L I S P / M T S

User's Guide

September 1974

Department of Computer Science

UNIVERSITY OF BRITISH COLUMBIA

07-23-76

by Bruce Wilcox and Carole Hafner

Table of Contents

I. Introduction

Welcome to the wonderful world of LISP/MTS. LISP is not like any other programming language. It combines a very simple syntactic structure with an extremely powerful and flexible semantic structure. This combination of characteristics puts a great burden on the programmer to use the language carefully. You should think of learning LISP as an adventure in the use of computers, and an exercise in logical thinking. Although you may have difficulty with the language at first, you will probably find that once you are accustomed to LISP, other programming languages will seem very cumbersome and restrictive.

In designing LISP/MTS we have attempted to embody the logical power of LISP in a language economical enough to be useful to many people. We have also added many of the user options, input/output capabilities, and de-bugging features that programmers expect to find in any programming language.

Throughout this manual, we have used mnemonics to represent LISP elements in concise representations of the formats of basic LISP operations. A, A1, A2 represent atoms; N, N1, N2 represent numeric atoms; L, L1, L2 represent lists. S, S1, S2 represent any LISP structure, LA, LA1, LA2 represent lists whose elements are atoms; and FN, FN1, FN2 represent function specifications. By S1 . . . Sn we mean that any number of expressions of that type may be given, by <S> we mean that an expression of that type is optional, and by <A, LA> we mean that the user has a choice of one or the other.

Good Luck. We hope you will enjoy using LISP/MTS. Any comments, questions, or bugs should be reported to the authors at 2028 Mental Health Research Institute, Ann Arbor, Mich. Tel: 313-764-4220.

BRUCE WILCOX CAROLE HAFNER

Note: For a formal definition of the original LISP language, see McCarthy et. al., LISP 1.5 PROGRAMMERS GUIDE, M. I. T. Press, 1962.

The development and implementation of LISP/MTS was supported in part by National Science Foundation Grant Number GJ-31339X.

This document was converted into Format source by Mark DuMont and Vincent Manis, UBC Department of Computer Science. Furthur revision and modification was done by Paul Friedman, Wayne Hall, David McDonald, and Jim Davidson.

II. The LISP Language

A. Atoms, Buffers, and Arrays

The primitive data structures of LISP, called ATOMS, are similar in form to variables in other languages.

1. PNAME of an atom

Atoms are created implicitly and referenced through their PNAMEs, or print names. The PNAME of an atom may be any character string up to 255 characters long.

When an atom name, say BOOK, first appears in the input stream, an atomic structure with the PNAME "BOOK" is automatically created. Any future references to the atom BOOK will reference the same structure. The system OBJECT LIST maintains pointers to all atomic structures, and each atomic string which appears in the input stream is checked against this list.

2. Types of atoms

There are two types of atoms in LISP, literal atoms and numeric atoms. When an atom name appears in the input stream, the form of the name, and the current input number base determine the type of the atom.

If the input number base is 10 (the default case), then FORTRAN type integers and floating point numbers will be treated as decimal numbers, and will become numeric atoms. All other character strings will become literal atoms.

If the input number base is 16 (the user may change the number base by calling the STATUS function), FORTRAN type floating point numbers will still be treated as decimal numbers, and will become numeric atoms. However, any character string beginning with a decimal digit (0 - 9) and containing only hexadecimal digits (0 - 9, A - F) will be treated as a hexadecimal number, and will become a numeric atom with the value of that hexadecimal number.

If the input number base is 0, then all character strings will be interpreted as literal atom names, and no numeric atoms will be created.

Unlike literal atoms, numeric atoms are not stored on the OBLIST; instead a new atom is created each time a number appears in the input stream or when a new value is calculated. Thus, two occurrences of the number 17 will produce references to two distinct structures.

Note -- The interpreter recognizes two numbers as being EQUAL if their values are equal; they will also be EQ for all functions which use EQ tests, ie MEMQ, DELQ, etc.

3. Value of an atom

Atoms can have VALUEs, which may be any LISP structure. The VALUE of a literal atom is undefined until a value is given to it. All numeric atoms, by convention, have themselves as their VALUEs.

4. Special atoms

There are several special atoms in LISP, with pre-defined VALUEs. One is NIL , used throughout the system to indicate a null list, or a truth value of false. The VALUE of NIL is NIL. Another special atom is the atom T, used throughout the system to indicate a truth value of true. The VALUE of T is T.

Although the user can change the value of any atom, in general he should not alter the VALUEs of numeric atoms.

The VALUE of NIL must always remain NIL.

The pre-defined atoms of LISP, (and their general significance) is as follows:

```
NIL (Program Logic) = NIL
T (Program Logic) = T
LISPIN (Input/Output) = (Input Buffer . SCARDS)
LISPOUT (Input/Output) = (Output Buffer . SPRINT)
ERRIN (Input/Output) = (Error Input Buffer . GUSER)
ERROUT (Input/Output) = (Error Output Buffer . SERCOM)
*ERR* (Error Processing) = (DUMP)
*ATTN* (Error Processing) = (DUMP)
*PGNT* (Error Processing) = (DUMP)
*UNDEF* (VALUE of undefined atoms) = error 16 if EVALed
*FNS* (list of DEFUN'd functions) = NIL
All numeric atoms = themselves
```

5. Property lists

Besides a VALUE, an atom can have any number of properties, and each property has a property-value. For example, the atom BOOK may have a property COLOR with property-value BLUE, and a property PAGES with property-value 367. The name of a property is referred to as the property indicator, or IND, and the property-value is referred to as the PVAL.

Associated with each atom is a property-list (PLIST) of indicators and values. If an atom has no properties, then its PLIST is NIL.

The property list of the atom NIL is NIL, and may not be altered. Thus, NIL is always guaranteed to have a NIL value and a NIL PLIST.

Numeric atoms may not have property lists.

6. Buffers

LISP/MTS supports a data type called BUFFERS. Although buffers are not truly atoms (they may not be given VALUEs), they are like atoms in that they have PNAMEs. The PNAME of a buffer is the current contents of the buffer. The PNAMEs of atoms and list representations of LISP structures can be placed in a buffer by calling the system print functions. New atoms can be created whose PNAMEs are the contents of a buffer by calling the READ function. All input/output in the system takes place by printing the contents of a buffer on an MTS device, and by reading a record from an MTS device into a buffer. Buffer contents can be compared and translated by system functions.

Whenever a buffer is passed as an argument to a function, it is actually a buffer pointer structure (called an IOARG) which is passed, rather than the buffer itself. A full description of buffers may be found in the Section on Input/Output in LISP/MTS.

7. Arrays

LISP/MTS also supports arrays, where the value of an array element can be any LISP structure. For a description of the definition and use of arrays, see the DEFINE function.

III. Running the LISP Interpreter

LISP is an interpretive language. The system will read one S-expression from its input stream, evaluate it, and print out the value computed, then read another S-expression, etc. Since the top-level controller calls READ to get an S-expression, EVAL to evaluate it, and PRINT to print out the result, the top level function of LISP is often referred to as a READ-EVAL-PRINT loop.

A. The PAR= Run Field

LISP, like many other MTS programs, accepts various control parameters via the PAR= field of the $RUN command. The keyword parameters may appear in any order, and there may be any number of keywords given, e.g. "PAR=FCS=3,PDS=2,MAX=8". The keyword parameters recognized by LISP, and their significance are described below.

1. PAR=FCS= Indicates the number of pages of initial freespace. Default value is 3 pages.

2. PAR=MAX= Indicates the limit on the number of pages of freespace which will be allocated by the system. If this limit is allocated and more space is needed, the user will be prompted in interactive mode; and execution will be terminated in batch mode. Default value is 15 pages.

3. PAR=ERR= Indicates the initial status of interrupt traps.
   0 = program and attention interrupt traps enabled.
   1 = attention interrupt trap disabled.
   2 = program interrupt trap disabled.
   4 = both traps disabled.
   Default value is 0.

4. PAR=GC#= Number of cells of freespace which must be reclaimed during a garbage collection in order to suppress allocation of more space. initially set to 500.

5. PAR=INT= Allows numbers to be made common and placed on the OBLIST. All positive numbers less than this number will be made unique. initial value is 0.

6. PAR=PDS= Sets the initial number of pages of Stack space. Lisp/MTS will ask the user for confirmation of Stack extents beyond this limit in interactive mode. Batch runs will stop after this limit is reached. The default is 1 page.

7. PAR=OBJ= Indicates the number of hash buckets for the literal atom OBJECT LIST. The greater the number of buckets, the faster the resolution of atomic references should be. An odd number is recommended. Default is 69.

B. Input to LISP

Input to LISP is free format, with blanks, commas, periods, parentheses, and ends-of-line acting as separators. Any time a separator appears, it may be surrounded by any number of blanks. Extra right parentheses may be inserted at the beginning or the end of a top-level form, and they will be ignored. For example:) (A B C D)))) = (A B C D) at the top level.

If a semi-colon (;) appears anywhere in an input line, the system will ignore everything else that appears in the line, and will skip to the next line. Thus, the semi-colon is equivalent to an end-of-line. This allows the user to put comments in his input file without the expense of making an atom from every word.

***Warning: The semi-colon is an MTS carriage control character which will cause a line printer to skip to a new page if it is the first character in an output line. At the present time this warning does not seem to apply to MTS at UBC, but users should take note anyway.

Note: An exception is made to the treatment of the period as a separator when it occurs in a legal floating-point number. In that case, the period will be interpreted as part of the number. To make a dotted-pair of two numbers, merely surround the period with blanks. For example, (123.456) is a list of a single numeric atom, while (123 . 456) is a dotted-pair of two integers.

In order to allow the incorporation of separator characters into atom PNAMEs, LISP/MTS defines a special input convention. If a double-quote character (") occurs at the beginning and the end of an atom name, then all characters which occur between the double-quotes will be treated as the PNAME of a single atom. The closing double-quotes must be part of the same input line as the opening double-quotes, and the double-quotes will not be part of the PNAME of the atom. For example, if the input stream contains the atom name:

"AB CD.EF"

an atom with the PNAME: AB CD.EF will be created.

If two double-quotes in a row appear within a double-quoted string, they will be interpreted as a literal double-quote. For example, if "ABC""DE" is read in, the literal atom ABC"DE will be created.

Double-quotes which appear strictly within an atom name have no special significance, and are treated like any other character. If two double- quotes appear at the beginning of an atom name, however, this will generate a syntax error.

To insure balancing of parentheses, the characters < and > act as super parentheses. Upon reading a right super parenthesis (a >), enough right parentheses will be added to balence the s-expression begun with the most recent left super parenthesis. If there is no left super parenthesis, then enough right parentheses are added to finish off the entire expression. Extra right super brackets are ignored. A maximum of 100 pairs of super parentheses are allowed. For example

```
<((((A B>        is read as          (((((A B)))))

(COND <<NULL (CDR X>                 (COND ((NULL (CDR X))
        (CAR (CONS X X>    is read as        (CAR (CONS X X)))
      <T (CAR (CADR X>)                     (T (CAR (CADR X))))
      '((((A B>                      (QUOTE ((((A B))))))
```

C. Operation of EVAL

Evaluation of LISP expressions is done by the function EVAL. When LISP reads a form and sends it to EVAL, the first thing EVAL does is check to see if the form is a single atom. If so, then the value of the form is the VALUE of the atom.

If the form is not an atom, it must be a list. The first element, or the CAR of the list specifies a function to be called. The remaining elements of the list, or the CDR, represent the arguments of the function. If the CAR of the form is an atom, then LISP interprets it as the name of a function, and calls that function (We will see later that there are ways of invoking functions other than a direct call). For example, if the form read by LISP is (ADD X Y), then the function ADD will be called with the VALUE of X as its first argument, and the VALUE of Y as its second argument.

Notice that, as in other languages, it is not the name of the argument which is passed to the function, but its value. For this reason, we refer to the elements which actually appear in the form as argument-designators, and reserve the term "argument" for the values which are actually passed to the function.

Since EVAL calls itself in order to determine the values of the argument-designators, the argument-designators do not have to be atoms, but can be any LISP form which will evaluate to the desired argument. For example, if the VALUE of X is 2 and the VALUE of Y is 3, then EVALing the form (ADD X (ADD Y 1) will cause the function ADD to be invoked twice - the first time with arguments 3 and 1, and the final time with arguments 2 and 4. Naturally, the VALUEs of X and Y are not altered by this operation.

There are a number of built-in LISP functions which are invoked by a direct call as described above. In addition, the user can define new functions by composing these built-in functions in various ways, and then the user-defined functions can also be invoked by name.

D. Output and Termination

Whenever a LISP form is EVALed, a resulting value is returned. When the system reads and EVALs a form, it then prints out its (top-level) value before reading the next form. When we say only the top-level value is printed, this means that the evaluation of arguments, which may involve intermediate function calls, does not cause anything to be printed.

For example, if a user types in the form: (ADD X (ADD Y 1)) where the VALUE of X is 2 and the VALUE of Y is 3, the system will EVAL this entire expression and print the resulting value: 6.

Evaluating the form (STOP) at any level will terminate execution of LISP. Evaluating the form (MTS) will cause a return to MTS from which the user may restart.

IV. Basic LISP Functions

1. (QUOTE S)

It is important to remember that when a LISP form appears as an argument in a function call, this signifies that the value of the form is to be the argument of the function. However, many times LISP users wish to specify directly what an argument to a function should be. In order to facilitate this process, the function QUOTE is available.

The value of (QUOTE A) is the atom A. The value of (QUOTE (CAR (A B C))) is the list (CAR (A B C)).

If a user enters (CONS X Y) from the input stream, the system will call the function CONS with the respective VALUEs of X and Y as arguments. If the user enters (QUOTE (CONS X Y)), the system will merely type back (CONS X Y), since that structure is the value of the input form. If the user enters (CONS (QUOTE X) (QUOTE Y)), the system will execute CONS, but its arguments will be the atoms X and Y rather than their respective VALUEs. To make QUOTEing more convenient, a shorter notation for QUOTE is defined in the system. This is the ' character.

'A is equivalent to (QUOTE A). '(A (B C) D) is equivalent to (QUOTE (A (B C) D)).

A. Basic LISP Predicates

1. (ATOM S)

returns T if its argument is an atom, NIL otherwise.

```
Ex: (ATOM 'A) = T
(ATOM '(A B C)) = NIL
```

2. (NOT S)

returns T if its argument is NIL, NIL otherwise.

```
Ex: (NOT (CAR '(A NIL B))) = NIL
(NOT (CAR (CDR '(A NIL B)))) = T
```

3. (NULL S)

Same as (NOT S); returns T if its argument is NIL, and NIL otherwise.

4. (EQUAL S1 S2)

returns T if its arguments have the same LISP structure. NIL otherwise.

```
Ex. (EQUAL '(A B C) '(A A B C)) = NIL
(EQUAL '(A B C) (CDR '(A A B C))) = T
(EQUAL 8 (TIMES 2 4)) = T
```

5. (EQ S1 S2)

returns T if its arguments are the same LISP structure. NIL otherwise.

Numeric atoms are exceptions in that there values are compared instead of their address.

Since there are frequently multiple structures which represent the same S-expression, not every pair of elements which are EQUAL are EQ. EQ is almost always used with atomic arguments, since there is only one copy of each atomic name on the OBJECT LIST.

```
Ex: (EQ 'A 'A) = T
(EQ '(A B) '(A B)) = NIL
```

6. (NEQ S1 S2)
returns T if its arguments are not EQ, NIL otherwise. This function is equivalent to
(NOT (EQ S1 S2))
See EQ above.

7. (EQNAME A1 A2)
returns T if its arguments are literal atoms or buffer atoms which have the same PNAME. NIL otherwise.

EQNAME will be equivalent to EQ for normal atoms which are on the OBJECT LIST. However, for BUFFER atoms (see Section on I/O), and atoms created by GENSYM, EQNAME provides a new and useful function.

Ex: (EQNAME 'TEST 'TEST) = T
(EQNAME 'ANINPUTLINE IOARG) = T if the buffer associated with IOARG has as its contents "ANINPUTLINE".

8. (NUMBERP A)
returns T if its argument is a numeric atom NIL otherwise.

Ex: (NUMBERP 3) = T

9. (SORTP A1 A2)
returns T if the PNAME of its first argument is less than or equal to its second argument in standard EBCDIC collating sequence. NIL otherwise. A1 and A2 must be literal atoms or IOARGs.

Ex: (SORTP 'ABC 'ABB) = NIL
(SORTP 'ABB 'ABB) = T
(SORTP 'AB 'ABB) = T

10. (LISTP S)
Returns T if S is a CONS-cell, and NIL otherwise.

11. (UNDEFP A <S>)

Returns T if A is an undefined atom, and NIL otherwise.

If S is given, and A is undefined, the value of S is assigned to A.

```
Ex: (UNDEFP 'X) = T (if X is unbound)
(UNDEFP 'X 3) = 3 (X is SETQ'd to 3)
(UNDEFP 'X) = NIL (X is now bound.)
```

12. (TAILP L1 L2)

Returns L1 if L1 is a tail (i.e. some number of CDRs $\geq$ 0) of L2, and NIL otherwise.

```
Ex: if X has the value (A B C)
(TAILP '(B C) X) = NIL
(TAILP (CDR X) X) = T
```

B. List Searching Operations

The functions in this section enable the user to break down LISP structures into component structures in various ways. The result will frequently depend on finding some particular substructure.

1. (CAR L)

returns the CAR of any structure (i.e., the first element of any list or the VALUE of an atom).

```
Ex: (CAR '((B C) D (E F))) = (B C)
```

2. (CDR L)

returns the CDR of any structure (i.e., the list of remaining elements of any list or the PLIST of a non-numeric atom). The CDR of a numeric atom is an error.

```
Ex: (CDR '((B C) D (E F))) = (D (E F))
```

3. (C . . . R L)

These 28 functions perform all compositions of up to 4 instances of CARs and CDRs.

```
Ex: (CAAR L) = (CAR (CAR L))
(CAAAAR L) = (CAR (CAR (CAR (CAR L))))
(CADADR L) = (CAR (CDR (CAR (CDR L))))
(CDDDR L) = (CDR (CDR (CDR L)))
```

4. (MEMBER S1 L <S2>)

The list L is searched to see if S1 is an element. If so, then the rest of the list L, starting with S1, is returned.

If S1 is not an element of L, and no third argument is given, NIL is returned. If a third argument is given, it is EVALed and that result is returned.

```
Ex: (MEMBER 'A '((A B) C (D E) G)) = NIL
(MEMBER 'A '((A B) C (D E) G) '(ADD1 3)) = 4
(MEMBER '(D E) '((A B) C (D E) G)) = ((D E) G)
```

5. (MEMQ S1 L <S2>)

Same as MEMBER, but uses an EQ test instead of an EQUAL test.

6. (ASSOC S1 L <S2>)

The list L is searched to see if S1 is the CAR of any element. If so, then that element is returned. If S1 is not the CAR of any element, and no third argument is given, NIL is returned. If a third argument is given, it is EVALed and that result is returned.

```
Ex: (ASSOC 'A '((A B) (C D) (E G))) = (A B)
(ASSOC '(A B) '((A B) (C D) (E G)) ''FAIL) = FAIL
```

7. (ASSQ S1 L <S2>)

Same as ASSOC, but uses an EQ test instead of an EQUAL test.

8. (FIND S1 S2 <N>)

The structure S2 is searched for any substructure (subtree) whose CAR is EQUAL to S1. If N is given, the Nth such substructure is returned. If N is not given, the first such substructure is returned. If the substructure specified is not found, FIND returns NIL.

```
Ex: (FIND 'B '(A B C)) = (B C)
(FIND 'A '(A (B (A C) D))) = (A (B (A C) D))
(FIND 'A '(A (B (A C) D)) 2) = (A C)
(FIND '(A C) '(A (B (A C) D))) = ((A C) D)
(FIND '(A C) '(A (B (A C) D)) 2) = NIL
```

9. (NTH L N)

returns the sublist of L beginning with the Nth element of L. If N is zero or negative, NTH will return the last cell of L. If N is greater than the number of elements of L, NTH will return NIL.

```
Ex: (NTH '(A B C) 1) = (A B C)
(NTH '(A B C D) 3) = (C D)
(NTH '(A B C D) 0) = (D)
(NTH '(A B C D) 100) = NIL
```

10. (LAST S)

Returns the last top-level CONS-cell of a list.

```
Ex: (LAST '(A B C)) = (C)
(LAST '(A B C (D E)) = '((D E))
```

C. Functions that Create New LISP Structures

This section includes functions that, besides returning a value, create new LISP structures. Frequently, the value returned from a function in this section is precisely the new LISP structure which was created.

1. (CONS S1 S2)
returns the dotted-pair of S1 and S2.

```
Ex: (CONS 'A 'B) = (A .  B)
(CONS '(A B C) '(D E F)) = ((A B C) .  (D E F)) = ((A B  C)
D E F)
(CONS 'A '(B C (D E))) = (A B C (D E))
```

2. (LIST S1 . . . Sn)
returns the list of S1 through Sn.

```
Ex: (LIST 'A 'B) = (A B)
(LIST '(A B C) '(D E F)) = ((A B C) (D E F))
(LIST 'A '(B C D)) = (A (B C D))
```

3. (EVLIS L)
evaluates each element of L and returns a list of these values.

```
Ex: (EVLIS '((ADD 3 1) (ADD 5 6))) = (4 11)
```

4. (APPEND L1 . . . Ln)
returns a concatenated list of copies of lists L1 through Ln.

```
Ex: (APPEND '(A B C) '(D E F)) = (A B C D E F)
(APPEND '(A B C) NIL '(D E F)) = (A B C D E F)
```

5. (APPEND1 L S1 . . . Sn)
returns a copy of the list L, with S1 through Sn appended as elements to the end.

```
Ex: (APPEND1 '(A B C) 'D 'E 'F) = (A B C D E F)
(APPEND1 '(A B C) '(D E) 'F) = (A B C (D E) F)
(APPEND1 NIL 'C 'D 'E) = (C D E)
```

6. (APPEND* L1 . . . LN)
returns copies of L1 . . . LN-1, appended to the original list (not a copy) LN.

```
EX: (APPEND* '(A B C) '(D E) '(F G H)) = (A B C D E F G H)
```

7. (REVERSE L)
returns a list of the (top-level) elements of L, in reverse order.

```
Ex: (REVERSE '(A B (C (D E)) F)) = (F (C (D E)) B A)
```

8. (DREVERSE L)
returns a list of the (top level) elements of L, in reverse order. The original list is destroyed in the process.

EX: Suppose X has the value (A B (C D) E F) then:

```
(DREVERSE X) = (F E (C D) B A)
```

and X = (A).

9. (COPY S1 <S2 <S3>>)
returns a copy of structure S1.

If arguments S2 and S3 are given, each occurrence of S2 in the original structure (S1) will be replaced by S3 in the copy. S2 need not be a "top-level" element, but may be an element at any level. If S2 appears without S3, then all occurrences of S2 in the original structure (except as the CDR of a dotted-pair) will be deleted in the copy.

If the first argument to COPY is a literal atom, the value of COPY will be a new atom, not on the OBJECT LIST, with the same PNAME as the original atom.

```
Ex: (COPY '(A B C)) = (A B C)
(EQUAL L (COPY L)) = T
(EQ L (COPY L)) = NIL
(COPY '(A (B) C) 'B) = (A NIL C)
(COPY '(A B C (D B) E) 'B) = (A C (D) E)
(COPY '(A B C (D B) E) 'D '(L K)) = (A B C ((L K) B) E)
(COPY 'A) = A
(EQ (COPY 'A) 'A) = NIL
```

10. (DSUBST L S1 S2)
returns L with all occurences of S1 replaced by S2. The list L is physically changed.

EX: Suppose X has the value (A B C D) then:
(DSUBST X 'C '(D)) = (A B (D) D)
and X has the value (A B (D) D).

11. (GENSYM <A>)
returns a unique atom. If no argument is given, GENSYM creates atoms G1, G2, . . . etc. Every time GENSYM is called, the GENSYM counter is incremented by one. If a literal atom or an IOARG is given to GENSYM, the PNAME of that atom, or of the buffer associated with the IOARG will be used, followed by the current GENSYM counter. If the buffer portion of the IOARG is NIL, the current system output buffer will be used.

The GENSYM counter can be re-set by using the STATUS function.

Note: An atom created by GENSYM is not placed in the system OBJECT LIST. Thus, if an atom with the same PNAME is created during a READ, it will not refer to the same atom which was created by GENSYM. The user may remove any atom from the OBJECT LIST by calling the function REMOB (See the Section on OBLIST functions).

Ex: (SET 'GENSET (GENSYM 'ATOM)) = ATOM1
(EQ GENSET 'ATOM1) = NIL
(EQNAME GENSET 'ATOM1) = T

12. (MKATOM A1 . . . An)
The function MKATOM returns an atom whose PNAME is the string of all the PNAMES of its arguments. Each argument must EVAL to a literal atom.

EX: (MKATOM 'ABC 'DE 'FGHI) = ABCDEFGHI
(MKATOM (CAR '(THIS IS IT)) (CADR '(SO IS THIS)))
= THISIS

13. (EXPLODE A)

Returns a list of the single-character atoms of the PNAME of A.

A must be a literal atom, or an IOARG, in which case the PNAME of its associated buffer will be used. If the buffer portion of an IOARG is NIL, the system output buffer will be used.

14. (LDIFF L1 L2 <L3>)

L2 must be a tail of the list L1, i.e. EQ to the result of applying some number of cdr's to L1. LDIFF(L1,L2) returns a list of all elements of L1 up to L2, i.e. the list difference of L1 and L2. The value of LDIFF is always a new list structure unless L2 = NIL, in which case the value is L1 itself. If L3 is included as a parameter, then the value of LDIFF is effectively:

(NCONC L3 (LDIFF L1 L2))

i.e. the list difference is added at the end of list L3.

EX: Suppose X has the value (A B C D E F) then:

(LDIFF X (MEMQ 'D X)) = (A B C)

(LDIFF X NIL) = X = (A B C D E F)

(LDIFF X (MEMQ 'D X) X) = (A B C D E F A B C)

15. (UNION L1 L2)

returns a list which represents the set union of lists L1 and L2.

The members of L1 and L2 are treated as the elements of a set, and elements which are EQUAL will not be duplicated in the resulting list.

Ex: (UNION '(A (B C) (D E)) '((B C) D)) = (A (B C) (D E) D)
(UNION '(1 2 3) '(3 4 5)) = (1 2 3 4 5)

16. (UNIONQ L1 L2)
same as UNION, but uses an EQ test instead.

Note -- Due to the way this function is implemented, numbers are not recognized as being EQ.

EX: (UNIONQ '(A B C D) '(E F B D G)) = (G D B F E C A)

17. (INTERSECT L1 L2)
Returns a list of all elements of L1 which are also elements of L2. The test used to compare elements is the EQUAL test.

Ex: (INTERSECT '(A (B C) (F G) D) '((B C) D E)) = ((B C) D)

18. (INTERSECTQ L1 L2)
same as INTERSECT, but uses an EQ test.

Note -- Due to the way this function is implemented, numbers are not recognized as being EQ.

EX: (INTERSECTQ '(A B 1 2) '(1 2 C B A)) = (A B)

19. (EXCLUDE L1 L2)
Returns a list of all elements of L2 which are not elements of L1. The test used is EQUAL.

Ex: (EXCLUDE '(A (B C) D E) '((X Y) (B C) A Z)) =
((X Y) Z)

20. (EXCLUDEQ L1 L2)
same as EXCLUDE, but uses an EQ test.

Note -- Due to the way this function is implemented, numbers are not recognized as being EQ.

EX: (EXCLUDEQ '(A B C 1 2) '(D E A B 2)) = (2 E D)

21. (SORT L <SP>)

Returns list L sorted according to the function SP. SP should be a predicate of two arguments, if the first argument should be ahead of the second argument in the sorted list SP should return a NON-NIL value, otherwise SP should return NIL. SP defaults to the system function "SORTP".

Note -- SORT destructively sorts list L.

```
EX: (SORT '(B C A)) = (A B C)
    (SORT '(4 2 1 3) 'LESSP) = (1 2 3 4)
```

22. (MERGE L1 L2 <SP>)

Returns a merged list of the two sorted lists L1 and L2 according to SP. SP should be a predicate of two arguments. The next element of the first list is passed as the first argument to SP and the next element of the second list is passed as the second argumant to SP. SP should return a NON-NIL value if the first argument should be ahead of the second argument, otherwise it should return NIL. SP defaults to the system function "SORTP".

Note -- MERGE destructively merges the two lists.

```
EX: (MERGE '(A C E) '(B D F G)) = (A B C D E F G)
    (MERGE '(1 3 5) '(2 4) 'LESSP) = (1 2 3 4 5)
```

D. Functions That Modify Existing LISP Structures

1. (SET A1 S1 . . . An Sn)

The VALUE of Ai is set to Si for each i, and the value returned from SET is the last Si.

Ex: (SET 'X 'A 'Y '(B C)) = (B C), and the VALUE of X is set to A,
the VALUE of Y to (B C).

2. (SETQ A1 S1 . . . An Sn)
Sets arguments A1 . . . An to the values of arguments S1 . . . Sn, respectively.

The value returned from SETQ is the value of SN.

Ex: (SETQ X (CAR '(B C)) Y 'A) = A, and the VALUE of X becomes B,

Note: Suppose the VALUE of X is VALX. Then
(SET 'X '(B C) 'Y X) = VALX,
and X is set to (B C), Y is set to VALX, since the arguments to SET are EVALed before SET is called. However,
(SETQ X '(B C) Y X) = (B C),
and X is set to (B C), Y is set to (B C), since the SETQ performs an EVAL-SET-EVAL-SET loop.

3. (SETA ARR-ELT S)
sets the array element specified by ARR-ELT to the value of S. ARR-ELT is an array element specification of the same form used to get an array element.

SETA returns the value of S.

Ex: (SETA (B 3 4) '(X Y)) = (X Y), and the array element (B 3 4) is set to (X Y).
(SETA (B (ADD 2 2) (SUB1 5)) (B (ADD 1 1) 3)) will return the value of (B 2 3),
and the array element (B 4 4) will be
set to this value

4. (UNCONS L A)
returns the CAR of L, and, as a side effect, sets the VALUE of A to the CDR of L. Note that A, the second argument, is not evaled.

Ex: (UNCONS '(A B C) X) = A, and the VALUE of X becomes (B C).

If the VALUE of L is (A B C), then:
(SET 'M (UNCONS L L)) = A, and the VALUE of L becomes (B C),
and the VALUE of M becomes A.

5. (RPLACA S1 S2)
replaces the CAR of S1 with S2 and returns the new structure.

Ex: (RPLACA '(A B C) '(E F)) = ((E F) B C)

6. (RPLACD S1 S2)
replaces the CDR of S1 with S2 and returns the new structure

Ex: (RPLACD '(A B C) '(D E)) = (A D E)

Note: RPLACA and RPLACD actually modify the structures sent to them as arguments, unlike functions such as APPEND, APPEND1, and COPY, which create entirely new structures with the desired properties. Because of this, RPLACA and RPLACD should be used with great caution. It is very easy to create circular LISP structures using these functions, and attempts to process such structures can become very expensive by the time the user discovers his program is in an infinite loop.

7. (DELETE S L <N>)
Deletes up to N occurrences of expression S from the list L. If no N is given, all occurrences are deleted. S must occur as a top-level element of the list L. DELETE returns the new list L.

Ex: (DELETE 'C '(A B C D C D C D) 2) = (A B D D C D)
(DELETE 'C '(A B C D (C D) C D)) = (A B D (C D) D)

If the VALUE of L is (A B C), then (DELETE 'B L) = (A C), and the VALUE of L is (A C). However, (DELETE 'A L) = (B C), but the VALUE of L is still (A B C). Thus, DELETEing the CAR of a list L is merely equivalent to taking the CDR of L, but DELETEing any other element will cause an actual change in the list structure.

8. (REMOVE S L <N>)
Same as DELETE, but the original structure is not changed.

EX: Suppose X has the value (A (A B) (C D) (A B)) then:
(REMOVE '(A B) X) = (A (C D))
and X still will have the value (A (A B) (C D) (A B)).

9. (DELQ S L <N>)
Same as DELETE, but uses an EQ test instead of EQUAL.

10. (NCONC L1 . . . Ln)
creates a concatenated list of L1 through Ln by actually modifying list Li so that it becomes Li . . . Ln. Thus, list LN is "grafted" onto the end of list L(n-1), and then list L(n-1) is grafted onto the end of list L(n-2), etc.

Ex: If the VALUE of X is (A B), and the VALUE of Y is (C D) and the VALUE of Z is (E F), then:

(NCONC X Y Z) = (A B C D E F), and the VALUE of Z is (E F),
and the VALUE of Y is (C D E F),
and the VALUE of X is (A B C D E F)

Note: The same warnings given for RPLACA and RPLACD also apply to NCONC.

E. Operations on Property Lists

Although the property list of an atom is often treated as an unordered collection of property indicators and property-values, in fact the PLIST of an atom is a normal LISP list of the form (IND1 PVAL1 . . . INDN PVALN). With a few special exceptions, new property indicators and property-values are added at the front of the PLIST.

1. (PUT <A,LA> IND <PVAL>)
gives the atom A, or all the atoms in the list LA, the property IND with property value PVAL.

If PVAL is omitted, a system default of T is used. (This system default may be changed by calling the STATUS function).

If an atom already has property IND on its PLIST, then the previous PVAL associated with property IND is replaced by the new PVAL.

The value returned from PUT is PVAL.

Ex: (PUT '(A B) 'INCL 'X) = X,
and the property INCL with property-value X
is put on the PLIST of A and B.

2. (PUTL L IND <PVAL>)

is like PUT execept that it operates directly on the list it is given (i. e. as if it were a property list). The value returned is the new list.

```
Ex: (PUTL '(A INCL B BLUE) 'A 'EXCL)
  = (A EXCL B BLUE)

(PUTL '(A EXCL B BLUE) 'C 'GREEN)
  = (C GREEN A EXCL B BLUE)
```

3. (DEFPROP <A,LA> IND <PVAL>)

DEFPROP is the NEXPR version of PUT. It returns its first argument.

4. (ADDPROP <A,LA> IND <PVAL>)

works just like PUT except a new instance of IND is always put on the PLIST of A, or of the atoms in LA. Thus, using ADDPROP, it is possible to have duplicate instances of one property on the PLIST of an atom. Using ADDPROP in conjunction with (REM A IND 1), the user may operate a push-down stack of property-values for a particular property.

```
Ex: (PUT 'A 'INCL 'X) = X
(ADDPROP 'A 'INCL 'Y) = Y
(GET 'A 'INCL) = Y
(REM 'A 'INCL 1) = NIL
(GET 'A 'INCL) = X
```

5. (GET <A,L> IND <S>)

returns the property-value associated with the indicator IND on the PLIST of A. If A does not have property IND, and S is not given, then GET returns NIL.

If the third argument S is given, then S is a form to be EVALed if A does not have property IND. If S is EVALed, the value of S will be the value returned from GET.

```
Ex: (PUT 'A 'INCL '(X Y)) = (X Y)
(GET 'A 'INCL) = (X Y)
(GET  'A  'NOTON) = NIL, assuming NOTON is not on the PLIST
of A.
(GET 'A 'NOTON '(GET 'A 'INCL)) = (X Y)
```

If the first argument is a list, it will be searched directly, rather than having its P-list taken.

Ex: (GET '(A B C D) 'C) = D

6. (GETL <A,L> L <S>)

finds the first indicator on the PLIST of A which is a member of the list L. Returns the rest of the PLIST of A, starting with the indicator which was found.

If no indicator on the PLIST of A is a member of L, then if S is not given, GETL returns NIL. If S is given, it will be EVALed and this value will be returned from GETL.

Ex: If the PLIST of BOOK is (COLOR BLUE SIZE 367 TOPIC MATH), then

```
(GETL 'BOOK '(WEIGHT TOPIC SIZE)) = (SIZE 367 TOPIC MATH)
(GETL 'BOOK '(TOPIC) '(GET 'BOOK 'COLOR)) = (TOPIC MATH)
(GETL 'BOOK '(WEIGHT) '(GET 'BOOK 'COLOR)) = BLUE
```

If the first argument is a list, it will be searched directly, rather than having its P-list taken.

Ex: (GETL '(A B C D) '(X C)) = (C D)

7. (GETFN FN)

GETFN allows the user to inspect the function definition associated with a form. GETFN will consider its argument as a function specification, and will simulate the action of EVAL in determining how to apply it. If FN is a LAMBDA or LABEL expression, then the value returned from GETFN is just FN itself. If FN is an atom which is currently defined as a SUBR, FSUBR, or NSUBR, then the PVAL associated with the SUBR, FSUBR, or NSUBR indicator is returned as the value of GETFN. (This PVAL will generally be a SUBR or ARRAY type atom.)

If FN is an atom which is currently defined as an EXPR and the PVAL associated with the EXPR property is a LAMBDA-expression, then the LAMBDA-expression is the value returned from GETFN.

GETFN generates an error if it encounters an atom with no function definition whose VALUE is itself or *UNDEF*.

```
EX: (GETFN '(LAMBDA (X) X)) = (LAMBDA (X) X)
            (GETFN 'CAR) = *
```

The SUBR atom will be printed as an Asterisk, but it may be Dumped, compared to other addresses, or transfered to the PLISTs of other atoms.

8. (REM <A,LA> IND <N>)

removes up to N occurrences of the property IND from the PLIST of the atom A, or all the atoms in the list LA. If N is not given, all occurrences are removed.

The value of REM is NIL.

```
Ex: (PUT 'A 'INCL '(X Y)) = (X Y)
(GET 'A 'INCL) = (X Y)
(REM 'A 'INCL) = NIL
(GET 'A 'INCL) = NIL
```

F. Basic Numeric Predicates

1. (GREATERP N1 . . . Nn)

returns T if N1 . . . Nn is a strictly decreasing sequence of numbers. NIL otherwise.

2. (LESSP N1 . . . Nn)

returns T if N1 . . . Nn is a strictly increasing sequence of numbers. NIL otherwise.

3. (ZEROP N)

returns T if integer N=0. NIL otherwise

4. (EVENP N)

returns T if N is an even integer. NIL otherwise.

5. (MINUSP N)

returns T if N is a negative number. NIL otherwise.

G. Basic Numeric Operations

1. (LENGTH L)
   returns the length of the list L. LENGTH of an atom is 0.

2. (PLEN A)
   returns the length of the PNAME of the atom A. A must be a literal atom or ioarg.

3. (ADD1 N)
   returns integer N+1

4. (SUB1 N)
   returns integer N-1.

5. (MINUS N)
   returns number -N.

6. (ABS N)
   returns the absolute value of number N.

7. (FIX N)
   returns the integral (truncated) part of N.

   Ex: (FIX 3.91) = 3

8. (FLOAT N)
   returns the floating point equivalent of N.

9. (MAX N1 . . . Nn)
   returns the algebraic maximum of numbers N1 . . . Nn.

10. (MIN N1 . . . Nn)
    returns the algebraic minimum of numbers N1 . . . Nn.

11. (PLUS N1 . . . NN)
    returns the sum of N1 . . . NN. The function ADD has the same effect.

12. (DIFFERENCE N1 N2)
    returns N1-N2. The function SUB has the same effect.

13. (TIMES N1 . . . Nn)
    returns the product of N1 . . . Nn.

14. (DIVIDE N1 N2)
    returns the quotient of N1 and N2. Floating point division.

15. (REMAIN N1 N2)
    N1 and N2 must be integers. Returns the remainder of N1/N2.

16. (ADDRESS S)
    returns a numeric atom equal to the address of the LISP structure S.

17. (SHIFT N1 N2)
    N1 and N2 must be integers. Returns the number N1, shifted N2 bits to the left. If N2 is negative, the effect is a shift to the right.

18. (LAND N1 . . . Nn)
    N1 . . . Nn must be integers. Returns the result of a bitwise logical AND of N1 . . . Nn.

19. (LOR N1 . . . Nn)
N1 . . . Nn must be integers. Returns the result of a bitwise logical OR of N1 . . . Nn.

20. (LXOR N1 . . . Nn)
N1 . . . Nn must be integers. Returns the result of a bitwise logical EXCLUSIVE-OR of N1 . . . Nn.

```
Ex: (LAND 3 5) = 1
(LOR 3 5) = 7
(LXOR 3 5) = 6
(LXOR -1 3) = -4
(SHIFT 32 -1) = 16
(SHIFT 3 2) = 12
```

H. Control Functions

This section includes the functionals, which take as their arguments definitions of functions to be invoked;, as well as EVAL, PROG, REPEAT, DO, and PROGN, which control the evaluation of forms in LISP.

1. (EVAL S)
Evaluates its argument and returns the result.

Ex: If the VALUE of X is (A B C), and the VALUE of A is VALA, then
(EVAL (CAR X)) = VALA

2. (PROG LA S1 . . . Sn)
The PROG function, along with GO and RETURN, allows the LISP user to write subroutine-like sequences of LISP code, with branching, and with the ability to exit and return a value at any point.

LA is a list of local or PROG variables. The PROG variables are bound to NIL upon entry to the PROG, and unbound to their previous values upon exit from the PROG. Thus, the PROG variables may be used within a PROG as though they were distinct variables from any outside the PROG. Note that this "protection" of PROG variables applies only to their VALUEs. If the property list of a PROG variable is changed within a PROG, the change will not be undone upon exit from the PROG.

The PROG variable list may be NIL, but it may not be omitted. 5.

S1 . . . Sn are a sequence of forms to be evaluated in order. However, if any of these forms are atoms, they are not evaluated, but rather are interpreted as statement labels. If a form (GO A) appears in the PROG, and A is used as a statement label in the PROG, then evaluating (GO A) causes the flow-of-control to be transfered to the form which appears after the label A.

If the flow-of-control "drops through" the last form of the PROG, then the value of that form will be returned as the value of the PROG. However, if the last form of the PROG is an atom, then the atom itself, rather than its VALUE is returned as the value of the 6.
PROG.

3. (RETURN S <LEVEL>)

If at any point within a PROG, a form (RETURN S) is evaluated, then PROG immediately exits, and returns the value of S. RETURN takes an optional second argument, which is the level to be RETURNed from. This argument is a stack pointer and is specified in the same way as in UNEVAL, DISPLAY, and RES. If the second argument to RETURN is omitted, the return will be from the current 7.
dominating PROG.

(RETURN (CAR X)) returns to the closest enclosing PROG.
(RETURN (CAR X) 'FOO) returns from the closest enclosing call to FOO.

4. (GO A)

GO is used within the PROG function to branch to the PROG label A. GO is, like PROG , an N-type function. Thus, (GO A) will cause a branch to the form labelled by the atom A. However, if GO is given a non-atomic argument, it will EVAL this argument, and then attempt to "go" to the result. Ex: (GO (CAR A)) will evaluate (CAR A), and if the result is an atom, will branch accordingly. If the result is not an atom, GO will EVAL it in turn, and continue the process until an atom is
found. | fn3(PROG1,S1,'. . . Sn) returns S1, or its 8.
first argument. This function is useful when the user wants to do several different things in one step, and wants only the fist value to be returned.

(PROG1 'DONE S2 . . . Sn) = DONE

5. (PROGN S1 . . . Sn)

returns Sn, or its last argument. This function is useful when the user wants to do several different things in one step, and want only the last result returned. The argument designators will be EVALed as a side effect of calling PROGN. For example, at the top level, the user may wish to embed a number of forms in a PROGN in order to suppress printing of all but the last result.

```
Ex: (PROGN S1 . . . Sn 'DONE) = DONE
```

6. (REPEAT S N <EQUFAIL>)

Evaluates form S N times, or until the value of S is EQUAL to EQUFAIL. REPEAT returns the last computed value of S. If N is negative, an error results.

```
Ex: (SETQ N 1)
(REPEAT '(SETQ N (ADD1 N)) 12) = 13, and N = 13
(REPEAT '(SETQ N (ADD1 N)) 12 2) = 2, and N = 2
```

7. (DO VAR INITIAL INCR TEST S1 . . . Sn)

This function can be useful for writing FORTRAN or ALGOL like loops. It can be best explained with the following equivalent PROG.

```
(PROG (I)
      (SETQ I (INITIAL))
 LOOP (COND ((TEST) (RETURN I)))
      S1
      S2
      .
      .
      SN
      (SETQ I (INCR))
      (GO LOOP))
```

8. (APPLY FN L)

Causes the function FN to be invoked, where L is a list of its arguments. FN may be any LISP function specification.

```
Ex: (APPLY 'CAR '( (A B C) ) = A
(APPLY 'CONS '(X Y) ) = (X . Y)
```

9. (APPLY1 FN S1 . . . Sn)
Causes the function FN to be invoked, where S1 . . . Sn are the arguments of FN. FN may be any LISP function specification.

Ex: (APPLY1 'CAR '(A B C)) = A
(APPLY1 'CONS 'X 'Y) = (X . Y)

10. (MAP FN L1 . . . Ln)
Causes the function FN to be called, with L1 . . . Ln as its arguments, and then to be called again with (CDR L1) . . . (CDR Ln) as its arguments, and then to be called again with (CDDR L1) . . . (CDDR Ln) as its arguments, etc., until the shortest list is exhausted. Thus, when MAP is used, the arguments of FN will always be lists, never atoms.

MAP returns NIL.

11. (MAPC FN L1 . . . Ln)
Works like MAP, except the CAR of each successive list is used as the argument to FN. Thus, MAPC calls FN with (CAR L1) . . . (CAR Ln) as its arguments, and then with (CADR L1) . . . (CADR Ln), etc.

MAPC returns NIL.

12. (MAPLIST FN L1 . . . Ln)
Causes the function FN to be called with L1 . . . Ln as its arguments, and then with (CDR L1) . . . (CDR Ln), etc, just as in MAP. However, the value returned from MAPLIST is the list of all the successive values returned from FN.

13. (MAPCAR FN L1 . . . Ln)
Works just like MAPLIST except that the CAR of each successive list is used as the argument to FN. MAPCAR returns a list of all the successive values returned from FN.

14. (MAPCON FN L1 . . . Ln)

Causes the function FN to be called with L1 . . . Ln as its arguments, and then with (CDR L1) . . . (CDR Ln), just as in MAP. However, the value returned from MAPCON is a concatenated list of all the values returned from FN.

***NOTE: The user should be aware that the values returned from FN when called via MAPCON or MAPCAN must be lists, or an error will result.

***Warning: The user should be aware that MAPCON and MAPCAN call NCONC to create the concatenated list of values returned from FN. Thus, the actual structures returned from FN will be modified by MAPCON and MAPCAN. The possibilities for creating circular lists are the same as for NCONC, RPLACA, etc.

15. (MAPCAN FN L1 . . . Ln)

Works just like MAPCON, except the CAR of each successive list is used as the argument to FN. MAPCAN returns a concatenated list of all the values returned from FN.

```
Ex: Let the VALUE of X be ((D 7) (A 6) (N 5))
(MAPLIST 'REVERSE  X) = (((N 5) (A 6) (D 7)) ((N 5) (A 6))
((N 5)))
(MAPCAR 'REVERSE X) = ((7 D) (6 A) (5 N))
(MAPCON 'REVERSE X) = ((N 5) (A 6) (D 7) (N 5) (A 6) (N 5))
(MAPCAN 'REVERSE X) = (7 D 6 A 5 N)
```

I. OBJECT List Functions

LISP maintains a system list of atoms called the OBJECT LIST. The purpose of the OBJECT LIST is to allow references to atoms by name on input. Thus, whenever READ reads an atom, it compares the atom with the atoms on the OBJECT LIST. If they match, then the pointer created references the atom which was already on the OBJECT LIST, and no new atom is created. If there is no match, a new atom is created, and placed on the OBJECT LIST.

There may be atoms in the system which are not on the OBJECT LIST. For example, atoms created by GENSYM are guaranteed to be unique since they are not on the OBJECT LIST. A reference by PNAME to an atom which is not on the OBJECT LIST will cause a new atom be created with the same PNAME, and the original atom will not be referenced.

Atoms on the OBJECT LIST are considered active structure by the garbage collector, and are preserved.

1. (OBLIST)

   The function (OBLIST) of zero arguments returns a (long) list of all the atoms which are on the OBJECT LIST.

2. (REMOB A1 . . . An)

   The function REMOB removes literal atoms from the OBJECT LIST. Once an atom is REMOBed, it may no longer be referenced by PNAME, and will be destroyed during the next garbage collection, if it is not referenced by any active LISP structures.

3. (PUTOB A1 . . . An)

   The function PUTOB puts literal atoms on the OBJECT LIST. If an argument to PUTOB is already on the OBJECT LIST, then PUTOB has no effect on that atom. If PUTOB finds an atom on the OBJECT LIST with the same PNAME as one of its arguments, the PUTOB argument is put on the OBJECT LIST "in front of" the other atom, but the other atom is not remobed. Thus, the most recent atom with a particular PNAME is the one which will be found by READ, but if the most recent atom with a particular PNAME is REMOBed, then a previous atom with the same PNAME will become "active" (from the point of view of the READ function).

4. (MAPOB FN)

   Maps the function FN onto the OBLIST. Unlike the function OBLIST, MAPOB does no CONSes and will not pass the atom *UNDEF* to FN. The function FN must be a function of 1 argument.

J. Conditional Functions

1. (AND S1 . . . Sn)

evaluates the arguments S1 through Sn in turn until some Si has a value of NIL. AND then stops evaluating and returns NIL.

If none of the Si has a value of NIL, AND returns the value of Sn.

Ex: (AND (CAR Z) (SETQ Z A) (SETQ X 'DONE)) has the following effect:

If (CAR Z) is NIL, merely returns NIL.

Otherwise, Z is set to the VALUE of A, and if the VALUE of A is NIL, then returns NIL.

Otherwise, X is set to DONE, and DONE is returned.

2. (OR S1 . . . Sn)

Evaluates its arguments S1 . . . Sn until it finds one with a value which is not NIL. OR then returns that value. If all of the arguments evaluate to NIL, then OR returns NIL.

Ex: (OR (CAR Z) (SETQ Z A) (SETQ X 'DONE) (SETQ Y NIL)) has the following effecto:

If (CAR Z) is non-NIL, returns CAR Z.

Otherwise, sets Z to the VALUE of A. If the VALUE of A is non-NIL, then returns that value.

If the VALUE of A is NIL, then sets X to DONE, and returns DONE.

Y will never be set to NIL.

```
3.  (COND
       (P1 <S1 . . . SN>)
       (P2 <T1 . . . TN>)
       .  .  .
       (PN <U1 . . . UN>))
```

is the basic conditional execution format for LISP. The arguments to COND are one or more COND-expressions of the form: (P <S1 . . . Sn>).

COND starts with the first COND-expression, and evaluates P, which may be any LISP form. If the value of P is NIL, then COND will go on to the next COND-expression and repeat the process. If the value of P for the last COND-expression is NIL, then COND returns NIL.

If the value of P is non-NIL, then COND does not go on to the next COND-expression. COND will evaluate S1 . . . Sn successively, and the value returned from COND will be the value of Sn. If no Si are given, COND merely returns the value of P.

Ex: We can see that the functions AND and OR are merely sub-cases of COND.

```
(AND S1 . . . Sn) = (COND
                        ((NOT S1) NIL)
                        ((NOT S2) NIL)
                             .
                             .
                        ((NOT S(n-1)) NIL)
                        ((Sn)))

or: (AND S1 . . . Sn) = (COND
                            (S1 (COND
                                    (S2 (COND . . .
                                               .
                                               .
                                              (COND
                                                (S(n-1) Sn)
                                                 )...))))

Ex: (OR S1 . . . Sn) = (COND
                           (S1)
                           (S2)
                            .
                            .
                           (Sn))
```

```
4.  (SELECT EQUTHING
       (E1 <S1 . . . SN>)
       (E2 <T1 . . . TN>)
       . . .
       (EN <U1 . . . UN>)
       FAIL)
```

is similar to COND, except the values of E1 . . . En are tested to see if they are EQUAL to the value of EQUTHING. If so, then S1 through Sn are evaluated, and the value of Sn is returned as the value of SELECT.

If E1 does not match EQUTHING, then SELECT goes on to (E2 T1 . . . TN), etc. If E1 matches EQUTHING, and no Si are given, then SELECT merely returns the value of E1.

If none of the Ei match EQUTHING, then FAIL is evaluated, and its value is returned. It is important to understand that the last argument of SELECT is always treated as a form to evaluate in case of failure, and never as a (E1 S1 . . . Sn) type of expression. Thus, a FAIL expression must be given.

```
Ex: (SELECT (GET 'BOOK 'COLOR)
     ('BLUE (BLUEFN 'BOOK))
     ('RED (REDFN 'BOOK))
     ('GREEN (GREENFN 'BOOK))
    (PROGN (PRINT '(ERROR: BOOK ILLEGAL COLOR))
    (ERRCOLOR 'BOOK)))
```

```
5.  (SELECTQ EQUTHING
       (<A1,LA1> <S1 . . . SN>)
       (<A2,LA2> <S1 . . . SM>)
       . . .
       (<AN,LAN> <S1 . . . SI>)
       FAIL)
```

is similar to SELECT, except that the test conditions, if atoms, are compared with EQ, and if lists, with MEMQ.

EQUTHING is EVALed. If the first test condition is an atom (A1), then an EQ test is performed, and if successful, then the corresponding (S1 . . . Sn) are EVALed. If the first test condition is an atom and not EQ to EQUTHING, then the next clause is examined. If the first test condition is not an atom, it must be a list of atoms (LA1) and a MEMQ test is performed between EQUTHING and the list of atoms. If EQUTHING is an element of the list of atoms (the MEMQ returns a non NIL value), then the corresponding (S1 . . . Sn) are EVALed. If the MEMQ fails, the next clause is examined. If all clauses fail, then the FAIL condition is EVALed.

```
(SELECTQ (GET 'BOOK 'COLOR)
     (BLUE (BLUEFN 'BOOK))
     (RED (REDFN 'BOOK))
     ((GREEN BLACK) (ODDCOLOR 'BOOK))
    (PROGN (PRINT '(ERROR: BOOK ILLEGAL COLOR))
    (ERRORCOLOR 'BOOK)))
```

6. (TIMER ID N)

The TIMER function allows the user to set up his own interrupts after a specified amount of CPU time has elapsed. The ID argument allows different timer interrupts to be distinguished. ID may be any LISP atom.

The following table indicates the various uses of the TIMER arguments:

ID	N	MEANING
non-NIL	0<n<1001	Set up an interrupt identified by ID, to generate a timer interrupt error in N seconds of real time. When the timer error occurs, the error form which will be printed is ID. The value returned is ID.
non-NIL	N>1000	Set up an interrupt identified by ID, to generate a timer interrupt error in N microseconds of CPU time. When the timer error occurs, the error form which will be printed is ID. The value returned is ID.
non-NIL	T	If there is an outstanding request with an ID which is EQ to ID, then TIMER returns the clock time remaining (in microseconds) in that request. Otherwise TIMER returns NIL.
NIL	NIL	Cancel all outstanding TIMER requests. The value of TIMER is NIL.
non-NIL	NIL	Cancels the pending interrupt request, if any, associated with ID. The value

of TIMER is the remaining clock time (in microseconds) in that request.

7. (TIME TIMEX <TIMEN <TIMETYPE>>)

TIME is an NLAMBDA function, which executes TIMEX TIMEN times and prints out statistics about the computation. TIME returns as value the last value of the evaluation of timex.

TIMETYPE is used to specify the type of output which will be produced. the possible values of TIMETYPE and their meanings are:

1 Print number of cons cells created.

2 Print CPU time used for the computation in seconds, garbage collection time is subtracted out.

4 Print CPU time used for garbage collection in seconds.

8 Print real time used for the computation in seconds.

To obtain more than one statistic pass to TIME the sum of the numbers of the statistics wanted. TIMETYPE defaults to 3, i.e. the number of cons cells created and the CPU time used for the computation is printed.

TIMEN defaults to 1. If TIMEN is negative a bad integer argument error will be produced.

Note -- TIME may be be called recursively.

8. (MTS <A,IOARG>)

The MTS function, besides allowing the user to return to MTS with the option to restart by calling (MTS), also allows execution of a single MTS command, with an automatic restart. This allows the LISP programmer (as distinct from the user of the program) to execute MTS commands without the user's knowledge.

The PNAME of the atom A, or the contents of the buffer associated with IOARG, is executed as an MTS command, and an automatic restart is performed.

MTS always returns NIL.

9. (UNTIL PRED S1 S2 . . . Sn)

An UNTIL loop. The forms PRED, S1, S2, . . . Sn are EVALed repeatedly until PRED EVALs to a non-NIL value. This value is then returned from the UNTIL function.

10. (WHILE PRED S1 S2 . . . Sn)

A WHILE loop. the forms PRED, S1, S2, . . . Sn are EVALed repeatedly until PRED EVALs to NIL. The value returned is NIL.

```
Ex: (SETQ X 1)
(WHILE (LESSP X 10) (SETQ X (ADD1 X)) (PRIN1 X))
   will print the line
     2 3 4 5 6 7 8 9 10
```

11. (STOP)

This function will cause execution of LISP to terminate.

V. Function Definition

A. Lambda-Expressions

As we noted in previously, the CAR of a form being EVALed is interpreted as a function specification. We described the situation when this CAR is an atom. In that case, the atom is interpreted as the name of a function to be called.

However, the CAR of a form to be EVALed need not be an atom. It can be an explicit function specification, called a LAMBDA-expression. The basic form of a LAMBDA-expression is:

```
(LAMBDA (A1 . . . An) S1 . . . Sn)
```

When a LAMBDA-expression appears as a function specification, it is treated as a function where A1 . . . An are the dummy arguments, and S1 . . . Sn is the body of the function. The dummy arguments A1 . . . An are bound to the arguments which are passed to the function, and then S1 . . . Sn are EVALed in turn. Finally, A1 . . . An are unbound to their original VALUEs.

The value of the LAMBDA-expression is the value of Sn.

A LAMBDA-expression may appear any time a function specification is required, for example, as the first argument of APPLY, MAP, MAPLIST, etc.

```
Ex: ((LAMBDA (X) (CDR X)) '(A B C)) = (B C)
```

Note: When we say that an atom is bound to some value, we mean that its present VALUE is saved, and it is set to the new value. When the atom is unbound, its previous VALUE is restored. Within the scope of a LAMBDA-expression, the dummy arguments have as their VALUEs the arguments of the function. For example, within the LAMBDA-expression above, the VALUE of X is (A B C).

Note: The number of arguments to a LAMBDA-expression, as for any other function, must be the same as the number of dummy arguments, or an error will result. The dummy argument list may [illegible] in which case the function takes no arguments, but it may [illegible] itted.

B. No-Spread LAMBDAs

Another form of LAMBDA-expression may be defined which takes an indefinite number of arguments. The basic form of the no-spread LAMBDA-expression is:

```
(LAMBDA A S1 . . . Sn)
```

Here the dummy argument list is replaced by a single non-NIL atom. When a no-spread LAMBDA is executed, the dummy argument A is bound to the number of arguments which were given.

The value of any particular argument may be obtained by calling the function ARG, with the number of the desired argument. Thus (ARG 1) returns the first argument, (ARG 3) the third argument, etc. Calling ARG with a number greater than the number of arguments given will generate an error.

Because a no-spread LAMBDA-expression may occur within the scope of another no-spread LAMBDA-expression, the function ARG takes an optional second argument, which, if given, must be EQ to the dummy argument of a dominating no-spread LAMBDA. For example, ((LAMBDA A (ARG 1 'A)) '(C D)) = (C D). If no second argument is given to ARG, then the immediately dominating no-spread LAMBDA is implied.

```
Ex: (LAMBDA C
           (PROG (X N)
                 (SETQ N 1)
              A  (COND ((GREATERP N C) (RETURN X))
                       ((SETQ X (APPEND1 X (CAR (ARG N))))))
                 (SETQ N (ADD1 N))
                 (GO A)))
```

This function will return a list of the CARs of all of its arguments.

C. FLAMBDA and NLAMBDA Expressions

There are two alternate forms of LAMBDA-expressions, which allow the user to give explicit definitions of N-type functions. The first of these is the NLAMBDA-expression. The basic spread and no-spread forms of NLAMBDA-expresssions are as follows:

```
(NLAMBDA (A1 . . . An) S1 . . . Sn)
(NLAMBDA A S1 . . . Sn)
```

The NLAMBDA-expression operates like an ordinary LAMBDA, except that the argument-designators themselves, rather than their values, are used as the arguments to the NLAMBDA.

```
Ex: ((NLAMBDA (X) (CDR X)) (A B C)) = (B C)
    ((NLAMBDA A (CAR (ARG 1))) '(A B C)) = QUOTE
```

The third and last form of LAMBDA-expression is the FLAMBDA. The basic forms of FLAMBDA- expression are as follows:

```
(FLAMBDA (A) S1 . . . Sn)
(FLAMBDA A S1 . . . Sn)
```

The argument-passing conventions for FLAMBDA-functions are slightly different than for LAMBDA and NLAMBDA-expressions. The FLAMBDA-expression must always have exactly one dummy argument, which in the case of a spread type FLAMBDA is bound to a list of all the argument-designators. In the case of a no-spread type FLAMBDA, the dummy argument is always bound to the number 1, and the function (ARG 1) will return the list of all the argument-designators.

```
Ex: ((FLAMBDA (A) A) X Y Z) = (X Y Z)
    ((FLAMBDA A (ARG A)) X Y Z) = (X Y Z)
```

It is important to be aware of the effect of APPLYing the three types of functions. The arguments to APPLY and APPLY1 are always EVALed before being passed to these functions, and they will not be EVALed again. Thus, for the purposes of APPLY, the difference between LAMBDA and NLAMBDA-functions disappears. However, for FLAMBDA-type functions, the arguments given are made into a list in the case of APPLY1, or left in their list form in the case of APPLY, and thus when these functions are APPLYed they are guaranteed to receive a single argument. The following examples illustrate the process:

```
(APPLY '(LAMBDA (X Y Z) (LIST X Y Z)) '(A B C)) = (A B C)
(APPLY '(NLAMBDA (X Y Z) (LIST X Y Z)) '(A B C)) = (A B C)
(APPLY '(FLAMBDA (X) (LIST X)) '(A B C)) = ((A B C))
```

Note: In general, the term "LAMBDA-expression" is a generic term including the NLAMBDA and FLAMBDA-expressions.

D. Named LAMBDA-expressions (LABEL-expressions)

LISP traditionally provides a special syntax for writing LAMBDA-expressions which can call them- selves. This is the LABEL-expression. The basic form of a LABEL-expression is:

```
(LABEL NAME LAMBDA-EXP)
```

NAME may be any atom. First NAME is bound to the LAMBDA-expression which is the second argument of the LABEL-expression, and the evaluation continues as though the LAMBDA-expression had been given. The effect is to temporarily define NAME as the LAMBDA-expression, provided that NAME is not already defined as a function within the system.

Thus, within the LAMBDA-expression, explicit calls to NAME may be made, which will invoke the LAMBDA-expression recursively.

```
EX: ((LABEL COUNT (LAMBDA (L N)
                          (COND ((NULL L) N)
                                ((COUNT (CDR L) (ADD1 N))))))
     '(A B C D E) 0)  =  5
```

The effect of this LABEL-expression is to temporarily define a function COUNT, which will return the sum of its second argument and the number of elements in its first argument.

E. Accessing Defined Functions

When an atom is to be used as a function name, a link to the function definition is maintained on the property list of that atom. The following special system indicators are used to mark function definitions:

SUBR
NSUBR
FSUBR
EXPR
BUG

SUBR, NSUBR, and FSUBR are indicators which mark the three types of built-in LISP functions. SUBRs take their arguments EVALed, like LAMBDA-functions; NSUBRs take their arguments unEVALed like NLAMBDA-functions, and FSUBRs take their arguments in a list, like FLAMBDA-functions. The property-values associated with these indicators are pointers to the machine code for those functions. An attempt to print out one of these links will merely cause an asterisk (*) to be printed.

EXPR and BUG are the indicators used to mark the 2 types of user-defined functions. The property-value associated with an EXPR indicator will be a function specification (usually but not necessarily a LAMBDA-expression) which will be invoked when the "parent" atom is used as a function name.

If several special system indicators are on the property list of the same atom, the first (and most recent) one will be used as the function definition for that atom.

Note: There is nothing to stop the user from modifying or destroying the special system properties on the PLIST of an atom. In fact, since the PLIST of an atom is the CDR of the atom, the user may access this list like any other list. This may frequently be a good way to make corrections to a user-defined function. However, modifying or destroying the links to built-in LISP functions should be done carefully, if at all.

***See the description the the GETFN function.

F. Defining New Functions in LISP

1. (DEFUN NAME <TYPE> ARGLIST S1 . . . Sn)

DEFUN is an N-type function which provides an easy way for the user to define one new LISP function by the usual method of putting a LAMBDA-expression on its property list. NAME is the name of the function being defined. TYPE must be EXPR, NEXPR, or FEXPR. If TYPE is omitted, EXPR is assumed. ARGLIST is a list of dummy arguments, or NIL, for a spread type function; or a single atom for a no-spread type function. S1 . . . Sn is the body of the function.

If TYPE is EXPR, a LAMBDA-expression is created.
If TYPE is NEXPR, an NLAMBDA-expression is created.
If TYPE is FEXPR, an FLAMBDA-expression is created.

DEFUN always puts the LAMBDA-expression created on the property list of NAME under the indicator EXPR. The value returned from DEFUN is the atom NAME.

Note: If TYPE is omitted, then ARGLIST may not be EXPR, NEXPR, or FEXPR.

```
Ex: (DEFUN SAMPLE C
         (PROG (X N)
              (SETQ N 1)
        A     (COND ((GREATERP N C) (RETURN X))
                    ((SETQ X (APPEND X (CAR (ARG N))))))
              (SETQ N (ADD1 N))
              (GO A)))
```

This creates a function called SAMPLE, which returns a list of the CARs of all of its arguments. SAMPLE takes an indefinite number of arguments - including none.

```
(SAMPLE) = NIL
(SAMPLE '(S R T) '(P Q) '(R)) = (S P R)
```

2. (DEFINE (NAME <TYPE> DEFN) . . . (NAME <TYPE> DEFN))

DEFINE is the basic function for defining and naming new LISP functions. DEFINE is an N-type function which takes an indefinite number of definitions as arguments. NAME is always an atom, which is the name of the entity being defined. TYPE may be EXPR, MACRO, BUG, ARRAY, or SUBR, or may be omitted, in which case EXPR is assumed.

For an EXPR or BUG, the DEFN given is put on the PLIST of NAME exactly as it appears. Thus, to DEFINE an EXPR, the entire LAMBDA-expression must be written out.

The form and meaning of BUG definitions will be explained in the two sections to follow.

The ARRAY and SUBR definitions require special parameters which define LISP arrays, and which create linkage to external subroutines, respectively. The form and meaning of these definitions will be explained in sections H and I to follow.

The value returned from DEFINE is the name defined if only one definition was given, or a list of the names defined if more than one was given.

Ex: (DEFINE (TEST EXPR (LAMBDA (Y) (PRINT Y)))) = TEST

This defines a function TEST which merely prints its argument.

Note: DEFUN and DEFINE, which put properties on the PLISTs of atoms, do not work in the same way as PUT. They compare the current indicator being placed on the PLIST with the first indicator which is there, and if they are the same, the PVAL of that indicator is replaced with the new definition. If the current indicator does not match the first one on the PLIST, it is merely placed in front of it. This process guarantees that the most recent function definition of an atom will be the active one.

i. BUGs

In order to facilitate the writing of de-bugging routines in LISP, a new facility called a BUG has been added to LISP/MTS. A BUG is a pseudo-function definition which can be placed on the property list of an atom already defined as a function. The BUG will cause an interception of the function on entry and on exit. The user can display the arguments sent to the function, or any other LISP structures, can test entry conditions, and can display and alter the value being returned from the function. The basic form of a BUG definition is as follows:

```
(DEFINE (A BUG (DEFN1 . DEFN2)))
```

DEFN1 is a function specification (usually a LAMBDA-expression) which must either be an FLAMBDA-function or have the same number of arguments as the function A. Immediately prior to calling the function A, DEFN1 will be called. If it is an FLAMBDA-function, its dummy argument will be bound to a list of the arguments of A. If it is a LAMBDA or NLAMBDA function, its dummy arguments will be bound to the arguments of A. For the purposes of BUGs, LAMBDA and NLAMBDA functions are identical.

After DEFN1 is called, A will be invoked as if the BUG were not present. DEFN1 does not have the power to alter the arguments sent to A (except, of course, by physical modification of the argument structures), but it does have the power to abort the call entirely. (see Section IV of this manual on Debugging Features).

Upon returning from the function A, DEFN2 is called. DEFN2 may be a LAMBDA or NLAMBDA function of one argument, in which case that argument will be bound to the value returned from A. If DEFN2 is an FLAMBDA, its dummy argument will be bound to a list of the value returned from A. The value returned from DEFN2 will replace the value actually returned from the function A, as the final result of calling A. Thus the writer of BUGs who wishes to pass along the value returned from A must be certain to define DEFN2 to accomplish this.

Note 1: When a BUG is placed on the property list of an atom, and then a new function definition (EXPR OR MACRO) is placed on the same property list, the BUG will be ignored. In other words, BUGs must be the first indicator on a property list in order to be effective. Thus, in a call to DEFINE which defines a function and a BUG for the same atom, the function definition must precede the BUG definition.

Note 2: One or more BUGs appearing with no function definition on the property list of an atom A will generate an error if A is invoked as a function.

Note 3: Multiple BUGs appearing on the property-list of an atom, followed by a function definition, will be treated as "stacked" and invoked in order. The input-bug-functions will be executed from first to last, followed by the function itself, followed by the output-bug-functions, from last to first. The dummy argument of each output-bug-function will be bound to the value returned from the one following it on the property list.

Note 4: If either DEFN1 or DEFN2 is NIL, then that portion of the BUG will be ignored and the function A will be invoked or exited without intervention.

Example: A bug is put on the function COUNT, to trace the entry and exit, and to print out the arguments.

```
(DEFUN COUNT (L N) (COND ((NOT L) N)
                         ((COUNT (CDR L) (ADD1 N)))))

(DEFINE (COUNT BUG ((FLAMBDA (ARGS)
                    (PRINT 'ENTRY-TO-COUNT)
```

HASHFN may be any LISP function which returns a numeric atom as its value, or may be an external routine called from LISP.

iii. Calling External Routines from LISP

LISP/MTS provides the option of calling user-written or library subroutines. The major purpose of this feature is to allow the use of complex numeric function, hash functions, etc., which would be extremely slow if written in LISP.

The basic form used to define external subroutines in LISP is:

```
(DEFINE (FN SUBR (N FILENAME <ENTRY-NAME>)))
```

FN is an atom which will become the LISP name of the external function.

FILENAME is the name of an MTS file from which the external code is to be loaded.

ENTRY-NAME specifies which entry point in an object file, or which subroutine in a library file is to be associated with the LISP function name FN. If no ENTRY-NAME is given for an object file, the default MTS entry point will be used. If no ENTRY-NAME is given for a library file, an error will be generated.

If the ENTRY-NAME given is already in core, then nothing will be loaded, and the LISP function FN will be defined to be the ENTRY-NAME function. This means that ENTRY-NAME must be unique, not only within its own file, but within the entire set of files used in DEFINE statements.

N specifies the type of calling conventions to be used, and must be 0,1,2, or 3.

N=0
: signifies that LISP internal SUBR calling conventions will be used. Any number of arguments may be given, and these may be any LISP structures. This external mode is for the use of systems programmers who might wish to write extensions of the LISP interpreter, and requires familiarity with the internal structures of LISP.

N=1
: signifies FORTRAN function calling conventions, with a floating point return value. Any number of arguments may be given, and they must be numeric atoms. If an argument is a floating-point numeric atom, it will be passed to the function as a double-precision floating point number. (This allows the

user to call both single and double precision functions, although LISP numbers have only single precision significance.) If the argument is an integer numeric atom, it will be passed to the function as a full word integer. (Note that the numeric value of the atom will be passed, and not the atomic structure).

Upon return from the function, Floating Point Register 0 will be treated as a single-precision numeric return value from the function, and a numeric atom will be created with that value, and returned as the value of the external function.

N=2

signifies FORTRAN function calling conventions, with an INTEGER return value. Any number of arguments may be given, and their interpretation will be the same as the N=1 case.

Upon return from the function, General Register 0 will be treated as an integer return value from the function, and a numeric atom will be created with that value, and will be returned as the value of the external function.

N=3

signifies FORTRAN subroutine calling conventions. Any number of numeric arguments may be given, and their interpretation will be the same as the N=1 and N=2 cases. For this type of external function, the arguments may be modified by the function, just as if they were the values of FORTRAN variables.

Upon return from the subroutine, General Register 15 is checked first. If the return code is non-zero, then the value returned from the LISP function will be NIL. If the return code is zero, then a list of the (possibly modified) argument values will be returned as the value of the LISP function. Note that a FORTRAN program which modifies the values of its arguments does not alter the value of any LISP structure. The only effect of the modification is to return some new numeric atom as part of the returned value of the LISP function.

An argument which was originally passed as an integer will be interpreted upon return as an integer. An argument which was originally passed as a floating point number will be interpreted upon return as a single-precision floating point number.

Ex: (DEFINE (DEXP SUBR (1 *LIBRARY DEXP)))

VI. Input/Output

A. Default I/O Operations

In the simplest application of LISP input-output, all input is read from the system input device (SCARDS), and all output is directed to the system output device (SPRINT). I/O is always treated as a stream, with the syntactic boundaries between S-expressions constituting the divisions between I/O operations, rather than physical records. Thus, several S-expressions may be read from one input line or one S-expression may span several input lines. Similarly, the basic print function PRIN1 will "stream" output S-expressions into a single output buffer until it overflows. Then it will be printed, and the current expression being PRIN1ed will be continued as the start of a new buffer.

EX:

```
   (PROGN
     (PRIN1 'THIS)
     (PRIN1 'IS)
     (PRIN1 'A)
     (PRIN1 'TEST:)
     (TAB 35)
     (PRIN1 '"THAT'S ALL")
     (TERPRI)) )
```

is NIL, and the following line will be printed:

```
THIS IS A TEST:                   THAT'S ALL
```

B. I/O Data Types

LISP/MTS provides the option of a more flexible (and more complicated) input/output scheme than the defaults described above. The basic data structures involved in the scheme are: the I/O destination atom, the buffer, and the file.

1. I/O Destination Atoms

An I/O destination atom is a pointer atom whose VALUE is a buffer/file pair to be used in an I/O operation. All of the I/O functions described in the previous section accept such a pair as an optional argument, and if given, the buffer/file pair specified will be used for that operation. Such a buffer/file pair is called an I/O argument, or IOARG.

If an IOARG is given on input, data is read from the specified (rather than the system input) buffer, and if the buffer is used up, a new line is read from the specified (rather than the system input) file. On output, data is printed into the specified (rather than the system output) buffer, and if an overflow occurs (or the operation is a PRINT), data will be printed on the specified (rather than the system output) file.

Specifically, an IOARG (the VALUE of an I/O destination atom) is a dotted-pair (BUFFER . FILE), which may be used to direct input/output operations, and may also be used as a buffer pointer for performing operations on buffers (EXPLODE, etc.). If either component of an IOARG is NIL, then the appropriate system buffer or file will be used. The VALUE of the I/O destination atom LISPIN is the dotted-pair of the default system input buffer and system input file. The VALUE of the I/O destination atom LISPOUT is the dotted-pair of the default system output buffer and system output file. If the user changes the system default buffers or files using the STATUS function (the equivalent of a read or write select operation), he may still have access to the original system IOARGs through LISPIN and LISPOUT.

2. Buffers

A buffer is an atomic structure with a variable PNAME, which is accessed through one or more IOARGs. New buffers may be created and linked to I/O destination atoms by calling the OPEN routine. Buffers are used for input/output, and may also be viewed as character strings.

The maximum size of a buffer is 255 characters.

Any PRINT operation into a buffer will cause a representation of the argument to be placed in the buffer. Any READ operation from a buffer will create and return the LISP structure represented by the next S-expression in the buffer. Options available through the STATUS function allow the user to suppress the insertion of blanks between printed S-expressions, or to intercept the performance of physical I/O when a buffer overflows, and execute some user-written routine instead. The user may also define Read-Macro and Print-Macro atoms.

When a buffer is passed as an argument to a function, it will always be the IOARG whose CAR is the buffer, rather than the buffer itself, which is passed. For example, functions such as EXPLODE, which forms a list of one-character atoms from the characters in a buffer, or GENSYM, which will created an atom whose PNAME is the current contents of the buffer, expect an IOARG, rather than the buffer itself to be passed. The FILE portion of the IOARG will be ignored. Thus, the IOARG also serves as a buffer pointer throughout the system. However, when functions such as READLINE, TAB, and SKIP return buffer pointers, it is the actual buffer structure and not the IOARG which is returned.

The atomic structure of a buffer extends only to its PNAME. Buffers may not be given VALUEs and PLISTs by the user. However, a buffer may be part (or all) of the list-structure argument to PRINT or PRIN1. For printing purposes, a buffer is treated like any other atom, and its PNAME will be inserted into the output buffer.

Ex:

If (PRIN1 (CAR LISPIN) BUF1) appears as an input line under normal conditions of operation, the character string " (PRIN1 (CAR LISPIN) BUF1)" will be placed in the buffer associated with I/O Destination Atom BUF1.

3. Files

The FILE is an internal LISP structure which has no significance to the user except that it serves to direct input and output calls to MTS files and devices. A FILE may reference any MTS file name (MYFILE), device name (*T*, *SINK*), logical unit name (SCARDS), or logical unit number (0 - 9).

Several files can be attached to a single buffer, by creating several IOARGs with the same buffer component. If these IOARGs are used for output, data printed will all go to the same buffer, but if the buffer overflows, the file for that I/O operation will be used as the output file. Similarly, several buffers can be attached to the same file by creating several IOARGs with the same file component. In that case, output from all the attached buffers will be interleaved in the file.

C. Buffer and File Prefix Characters

Any LISP buffer may have a prefix of up to 255 characters, which may be set and unset by calling the STATUS 9 function. The purpose of the buffer prefix is to allow prefix strings to precede output lines. All PRINT operations, including TAB and SKIP, will treat a buffer with an active prefix as though it begins after the prefix.

Note: Prefix characters use up character positions at the beginning of a buffer, and are included in the buffer size limit of 255 characters.

Warning: Since READ operations do not recognize buffer prefixes, a physical read operation into a buffer with a prefix will destroy or replace the prefix.

A file prefix character may be attached to any LISP file by calling the STATUS 14 function. This has the effect of calling the MTS function SETPFX which will cause any input from or output to the terminal or line printer to be prefixed by the prefix character.

Ex: Here is a sample run in which a buffer is created, given a prefix, the prefix is used, and then the prefix is turned off. Lines which are not indented are typed in by the user.

```
* (OPEN (ABUF 132))      ;a buffer is created with length 132.
                         ;ABUF is the I/O destination atom.
                         ;The file portion of the IOARG
                         ;created will be NIL.
>      NIL
* (READ ABUF)            ;causes a line to be read
                         ;from the system input device into ABUF,
                         ;and the first S-expression found
                         ;to be returned as the value of READ.
 THIS IS A TEST          ;here is the input line.
>      THIS
* (STATUS (10 ABUF T))    ;makes the current contents
                         ;of ABUF a prefix.
>      0
* (TERPRI ABUF)          ;this has no effect, since the prefix
                         ;is not treated as buffer contents.
>      NIL
* (PRINT 'PRINT2 ABUF)
  THIS IS A TEST PRINT2
>      PRINT2
* (STATUS (10 ABUF NIL)) ;turns off the prefix.
>      14
* (PRINT 'PRINT3 ABUF)
  THIS IS A TEST
  PRINT3
>      PRINT3
```

```
* (PRINT 'PRINT2 ABUF)
  PRINT2
>        PRINT2
```

D. Buffer Overflow Interception

The user may, by including an optional argument, attach a read intercept function or a print intercept function to an I/O call. The argument must have as its value a function which takes one argument. If a read intercept function is included, on any attempt to do a physical read into that buffer, the intercept function will be called first. The IOARG for that READ will be passed to the intercept function as its argument. If a write intercept function is specifed in a PRINT, PRIN1, or TERPRI call, on any attempt to do a physical write from the buffer, the intercept function will be called first. The IOARG for that PRINT operation is passed as the argument to the intercept function.

Upon return from an intercept function, the LISP system will complete the I/O operation.

E. End-of-file Processing

Each file has an EOF function, which will be called if an end-of-file is encountered while reading from that file. An EOF function may be attached to a file by calling STATUS 12.

An EOF function must be a function of one argument. When the function is called, the IOARG for the READ operation will be passed to it.

All files initially use the system default EOF function, called EOF, which causes the file to be closed. Whenever a file is closed, it is changed to reference *MSOURCE*. An end-of-file encountered on *MSOURCE* will cause the user to be asked if he wishes to continue if the run is interactive. In batch mode, an end-of-file on *MSOURCE* causes immediate termination of execution. The value of the function EOF is NIL.

The action which will be taken on return from an EOF function is determined by the value returned. If the value is non-NIL, the READ is aborted, and that value is returned as the value of READ. If the value returned from the EOF function is NIL, the READ will be tried again.

F. READMACRO and PRINTMACRO Functions

It is possible for the LISP user to define functions which will be called whenever a particular atom or character is encountered in the input stream, or whenever a particular atom appears in the output stream. A Readmacro or Printmacro function must be a function with one dummy argument. An atom is defined as a Readmacro or Printmacro by calling STATUS functions 2, 3, or 4 with the appropriate arguments.

1. Immediate READMACRO Atoms

(STATUS (2 HIT T)) defines the atom HIT as an immediate Readmacro. If HIT is encountered in the input stream during a READ operation, the function associated with HIT will be invoked immediately. The function HIT must be a function of 1 argument (the IOARG).

Upon return from the HIT function, the following action will be taken:

> If the value returned from HIT is an atom, then HIT will simply be "spliced out" of the input stream, and the READ will continue.
>
> If the value returned from HIT is a list, then the elements of that list will be "spliced in" to the input stream in place of HIT, and the READ will continue.

The Readmacro function may itself call READ, in which case the S-expression immediately following the atom HIT in the input stream will be returned.

Ex:

```
*
* (DEFUN HIT (X)
*    (COND ((ATOM (SETQ X (READ)))
*           (LIST (LIST X 'HIT)))
*          ((LIST (MAPCAN '(LAMBDA (A)
*                                    (LIST A 'HIT))
*                          X>
>    HIT
* (STATUS (2 HIT T))
>    NIL
 '(A B C HIT (D E F) G)
    (A B C (D HIT E HIT F HIT) G)
 '(A B C HIT D E F)
    (A B C (D HIT) E F)
```

2. Delayed READMACRO Atoms

(STATUS (3 HIT T)) defines the atom HIT as a delayed Readmacro. If HIT is encountered in the input stream during a READ operation, the function associated with HIT will be invoked, but not until after the current READ has been completed.

Thus, if the HIT function calls READ, it cannot read part of the "current" S-expression, but will return the S-expression following it.

Upon return from the HIT function, the value returned (if it is a list) will be "spliced in" to the original S-expression which was read, at the point where the Readmacro atom was encountered. If the value returned is an atom, then the Readmacro atom will merely be "spliced out" of the original S-expression.

Ex: Using the same definition of the Readmacro HIT:

```
*  (STATUS (3 HIT T))
>     NIL
* '(A B C HIT D E F)
* (SPLICE THIS)
>      (A B C (SPLICE HIT THIS HIT) D E F)
```

Note for Readmacro Users: The ' feature in LISP/MTS is a substitution (not a Readmacro), and does not involve an extra call to READ.

3. PRINTMACRO Atoms

Printmacros have been implemented slightly differently from Readmacros. (STATUS (4 ATM T)) will define ATM as a Printmacro atom. Whenever an attempt is made to print a <u>list</u> whose CAR is ATM, the Printmacro function will be invoked. Upon return from the Printmacro function, its value will determine what action is to be taken. If the value returned is NIL, the list (whose CAR is ATM), will be printed normally, as if no Printmacro were there. If the value returned is non-NIL, printing resumes, ignoring the list passed to the Printmacro function. (It is assumed that the Printmacro function printed the list.)

Printmacros are not defined as Exprs, but as PMACROs. They must be functions of 1 argument which will be bound to a CONS cell whose CAR is the list which was to be printed, and whose CDR is the IOARG.

Ex: To re-insert the character ' for the QUOTE function,

```
*
* (DEFPROP QUOTE PMACRO
     (LAMBDA (X)
```

```
*              (COND ((EQ (LENGTH (CAR X)) 2)
*                     (PRIN1 '"'" (CDR X) 2)
*                     (PRIN1 (CADAR X) (CDR X) 2)
*                     T>
>     QUOTE
* (STATUS (4 QUOTE T))
>     NIL
* '(X 'Y '(A B) (QUOTE) QUOTE 'QUOTE)
>     (X 'Y '(A B) (QUOTE) QUOTE 'QUOTE)
* '( (QUOTE) (QUOTE A) (QUOTE A B) '''D)
>     ((QUOTE) 'A (QUOTE A B) '''D)
*
```

4. The READMACRO Character Characteristic

A single-character immediate or delayed Readmacro atom may be given the additional effect of a READMACRO character by altering its disposition in the READ scan table. (See STATUS Code 5 description). A READMACRO character need not occur as an atom, but may occur at the beginning of any of any S-expression. However, a READMACRO character which is strictly embedded in an atom, or which occurs at the end of an atom, will not be recognized. Ex: Re-define the character Q as a Readmacro equivalent to the system ' substitution function.

```
(DEFUN Q (X) (LIST (LIST 'QUOTE (READ))))
(STATUS (5 Q 28) (2 Q T))
QA = A
Q(A B C) = (A B C)
QQ(A B C) = (QUOTE (A B C))
```

G. The FLAGS Argument of I/O Functions

Many of the I/O functions contain a FLAGS argument which specifies certain conditions on the operation. This argument is an integer which is the sum of all specifications. If a buffer intercept function is to be included, then this number must be present (if no special processing is to take place, then 0 should be used).

The FLAGS specifications have the following meaning:

1	No Readmacro Processing
2	No Spacing Between s-expressions On Output
4	Place Double Quotes Around Special Atoms
8	Output In Terse Mode (one line only)

The FLAGS argument will have as it value the sum of the desired specificaions. For example, if No Macro Processing and Double Quoting is desired, then the value if FLAGS will be 1 + 4 or 5.

H. Input/Output Function Descriptions

1. (OPEN (IODA BUFFER <FILE>) . . . (IODA BUFFER <FILE>))

Establishes any number of new I/O destination atoms. IODA must be an atom, and its VALUE will be set to the new buffer-file pair which is created. BUFFER must be an integer between 1 and 255, or a previously defined I/O destination atom, or NIL. If it is an integer, a new buffer will be created with that initial size. If it is an I/O destination atom, the buffer attached to that atom will be used. If it is NIL, then the buffer portion of the IOARG created will be NIL, and the system input and output buffers will be used whenever that IOARG is specified in an I/O call.

FILE must be an atom, a list of a single atom, or a previously defined I/O destination atom. If it is a (non IODA) atom, then that atom is interpreted as an MTS file or device name, e. g. MYFILE, *T*. If it is a list of a single atom, then that atom is interpreted as a logical unit number or name, e.g. (3), (SCARDS). If FILE is a previously created I/O destination atom, then the FILE portion of that atom will be used. (This feature allows the user to associate multiple buffers with one file). If the FILE argument is omitted, then the FILE portion of the IOARG will be NIL. When the IOARG is specified in an I/O call, the system default file will be used.

OPEN is a special-form type function (an FSUBR), which takes its arguments unevaluated. The value returned from OPEN is NIL.

2. (EOF IOARG)

closes the file associated with its argument and re-associates it to *MSOURCE*. An end-of-file on *MSOURCE* will cause a CONTINUE? prompt in interactive mode, and termination of execution in batch mode.

The function CLOSE has the same effect.

3. (READ <IOARG <FLAGS <INTERCEPT>>>)

The READ function takes a number of optional arguments. If any optional argument is given, all preceding ones must also be given.

READ causes the next S-expression in the current buffer to be read (beginning with the next atom or left parenthesis), and the corresponding LISP structure to be returned as the value of READ. If the current buffer is exhausted, a new line is read from the current file, and the operation continues.

IOARG identifies the buffer-file pair to be used for the READ. If IOARG is not given, or is NIL, the system input buffer-file pair will be used.

FLAGS, if included, specifies the special operation (possible disabling of Readmacros).

INTERCEPT must evaluate to a function of one argument which is the buffer intercept function.

. (READCH <IOARG <FLAGS <INTERCEPT>>>)

READCH works just like READ, except that each character in the buffer is treated as a separate S-expression, and is returned as a one-character atom. Commas, parentheses, periods, double-quotes, blanks, and other special characters are treated like any other characters, and simply formed into single-character atoms.

WARNING: The user should beware of single-character Read-Macros which will be activated by READCH if the character appears, even incorporated in a character string. Similarly, multiple-character Read-Macros cannot be activated by READCH. 8.

5. (READLINE <IOARG <FLAGS <INTERCEPT>>>)

READLINE causes a new line to be read into the current buffer. The previous contents of the BUFFER ARE DESTROYED.

IOARG, if given, identifies the buffer-file pair to be used for the READ. If IOARG is not given or is NIL, the system input buffer-file pair will be used. 9.

6. (PRINT S <IOARG <FLAGS <INTERCEPT>>>)

PRINT takes three optional arguments. If an optional argument is given, the preceding arguments must also be given.

S is the S-expression which is to be printed. PRINT will perform a TERPRI, will print the expression into the current buffer, and will TERPRI again. The value returned from PRINT is S.

IOARG identifies the buffer-file pair for the print operations. If IOARG is not given, or is NIL, the system output buffer-file pair will be used.

FLAGS specifies what type of special processing is to take place. 10.

INTERCEPT is the optional buffer intercept fuction. INTERCEPT must evaluate to a function of one argument.

7. (PRIN1 S <IOARG <FLAGS <INTERCEPT>>>)

PRIN1 simply places the print-name of S in the current buffer, after any previous contents of the buffer. If the buffer overflows, its contents are printed on the current file, and the operation continues. The arguments of PRIN1 have the same meaning as those of PRINT.

8. (TERPRI <IOARG <FLAGS <INTERCEPT>>>)

TERPRI causes the contents (if any) of the current buffer to be printed out in the current file. If the buffer is empty, TERPRI does nothing. The value of TERPRI is normally NIL, however, if the print operation is intercepted, the value returned from the intercept function will be passed back as the value of TERPRI.

The IOARG and FLAGS arguments have the same meaning as they do for PRINT.

9. (TAB N <IOARG <FILL>>)

TAB causes a tab operation to position N in the current buffer. (The first position in a buffer is 1; thus (TAB 1) is a way to clear a buffer without printing it). If the buffer has a prefix, TAB operates relative to the prefix. If N is non-positive, or larger than the buffer size, an error is generated.

IOARG identifies the current buffer for the TAB operation. If IOARG is not given, or is NIL, then the system output buffer is used. The file portion of IOARG is ignored.

FILL, if given, must be an atom or a buffer pointer (IOARG). The PNAME of FILL will be used as a filler for any positions skipped during a TAB operation to the right.

10. (SKIP N <IOARG <FILL>>)

SKIP causes a skip operation N spaces to the right. If N is negative, the skip will be to the left. An attempt to SKIP outside the buffer will generate an error.

IOARG identifies the current buffer for the skip operation. If IOARG is not given, or is NIL, then the system output buffer is used. The file portion of IOARG is ignored.

FILL, if given, must be an atom or an buffer pointer (IOARG). The PNAME of FILL will be used as a filler for any positions skipped during a SKIP operation to the right.

Note: TAB and SKIP affect the value of the buffer length for output only. These routines cannot be used for the purpose of skipping around in a buffer to READ various positions.

VII. Error and Debugging Functions

A. Error Atoms, Forms, and Expressions

There are 39 different errors that are recognized by the LISP system. When an error of type N occurs, the error message for that type becomes the "current" error message, the expression which generated the error (e.g., the illegal argument) becomes the "current" error expression, and the error form associated with that type is evaluated. After the error form is evaluated, LISP is re-started at the top level.

The error form for an error number is accessed through an atom, called the error atom. A call to the STATUS 1 function will associate an error number with a given atom. From that time on, whenever that error type occurs, the VALUE of that atom will be used as the error form.

At present, there are three pre-defined error atoms within the LISP system. The atom *ATTN* is the error atom for error number 1, which occurs whenever an attention interrupt is generated. The atom *PGNT* is the error atom for error number 0, which occurs whenever a non-numeric program interrupt occurs. The atom *ERR* is the error atom for all other errors.

ATTN,*ERR*, and *PGNT* are initially set to the form (DUMP). See the description of DUMP below.

B. System Error IOARGs

We have seen that there are initially two buffers maintained by the LISP system, the system input and output buffers, and the the two IOARGs LISPIN and LISPOUT initially point to these buffers (in their paired form with the system I/O files). There are also two system error buffers maintained by the LISP system, and the two IOARGs ERRIN and ERROUT initially point to these buffers (in their paired form with the system error I/O files).

The system default error input file is GUSER, and the default error output file is SERCOM.

Whenever a BREAK loop is entered, the system error IOARGs are used instead of the normal IOARGs for the READ-EVAL-PRINT loop and for all user-generated I/O which does not specify its own IOARGs.

C. Error Functions

1. (BREAK <S>)

Calling BREAK causes the system to enter a break loop. A break loop is a READ-EVAL-PRINT loop identical to the top-level loop of LISP, except that the ERRIN and ERROUT buffers and files are used for reading and printing respectively. After exiting from the BREAK loop, execution continues normally.

S is an optional argument which, if given, will be evaluated before the BREAK loop is entered.

The way to exit from a break loop is to evaluate NIL at the break level (i.e., just type in NIL). The value returned from BREAK is always NIL.

Note: The file Prefix characters for LISPIN and LISPOUT are * and > respectively. The file prefix characters for ERRIN and ERROUT are ? and + respectively. Thus, the user can easily tell whether or not he is in a break loop.

2. (DUMP <N <SW>>)

DUMP is the basic system dumping and trace-back program. DUMP can be called in two modes. The first mode occurs when no second argument is given. In this mode, the status of the rightmost eight bits of N indicate whether various error recovery actions should be performed. The code values described below should be added together to specify the actions desired. (The numbers in parentheses after the action specification indicate the relative order of performance of the various actions). If the first argument is omitted, the default value is 7.

Code Value	Action
1	Print current error message and expression which generated the error (1).
2	Print a backtrace of EVAL forms. The number of levels to be printed is determined by the system backtrace number - STATUS Code 26 (5).
4	Call BREAK (6).
8	Print PSW and contents of General Registers (2).
16	Dump 32 bytes of core starting 16 bytes before PSW location (3).
32	Dump 32 bytes of LISP stack data (for system programmer) (4).

There are three parameters controlling the backtrace produced by DUMP which may be altered by calling STATUS. The first is STATUS 30, the terse mode switch. Ordinarily, only the first output line of each expression is printed in order to eliminate long trace-backs. This switch may be reset to give a complete trace-back. The second parameter which may be controlled is STATUS 27, which controls the printing of arguments. Ordinarily the CAR (function specification), and CDR (argument list) of each form in the backtrace is printed. The user may, by changing this switch, suppress the printing of the argument lists. A third parameter, accessed by STATUS 26, controls the number of forms which will be backtraced. The default is 3.

(DUMP 0) is a special code which causes a full EVAL form back-trace to be printed.

Note: DUMP codes (other than 1 and 4) begin the dump at the location of the most recent error block on the stack. These DUMP codes should only be used within an error block.

The second mode of DUMP operation occurs when a SW argument is given. If SW is an integer, then that number of bytes, starting at address N, will be dumped in hexadecimal. (SW is rounded to a multiple of 16). If SW is not a number, then N is assumed to be the address of some LISP structure, and that structure is printed. Note that the number N is normally treated as a DECIMAL number; this can be changed thru status 24.

Note: The user can very easily generate a type 0 error (program interrupt) by asking DUMP to print a LISP structure, and giving it an address which is not a LISP structure. This will not do any harm, however.

DUMP always returns NIL.

3. (UNEVAL <N,S1> <<T,S2>>)

UNEVAL allows the user to look back on the system stack and trace the path that was followed by the system to get to its current position. It may be used from an error form or break loop to restart from any given point.

Each time EVAL is called internally, a block of information called an EVAL block is stored on the stack. The EVAL block contains the form which was to be EVALed, plus all revelant information needed to restart at that level. When the first argument to UNEVAL is an integer, it refers to the Nth previous EVAL block on the stack. 4.

For example, if you are in a break loop, and you type in (UNEVAL 1), the last form sent to EVAL will be returned. This will be (BREAK) if you entered the BREAK loop by calling BREAK directly, or (DUMP N) if the BREAK loop was entered as part of a DUMP operation. (UNEVAL ignores its own EVAL block).

If the first argument to UNEVAL is some expression S which is not an integer, then it refers to the most recent call to EVAL for which the CAR of the form to be EVALed was EQUAL to S. For example, if you evaluate (UNEVAL 'ASSOC), UNEVAL will return the most recent outstanding EVAL-form which has ASSOC as its CAR.

If the first argument to UNEVAL is a number larger than the current EVAL depth, or if it is a structure which is not EQUAL to any function specification on the stack, and the second argument is present, an error is generated. If the first argument to UNEVAL is a negative number, UNEVAL interprets this as a reference from the top-level form, and either returns that form, or unbinds to it (depending on the value of the second argument).

Once UNEVAL identifies the correct EVAL block, the second argument determines the action to be taken. If no second argument is given, UNEVAL returns the form that was sent to EVAL at that level. Thus, a call to UNEVAL with no second argument does NOT change the current level of execution. If the second argument to UNEVAL is T, then execution is re-started at that level. Thus, if you evaluate (UNEVAL 'ASSOC T), the system will exit from its current level, unbind all bindings down to the last time ASSOC was called, and re-start the call to ASSOC.

If the second argument to UNEVAL is anything other than T, then execution is re-started at the indicated level, but the form given as the second argument is substituted for the form which was originally sent to EVAL. Thus, if you evaluate (UNEVAL 4 '(APPEND X Y)), the system will unbind to the 4th previous EVAL block, and will then proceed to evaluate (APPEND X Y) in place of the form which was originally given.

Note: The user should be aware that unbinding to a previous LISP level will not restore altered data structures, property lists, or VALUEs changed via SET or SETQ.

4. (DISPLAY <N,S1> <B,F,L> <A>)

The DISPLAY function allows the user to locate a position on the stack with reference to an EVAL block, and then display one of the following:

a. The first bound value of a particular atom A, that occurred after that EVAL block was created.

b. If the EVAL block is a COND, a PROG, a SELECT, a LAMBDA-expression, or any function specification which eventually produced a LAMBDA-expression to be applied, then DISPLAY will return the next COND or SELECT expression to be processed, the next PROG expression to be EVALed, or the next sub-form of the LAMBDA to be EVALed.

c. The level in the stack (a negative number, counting from the top level).

d. The value ARG would return at that eval block.

The first argument to DISPLAY has the same significance as the first argument of UNEVAL. If it is an integer, it refers to an EVAL block N before the current block. If it is not an integer, it refers to the most recent EVAL block which has S1 as its CAR. As in UNEVAL, a negative integer references the top-level form. If the EVAL block referenced does not exist, NIL is returned.

The second argument to DISPLAY is either B, F, L or a number: B for binding (option a. above) and F for form (option b. above), L for level (option c. above), or a number which is taken as the first argument to ARG (option d. above).

7.

The third argument to DISPLAY is given whenever the second argument is B or a number. It is the atom whose binding is to be found. If A was never bound after the EVAL block referenced was created, then the current VALUE of A is returned. If a binding of A is found, then the value stored on the stack will be returned. (This is the old VALUE of A, that is, the VALUE which was saved away to be restored on exit from a PROG or LAMBDA). If the second argument was a number, the third is the optional second argument to ARG (the dummy variable name).

Note: In DISPLAY mode F, it is possible to find a COND, SELECT, PROG, or LAMBDA block on the stack which is not yet being executed. This will occur if the user interrupts during the binding of the PROG-variables, or during evaluation of the arguments of a LAMBDA function. In this case, there is no "next form" defined for that block, and an error type 37 will be generated.

DISPLAY is an N-type function, and its arguments are not EVALed.

5. (MODIFY <N,S1> <B,F> <A> S2)

The MODIFY function allows the user to modify one of the bindings or expressions accessible from DISPLAY

The arguments of MODIFY have the same significance as those of DISPLAY, except that S2 will replace the saved VALUE of A (in B mode) or the next expression to be processed (in F mode). MODIFY, like DISPLAY is an N-
type function. However, S2 will be EVALed and its value 8.
will be used as the replacement binding or expression. The value returned from MODIFY is the value of S2.

6. (ERR S)

This function generates a type 15 error, with S treated as the expression which generated the error (error expression). In addition, the atom ERR is set to
S. 9.

7. (RES <N>)

RES is the LISP internal restart function, and may be called to restart after an attention interrupt or STEP error call, or a timer interrupt. These interrupts are processed by LISP as follows: A single attention interrupt will cause a flag to be set, and when LISP reaches a state from which it can be restarted, the interrupt will be processed, and the error form associated with a type 1 error will be EVALed.

If a second attention interrupt is issued before the first one is processed, it will be recognized immediately and the error form will be EVALed. However, when this occurs, no restart is possible.

Assuming that only one interrupt has been issued, a call to RES with no arguments will cause exeuction to be resumed at the point where it was interrupted. If the argument N is given, it must be a positive integer, and the Nth previous outstanding interrupt will be restarted.

TIMER interrupts are always deferred until the system reaches a state from which it can be restarted. However, upon receiving a TIMER interrupt, the system immediately prints a comment on *MSINK* acknowledging the TIMER interrupt. At that point, the user may interrupt if he so desires. If an attention interrupt is issued while a timer interrupt is still pending, it will be processed immediately (and no restart will be possible).

8. (TRACE FN1 . . . FNn)

The function TRACE turns on an indicator on each atom FN1 . . . FNn which will be detected which will be detected when that atom's function is EVALed. The list of arguments will be printed on entry and the value will be printed on exit. FN may have an EXPR, SUBR, or any type of function.

9. (UNTRACE FN1 . . . FNn)

Untrace undoes the flagging done by the function TRACE.

10. (STEP N)

This function can be used to step through the execution of a form at a controlled rate. N specifies the number of forms which will be evaluated, after which an error (error number 24) will be generated. Calling (STEP 0) will turn off the step process without causing an error. Thus, for example, (STEP 1) will cause an error after executing the next form.

Note -- Any LISP error will automatically turn off the STEP function, as will a return to top-level LISP.

D. Error Messages

Following is a list of the errors recognized by the system. Each type of error sets up an error message and an error expression, which may be obtained (or altered) by calling STATUS, or which may be printed by calling DUMP. Since the default error form for all errors is (DUMP), which includes a printout of the current error message and errpr expression, these will normally be printed every time an error occurs. Error types 1-7 do not generate an error expression. Errors type 8 and above use as an error expression the argument which caused the error, unless otherwise noted.

Code	Meaning
0	Program Interrupt
1	Attention Interrupt
2	Timer Interrupt (See the Section on Control functions for a description of the TIMER function).
3	A function was called with too few arguments.
4	A function was called with too many arguments.
5	Numeric operation failure - numeric overflow, division by 0, etc.
6	An array specification contained the wrong number of subscripts.
7	PVAL of SUBR indicator not a SUBR.
8	A list was required as an argument, but something else was given.

9 An atom was required as an argument, but something else was given.

10 A numeric atom was required as an argument, but something else was given.

11 An integer atom was required as an argument, but something else was given.

12 A buffer (IOARG) was required as an argument, but something else was given.

13 A file (IOARG) was required as an argument, but something else was given.

14 An array name was required as an argument, but something else was given.

15 A call to the ERR function has occurred.

16 Attempt to EVAL an atom which is undefined.

17 Undefined function - the CAR of the form being EVALed is neither a valid function nor a Lambda expression.

18 Syntax error detected by READ. The error expression is the contents of the READ buffer.

19 Attempt to OPEN a buffer with a size which is non-positive or greater than 255

20 Invalid request code number in a call to STATUS.

21 Invalid error number given in a STATUS Code 1 call.

22 Attempt to set a "get-only" STATUS Code. (See the STATUS function).

23 Attempt to re-set a buffer to a size less than its current contents.

24 STEP counting completed.

25 A sytax error in a PROG. The list of PROG variables was not a list of atoms. The error expression is the PROG variable list which was given.

26 An atomic argument to GO was not the name of any current GO-label.

27 ARG was called where there is no outstanding No-spread function, or ARG was called with two arguments, and the second argument is not the name of any outstanding No-

spread dummy argument.

28 ARG was called with a number which is non-positive or greater than the number of arguments passed to the No-spread function.

29 An attempt to DEFINE an external SUBR with a type which is not defined.

30 LISP couldn't find or couldn't load an external routine which was DEFINEd. The error expression is the file name or entry point name which was given.

31 A subscript in an array specification was non-positive or exceeded the limits of that subscript position.

32 GETWORLD was called with an argument which is not valid ticket.

33 A call to RES was attempted when there was no outstanding attention or timer interrupt at that level, or the attention interrupt was an immediate (double) attention.

34 An attempt to call CHECKPOINT which was not at the top level of LISP, or a call to CHECKPOINT or RESTORE which did not specify a sequential file, or a call to RESTORE which specified a file which was not produced by CHECKPOINT.

35 An attempt to expand a Readmacro which is defined as both immediate and delayed.

36 A call to UNEVAL, DISPLAY, or MODIFY tried to reference an EVAL block which did not exist.

37 A call to DISPLAY or MODIFY which specified F mode identified an EVAL block which was not an executing PROG, COND, SELECT, or function with a LAMBDA definition.

38 More than 100 left super brackets were encountered.

39 The second parameter to LDIFF was not EQ to some number of CDR's of the first parameter.

VIII. Special System Functions

. The STATUS Function

The STATUS function is used for two purposes - to get and to et the values of system switches and parameters. There are two ypes of status call. One which merely interrogates the system nd returns the value of a system parameter, and one which upplies a value which is to replace the system parameter.

The various system parameters are identified by STATUS umbers. Numbers 1 through 30 are used to get and set parameters ssociated with buffers, files, arrays, and atoms. To get one of hese parameter values, the argument to STATUS will be of the orm:

```
(STATUS-NUMBER  NAME)
```

here NAME is the name of the appropriate buffer, file, array, or tom.

To set one of these parameters, the argument to STATUS will e of the form:

```
(STATUS-NUMBER NAME VALUE)
```

here VALUE is the new value for the parameter.

STATUS numbers 17 and above are used for general system witches and parameters. To get and set these parameters, the rgument to STATUS will be of the form:

```
STATUS-NUMBER                  to get, and
(STATUS-NUMBER VALUE)          to set.
```

Whether getting or setting a system parameter value, the evious value will be returned from STATUS. If more than one gument to STATUS is given, a list of the previous values of all e parameters used in the call will be returned.

Note: In a call to STATUS, the STATUS number parameter may any atom, and its VALUE (which must be a legal STATUS Code) ll be used as the actual STATUS Code. This allows mnemonic finitions to be given to STATUS Codes, e. g. (STATUS (SETPFX NIL)), where the VALUE of SETPFX is 8.

1. Type I STATUS Codes

This group of STATUS functions are for Buffer, File, Array, and Atom Characteristics.

Code Meaning

1 This status number is used to get or set the error atom associated with a particular error number. (See the Section on Error Recovery for an explanation of the error atom). The get form is (STATUS (1 N)), which will return the error atom associated with error number N. The set form is (STATUS (1 N A)), in which case A will be the new error atom associated with error number N. From that time on, a Type N error will cause the VALUE of A to be used as the error form.

2 This STATUS number is used to get or set the immediate readmacro switch for an atom. Its argument must be an atom. If the readmacro switch is NIL, then the atom will not be recognized as an immediate readmacro. If the switch is non-NIL, then whenever the atom appears as part of an S-expression read in, it will be treated as an immediate readmacro as described in the Section on I/O routines. The initial value of this parameter for all atoms is NIL.

3 This STATUS number is used to get or set the delayed readmacro switch for an atom. It has the same significance as the immediate readmacro switch, except that if this switch is on, whenever the atom appears as part of an S-expression read in, it will be treated as a delayed readmacro.

4 This STATUS number is used to get or set the printmacro switch for an atom. It has the same significance as the readmacro switches, except that if this switch is on whenever the atom is printed into a buffer, it will be treated as a printmacro as described in the Section on I/O routines.

5 This STATUS code is used to get or set the disposition of characters in the READ scan table. It allows the user to alter LISP syntax. The argument must be a literal atom. The parameter value given will replace the scan table value for the first character of that atom. The legal scan table values, and their significance to READ, are as follows:

0 Insignificant (non-printing) characters.
4 Left parenthesis "("
8 Right Parenthesis ")"

12	End of line (including semi-colon).
16	Period: dotted-pair or number.
20	Plus sign "+": beginning of a number.
24	Minus sign "-": beginning of a number.
28	Single character atom. (For Readmacro characters).
32	Quote character. Special processing.
36	Number starter (0 - 9).
40	Literal starter. (A-Z, etc.)
44	Double-quote char. Special processing.
48	Right super parenthesis ">"
52	Left super parenthesis "<"

6 This STATUS code is used to get or set the disposition of characters in the READ literal break table. The argument given must be a literal atom. The parameter value given will replace the break table value for the first character of that atom. The break table values are:

0	May be part of a literal atom's PNAME.
1	Break character - end of literal PNAME.

7 This STATUS code is used to get or set the disposition of characters in the READ number break table. The argument given must be a literal atom. The parameter value given will replace the break table value for the first character of that atom. The number break table values are as follows:

0	Numeral (0-9)
1	Normal end of a number. (Blank, comma, end-of-line, etc.).
2	Floating-point Period.
3	Hexadecimal digit (A-F).
4	Neither a break character nor part of a number. Back up and process this atom as a literal atom.

Note: Codes 0,2, and 3 must be used only with the characters listed after them. Attempts to do numeric conversion after improper use of these codes will generate numeric exceptions.

8 This STATUS number is used to get the number of dimensions of an array. Its argument must be an array name.

9 This STATUS number is used to get or set the size of a buffer (effectively the right margin). The buffer size includes the buffer prefix (if any), and may not exceed 255.

10 This STATUS number is used to get or set the buffer prefix characteristic for a buffer. Evaluating (STATUS (10 IODA T)) freezes the current contents of the buffer associated with IODA as a prefix. Evaluating (STATUS (10 IODA NIL)) releases the prefix. At that point, the prefix will be treated as the contents of the buffer, and will appear at the beginning of the next output line, unless a (TAB 1) or (TERPRI) is performed to get rid of it.

11 This status number is used to get or set the current READ pointer for a buffer. The argument given must be an I/O destination atom. The value of this parameter is not computed relative to any prefix which may exist. It is not affected by doing print operations into the buffer, but it is re-set to 0 whenever a TERPRI or a 2.
physical write operation is performed. A TAB or SKIP to a smaller number will reset the pointer to the smaller number.

12 This STATUS number is used to get or set the default EOF function for a LISP file. The argument given must be an I/O destination atom. If an end-of-file is encountered on a read operation from the file, the EOF function will be invoked, unless it has been explicitly overridden in the call to READ. For a description of the form of the EOF function and the significance of the value returned from it, see the Section on I/O functions. The initial value of this parameter for all files is the system function EOF.

13 This STATUS number is used to get or set the echo characteristic for a LISP file. The argument given must be an I/O destination atom. If the parameter value is non-NIL, all input read from the file will be echoed on *MSINK*. If the value is NIL, echoing will not occur. The global echo switch (STATUS Code 31) overrides the individual file switches if the global switch is non-NIL. Otherwise, the individual file switches control echoing. The initial value of this parameter for all files is NIL.

14 This STATUS number is used to get or set the file prefix character for a LISP file. The argument given must be an I/O destination atom. The parameter must be a literal atom, whose first character will be used as the file prefix character for the file. The value returned will be an integer between 0 and 255, which represents the byte value of the prefix character.

15 This STATUS number is used to get or set the line number for a LISP file. The argument given must be an I/O Destination atom. The parameter value must be an

integer atom which represents the line number parameter to be used in the next I/O operation involving the file.

16 This STATUS number is used to get or set the modifier word for a LISP file. The argument given must be an I/O destination atom. The parameter value must be an integer atom which represents the modifier word to be used in all subsequent I/O operations involving the file (that is, until this parameter is changed). See the MTS Manual, Volume 3, for a description of the significance of modifier values. The initial value of this parameter for all files is 0.

2. Type II STATUS Codes

These STATUS functions access System switches.

Code	Meaning
17	Bytes of freespace currently allocated. (Get only).
18	Number of bytes currently allocated to stack. (Get only).
20	System standard input IOARG. Initially set to the dotted pair of the system input buffer (size 255) and SCARDS.
21	System standard output IOARG. Initially set to the dotted- pair of the system output buffer (size 70) and SPRINT.
22	System error input IOARG. Used in BREAK loops in place of standard input IOARG. Initially set to the dotted-pair of the system error input buffer (size 255) and GUSER.
23	System error output IOARG. Used in BREAK loops in place of standard output IOARG. Intially set to the dotted-pair of the system error output buffer (size 70) and SERCOM.
24	Input number base (10, 16, or 0). 0 signifies no numerics. Initially 10.
25	Output number base (10 or 16). Initially 10.
26	Number of levels of forms to print on EVAL form backtrace. (0 = none, -1 = all). Default is 3.

27 Trace-back argument switch. 0= print only function specifications on EVAL form trace-back. >0 = print function specifications and argument list. Initially 1.

28 Most recent expression which generated an error. (Get only).

29 Error number of most recent error. (Get only).

30 Only one line of each backtrace form is printed.

31 Global switch for echoing input lines on *MSINK*. T = echo, NIL = do not echo. Initially NIL.

32 System message switch.
0 = no mesages.
1 = Print Garbage Collector messages.
2 = Print Checkpoint message.
4 = print function redefinition messages
default = 7. (print all)

33 Batch/Terminal switch.
4 = batch
0 = interactive.

34 Interrupt trap switch. Initially 0 (all traps on).
1 = Disable program interrupt trap.
2 = Disable attention interrupt trap.
3 = Disable both.

35 Value of STEP counter.

36 Value of GENSYM counter.

37 Initialization call for TIME (Set form only). Automatically initialized at the start of each run.

38 CPU time used, relative to previous initialization. (Milliseconds, get only).

39 Elapsed time, relative to previous initialization. (Milliseconds, get only).

40 Supervisor state time, relative to initialization. (Timer units, get only).

41 Problem state time, relative to initialization. (Timer units, get only).

42 Time of day. Returns literal atom: AA:BB:CC, where AA = hour, BB = minutes, CC = Seconds. (Get only). Note: The atom returned is not on the OBJECT LIST.

43 Date. Returns a literal atom of the form "MMM DD, YYYY", where MMM = month, DD = day, YYYY = year. (Get only). The atom is not put on the OBLIST.

44 CHECKPOINT restore switch. 0 = exit after CHECKPOINT. 1 = automatic RESTORE after CHECKPOINT. Initially 1.

45 Function record switch. 1 = save list of all functions on atom *FNS*. 0 = don't. Default is 0.

46 ID. Returns the user's ID as a literal atom (get only). The atom is not put on the OBLIST.

3. Direct Core Modification

This special STATUS code permits the user to alter up to 7 consecutive bytes of core to any value. Obviously, the user does so at his own risk.

A1 must be an atom, whose VALUE is a numeric atom. That number is the first address which will be modified.

A2 is an I/O destination atom, whose associated buffer or a literal atom whose PNAME contains the data to be inserted in core, starting at address A. The first character in the PNAME must be the character X. It must be followed by an even number of hexadecimal digits, up to a maximum of 14, representing half that number of bytes to be modified.

```
    EX: (SETQ MODA (ADDRESS 'ZAP))
             (STATUS (0 MODA TBUF))
```

If the buffer TBUF contains the characters X00000000, then the VALUE of the atom ZAP would be destroyed. An attempt to evaluate (CAR ZAP) would generate a program interrupt.

B. The Garbage Collector

This section is included only to mention that there is a garbage collection routine in the LISP system which is activated when a job runs out of space to create new LISP structures. The garbage collector releases space which is occupied by unreferenced structures, allocates more space if necessary (controlled by STATUS Codes 33 and 34), and prompts the user if the maximum allowable space is exhausted.

The user may optionally receive a message at the end of each garbage collection indicating the type of collection that occurred (relevant to the programmer but not the user), the number of LISP cells "collected", the amount of additional space allocated, and the current depth of the stack.

It should be noted that attention interrupts which occur during a garbage collection are deferred until immediately after the garbage collection is completed.

1. (RECLAIM)

   This function forces a garbage collection to occur and returns as value the number of cells collected.

C. CHECKPOINT and RESTORE

1. (CHECKPOINT FILE <S>)

2. (RESTORE FILE)

   CHECKPOINT and RESTORE allow the user to save a "snapshot" of his current system, and restore the same system at a later time. A CHECKPOINTed system takes up less space on disk, and requires considerably less time to load than a LISP system stored in source (S-expression) form.

   (CHECKPOINT A) saves the current system in the MTS file A. The file must be sequential or CHECKPOINT will generate an error.

   (RESTORE A) restores the LISP system previously saved by CHECKPOINT in MTS file A.

   (CHECKPOINT A S) checkpoints only the LISP structure S. On RESTORE of the file A, the system will be augmented by structure S. Any atoms which have the same PNAME as an atom which is part of S will be REMOBed and replaced with the CHECKPOINTed atom.

*******NOTE: The arguments to CHECKPOINT and RESTORE are NOT IOARGs. They are honest to goodness file names. The user should not attempt to OPEN a file for the purpose of CHECKPOINT and RESTORE.

At the present time, a call to CHECKPOINT may occur at any level of LISP, however a RESTORE of the _entire system_ always returns to the top level.

When CHECKPOINT terminates, a message is printed on *MSINK* which informs the user of the pages of core used by his program. In adition, (CHECKPOINT A), which destroys freespace, immediately initiates a RESTORE of the system. STATUS 44 may be used to prevent this RESTORE.

Neither CHECKPOINT nor RESTORE evaluates its arguments.

Note -- On the RESTORE of a specific structure S, it may happen that an atom A occurs in the structure being Restored, and there is already an atom A on the OBLIST. Both the VALUE and Property List of A will be set to the value they had at the time the CHECKPOINT was done; the current values disappear. The user can reverse this effect by setting the PLIST of the atom to *UNDEF* before the CHECKPOINT. In this situation, the RESTOREd atom A will reference the current atom A and the VALUE and Property List will _not_ be changed.

Note -- After a total system CHECKPOINT file is Restored, the system will begin reading from the current input buffer (LISPIN). If the user wants some initialization performed after a RESTORE, he can CHECKPOINT the initialization form into his file by putting it on the same input line.

e.g. (CHECKPOINT MYFILE) (REINIT)

Note -- A call to CHECKPOINT with a specific structure S will not do an automatic RESTORE, but will always terminate execution.

Note -- Two attention interrupts occuring during a CHECKPOINT or RESTORE will cause an immediate return to MTS. Use a $RESTART to continue.

Note -- The user should be aware that if LISP I/O units have been modified before a CHECKPOINT is performed, they will be in effect after the RESTORE.

D. Miscellaneous Functions

1. (LTR S SW)

The LTR function, the product of a diabolical mind, should never be used by anyone. It may be invoked any time the LISP system is doing an iterated EVAL through a list of S-expressions, in particular, during a LAMBDA, a PROG, or the "S1 . . . SN" portion of a COND; and also during evaluation of a sequences of arguments to be passed to a function. Its purpose is to allow conditional evaluation of arguments.

S is the value to be returned from LTR.

SW is a switch which determines what will happen to the rest of the forms in the list, which would be iteratively evaluated if the LTR were not present.
SW = NIL - don't evaluate any more forms. S is then effectively the last value in the list.
SW = T - continue normally through the list.
SW = anything else - in this case SW must be a new list of forms, which will be substituted for the rest of the original list, and evaluation will continue.

Ex: (REM (READ) (LTR (READ) X) (READ))

If X is NIL, then the effect of this is (REM (READ) (READ)) If X is T, then the effect of this is (REM (READ) (READ) (READ)) If X is (S) then the effect of this is (REM (READ) (READ) S)

LTR stands for "list terminate or re-direct".

E. Undoable Functions - the Transport System

LISP/MTS incorporates a simple mechanism for creating and altering data structures "hypothetically", for backing up to a previous state of the data structures, and for maintaining several alternative structures at once and switching back and forth among them.

This mechanism, called the Transport System, is useful for LISP implementations of problem solving, game playing, and automatic programming algorithms.

If we consider the state of all LISP structures at a particular moment to be a possible world, then the Transport System allows the user to obtain a "ticket" which will return him to that world at a later time.

Within the Transport System, there is always one unique world which has the status of Reality. This is the state of LISP structures before any "hypothetical" changes have been made. We can picture a system of hypothetical worlds as a tree structure, with Reality at the root. World A dominates World B if the user started in World A and, by making various hypothetical changes in his data structures, reached World B. Thus, all worlds are dominated by Reality.

The tickets which are created by the Transport System are actually lists of alterations of LISP structure. When the user returns to a dominating world, the alterations he has performed are un-done, or reversed. If he returns to a world which does not dominate the world he is currently in, alterations are reversed until the closest common dominating world is reached, and then the alterations which were performed to get to the desired world are repeated.

The following "undoable" functions exist:

SETQ2, SET2, SETA2
RPLACA2, RPLACD2, UNCONS2
NCONC2, DELQ2, DELETE2
PUT2, REM2, PUTPROP2, ADDPROP2

These functions behave exactly like their non-undoable counterparts, but also save the information necessary to undo the changes they make.

1. (NEWWORLD <<T,NIL>>)

The NEWWORLD function has three uses. (NEWWORLD) returns a ticket to the current state of LISP structure. By calling NEWWORLD, a state becomes a reachable world in the Transport System.

EX: (SETQ EARTH (NEWWORLD))

SAVES THE TICKET AS THE VALUE OF EARTH.

(NEWWORLD T) returns a ticket to Reality. This is provided in case the user wishes to return to Reality, but has not saved a ticket to get there.

(NEWWORLD NIL) returns a ticket to the closest reachable world which dominates the current state.

Note: NEWWORLD does not cause a transfer to any other world. Its purpose is to create tickets.

2. (GETWORLD S)

The GETWORLD function performs the transportation in the system. Its argument must be a valid ticket (Error 39 will be generated if not), and it causes a transfer to the world identified by that ticket.

EX: (GETWORLD EARTH)

3. (REALWORLD)

REALWORLD, the most amazing function of all, takes the current state of LISP structure, and causes it to become Reality. What was once Reality is now lost forever, and all previously created tickets will no longer be valid.

IX. Index