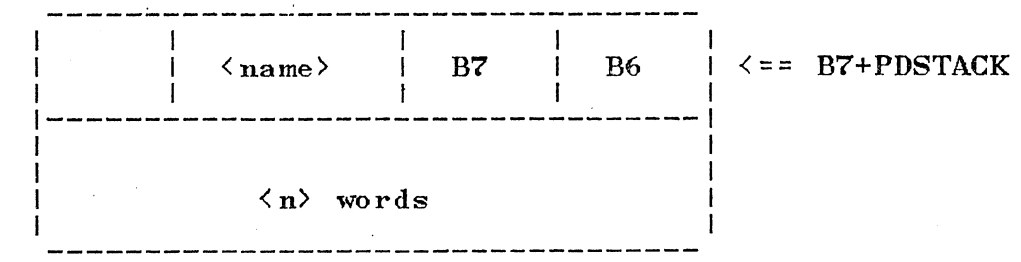
Any value computed by the function is expected to be in register A1 on exit from the function.

### 6.3.9.3 Coding Examples

1) Saving information on the stack

```
(SB(1) "<function-name>)
(SB(2) <n>)
(RJ SAVE76ER)
```

The above code sequence saves the <function-name>, B6, and B7 on the stack in the expected form so that the <function-name> can appear in the backtrace (see section 8.4) if an error occurs within the function. If <n> is greater than 0, <n> words are allocated below the top of the stack for use in saving other information. The stack configuration after execution is:




The value of B7 saved in the top word is that value B7 had before the additional entry was made. Other data to be stored on the stack may be referenced by the address expression.

```
B(7) + (+ PDSTACK i)
```

where i = 1 to <n>.

It is recommended that this code sequence be executed at the beginning of each LISP-callable function.

Information saved on the stack must be explicitly removed before exiting the function.

2) Restoring the stack

```
(SA(i) B(7) + PDSTACK)
(SB(6) X(i))
(AX(i) (18))        (i in range 1-5 inclusive)
(SB(7) X(i))
```

This code sequence restores B6 and B7 to their values stored at the top of the stack. Other data on the stack must be retrieved by knowing their positions relative to B7.

3) Binding variables

```
(SB(1) "<list-of-variables>)
(RJ BINDIT)
```

Execution of this code binds all the arguments passed in A1-A5 to the atoms contained in the list of variables. The arguments can then be referenced as the values of those variables. BINDIT places a pointer to the list of variables on the stack and decrements B7 by 2. Variables bound in this way must be explicitly unbound before leaving the function.

4) Accessing the value of a variable

```
(SA(i) "<variable>)          (i in range 1-7 inclusive)
(SA(i) X(i))
```

This code places the variable's value into register A(i).

5) Unbinding variables

```
+ (SB(3) (+ * 1))
  (EQ APPLLG)
```

This code retrieves the list of variables saved on the stack and unbinds each atom. Register B7 is incremented by 2. This code destroys the contents of A1.

6) Calling a SUBR or FSUBR

```
(code to put arguments in A1-A5)

+  (SB(6) (+ * 1))
   (EQ <name>)
```

7) Calling an EXPR

```
        (code to build a list of arguments in A2)

           (SA(1) "<name>)
        +  (SB(6) (+ * 1))
           (EQ APPLY)
```

8) Getting a word of free space

```
           (SA(i) A(0))          (i in range 1-5 inclusive)
           (SA(0) X(i))
        +  (NZ X(i) (+ * 1))
           (RJ GARBAGE)
```

This operation must be performed in this manner. At the end of the sequence, register A(i) contains a pointer to the free word, which can now be used, and A0 has been properly updated.

9) Get a word of full space

```
           (SA(i) FULLLIS)       (i in range 1-5 inclusive)
           (SA(i) X(i))
           (BX(j) X(i))          (j equals 6 or 7)
           (SA(j) FULLLIS)
        +  (NZ X(i) (+ * 1))
           (RJ FGARBAG)
```

This operation must be performed exactly as indicated. At the end of this sequence, the address of the full-space word, which can now be used, is in register A(i).

10) Exiting a function

```
           (JP B(6))
```

Prior to executing this instruction, it will be necessary to ensure that B6 contains the same value it had on entry to the function.

## 6.4 THE LISP LOADER

This section explains how compiled and/or assembled code is placed into memory. If the variable LOADFLAG is non-NIL at assembly time, the assembler automatically calls the loader during its second pass over the program. However, to use the intermediate form of code which has been saved on a file, the loader must be called by use of the LISP function READLAP.

READLAP is an FSUBR function which may have 0, 1, or 2 arguments:

```
(READLAP <filename> <lat>)
(READLAP <filename>)
(READLAP)
```

<filename> is the unquoted name of a file from which READLAP is

to read functions in the intermediate form produced by LAP. If the <filename> is omitted, then READLAP reads from the current system input file (SYSIN). The <lat>, if given, is a list of function names whose definitions may appear in the set to be loaded, but which are to be ignored if encountered. This feature is useful to avoid loading superceded functions without having to regenerate the entire file of functions of which they may be a part.

READLAP begins reading at the current position of the indicated file. Each function is read and pass 2 of the assembler is performed on it, loading the functions into memory one after the other. Reading stops when the end-of-file is encountered or when the atom //ENDLAP is read, whichever occurs first.

### 6.4.1 The Loading Process

Pass 2 of the assembler evaluates address expressions and combines completely assembled instructions into full-word units. As each word of information is completed it is stored into the next available word of binary program space (see chapter 3). If there is not sufficient memory remaining in binary program space to accommodate a function, a request is made to the operating system to allocate more memory. Such requests are made in multiples of 1000 (octal) words.

A known amount of binary program space can be allocated in LISP by means of the X parameter on the control card. If the needed amount is known, this option can be used to avoid making requests on the operating system.

### 6.4.2 Output from READLAP

As each function is read, its name is printed by READLAP. The names are printed across the page, as a record of the functions loaded.

If pass 2 of the assembler detects errors, the error messages given in section 6.3.7 are printed. The last function name printed before the error messages is the name of the function in which the errors were detected. If any errors occurred, the message:

TOTAL ERRORS = <number>

is then printed. Any function in which an error was detected is not loaded into memory.

After all functions have been processed, the following message is printed:

TOTAL LOAD TIME = <number>

<number> is the total time in milliseconds spent in the loading process.

The value returned by READLAP is a list giving the entry points and names of all functions loaded. The list contains the entries in the reverse order of their actual loading. The form of the list is:

```
((((<name> <type> <address>)
     (<name> <type> <address>) ...
       (<name> <type> <address>))
 ((<name> <type> <address>) ...
          ...))
```

Each sublist contains the names, types and addresses of all entry points to one function.

## 6.5 FINAL COMMENTS

The compiler and assembler are themselves large programs which get loaded into binary program space. There is no way to reuse the space they occupy. It is usually wise, therefore, to compile and assemble in one LISP run, saving the intermediate code, and then to load and execute the functions in a separate LISP run.

If several runs of the compiler and assembler are expected, it may be wise to generate and save a DEFSYS of the system immediately after loading LAP and LCOMP.

LCOMP may be loaded separately from LAP if space is at a premium. Setting PRNTFLAG to true causes a listing of the compilation result to be output to //WRS. If this listing is slightly altered to the form acceptable by LAP and if the macros which begin the LCOMP file are loaded, then the output of the compiler may be assembled in a second pass.

References:

1. Cohagan, W. L., A LISP Compiler/Assembler System for the CDC 6400/6600. Master's Thesis, University of Texas at Austin, 1971.

2. Control Data 6000 Series Computer System/6000 Compass Version 2 Reference Manual, Pub. No. 60279900.

# 7. LISP OVERLAYS, THE FORTRAN INTERFACE, AND VIRTUAL MEMORY

This chapter describes the facilities in UT LISP which answer the following three questions:

1) How can a LISP program be saved for future use so that it does not have to be reloaded from source language input each time?

2) How can a large LISP program be partitioned with dynamic linkage of the pieces during execution?

3) How can a FORTRAN program be used in conjunction with a LISP program?

All three of these capabilities exist in UT LISP. The first two questions are answered by the use of LISP overlays, the third by use of the FORTRAN overlay system.

## 7.1 THE LISP OVERLAY

In chapter 3 we indicated that the memory allocated to a running LISP program is divided into several distinct areas. All user-defined programs and data reside in an area of memory which starts at the address following the interpreter code and extends to the address specified by the field length. This area plus a small amount of memory containing pointers and system status information constitutes the LISP overlay area and can be written onto a disk file at the user's command.

The overlay is written onto the file as two logical records. The first record contains all system status information including the field length in effect at the time of overlay definition, the return address for the function in execution when the overlay was defined, and a version number for LISP. The second record contains everything else. Once the overlay is on the disk file, it may be saved and/or manipulated in any way that any disk file may be.

Functions exist within LISP to load the overlay back into memory in a two-step process. In the first step the first record of the overlay is read into memory. If the version number contained in the overlay does not agree with the current LISP version number, LISP terminates with a message. It is not possible to use an overlay that was created under one version with a different version. If the version numbers match, the field length of the job is dynamically updated to that required for the overlay, and then the rest of the file is read into memory. Finally execution resumes at the point where it was suspended when the overlay was created. In this sense, the overlay is a dynamic snapshot of the LISP program.

## 7.2 CREATING A LISP OVERLAY

LISP overlays are created by function DEFSYS.

(DEFSYS <filename> <boolean>)

pseudofunction; SUBR

DEFSYS writes the current LISP program onto the file <filename> as a LISP overlay. <filename> is rewound before being written. If <boolean> is true, execution terminates immediately after the overlay is written. If <boolean> is NIL, execution of LISP continues, and DEFSYS prints a message:

FIELD LENGTH: <number>

where <number> is the current field length needed for the overlay. The value of DEFSYS is *T*.

## 7.3 REFERENCING A LISP OVERLAY

LISP overlays may be used in three different ways. They may be simply loaded, loaded and control unconditionally passed to them, or loaded and used as an extension to the running program. When loaded via the S parameter on the LISP control command, they are used as programs saved for use by being simply loaded without reinitialization.

### 7.3.1 Simple Loading of a LISP Overlay

A LISP overlay is loaded and replaces the running LISP program by execution of function LOADSYS.

(LOADSYS <filename>)

pseudofunction; SUBR

LOADSYS rewinds and loads file <filename>. Unpredictable results will be obtained if <filename> does not contain a LISP overlay. LOADSYS itself returns NIL and execution resumes in the newly-loaded overlay where execution was suspended when the overlay was defined. The use of S=<filename> on the LISP control command (see chapter 2) is equivalent to executing (LOADSYS <filename>) before any other processing is done.

### 7.3.2 Linking to a Particular Function in an Overlay Without Return

By executing function OVERLAY a user can load a particular overlay and request execution of some particular function defined in the newly-loaded program.

(OVERLAY <filename> <function> <list> <lat>)

pseudofunction; SUBR

<filename> specifies the overlay to be loaded. <function> is a function to be executed in the new program. <list> is a list of actual arguments to which <function> is to be applied. <lat> defines an environment for the execution of <function>. The elements of <lat> should be variables which are defined in the initial LISP program and referenced by <function>. The values of these variables are passed into the newly-loaded overlay and redefined there so that they will be defined when <function> references them. OVERLAY writes <function>, <list> and the environment onto a scratch file named ARGS (there must be a buffer available for this file), then it loads the overlay from <filename>. After loading the overlay, OVERLAY then rewinds ARGS and reads <function>, <list>, and the environment, establishes the variables in the environment, and applies <function> to <list>. After executing <function>, control returns to SYSIN as though <function> had been executed from the top level of LISP.

### 7.3.3 Linking to a Particular Function in an Overlay With Return

By use of the function CALLSYS, one can not only load an overlay and execute a function in it, but also save the calling environment so that the environment can be restored when the overlayed function finishes. Thus CALLSYS allows LISP programs to be partitioned into several parts with almost transparent linkages between them. Overlays may even link to themselves recursively.

(CALLSYS <filename> <function> <list> <lat> <boolean>)

pseudofunction; SUBR

The first four arguments of CALLSYS are identical in meaning and function to those of OVERLAY. The operation of CALLSYS is similar to that of OVERLAY except that before loading the new overlay, the calling program is saved on a temporary disk file. Then after executing <function> the value of the execution and the environment are written onto ARGS, the saved program is reloaded, the value and environment are read from ARGS, the environment is re-established, and the value is returned as the value of CALLSYS. The environment is passed back into the calling program so that any changes made in it by the called function are properly reflected. The temporary file holding the saved programs operates like a stack so that chains of overlays may be properly entered and exited. An exception to this operation occurs if the fifth argument, <boolean>, is true. If so, the calling program is not saved, and return from the loaded overlay goes to the overlay which loaded the overlay that executed CALLSYS with <boolean> true. This feature is properly used when a return from CALLSYS would cause an immediate return to a lower-level overlay. Use of this feature eliminates the I/O charges otherwise incurred by saving and reloading a program which will not be used.

Example: Suppose the following run were made. Only the interesting aspects of the run are shown.

```
command:        LISP,S=AA.  (control command)

action:         load AA and resume execution

command:        (DIFFERENCE
                  (CALLSYS "BB "FX (LIST P Q) NIL NIL)
                  (FY PQ))

action:         save AA
                load BB
                execute function FX
                reload AA and return value FX
                execute function FY
                execute function DIFFERENCE

command:        (CALLSYS "CC "RST NIL NIL NIL)

action:         save AA
                load CC
                execute  RST,  which contains the following
                command

command:        (CALLSYS "DD "PQR NIL NIL T)
                  (determine value of RST)

action:         load DD
                execute PQR
                reload AA and return value of PQR as  value
                of RST, etc.
```

### 7.3.4 Hints and Warnings About LISP Overlay Use

The file operations of creating and loading overlays are done by UT LISP as efficiently as possible. The files involved, however, can be quite large and consequently the I/O charges for these operations can be very heavy. For example, a 77000 (octal) word program in overlay form occupies about 350 sectors on the disk. At 4 milliseconds of charge time per sector transferred, a LOADSYS of such an overlay costs 1.4 seconds, and a CALLSYS between one such overlay and another costs 4.2 seconds in I/O charges alone. Overlays should be used judiciously.

Information can be passed between LISP overlays in only three ways via OVERLAY or CALLSYS: through the argument list for the function, through the environment specified, or by means of an external file written by the caller and read by the callee. It is particularly important to note that information stored on the property lists of literal atoms in the calling program is not and cannot be automatically passed to the called program. One solution to this limitation is to make sure that the same atoms and property lists are kept in both overlays. Another solution is to keep such property list information in an overlay by itself and reference that overlay any time the information is needed.

### 7.3.5 Error Return From CALLSYS

If an error occurs while executing functions in an overlay loaded by CALLSYS, the appropriate error message and backtrace are printed if enabled. If the error occurs within the scope of an ERRORSET in the last-loaded overlay, then control stays in

that overlay. If, however, no ERRORSET has occurred in the last-loaded overlay, LISP backtracks through the chain of loaded overlays until either the top-level program is restored, or a program is restored which had an ERRORSET call in it.

The backtracking process involves restoring each saved core-image in the reverse order of entry. After each is restored, the message

***** ERROR: CALLSYS

is printed, followed by the backtrace of the overlay just restored, as if it was at the point of entry to the overlay just exited. Thus a complete history of the entire chain of overlay entries can be viewed.

Example:

Suppose program AA has CALLSYSed overlay BB which has CALLSYSed overlay CC which has CALLSYSed overlay DD and an error occurs in overlay DD. Assume that overlay BB has called ERRORSET. Then the output will contain

```
***** error: (message for the error)
>>>>> STACK: (backtrace for overlay DD)
***** ERROR: CALLSYS
>>>>> STACK: (backtrace for overlay CC)
***** ERROR: CALLSYS
>>>>> STACK: (backtrace for overlay BB)
```

and overlay BB will retain control.

## 7.4 THE LISP - FORTRAN INTERFACE

A properly-coded FORTRAN overlay may be accessed from a LISP program to perform some kind of calculation to which FORTRAN is better suited. The facility provided is minimal, but effective. Note that LISP may link to a FORTRAN program, but FORTRAN may not link to a LISP program. The overlay must have been programmed in FORTRAN, not some other language similar to FORTRAN. From LISP, a FORTRAN overlay is accessed by executing the function FORTRAN:

(FORTRAN <filename>)

pseudofunction; SUBR

<filename> is a disk file containing FORTRAN overlays. The FORTRAN function first saves almost all of the current core image on a disk file. Only I/O buffers and enough code to reload the saved image are retained. A (1,0) overlay is then loaded from file <filename> and control passed to it. The FORTRAN overlay must obey certain restrictions (see below). Communication between LISP and the FORTRAN program is via disk files only, and must be explicitly programmed by the user. The FORTRAN program may access a maximum of 6 files, and these must have been opened by LISP before loading the FORTRAN program. The FORTRAN function returns a value of NIL if the overlay load was unsuccessful, or *T* if loading and execution proceeded normally.

Only programs compiled by the RUN FORTRAN compiler can be used. Since a small portion of LISP code remains in memory while the FORTRAN program is active, it is necessary that the FORTRAN

overlays be so structured that they do not overlay that code. When the overlays are created by the system loader, a dummy (0,0) overlay must be included in the relocatable binary input so that the load address of the (1,0) overlay will be at the proper point. This (0,0) overlay is never used and can best be implemented with the following COMPASS program:

```
        IDENT     DUMMY
        LCC       OVERLAY(<file>,0,0)
        ENTRY     DUMMY
DUMMY   BSS       <number>
        END       DUMMY
```

<file> is the name of the file on which the absolute overlays are to be written and <number> is the appropriate value to assure that the (1,0) overlay starts in the right place. This value may change with different versions of LISP, but is always available as the value of the atom FORFIRST. This <number> is not absolutely critical in the sense that the (1,0) overlay may be loaded at a higher address and still work; in this case, LISP reloads it at the correct address.

The FORTRAN program must fit into the field length at which LISP is running when the FORTRAN function is called.

## 7.5 WARNINGS ABOUT RESERVED FILE NAMES

The file used by the LISP functions when saving core images is named FROM, and the file used to save core images by FORTRAN is named FTEMP. These are not LISP files, in the sense that they do not use the regular LISP buffers. If the user has local files of these names, they will be destroyed by LISP if overlays are used.

## 7.6 VIRTUAL MEMORY FOR FUNCTIONS

LISP programs using many functions dynamically often cannot be broken into overlays conveniently or become extremely expensive to run when so partitioned. LISP users may find the virtual memory facility useful in these situations. Virtual memory programs make implicit use of the random access functions to create a file containing the user's designated virtual functions. These functions are then capable of having their in-core definitions removed when they are not being executed by the running LISP system and can subsequently be recreated from the information stored in the virtual function file.

The virtual function file contains pointers to the actual locations of atoms in core instead of the original symbolic representations of these atoms. Users must therefore either save the virtual function file along with a DEFSYS of the LISP system containing their loaded program or recreate the virtual function file on each subsequent run. DISKOUT makes a virtual function file random. However, if an overlay is created with the virtual functions, re-entry of the overlay will not cause the virtual function file to be random and a RANDOM I/O error may occur. The following functions describe the use of the virtual memory facility.

(DISKOUT <fixnumber1> <fixnumber2> <list>)

pseudofunction; SUBR

DISKOUT is the basic function of the virtual memory system. This function writes the definition of the functions whose names appear in <list> onto the virtual function file VIRFN. The user may change the name of this file by changing the value of the system variable VIRFN. Each function name given in <list> is assumed to have previously been defined.

DISKOUT also puts a VIRFN property indicator on the property list of each function name given in <list>. This indicator tells the system that the function has been defined on the virtual function file. Subsequent redefinition of any function previously used in a call to DISKOUT causes the disk-resident definition of that function to be lost.

There are two numeric bounds on the amount of memory the LISP system allows for in-core definitions of virtual functions. <fixnumber1> and <fixnumber2> determine the respective lower and upper bounds on the size of the in-core virtual function memory. Two system variables //CODEMIN and //CODEMAX are bound to these values. Typical values for //CODEMIN and //CODEMAX are respectively 4000 and 7000, allowing for 4000 to 7000 words of in-core virtual function definitions.

Whenever a function which is not in core is called by the LISP program, LISP checks to see if it will fit in core without exceeding //CODEMAX (i.e., <fixnumber2>). If so, the function is read into core. If not, functions which have resided longest in in-core virtual function memory are removed until the amount of in core code is less than //CODEMIN; then requested function is brought into core and execution continues. This transfer is of course performed in a manner which is transparent to the running program.

DISKOUT returns <list> as its value.

(GETD <litatom>)

pseudofunction; SUBR

GETD returns the functional definition of <litatom> by getting the value associated with the EXPR or FEXPR indicator on <litatom>'s property list. GETD can be used to force the retrieval of <litatom> from the virtual function file if its definition is not presently in core, provided <litatom> has been the argument of a previous call to DISKOUT. For more information on GETD see section 4.7.

# 8. DEBUGGING THE LISP PROGRAM

This chapter explains the causes and cures of the errors detected by UT LISP and offers general advice for debugging LISP programs. In the interpretive mode, LISP has very powerful error-checking capabilities, and the associated messages can provide the user with considerable information to aid in debugging his program. Extensive error checking is not provided in compiled LISP code in order that it may run faster; it is assumed that compiled functions have been thoroughly debugged before compilation.

## 8.1 DAYFILE ERROR MESSAGES

The following DAYFILE error messages are currently given by UT LISP. All errors indicated by DAYFILE messages are absolutely fatal to execution.

INSUFFICIENT FIELD LENGTH

This message is given when the LISP system finds that the field length provided permits an impractically small amount of free and full-word storage. A field length of 50000 (octal) is recommended as the minimum practical size.

PARAMETER ERROR <letter>

This message is given when parameter <letter> on the LISP control command violates the restrictions described in section 2.1.1. These restrictions are that X, S, I, and O can appear only once each, and that C cannot appear in combination with I or O. Also, single-letter names may not be used for user-defined subsystems.

CANT LOAD <filename>

This message is given when <filename> appears on the control command as a subsystem name and there is some error during loading. This error implies that there is something wrong with the file.

## 8.2 UT LISP ERROR MESSAGES AND THEIR MEANINGS

Three types of error messages may appear during a LISP run:

(a) Informative messages. These messages report conditions which are not fatal to execution but should demand user attention. After printing the message, the LISP system continues execution as best it can. These messages are preceded by /////.

(b) Recoverable errors. Errors represented by these messages cause execution of the current expression or doublet to be terminated. The LISP system is able to recover, however, and continue with the next expression or doublet. Each message is preceded by *****, and is associated with an "error code" atom whose value is passed to EVAL upon detection of the associated error (see section 8.5).

(c) Non-recoverable errors. Errors represented by these messages cause the entire LISP run to be terminated. These messages are preceded by !!!!!.

All error messages are set off from the rest of the output by blank lines before and after, and therefore should be easily spotted by the user. Most recoverable error messages are accompanied by a backtrace. Informative messages and non-recoverable error messages are not so accompanied. The backtrace is a reduced listing of the contents of the LISP stack at the time of the error, and the primary information it conveys to the user is the sequence of functions which have been entered, but not completed. In form the backtrace is:

>>>>> STACK: (<name> <name> <name> ... <name>)

for as many lines as are necessary. In the case of EXPR and FEXPR functions, <name> is actually a list containing the function name and the actual parameters with which it was called. Occasionally, information other than function names, such as lists and numbers, may appear, but the chief fact of interest to the user is that every function entered, both user-defined and system-defined, is named on the stack. The names are printed in forward order, with the most-recently-entered function appearing last (see section 8.4).

### 8.2.1 Errors Detected During Input

These errors imply that the user's data is incorrect. They can be eliminated only by revising the data.

///// ILLEGAL DOTTED PAIR SYNTAX

While reading an S-expression, LISP detected one of the forms

--- S-exp1 . S-exp2 . S-exp3)
or --- S-exp1 . S-exp2 S-exp3)

Only --- S-exp1 . S-exp2) is a legal dotted pair. S-exp3 is completely ignored, and LISP continues. If the file is being listed as it is read, this message appears directly beneath the line containing the error.

///// ILLEGAL SEQUENCE .)
///// ILLEGAL SEQUENCE (.

These two character sequences are illegal in an S-expression. The period is ignored in both cases. If the file is being listed as it is read, the message appears beneath the line containing the error.

///// CONSECUTIVE DOTS

Two or more consecutive periods were found in a dotted pair. All but the first period are ignored. If the file is being listed as it is read, this message appears beneath the line containing the error.

///// UNMATCHED RIGHT PARENTHESES

This message occurs if the first non-blank character of a new S-expression is a right parenthesis. The implication is that there is an excess of right parentheses over left parentheses. The extra parentheses are ignored. If the file is being listed as it is read, this message appears directly beneath the line containing the error.

///// EXTRANEOUS DOTS

This message occurs if the first character read for an S-expression is a period. That is, one or more periods appear between S-expressions on the input file. The periods are ignored. If the file is being listed as it is read, this message appears directly beneath the line containing the error.

***** UNMATCHED LEFT PARENTHESES

Code: ERRI1

This message occurs if an end-of-file is read before sufficient right parentheses have been read to match all left parentheses. It is symptomatic of missing data cards.

### 8.2.2 Errors Detected During Output

See section 5.3.

### 8.2.3 Errors Detected by File Manipulation Functions

***** ATTEMPTED TO OPEN MORE THAN SIX FILES

Code: ERRI2

This error occurs for calls to the RDS, WRS, REWIND, ENDFILE, or OPEN functions which attempt to open a seventh file. LISP can accommodate only six files at one time. If more than six files are needed, the advised approach is to CLOSE a file and open it again later if necessary.

***** ATTEMPTED TO CLOSE SYSIN OR SYSOUT

Code: ERRI3

This error is detected by the CLOSE function. It is never permissible to close the file which is equated to either SYSIN or SYSOUT. SYSIN and SYSOUT must always exist. They may be changed, however (see chapter 5).

***** RANDOM I/O ERROR: <filename>

Code: ERRR1

This error is detected by either RANIN or RANOUT when a random I/O operation is attempted on a file which has not been opened in random mode. (See section 5.2 for opening a file in random mode.)

### 8.2.4 Errors Detected by the Garbage Collector

***** INSUFFICIENT FREE SPACE

Code: ERRGC2

***** INSUFFICIENT FULL SPACE

Code: ERRGC3

This error occurs when the system was only able to recover less than 1/64th of the space initially allocated to the indicated area. The implication is that it would be futile to continue execution, since garbage collection would occur more and more frequently with less and less gain. When this error occurs, the system returns control to the last occurrence of ERRORSET or to SYSIN, whichever occurs first. The problem can be cured most easily by increasing the field length for the next run. Alternatively, the A parameter on the LISP control command may be adjusted (see chapter 2). Decreasing the value of the A parameter effectively allocates more of the given space to the full-word space region, whereas increasing the value allocates more to the free space.

///// FREE SPACE IS CROWDED
///// FULL SPACE IS CROWDED

One of these two informative messages occurs if the garbage collector finds that less than 1/16th of the allocated space in the respective region is available. These messages indicate that the program may run more efficiently either in a larger memory space or else with adjustment in the A parameter on the LISP control command (see chapter 2).

///// GARBAGE COLLLECTED: <n1> <n2>

This informative message appears every time a garbage collection occurs if the G parameter is specified on the LISP control command, or if it is turned on by setting //GC to true (see section 4.14). In the message, <n1> is the number of free-space words which were recovered and <n2> is the number of full space words which were recovered.

### 8.2.5 Errors Detected by the Interpreter

***** UNDEFINED FUNCTION: <name>

Code: ERRA3

The interpreter has encountered an atom, <name>, which is syntactically in the position of a function name and for which it was unable to find a functional definition. Most commonly this error is caused by an extra set of parentheses around a form within a LISP expression. Such a pair of parentheses causes the LISP interpreter to treat the results of the enclosed form as another call on a function. Typically, the result is some data list whose first element is not a function name. Another cause is the omission of a necessary QUOTE in front of a data list, causing the data list to be interpreted as a form. Finally, this error may arise if there is a parenthesis error within the expression or doublet which (is supposed to have) defined the function. Such an error may cause the function not to be recognized by DEFINE or DEFLIST. The backtrace indicates the function within which the erroneous function call occurred.

***** NUMBER TREATED AS FUNCTION: <number>

Code: ERRA4

This error occurs when the interpreter finds <number> in a position where it expects a function name. The causes of the error are the same as for the "undefined function" error, that is, an extra set of parentheses around a form or else a missing QUOTE. Again, the backtrace indicates which function contained the erroneous information. The number itself is printed in the message.

***** RECURSION LIMIT EXCEEDED: <list>

Code: ERRA5

Certain information is placed on the stack when a function is entered and is removed from the stack when the function is exited. Typically the amount of information placed on the stack is from two to six words. This error occurs when there is no more room on the stack. It is impossible to specify how many levels of recursion are allowed by the system, since the limit depends on the particular sequence of functions which are entered. In general, though, a stack size of one thousand words seems to allow two hundred to three hundred levels of recursion. Technical reasons for this error to occur are the failure of the programmer to include terminating conditions in a recursive function. The error may occur in certain system functions such as PRINT and APPEND, if they are given a circular list to process. Finally, it is possible that a recursive function is operating on data which causes it to recur too many times. If the latter is the case, then it is advisable to rewrite that function in an iterative manner using the PROG feature. This problem can also be eliminated by increasing the stack size via the B parameter on the LISP control command. No backtrace is given with this error. <list> is a list of the last eight entries on the stack.

***** UNBOUND VARIABLE: <name>

Code: ERRA6

A variable which has not been assigned a value can be detected by the LISP system. This message occurs when the value of such a variable is required in a computation. <name> is the supposed variable and information about the function in which it was encountered is given in the backtrace. Typical causes of this error are a keypunch error or a missing QUOTE around a data atom, in which case the data atom is evaluated as a variable. This error is not detected within compiled functions.

***** TOO MANY ARGUMENTS: <name>

Code: ERRA7

***** TOO FEW ARGUMENTS: <name>

Code: ERRA8

One of these errors occurs when a user-defined EXPR function is called with either too many or too few arguments. These conditions are not detected for machine-coded SUBRs or compiled functions. <name> is the function which was improperly called and the name of the calling function appears in the backtrace information. The user must redefine his function or correct the function call to correct this error.

***** NON-HIERARCHICAL FUNARG

An error has occurred in "unpeeling" the stack, probably caused by incorrect use of "funarg" expressions. No backtrace is given.

### 8.2.6 Errors Detected Within Particular LISP Functions

***** ILLEGAL ARGUMENT: (<fn> . <arg>)

Code: ERRA1

This error is detected by several LISP functions. In this message the system prints the result of (CONS <fn> <arg>), where <fn> is the name of the function which received <arg> as its illegal argument. The backtrace gives the name of the function within which the call was made. The user should consult the function descriptions (chapters 4, 5, 6, 7, and 9) to determine the reasons for the error. This error typically occurs when a function is given an atom when a list is expected or vice versa, or else when a numeric function is given a non-numeric argument. At times the function name given may not seem to be related to the function called by the user, since certain of the LISP system functions call other functions.

***** ILL-FORMED ARGUMENT: <name>

Code: ERRA2

This error is similar to the illegal argument error. It occurs when a non-list dotted pair is given to a function which expects a list and looks for the NIL terminator of that list. <name> is the function receiving the ill-formed argument. The user must rewrite his call on the function to correct the error.

***** ERROR: <s>

Code: ERRA0

This message occurs for a user-detected error when the user calls the system ERROR function. The message includes the S-expression <s>, which is the argument to the ERROR function and can be anything the user desires.

***** RETURN OR GO OUTSIDE A PROG

Code: ERRP1

This error occurs if a RETURN or GO function was called within another function which was not executing within the overall control of a PROG expression. This situation is not allowed and the user must redefine the function to correct the error.

***** GO TO NON-EXISTENT LABEL: <name>

Code: ERRP2

Within a PROG expression the function GO was called with a non-existent label, <name>. The most common cause of this error is a keypunch error.

***** CHARACTER BUFFER EXCEEDED

Code: ERRC1

Detected by the PACK function, this error occurs when an attempt is made to pack more than the limit of 120 characters into the character buffer.

!!!!! NON-MATCHING OVERLAY

The UT LISP system is not a static one, but evolves in time. LISP overlays created under one version of the system are not likely to be able to run under another version of the system. The system contains internal information specifying which version it is, and each overlay contains the version under which it was created. The two version numbers are compared when an overlay is loaded and this error occurs if they do not match. It is a fatal error since execution would almost surely be unsuccessful. To correct the error the user must redefine this overlay under the current version of the system.

!!!!! ILLEGAL FORMAT FOR //MODE

The system control variable //MODE determines the top-level behavior of LISP. It must have the format

(<function> . <positive-integer>)

Any deviation of //MODE from this format terminates execution. However, for conversational mode, LISP sets //MODE to (EVAL . 1) after printing the error message and then returns to execution of SYSIN.

!!!!! KILLED: <s>

This message occurs when the user purposely terminates a program by executing function DIE with <s> as its S-expression argument.

## 8.3 WHAT TO DO IF THE ANSWER IS WRONG

It is often the case that a program executes without any of the errors previously described, yet gives incorrect results. In such a case the user must apply a debugging procedure to determine the errors in his own functions. We shall try here to indicate a reasonable procedure for debugging running LISP programs.

It is assumed that the user is familiar with the purposes of each of the functions that he is attempting to use. If this is true, then a study of the form of the incorrect result often indicates approximately where within the set of functions the error is occurring. This location greatly reduces the amount of debugging effort required, since the user's attention can then be focused on that area. In any event the first step is to carefully review the form of the functions as they were input to the LISP processor. This review is facilitated by a listing of the functions which shows the parenthesis level count, obtained by using the P parameter on the LISP control command (see chapter 2). With such a listing, the user should carefully scrutinize his function definitions, making certain that they are syntactically correct. It is possible for simple parenthesis errors to cause a function to have a form such that it neither serves the intended purpose nor causes a LISP system error. Any errors in syntactic form discovered in this way should be corrected, and the job rerun to determine if an erroneous answer still occurs.

Once the syntactic forms of the user's functions are correct and an erroneous result still occurs, then the various tracing facilities of the LISP system can be used to gain more information about the behavior of the functions. The TRACE function described in chapter 4 is used to turn on tracing for any selected set of user and/or system-defined functions. Its converse, the UNTRACE function, can be used to selectively turn off such tracing. When a function is traced, then when that function is entered the arguments which were actually received by it are printed, and when it is exited the value it generates is printed. The trace output is produced even when the function is entered via recursion from within itself. Thus, tracing a recursive function yields complete information about the way it operates on a given set of data.

Tracing of functions should be used very carefully. The extra printing involved in the tracing information costs the user both pages of output and extra time. If a great many functions are traced at one time, these extra costs can add considerably to the cost of a job. It is suggested, therefore, that the user trace only those functions of which he suspects erroneous operation. Also, he should trace those functions while they are operating on a minimal-sized data set which will reproduce the error. A very useful procedure to apply is to make one run tracing all of the functions in which a suspected error is occurring. Then, by reviewing the output, the user can determine which of the traced functions are operating correctly, and perhaps which of the functions is operating incorrectly. If the erroneous function(s) cannot be determined for one reason or another, the tracing output will usually point to some other functions which may not have been traced that are better candidates to be traced the next time. Then the next test run should be made without tracing those functions which are known to be operating correctly. Thus, through a progressive sequence of runs the erroneous function can be isolated.

Tracing, as has just been described, is very useful for debugging recursive functions. It does not, however, always yield complete information about iterative functions which have been written using the PROG feature. In this case, tracing shows the arguments which were received by the function; and shows the value which was returned by the function; but shows nothing of what happened in between, unless the iterative function is also recursive. For such functions another tracing procedure is available, namely, the TRACESET function. This function and its converse, the UNTRACESET function, are both described in chapter 4. They allow one to selectively turn on and turn off the tracing of all SET and SETQ value assignment statements within a PROG expression. When TRACESET has been applied to a given function then every time a variable is assigned a value via the SET or SETQ functions within the PROG expression (or in any recursive function which is executed under the control of the PROG expression) a line is printed showing the variable and the value which was assigned to the variable at that time. Since iterative functions use variables to hold temporary results, to count, and so on, the TRACESET option allows one to obtain almost complete information about an iterative function. The same warning about the cost of using the TRACE function is given with respect to the TRACESET function.

By judicious use of the TRACE and TRACESET functions the user can normally determine very closely the point at which an erroneous answer is being generated. In the unlikely event that the error appears to occur within one of the system-defined functions, the user should report such a finding to one of the Computation Center's consultants, showing him the output necessary to support the conclusion.

Tracing is not always as flexible as might be desired. For instance, it might be desirable to trace a certain function only when it is called by some particular function, instead of every time the function is called. Such selective tracing may be accomplished if within the particular calling function, the call to the function to be traced is immediately preceded by a TRACE call and immediately followed by an UNTRACE call. Or the user might wish to actually program the printing of intermediate results. This is quite easily done in LISP since the PRINT function is an identity function. That is, it returns its unchanged argument as its value. Therefore, any form which

appears in the LISP program may be made the argument to PRINT without affecting the result of the program.

To reiterate, we shall state the debugging procedure as a step-wise process.

(1) From a listing with parenthesis counts, check for and correct syntactic errors.

(2) Use TRACE and/or TRACESET to selectively trace the functions which are believed to be in error, and progressively isolate the functions which actually are in error.

(3) Correct the erroneous function(s).

## 8.4 UNDERSTANDING THE UT LISP BACKTRACE

The backtrace is a condensed listing of the content of the system's pushdown stack. It contains a history of the activity of the system up to the point in time at which the backtrace is taken.

Of most interest is the appearance of the names of all functions which were entered but have not yet completed. Often these names appear twice, side-by-side in the backtrace, because in addition to either EVAL or APPLY the function itself put its name on the stack. The user should not be alarmed by this duplication.

Other symbols, particularly EVAL and EVLIS, may appear even though the user did not call these functions, since they are used internally by the LISP interpreter. If the backtrace is taken while a set of function arguments is being evaluated, then the already-evaluated arguments appear on the stack.

## 8.5 PROGRAM DETERMINATION OF ERROR TYPE

Some of the error messages listed in section 8.2 have a code associated with them. This code identifies a particular error type within LISP. Each code is an atom.

Whenever one of these errors occurs, the code associated with it is made the value of the atom ERRORTYPE. By interrogating the value of ERRORTYPE, one can determine the type of the last-occurring error. ERRORTYPE initially has the value NIL. This facility is useful in conjunction with ERRORSET, which allows a program to recover from an error, or when running interactively with the Z control command parameter (see section 2.1). The further course of the computation may be guided by knowing what error occurred.

An additional facility provided by LISP is that of error trap procedures. Each error code atom has a value which is considered to be a form to be evaluated when the error occurs. Initially these atoms are all bound to NIL. If the user, however, binds some other form to the atom then that form is evaluated, but only after any interrupt expression has itself been evaluated (see chapter 9).

# 9. INTERACTIVE USE

The previous chapters have discussed most of the facilities of UT LISP. These facilities are available to both the batch user and the interactive user. This chapter explains some slight peculiarities in the behavior of LISP when used in the interactive mode and an additional facility, the interrupt, which is primarily useful to the interactive user.

## 9.1 INTERACTIVE I/O BEHAVIOR

When the C parameter is specified on the LISP control command, LISP is initialized to operate interactively (conversationally). Three things happen when this is done:

1) SYSIN and SYSOUT are both equated to file TTY and are associated with the same buffer area.

2) The size of that buffer is reduced from 512 words to 10 words and the output line length is set to 70 characters.

3) Printing of top-level input on SYSOUT is disabled.

LISP outputs information only when the buffer for a file is full or whenever for a given file a read operation occurs after a write operation. Item 1 above assures that all generated output will be sent to the terminal before LISP requests its next input line, keeping the proper sequence of input and output. The second change above causes output to be sent to the terminal sooner than if a large buffer were used. This means that what the user sees at his terminal is more in synchrony with what LISP is actually doing at that moment. The third item merely recognizes that if the user is typing his input at a terminal, the input is already visible to him.

The fact that SYSIN and SYSOUT share a buffer in interactive mode does impose some restrictions that do not apply in batch mode:

1) Top-level expressions may not be typed accumulatively. That is, a new one should not be typed until all output generated by the current one has been sent to the terminal.

2) If a data input line contains several S-expressions to be read by several executions of READ, no output functions may be executed until the entire line has been read. For example:

   If your program contains

   ```
   (PROGN (PRINT (READ)) (PRINT (READ)))
   ```

   and on the first input request you type

   ```
   A B
   ```

   then

   ```
   A
   ```

will be printed and LISP will request more input instead of reading B from the first input line.

Note that these comments apply only to terminal input. Interactive LISP programs may manipulate disk files in the same way as batch programs.

3) It is not possible to get a parenthesis count line for terminal input.

## 9.2 INTERRUPTS

An interrupt is a means for suspending the normal execution of a program and performing a task (perhaps independent of that program) in such a way that the suspended program execution can be resumed as if no interruption had occurred. Interrupts are provided in the hardware of many machines. They are usually used to enable processing intermittent, real-time events, such as the occurrence of errors or the input of characters from the keyboard terminals of a timesharing system. The CDC 6000 series computers do not have hardware interrupts, but UT LISP provides for the simulation of up to 12 interrupts. The UT LISP interrupt feature is most useful in interactive execution mode, since the TAURUS timesharing system supplies the mechanism for simulating real-time interrupts (see section 9.2.2). However, UT LISP interrupts can also be effected during batch executions.

When an interrupt occurs, UT LISP preserves the state of the program, evaluates a specified "interrupt expression", and then restores the preserved state. By using DEFINT (see section 9.2.2), the user can define a unique expression to be evaluated for each interrupt. The interrupt expressions may do anything valid in the context in which they are evaluated, including interaction with the user. User interaction is most easily achieved by calling function SYSIN (see section 4.12) within the interrupt expression.

### 9.2.1 Uses for LISP Interrupts

There are three main purposes for using interrupts with a LISP program: (1) to query the program status, (2) to determine the cause and cure of an error, or (3) to effect special control.

If the expression evaluated in response to an interrupt converses with the user, he can determine something about the status of his program. He can execute BACKTRACE (see section 4.12) to see what is on the stack, evaluate variables, look at function definitions, and so on. When the interrupt expression has completed evaluation, LISP continues execution of the interrupted process. Notice, however, that any changes in variable values, etc., which occurred during evaluation of the interrupt expression are effective when the interrupted process resumes.

When an error occurs, its corresponding message is printed. Then, if system variable //ZAP (see section 4.14) is non-NIL, the interrupt corresponding to the value of //ZAP occurs. If the interrupt routine permits doing so, the status of the program may be checked and the cause of the error determined. Then it is sometimes possible to correct the cause of the error and resume execution as if the error had not occurred (see RETFROM, section 9.3). There is no useful general procedure for such recovery; each case requires special attention.

Two interrupts are automatically evoked by actions of the LISP system. Interrupt 3 occurs at the first application of EVAL after a garbage collection occurs. By default, the value associated with interrupt 3 is NIL and no action occurs. If some action is desired, the user may specify an interrupt expression for interrupt 3 by using function DEFINT (section 9.2.2). Interrupt 2 is triggered by use of the function EVALTRAP (see section 9.2.3).

Finally, an interrupt expression may be used to cause some side effect which influences the course of the computation. Such a use could be a potentially powerful tool.

### 9.2.2 Effecting Interrupts

The interrupt expression evaluated when an interrupt occurs is the bound value of the system variable //INT<n>, where <n> is an integer in the range 1 through 12. The default interrupt expressions present in the UT LISP system are shown in the table below.

| Interrupt | Default Interrupt Expression |
|---|---|
| 1-2 | (PROGN (PRINT $$$///// INTERRUPT$) (SYSIN (QUOTE TTY))) |
| 3-5 | NIL |
| 6 | (PROGN (SETQ //INPUT "(INPUT //SYSIN)) (SETQ //MODE "(EVAL . 1))) |
| 7-12 | NIL |

It is possible to lose control of LISP if //MODE or //INPUT get set to bad values. Interrupt 6 may be used to try to recover control.

Users may establish an interrupt expression by SETting the appropriate variable //INT<n> or by executing function DEFINT.

(DEFINT <fixnumber><exp>)

pseudofunction; SUBR

DEFINT binds <exp> to the system variable //INT<fixnumber>, where <fixnumber> must be in the range 1-12. <exp> is subsequently evaluated when interrupt <fixnumber> occurs.

During interactive execution, a LISP interrupt can be caused either by the program itself or by the TAURUS user. During batch execution, only the program can cause an interrupt. A LISP interrupt is evoked under program control by executing the LISP INTERRUPT function.

(INTERRUPT <fixnumber>)

pseudofunction; SUBR

INTERRUPT evokes the simulated interrupt <fixnumber>, which results in the evaluation of the interrupt expression bound

to the system variable //INT<fixnumber>. <fixnumber> must be in the range 1-12 or be NIL. If NIL, interrupt 1 is assumed.

The TAURUS user can cause a LISP interrupt by typing the appropriate TAURUS INTERRUPT command shown below.

| LISP Interrupt | TAURUS Command Long Form | TAURUS Command Short Form |
|---|---|---|
| <n> | <BEL> INTERRUPT=L<n><CR> | <BEL> I=L<n><CR> |
| 6+<n> | <BEL> INTERRUPT=S<n><CR> | <BEL> I=S<n><CR> |

In the TAURUS commands, <n> must be in the range 1-6, <BEL> represents the TAURUS "bell" issued by simultaneously striking the "CNTRL" and "G" keys, and <CR> represents the RETURN key. The "=L<n>" form of the command sets sense light <n>, which corresponds to the LISP interrupt <n> (i.e., interrupts 1-6). The "=S<n>" form of the command sets sense switch <n>, which corresponds to the LISP interrupt 6+<n> (i.e., interrupts 7-12).

EVAL monitors the sense lights and switches during its execution and when one is ON, a simulated interrupt occurs. The response to the TAURUS INTERRUPT command is not immediate, but should be fairly fast when programs are being interpreted. When LISP execution is primarily in compiled code, response may be slow, since response is a function of how frequently EVAL is executed.

### 9.2.3 The Trap Function

(EVALTRAP <fixnumber>)

Pseudofunction; SUBR

EVALTRAP resets an internal counter to <fixnumber>. Each time the interpreter invokes EVAL this counter is decremented by 1. When the counter becomes negative the system resets the counter to 2**59 - 1 (the default value) and causes interrupt 2 to occur. Thus EVALTRAP acts as a bound on the amount of work the LISP system can do before an interrupt occurs. The user can control what occurs when interrupt 2 is triggered by using DEFINT to associate expressions with //INT2.

Possible applications of EVALTRAP include simulating a "time trap", checking for infinite loops or running an interactive program requiring periodic inspection of its progress. EVALTRAP returns its argument as its value.

## 9.3 RETURN FROM NESTED FUNCTION INVOCATIONS

This section is included here because the facilities described are most often useful when the interrupt-on-error capability is activated in interactive mode. These facilities allow one to a) pinpoint the stack entry for a previous invocation of some function, and b) exit directly from that particular invocation

with a value as though all subsequent function invocations were completed. Thus, suppose that on the [n]th invocation of some function, an error interrupt occurs, and the user knows that if no error had occurred the result of the [n - i]th invocation should have been <x>. Then the user can specify to exit directly from the [i]th most recent invocation with value <x>, and the computation will proceed as though the error had not occurred. Two functions are used:

(NTHFNBK <atom> <fixnumber>)

pseudofunction; SUBR

NTHFNBK searches the stack, starting from the current top, for the <fixnumber>th occurrence of the function name <atom>. NTHFNBK returns the stack index of that entry.

(RETFROM <fixnumber> <exp>)

pseudofunction; SUBR

The <fixnumber> is a stack index for some function invocation as found by NTHFNBK. RETFROM causes the stack to be "peeled" back to that point, and then exits the indexed function with the value of <exp> as the value of the function. Note that <exp> is evaluated after the stack has been "peeled" back, in the environment that held when the function was invoked. Also, the effects of TRACE and TRACESET may be affected by a call to RETFROM.

# A. ALPHABETIC INDEX OF UT LISP SYSTEM FUNCTIONS

| Name | Type | Section | Arguments |
|---|---|---|---|
| ABOLISH | SUBR | 5.9 | <filename> |
| ADDR | SUBR | 4.12 | <s> |
| ADDRP | SUBR | 4.12 | <s> |
| ADD1 | SUBR | 4.9 | <number> |
| ADVANCE | SUBR | 5.5 | <boolean> |
| ALIST | SUBR | 4.12 | |
| ALPHAP | SUBR | 4.3 | <litatom1><litatom2> |
| AND | FSUBR | 4.4 | <exp1><exp2> ... <exp[n]> |
| APPEND | SUBR | 4.6 | <list><s> |
| APPLY | SUBR | 4.5 | <function><list> |
| ATOM | SUBR | 4.3 | <s> |
| A+ | SUBR | 4.3 | <s1><s2> |
| A- | SUBR | 4.3 | <s1><s2> |
| BACKTRACE | FSUBR | 4.12 | |
| CALLSYS | SUBR | 7.3 | <filename><function><br><list><lat><boolean> |
| CAR | SUBR | 4.3 | <nats> |
| CDR | SUBR | 4.3 | <nats> |
| CHLEX | SUBR | 5.7 | <character><fixnumber> |
| CLARRAY | SUBR | 4.13 | <litatom> |
| CLEARBUFF | SUBR | 4.10 | |
| CLOSE | SUBR | 5.2 | <filename> |
| COMMENT | FSUBR | 4.3 | <s1><s2> ... <s[n]> |
| COMPILE | SUBR | 6.2 | <lat> |
| COMPRESS | SUBR | 4.10 | <lat><boolean> |
| CONC | FSUBR | 4.6 | <list1><list2> ... <list[n]> |
| COND | FSUBR | 4.5 | (<boolean1><exp> ... <exp>)<br>(<boolean2><exp> ... <exp>) ...<br>(<boolean[n]><exp> ... <exp>) |
| CONS | SUBR | 4.3 | <s1><s2> |
| COPY | SUBR | 4.6 | <s> |
| CP | SUBR | 4.12 | |
| CSR | SUBR | 4.3 | <nats> |
| C...R | EXPR | 4.3 | <nats> |
| DATE | SUBR | 4.12 | |
| DEADSTART | SUBR | 4.12 | |
| DEF | FSUBR | 4.7 | (<litatom1><lat1><s1>)<br>(<litatom2><lat2><s2>) ...<br>(<litatom[n]><lat[n]><s[n]>) |
| DEFF | FSUBR | 4.7 | (<litatom1><lat1><s1>)<br>(<litatom2><lat2><s2>) ...<br>(<litatom[n]><lat[n]><s[n]>) |
| DEFINE | SUBR | 4.7 | ((<litatom1><s1>)<br>(<litatom2><s2>) ...<br>(<litatom[n]><s[n]>)) |
| DEFINT | SUBR | 9.2 | <fixnumber><exp> |
| DEFLIST | SUBR | 4.7 | ((<litatom1><s1>)<br>(<litatom2><s2>) ...<br>(<litatom[n]><s[n]>))<litatom> |

| Name | Type | Section | Arguments |
|---|---|---|---|
| DEFSYS | SUBR | 7.2 | <filename><boolean> |
| DIE | SUBR | 4.11 | <s> |
| DIFFERENCE | SUBR | 4.9 | <number1><number2> |
| DIGIT | SUBR | 4.10 | <s> |
| DISKOUT | SUBR | 7.6 | <fixnumber1><fixnumber2><list> |
| DIVIDE | SUBR | 4.9 | <number1><number2> |
| EFFACE | SUBR | 4.6 | <s><list> |
| ENDFILE | SUBR | 5.9 | <filename> |
| ENDREAD | SUBR | 5.5 | |
| EQ | SUBR | 4.3 | <s1><s2> |
| EQN | SUBR | 4.3 | <s1><s2> |
| EQUAL | SUBR | 4.3 | <s1><s2> |
| ERROR | SUBR | 4.11 | <s> |
| ERRORSET | SUBR | 4.11 | <exp><boolean1><boolean2> |
| EVAL | SUBR | 4.5 | <exp> |
| EVALQUOTE | SUBR | 4.5 | <function><list> |
| EVALTRAP | SUBR | 9.2 | <fixnumber> |
| EVLIS | SUBR | 4.5 | <list> |
| EXIT | SUBR | 4.5 | <litatom><s> |
| EXPLODE | SUBR | 4.10 | <atom> |
| FIX | SUBR | 4.9 | <flnumber> |
| FIXP | SUBR | 4.9 | <number> |
| FLAG | SUBR | 4.7 | <lat><litatom> |
| FLOAT | SUBR | 4.9 | <fixnumber> |
| FLOATP | SUBR | 4.9 | <number> |
| FORTRAN | SUBR | 7.4 | <filename> |
| FQUOTE | FSUBR | 4.3 | <s> |
| FREE | SUBR | 4.12 | |
| FULL | SUBR | 4.12 | |
| FUNCTION | FSUBR | 4.5 | <function> |
| GENSYM | FSUBR | 4.12 | <letter> |
| GET | SUBR | 4.7 | <litatom1><litatom2> |
| GETD | SUBR | 4.7 | <litatom> |
| GETEL | SUBR | 4.13 | <litatom> (<fixnumber1><fixnumber2> ... <fixnumber[n]>) |
| GETPN | SUBR | 4.7 | <litatom> |
| GO | FSUBR | 4.5 | <litatom> |
| GRADP | SUBR | 4.3 | <s1><s2> |
| GREATERP | SUBR | 4.9 | <number1><number2> |
| IMAGEL | SUBR | 4.10 | <atom><boolean> |
| INBIN | SUBR | 5.10 | |
| INPUT | SUBR | 5.4 | <filename> |
| INTERN | SUBR | 4.10 | <fwl> or <litatom> |
| INTERRUPT | SUBR | 9.2 | <fixnumber> |
| ISPACE | SUBR | 5.7 | <number> |
| ITAB | SUBR | 5.7 | <number> |
| LABEL | FORM | 4.5 | <litatom><lambda expression> |
| LAP | SUBR | 6.3 | <s1><s2> |
| LEFTSHIFT | SUBR | 4.9 | <number><fixnumber> |
| LENGTH | SUBR | 4.6 | <s> |
| LESSP | SUBR | 4.9 | <number1><number2> |

| Name | Type | Section | Arguments |
|---|---|---|---|
| LIST | FSUBR | 4.3 | <s1><s2> ... <s[n]> |
| LISTING | SUBR | 4.14 | <s> |
| LITER | SUBR | 4.10 | <s> |
| LOADSYS | SUBR | 7.3 | <filename> |
| LOGAND | FSUBR | 4.9 | <number><number> ... <number> |
| LOGOR | FSUBR | 4.9 | <number><number> ... <number> |
| LOGXOR | FSUBR | 4.9 | <number><number> ... <number> |
| LOOK | SUBR | 4.11 | <fixnumber> |
| MAP | SUBR | 4.8 | <list><fnexp> |
| MAPC | SUBR | 4.8 | <list><fnexp> |
| MAPCAR | SUBR | 4.8 | <list><fnexp> |
| MAPCON | SUBR | 4.8 | <list><fnexp> |
| MAPLIST | SUBR | 4.8 | <list><fnexp> |
| MAX | FSUBR | 4.9 | <number><number> ... <number> |
| MEMBER | SUBR | 4.3 | <s><list> |
| MEMQ | SUBR | 4.3 | <s><list> |
| MIN | FSUBR | 4.9 | <number><number> ... <number> |
| MINUS | SUBR | 4.9 | <number> |
| MINUSP | SUBR | 4.9 | <number> |
| MKARRAY | SUBR | 4.13 | <litatom> (<fixnumber1><fixnumber2> ... <fixnumber[n]>) |
| MKNAM | SUBR | 4.10 | |
| NCONC | SUBR | 4.6 | <list><s> |
| NFORMAT | SUBR | 5.8 | <fixnumber1><fixnumber2> |
| NOT | SUBR | 4.4 | <s> |
| NTH | SUBR | 4.6 | <list><fixnumber> |
| NTHFNBK | SUBR | 9.3 | <atom><fixnumber> |
| NULL | SUBR | 4.3 | <s> |
| NUMBERP | SUBR | 4.3 | <s> |
| NUMOB | SUBR | 4.10 | |
| NUMTOATOM | SUBR | 4.10 | <number> |
| OCTAL | SUBR | 4.9 | <number> |
| ONEP | SUBR | 4.9 | <number> |
| OPCHAR | SUBR | 4.10 | <s> |
| OPEN | SUBR | 5.2 | <filename> ((<c1> . <v1>) (<c2> . <v2>) ... (<c[n]> . <v[n]>)) |
| OPENFILES | SUBR | 5.2 | |
| OR | FSUBR | 4.4 | <exp1><exp2> ... <exp[n]> |
| OSPACE | SUBR | 5.8 | <fixnumber> |
| OTAB | SUBR | 5.8 | <fixnumber> |
| OUTBIN | SUBR | 5.10 | <fw1> |
| OUTPUT | SUBR | 5.3 | <filename><s><boolean> |
| OUTPUT1 | SUBR | 5.3 | <filename><s><boolean> |
| OVERLAY | SUBR | 7.3 | <filename><function><list><lat> |
| PACK | SUBR | 4.10 | <character> |
| PAIR | SUBR | 4.6 | <list1><list2> |
| PLUS | FSUBR | 4.9 | <number><number> ... <number> |
| PP | SUBR | 4.12 | |
| PPRINT | SUBR | 5.3 | <s><boolean> |

| Name | Type | Section | Arguments |
|---|---|---|---|
| PRINT | SUBR | 5.3 | <s><boolean> |
| PRIN1 | SUBR | 5.3 | <s><boolean> |
| PROG | FSUBR | 4.5 | <lat><s1><s2> ... <s[n]> |
| PROGN | FSUBR | 4.5 | <exp1><exp2> ... <exp[n]> |
| PROG2 | SUBR | 4.5 | <exp1><exp2> |
| PROP | SUBR | 4.7 | <litatom1><litatom2><fnexp> |
| PUT | SUBR | 4.7 | <litatom1><litatom2><s> |
| PUTD | SUBR | 4.7 | <litatom><s> |
| QKEDIT | SUBR | 4.7 | <litatom><s1><s2> |
| QUOTE | FSUBR | 4.3 | <s> |
| QUOTIENT | SUBR | 4.9 | <number1><number2> |
| RANDOM | SUBR | 4.9 | <number> |
| RANIN | SUBR | 5.6 | <filename><address> |
| RANOUT | SUBR | 5.6 | <filename><s><boolean> |
| RDS | SUBR | 5.1 | <filename> |
| READ | SUBR | 5.4 | |
| READCH | SUBR | 5.5 | <boolean> |
| READLAP | FSUBR | 6.4 | <filename><lat> |
| RECIP | SUBR | 4.9 | <number> |
| RECLAIM | SUBR | 4.12 | |
| REMAINDER | SUBR | 4.9 | <number1><number2> |
| REMFLAG | SUBR | 4.7 | <lat><litatom> |
| REMOB | SUBR | 4.12 | <litatom><boolean> |
| REMPROP | SUBR | 4.7 | <litatom1><litatom2> |
| RETFROM | SUBR | 9.3 | <fixnumber><exp> |
| RETURN | SUBR | 4.5 | <s> |
| REVERSE | SUBR | 4.6 | <list> |
| REVERSIP | SUBR | 4.6 | <list> |
| REWIND | SUBR | 5.9 | <filename> |
| RIN | SUBR | 5.6 | <filename><fixnumber> |
| ROUT | SUBR | 5.6 | <filename><s><boolean> |
| RPLACA | SUBR | 4.3 | <nats><s> |
| RPLACD | SUBR | 4.3 | <nats><s> |
| RPLACS | SUBR | 4.3 | <nats><s> |
| SASSOC | SUBR | 4.8 | <s><list><fnexp> |
| SEARCH | SUBR | 4.8 | <list><fnexp1><fnexp2><fnexp3> |
| SECTORS | SUBR | 4.12 | |
| SELECT | FSUBR | 4.5 | <exp><br>(<exp[1,1]><exp[1,2]> ...<br><exp[1,n1]>)<br>(<exp[2,1]><exp[2,2]> ...<br><exp[2,n2]>) ...<br>(<exp[m,1]><exp[m,2]> ...<br><exp[m,n[m]]>) <exp[m+1]> |
| SET | SUBR | 4.3 | <atom><exp> |
| SETEL | SUBR | 4.13 | <litatom><br>(<fixnumber1><fixnumber2> ...<br><fixnumber[n]>)<s> |
| SETQ | FSUBR | 4.3 | <atom><exp> |
| STARTREAD | SUBR | 5.5 | |
| SUBLIS | SUBR | 4.6 | ((<s1> . <s2>)<br>(<s3> . <s4>) ....<br>(<s[n-1]> . <s[n]>))<s> |

| Name | Type | Section | Arguments |
|---|---|---|---|
| SUBST | SUBR | 4.6 | <s1><s2><s3> |
| SUB1 | SUBR | 4.9 | <number> |
| SYSIN | SUBR | 5.1 | <filename><character> |
| SYSOUT | SUBR | 5.1 | <filename> |
| TEMPUS | SUBR | 4.12 | |
| TEREAD | SUBR | 5.5 | |
| TERPRI | SUBR | 5.3 | |
| TIME | SUBR | 4.12 | |
| TIMES | FSUBR | 4.9 | <number><number> ... <number> |
| TM | SUBR | 4.12 | |
| TMLEFT | SUBR | 4.12 | |
| TRACE | SUBR | 4.11 | <lat> |
| TRACESET | SUBR | 4.11 | <lat> |
| TTYCOPY | SUBR | 5.3 | <filename> |
| UNPACK | SUBR | 4.10 | <fw> |
| UNTRACE | SUBR | 4.11 | <lat> |
| UNTRACESET | SUBR | 4.11 | <lat> |
| WRITE | SUBR | 5.3 | <s><boolean> |
| WRS | SUBR | 5.1 | <filename> |
| ZEROP | SUBR | 4.9 | <number> |
| + | FSUBR | 4.9 | <number><number> ... <number> |
| - | SUBR | 4.9 | <number1><number2> |
| * | FSUBR | 4.9 | <number><number> ... <number> |
| / | SUBR | 4.9 | <number1><number2> |
| = | SUBR | 4.3 | <s1><s2> |
| < | SUBR | 4.9 | <number1><number2> |
| > | SUBR | 4.9 | <number1><number2> |

# B. LISP SUBSYSTEMS

A LISP subsystem is a "canned" set of function and/or constant definitions which constitutes an extension of the facilities provided by the normal LISP system. In particular, a subsystem may be used to establish a set of application-oriented primitives which can then be used to construct user programs in some particular application area. In form, a subsystem is a single file containing a block of text in normal LISP input format (without operating system control commands).

When LISP begins execution, each subsystem named on the control command is read by LISP. Each expression of a subsystem is evaluated as if the text were normal input, except that none of the system output usually generated by LISP is produced. Each file is interpreted as a sequence of expressions for EVAL. Of course the evaluation mode within the file can be changed by binding //MODE (see section 4.14) to an appropriate dotted pair. When all subsystems have been so processed, LISP begins reading user-supplied input in the usual way.

In this way a subsystem defines a collection of facilities which can be used by user programs subsequently input. LISP overlays provide a similar capability. However, an overlay represents a snapshot of the results of some evaluation activity, and always executes with the same global options and field length for which it was defined. A subsystem, on the other hand, can be used with any combination of control command parameters and in any field length. It is important to recognize that subsystem use is costly because of the extra I/O and evaluation time required to process a subsystem. On the other hand, an overlay is much less expensive since it is an absolute memory image which needs only to be loaded.

Four subsystems are currently available at UT:

LAP - the LISP assembler
LCOMP - the LISP compiler
GRASP - a graph-processing extension to LISP
LISPED - an interactive internal function editor

## C. SYSTEM VARIABLES

The system variables are literal atoms which exist in the standard LISP system and which are defined at system definition time to have the values shown below.

| Variable | Section | Value |
|---|---|---|
| ANDSIGN | | ^ |
| BLANK | | (atom whose name is the blank character) |
| COLON | | : |
| COMMA | | , |
| DARROW | | ! |
| DASH | | - |
| DOLLAR | | $ |
| EOF | | $EOF$ |
| EOR | | $EOR$ |
| EQSIGN | | = |
| EQUIV | | " |
| ERRA0 | 8.2.6 | NIL |
| ERRA1 | 8.2.6 | NIL |
| ERRA2 | 8.2.6 | NIL |
| ERRA3 | 8.2.5 | NIL |
| ERRA4 | 8.2.5 | NIL |
| ERRA5 | 8.2.5 | NIL |
| ERRA6 | 8.2.5 | NIL |
| ERRA7 | 8.2.5 | NIL |
| ERRA8 | 8.2.5 | NIL |
| ERRC1 | 8.2.6 | NIL |
| ERRGC2 | 8.2.4 | NIL |
| ERRGC3 | 8.2.4 | NIL |
| ERRI1 | 8.2.1 | NIL |
| ERRI2 | 8.2.3 | NIL |
| ERRI3 | 8.2.3 | NIL |
| ERRORTYPE | 8.5 | NIL |
| ERRP1 | 8.2.6 | NIL |
| ERRP2 | 8.2.6 | NIL |
| ERRR1 | 8.2.3 | NIL |
| F | | NIL |
| FORFIRST | 7.4 | (location of origin of FORTRAN overlays) |
| GARLIST | | NIL |
| GREATER | | > |
| GREATEREQ | | ≥ |
| LAPUNCH | 6.3.6 | NIL |
| LBRACK | | [ |
| LESS | | < |
| LESSEQ | | ≤ |
| LOADFLAG | 6.3.6 | NIL |
| LPAR | | ( |

| Variable | Section | Value |
|---|---|---|
| NEQUAL | | # |
| NIL | | NIL |
| NOTSIGN | | ' |
| OBLIST | 3.3.7 | (bucket-sorted list of all literal atoms) |
| ORSIGN | | & |
| PERIOD | | . |
| PLUSS | | + |
| PREVIOUS | 4.14 | NIL |
| PRNTFLAG | 6.2.1 | NIL |
| RARROW | | , |
| RBRACK | | ] |
| RPAR | | ) |
| SEMICOLON | | ; |
| SLASH | | / |
| STAR | | * |
| T | | *T* |
| UPARROW | | ^ |
| VIRFN | 7.6 | VIRFN |
| + | | + |
| - | | - |
| * | | * |
| *T* | | *T* |
| / | | / |
| //CODEMAX | 4.14 | 0Q |
| //CODEMIN | 4.14 | 0Q |
| //EXPERT | 4.14 | NIL |
| //FATAL | 4.14 | NIL |
| //FL | 4.14 | 18944 (i.e., 45000 octal) |
| //FRS | 4.14 | |
| //FUS | 4.14 | |
| //GC | 4.14 | NIL |
| //GFR | 4.14 | 0 |
| //GFU | 4.14 | 0 |
| //INPUT | 4.14 | (INPUT (QUOTE SYSIN)) |
| //INT1 | 9.2.2 | (PROGN (PRINT $$$///// INTERRUPT$) (SYSIN (QUOTE TTY))) |
| //INT2 | 9.2.2, 9.2.3 | (PROGN (PRINT $$$///// INTERRUPT$) (SYSIN (QUOTE TTY))) |
| //INT3 | 9.2.1, 9.2.2 | NIL |
| //INT4 | 9.2.2 | NIL |
| //INT5 | 9.2.2 | NIL |
| //INT6 | 9.2.2 | (PROGN (SETQ //INPUT (QUOTE (INPUT //SYSIN)) (SETQ //MODE (QUOTE (EVAL . 1)))) |
| //INT7 | 9.2.2 | NIL |
| //INT8 | 9.2.2 | NIL |
| //INT9 | 9.2.2 | NIL |

| Variable | Section | Value |
|---|---|---|
| //INT10 | 9.2.2 | NIL |
| //INT11 | 9.2.2 | NIL |
| //INT12 | 9.2.2 | NIL |
| //MODE | 4.14, 8.2.6 | (EVAL . 1) |
| //NFR | 4.14 | 0 |
| //NFU | 4.14 | 0 |
| //OUTPUTA | 4.14 | (LAMBDA (=====/////) (OR (AND (NOT (ATOM =====/////)) (EQ (CAR =====/////) (QUOTE DEFINE))) (OUTPUT //SYSOUT =====/////))) |
| //OUTPUTB | 4.14 | (LAMBDA (=====/////) (OUTPUT //SYSOUT =====/////)) |
| //PCSR | 4.14 | NIL |
| //PLEVEL | 4.14 | 65536 |
| //PLIMIT | 4.14 | 65536 |
| //RDS | 4.14, 5.1.2, 5.1.3 | SYSIN |
| //SAVING | 4.14 | NIL |
| //STS | 4.14 | 1000 |
| //SYSIN | 4.14, 5.1.1, 5.1.3 | SYSIN |
| //SYSOUT | 4.14, 5.1.1, 5.1.3 | SYSOUT |
| //TIMING | 4.14 | NIL |
| //TPLEVEL | 4.14 | 4 |
| //TPLIMIT | 4.14 | 4 |
| //WRS | 4.14, 5.1.2, 5.1.3 | SYSOUT |
| //ZAP | 4.14, 9.2.1 | NIL |
| ( | | ( |
| ) | | ) |
| $ | | $ |
| $EOF$ | | $EOF$ |
| = | | = |
| <blank> | | <blank> |
| , | | , |
| . | | . |
| " | | " |
| [ | | [ |
| ] | | ] |
| : | | : |
| # | | # |
| & | | & |
| % | | % |
| ^ | | ^ |
| ! | | ! |
| < | | < |
| > | | > |
| @ | | @ |
| ? | | ? |
| \ | | \ |
| ; | | ; |

# D. COMPARISON OF UT LISP WITH MIT LISP 1.5

| MIT LISP* | UT LISP |
|---|---|
| | ABOLISH |
| | ADDR |
| | ADDRP |
| ADD1 | ADD1 |
| ADVANCE | ADVANCE |
| | ALIST |
| | ALPHAP |
| AND | AND |
| APPEND | APPEND |
| APPLY | APPLY |
| ARRAY | |
| ATOM | ATOM |
| ATTRIB | |
| | A+ |
| | A- |
| | BACKTRACE |
| | CALLSYS |
| CAR | CAR |
| | CA...R |
| CDR | CDR |
| | CD...R |
| | CHLEX |
| | CLARRAY |
| CLEARBUFF | CLEARBUFF |
| | CLOSE |
| COMMON | |
| | COMMENT |
| COMPILE | COMPILE |
| | COMPRESS |
| CONC | CONC |
| COND | COND |
| CONS | CONS |
| COPY | COPY |
| COUNT | |
| | CP |
| CP1 | |
| CSET | |
| CSETQ | |
| | CSR |
| | CS...R |
| DASH | |
| | DATE |
| | DEADSTART |
| | DEF |
| | DEFF |
| DEFINE | DEFINE |
| | DEFINT |

---

* As defined in J. McCarthy et al., LISP 1.5 Programmer's Manual (M.I.T. Press, Cambridge, Mass.) 1962.

| MIT LIST | UT LIST |
|---|---|
| | DEFLIST |
| DEFLIST | DEFSYS |
| | DIE |
| DIFFERENCE | DIFFERENCE |
| DIGIT | DIGIT |
| | DISKOUT |
| DIVIDE | DIVIDE |
| DUMP | |
| | |
| EFFACE | EFFACE |
| | ENDFILE |
| ENDREAD | ENDREAD |
| EQ | EQ |
| | EQN |
| EQUAL | EQUAL |
| ERROR | ERROR |
| ERROR1 | |
| ERRORSET | ERRORSET |
| EVAL | EVAL |
| EVALQUOTE | EVALQUOTE |
| | EVALTRAP |
| EVLIS | EVLIS |
| EXCISE | |
| | EXIT |
| | EXPLODE |
| EXPT | |
| | |
| | FIX |
| FIXP | FIXP |
| FLAG | FLAG |
| | FLOAT |
| FLOATP | FLOATP |
| | FORTRAN |
| | FQUOTE |
| | FREE |
| | FULL |
| FUNCTION | FUNCTION |
| | |
| GENSYM | GENSYM |
| GET | GET |
| | GETD |
| | GETEL |
| | GETPN |
| GO | GO |
| | GRADP |
| GREATERP | GREATERP |
| | |
| | IMAGEL |
| | INBIN |
| | INPUT |
| INTERN | INTERN |
| | INTERRUPT |
| | ISPACE |
| | ITAB |
| | |
| LABEL | |
| LAP | LAP |
| LEFTSHIFT | LEFTSHIFT |
| LENGTH | LENGTH |
| LESSP | LESSP |

| MIT LISP | UT LISP |
|---|---|
| | LIST |
| | LISTING |
| LITER | LITER |
| LOAD | |
| | LOADSYS |
| LOGAND | LOGAND |
| LOGOR | LOGOR |
| LOGXOR | LOGXOR |
| | LOOK |
| MAP | MAP |
| | MAPC |
| | MAPCAR |
| MAPCON | MAPCON |
| MAPLIST | MAPLIST |
| MAX | MAX |
| MEMBER | MEMBER |
| | MEMQ |
| MIN | MIN |
| MINUS | MINUS |
| MINUSP | MINUSP |
| | MKARRAY |
| MKNAM | MKNAM |
| NCONC | NCONC |
| | NFORMAT |
| NOT | NOT |
| | NTH |
| | NTHFNBK |
| NULL | NULL |
| NUMBERP | NUMBERP |
| NUMOB | NUMOB |
| | NUMTOATOM |
| | OCTAL |
| ONEP | ONEP |
| OPCHAR | OPCHAR |
| OPDEFINE | |
| | OPEN |
| | OPENFILES |
| OR | OR |
| | OSPACE |
| | OTAB |
| | OUTBIN |
| | OUTPUT |
| | OUTPUT1 |
| | OVERLAY |
| PACK | PACK |
| PAIR | PAIR |
| PAUSE | |
| PLB | |
| PLUS | PLUS |
| | PP |
| | PPRINT |
| PRINT | PRINT |
| PRINTPROP | |
| PRIN1 | PRIN1 |
| PROG | PROG |
| | PROGN |
| PROG2 | PROG2 |
| PROP | PROP |

| MIT LISP | UT LISP |
|---|---|
| PUNCH | |
| PUNCHDEF | |
| PUNCHLAP | |
| | PUT |
| | PUTD |
| | QKEDIT |
| QUOTE | QUOTE |
| QUOTIENT | QUOTIENT |
| | RANDOM |
| | RANIN |
| | RANOUT |
| | RDS |
| READ | READ |
| | READCH |
| READLAP | READLAP |
| RECIP | RECIP |
| RECLAIM | RECLAIM |
| REMAINDER | REMAINDER |
| REMFLAG | REMFLAG |
| REMOB | REMOB |
| REMPROP | REMPROP |
| | RETFROM |
| RETURN | RETURN |
| REVERSE | REVERSE |
| | REVERSIP |
| | REWIND |
| | RIN |
| | ROUT |
| RPLACA | RPLACA |
| RPLACD | RPLACD |
| | RPLACS |
| SASSOC | SASSOC |
| SEARCH | SEARCH |
| | SECTORS |
| SELECT | SELECT |
| SET | SET |
| | SETEL |
| SETQ | SETQ |
| SPEAK | |
| SPECIAL | |
| STARTREAD | STARTREAD |
| SUBLIS | SUBLIS |
| SUBST | SUBST |
| SUB1 | SUB1 |
| | SYSIN |
| | SYSOUT |
| | TEMPUS |
| TEMPUS-FUGIT | |
| | TEREAD |
| TERPRI | TERPRI |
| | TIME |
| TIMES | TIMES |
| | TM |
| | TMLEFT |
| TRACE | TRACE |
| TRACESET | TRACESET |
| | TTYCOPY |

| MIT LISP | UT LISP |
|---|---|
| UNCOMMON | |
| UNCOUNT | |
| UNPACK | UNPACK |
| UNSPECIAL | |
| UNTRACE | UNTRACE |
| UNTRACESET | UNTRACESET |
| | WRITE |
| | WRS |
| ZEROP | ZEROP |
| | + |
| | - |
| | * |
| | / |
| | = |
| | < |
| | > |