M T S

The Michigan Terminal System

Volume 8: LISP and SLIP in MTS

June 1976

Updated February 1979 (Update 1)
Updated January 1983 (Update 2)

The University of Michigan Computing Center
Ann Arbor, Michigan

DISCLAIMER

The MTS Manual is intended to represent the current state of the Michigan Terminal System (MTS), but because the system is constantly being developed, extended, and refined, sections of this volume will become obsolete. The user should refer to the Computing Center Newsletter, Computing Center Memos, and future updates to this volume for the latest information about changes to MTS.

June 1976

PREFACE

The software developed by the Computing Center staff for the operation of the high-speed processor computer can be described as a multiprogramming supervisor that handles a number of resident, reentrant programs. Among them is a large subsystem, called MTS (Michigan Terminal System), for command interpretation, execution control, file management, and accounting maintenance. Most users interact with the computer's resources through MTS.

The MTS Manual is a series of volumes that, when completed, will describe in detail the facilities provided by the Michigan Terminal System. Administrative policies of the Computing Center and the physical facilities provided are described in a separate publication entitled Introduction to the Computing Center.

The MTS volumes now in print are listed below. The date indicates the most recent edition of each volume; however, since volumes are updated by means of CCMemos, users should check the Memo list, copy the files *CCMEMOS or *CCPUBLICATIONS, or watch for announcements in the Computing Center Newsletter, to ensure that their MTS volumes are fully up to date.

Volume 1: The Michigan Terminal System, December 1979
Volume 2: Public File Descriptions, April 1982
Volume 3: System Subroutine Descriptions, April 1981
Volume 4: Terminals and Tapes, November 1980
Volume 5: System Services, April 1980
Volume 6: FORTRAN in MTS, December 1978
Volume 7: PL/I in MTS, September 1982
Volume 8: LISP and SLIP in MTS, June 1976
Volume 9: SNOBOL4 in MTS, September 1975
Volume 10: BASIC in MTS, December 1980
Volume 11: Plot Description System, August 1978
Volume 12: PIL/2 in MTS, December 1974
Volume 14: 360/370 Assemblers in MTS, August 1978
Volume 15: FORMAT and TEXT360, April 1977
Volume 16: ALGOL W in MTS, September 1980
Volume 17: Integrated Graphics System, December 1980
Volume 18: The MTS File Editor, September 1982

Other volumes are in preparation. The numerical order of the volumes does not necessarily reflect the chronological order of their appearance; however, in general, the higher the number, the more specialized the volume. Volume 1, for example, introduces the user to MTS and describes in general the MTS operating system, while Volume 10 deals exclusively with BASIC.

June 1976

The attempt to make each volume complete in itself and reasonably independent of others in the series naturally results in a certain amount of repetition. Public file descriptions, for example, may appear in more than one volume. However, this arrangement permits the user to buy only those volumes that serve his or her immediate needs.

Richard A. Salisbury,

General Editor

June 1976

Contents

OVERVIEW OF LIST-PROCESSING LANGUAGES IN MTS

This volume contains the language descriptions of the two major list-processing languages that are supported in MTS, namely, LISP and SLIP.

LISP is a programming language designed mainly for list-processing applications. The principal applications have been in artificial intelligence research. LISP was originally developed by J. McCarthy in the early 1960s as a formal language based on Church's lambda calculus. The syntax was, and still is, extremely simple. The language has been greatly extended by the addition of many special functions and is now a very powerful and efficient system when used for the appropriate applications. An important feature of LISP is that programs and data are represented by list structures so that one function can create or modify other functions, or even modify itself.

LISP programs are normally executed interpretively and require no translation. A compiler is also available which translates LISP programs into machine language. There are also two other LISP subsystems available that are described in this volume--a data structure editor and a debugging package.

Many different extended versions of LISP are now available throughout the country and there is no standardization across versions. A LISP program from another system will normally have to be modified to make it run properly on the MTS LISP system.

The MTS LISP system and the description in this volume were produced | by Bruce Wilcox and Carole Hafner of the Mental Health Research Institute at the University of Michigan. It is based on LISP systems currently in use at the Massachusetts Institute of Technology and elsewhere.

| SLIP (Symmetric List Processor) is a set of subroutines to allow list structures to be easily built and maintained in higher-level languages that do not have list-processing capabilities (e.g., FORTRAN). SLIP may be used with almost any application involving list structures, e.g., computer graphics data structures, memory lists for data base management systems, and mathematical applications involving lists of terms. The SLIP user can build both simple and complicated list structures using the same basic building blocks. Primitives are provided to read down a list, perform insertion, deletion, list copying, and sublist creation operations.

The description of SLIP in this volume was produced by Bertram Herzog, formerly of the Department of Industrial and Operations Engineering at the University of Michigan.

Page Revised February 1979 June 1976

LISP

INTRODUCTION

LISP is a programming language that combines a very simple syntactic structure with an extremely powerful and flexible semantic structure. This makes LISP unlike most other programming languages and places a great burden on the programmer to use the language carefully.

In the design of LISP, an attempt was made to embody the logical power of LISP in a language which is economical enough to be useful to many people. Many of the user options, input/output capabilities, and debugging features that programmers expect to find in any programming language have been added to LISP.

Throughout this section, various mnemonics have been used to represent LISP elements in describing the formats of basic LISP operations. A, A1, and A2 represent atoms; N, N1, and N2 represent numeric atoms; L, L1, and L2 represent lists. S, S1, and S2 represent any LISP structure; LA, LA1, and LA2 represent lists whose elements are literal atoms; and FN, FN1, and FN2 represent function specifications. S1...SN indicates that any number of expressions of that type may be given, and Si denotes any one of these expressions. <S> indicates that an expression of that type is optional, and <A,LA> indicates that the user has a choice of one or the other.

The development and implementation of LISP was supported in part by National Science Foundation Grant Number GJ-31339X. For a formal definition of the original LISP language, see J. McCarthy, et al., LISP 1.5 Programmer's Guide, M. I. T. Press, 1962.

THE LISP LANGUAGE

Atoms, Buffers, and Arrays

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

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.

For example, when the atomic string "BOOK" first appears in the input stream, an atomic structure with the PNAME "BOOK" is automatically created. Any future references to the atom BOOK 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.

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 single-precision floating-point numbers are treated as decimal numbers and become numeric atoms. All other character strings 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 are still treated as decimal numbers and become numeric atoms. However, any character string beginning with a decimal digit (0-9) and containing only hexadecimal digits (0-9, A-F) are treated as a hexadecimal number and become a numeric atom with the value of that hexadecimal number.

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

Unlike literal atoms, numeric atoms are not stored on the system OBJECT LIST; instead, a new atom is created each time a number appears in the input stream. Thus, two occurrences of the atomic string "17" produce references to two distinct structures.

VALUE of an Atom

Atoms can have VALUEs, which may be any LISP structure. The VALUE of a literal atom is undefined (set to the special system atom *UNDEF*) until a value is given to it. All numeric atoms, by convention, have themselves as their VALUEs. The VALUE of an atom is the CAR of the atomic structure (see the discussion of CAR and CDR below).

Property-Lists

Besides a VALUE, a literal 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 literal atom is a property-list (PLIST) of indicators and values. If an atom has no properties, then its PLIST is NIL. Numeric atoms do not have PLISTs. The PLIST of an atom is the CDR of the atomic structure.

Special Atoms

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

| An attempt to alter the VALUE or PLIST of NIL or the VALUE or PLIST of any numeric atom generates a "BAD ATOMIC ARGUMENT" error message.

The following are the predefined atoms of LISP and their values:

```
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)
All numeric atoms = themselves
Autoload atoms - see the subsection "Automatic Restoration
                 of LISP Functions."
```

Buffers

LISP supports a data type called BUFFERS. Although buffers are atoms, they may not be given VALUEs or PLISTs. The PNAME of a buffer is the current contents of the buffer. Character representations of LISP structures can be placed in a buffer by calling the system print functions. New atoms whose PNAMEs are the contents of a buffer can be created by calling the READ function. All input/output in the system takes place by printing the contents of a buffer onto an MTS file or device, and by reading a record from an MTS file or device into a buffer.

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 is given in the subsection "I/O Data Types."

Arrays

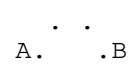
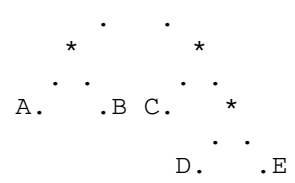
LISP also supports arrays, where the value of an array element can be any LISP structure. The subsection "Arrays" describes the creation and use of arrays.

S-Expressions

The basic LISP structure is a binary tree with atomic terminal nodes. To represent these trees syntactically, symbolic expressions, called S-expressions, are used. An S-expression consists of one of the following:

(1) an atom,
(2) a dotted pair of S-expressions (S1 . S2), or
(3) a list of S-expressions (S1...SN)

Examples:

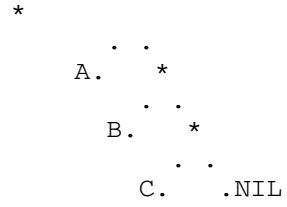
	Syntax	Tree Structure
(1)	A	.A
(2)	(A . B)	
(3)	((A . B) . (C . (D . E)))	

CAR and CDR

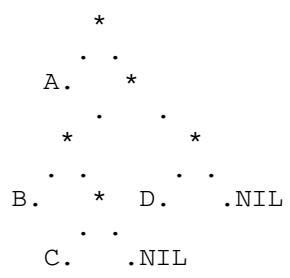
For any S-expression, the CAR is defined to be its entire lefthand branch and its CDR to be its entire righthand branch. Thus the CAR of ((A . B) . (C . (D . E))) is (A . B) and its CDR is (C . (D . E)). Similarly, the CAR of (A . B) is A and its CDR is B. The CAR of an atom is its VALUE, and the CDR of a literal atom is its PLIST.

Lists

The list notation defined in LISP provides a convenient shorthand which allows a subset of binary trees to be viewed as a list structure. The prototype of a LISP list is the following structure:



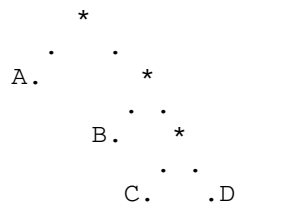
This represents the list (A B C). According to the definitions given of CAR and CDR, the CAR of a list is always its first element, and the CDR of a list is always the rest of the list. The end of a list is always signified by a CDR of NIL, which indicates that there are no more elements. Of course, an element of a list need not be an atom; thus, the structure



represents the list (A (B C) D). The CAR of this list is A. The CDR is ((B C) D).

It should be noted that the dotted-pair notation for (A B C) is (A . (B . (C . NIL))), and that these two LISP expressions are entirely equivalent. Any list can be written as a dotted pair; however, the converse is not true.

For structures similar to lists that have terminating atoms which are not NIL, a special syntax is available. The structure:



is represented by the expression (A B C . D).

Note: In general, the expression "element" refers to a top-level element of a list, "sublist" refers to a substructure which may be obtained by repeated CDRs, and a "substructure" indicates any subtree of the LISP structure. For example, the elements of the list (A B (C (D E)) F) are A, B, (C (D E)), and F. The sublists are (B (C (D E)) F), ((C (D E)) F), etc. The substructures,

however, include C, (D E), and all of its sublists, as well as all the sublists of the top-level list.

NIL

The list of zero elements ( ) is equivalent to the atom NIL within the LISP system. Thus, the CDR of (A) is ( ), or NIL. The result of this dual interpretation is that NIL is treated as an atom for most purposes, and as a list for some other purposes. For example, the CONC function which accepts only list arguments, and the COPY function, whose argument may be a list or an atom, both treat NIL as the null list ( ).

The LISP Interpreter

LISP is an interpretive language. The system reads one S-expression, or form, from its input stream, evaluates it, and prints out the value computed, then reads 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.

Input to LISP

Input to LISP is free format, with blanks, commas, periods, parentheses, angle brackets (< and >), 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; they are ignored. For example:

```
)) (A B C D)))) = (A B C D)
```

If a semicolon (;) appears anywhere in an input line, the system ignores the remainder of the line, and skips to the next line. Thus, the semicolon 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 semicolon is an MTS carriage-control character which causes a line printer to skip to a new page if it is the first character in an output line.

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 is interpreted as part of the number. To make a dotted pair of two numbers, the period must be surrounded by blanks. For example, (123.456) is a list of a single numeric atom, while (123 . 456) is a dotted pair of two integers.

A special feature of LISP which is not strictly part of the LISP syntax is the angle bracket. Any time a left angle bracket (<) occurs, it is treated as a normal left parenthesis, and the level at which it occurred is remembered. When a right angle bracket (>) occurs, it has the effect of inserting enough right parentheses to close out the most recent left angle bracket. A left angle bracket may not be closed out by a normal right parenthesis. If angle brackets are not balanced correctly, an error is generated.

In order to allow the incorporation of separator characters into atom PNAMEs, LISP defines a special input convention. If a quote character (") occurs at the beginning and the end of an atom name, all characters which occur between the quotes are treated as the PNAME of a single atom. The closing quotes must be part of the same input line as the opening quotes; the quotes are not 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 is created.

If two quotes in a row appear within a quoted string, they are interpreted as a literal quote. If quotes appear at the beginning of an atom name, however, this generates a syntax error. For example, if "ABC""DE" is read in, the literal atom ABC"DE is created.

Quotes which appear strictly within an atom name have no special significance, and are treated like any other character.

Operation of EVAL

Evaluation of LISP expressions is done by the function EVAL. If the form being EVALed is an atom, 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. (It will be seen 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 is 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, elements which actually appear in the form are referred to as argument-designators, and the term "argument" is reserved 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))

causes 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.

Output and Termination: (STOP) and (MTS)

Whenever a LISP form is EVALed, a resulting value is returned. When the system reads and EVALs a form, it prints its (top-level) value before reading the next form. When it is said that 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 enters the form

(ADD X (ADD Y 1))

where the VALUE of X is 2 and the VALUE of Y is 3, the system EVALs this entire expression and prints the resulting value of 6.

Evaluating the form (STOP) at any level terminates execution of LISP. Evaluating the form (MTS) returns control to MTS command mode from which the user may subsequently restart LISP via the $RESTART command.

Basic LISP Functions

(QUOTE S) and 'S

It is important to remember that when a LISP form appears as an argument-designator 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 echo (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.

Examples:

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

Basic LISP Predicates

In general, a LISP predicate will return either the atom T, indicating that it is true, or NIL, indicating that it is false. Examples of these predicates follow each description.

(ATOM S) — Returns T if its argument is an atom; otherwise, returns NIL.

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

(NOT S) — Returns T if its argument is NIL; otherwise, returns NIL.

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

(EQUAL S1 S2) — Returns T if its arguments have the same LISP structure or represent the same number; otherwise, returns NIL.

```
(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
```

(EQ S1 S2) — Returns T if its arguments are the same LISP structure; otherwise, returns NIL. Since there are frequently multiple structures which represent the same S-expression, not every pair of elements which are EQUAL are EQ; in particular, numeric atoms which represent the same number are not generally EQ. EQ is almost always used with literal atomic arguments, since there is only one copy of each atomic name on the OBJECT LIST.

```
(EQ 'A 'A) = T
(EQ '(A B) '(A B)) = NIL
(EQ 8 8) = NIL
```

(EQNAME A1 A2) Returns T if its arguments are literal atoms or buffer atoms which have the same PNAME; otherwise, returns NIL. EQNAME is equivalent to EQ for normal atoms which are on the OBJECT LIST. However, for buffer atoms and atoms created by GENSYM, EQNAME provides a new and useful function.

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

(NUMBER A) Returns T if its argument is a numeric atom; otherwise, returns NIL.

```
(NUMBER 3) = T
```

(SORT 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; otherwise, returns NIL. A1 and A2 must be literal atoms or IOARGs.

```
(SORT 'ABC 'ABB) = NIL
(SORT 'ABB 'ABB) = T
(SORT 'AB 'ABB) = T
```

(NULL S) NULL is equivalent to NOT.

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 some particular substructure being found. Examples of these functions follow each description.

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

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

(CDR L) Returns the CDR of any structure (i.e., the list of remaining elements of any list or PLIST of a literal atom). An attempt to take the CDR of a numeric atom will generate a type 0 error.

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

(C...R L)

These 28 functions perform all combinations of up to four instances of CARs and CDRs.

```
(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))))
```

(MEMBER S1 L)

The list L is searched to see if S1 is EQUAL to any element. If so, the sublist of L starting with S1 is returned. If that element is not EQUAL to any element of L, NIL is returned.

```
(MEMBER 'A '((A B) C (D E) G)) = NIL
(MEMBER '(D E) '((A B) C (D E) G)) = ((D E) G)
```

(ASSOC S1 L)

The list L is searched to see if S1 is EQUAL to the CAR of any element. If so, then that element is returned. If S1 is not EQUAL to the CAR of any element, NIL is returned.

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

(FIND S L <N>)

The structure L is searched for any substructure (subtree) whose CAR is EQUAL to S. If N is given, the Nth such substructure is returned. If N is not given, the first such substructure is returned. If N is zero or negative, FIND will return the last such substructure. If the substructure specified is not found, FIND returns NIL.

```
(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
```

(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.

```
(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
```

Functions That Create New LISP Structures

This section includes functions that create new LISP structures as well as returning a value. Frequently, the value returned from a function in this section is precisely the new LISP structure which was created. Examples of these functions follow each description.

(CONS S1 S2) Returns the dotted pair of S1 and S2.

```
(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))
```

(LIST S1...SN) Returns the list of S1 through SN.

```
(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))
```

(EVLIS L) Evaluates each element of L and returns a list of the values.

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

(CONC L1...LN) Returns a concatenated list of copies of lists L1 through LN.

```
(CONC '(A B C) '(D E F)) = (A B C D E F)
(CONC '(A B C) NIL '(D E F)) = (A B C D E F)
```

(APPEND L S1...SN) Returns a copy of the list L, with S1 through SN appended as elements to the end.

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

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

```
(REVERSE '(A (B (C D)) E)) = (E (B (C D)) A)
```

(COPY S1 <S2 <S3>>) Returns a copy of structure S1. If arguments S2 and S3 are given, each substructure of the original structure (S1) which is EQUAL to S2 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 other than

NIL, the value of COPY will be a new atom, which is not found on the OBJECT LIST, with the same PNAME as the original atom. The value of (COPY NIL) is NIL.

```
(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
```

(UNION L1 L2) Returns a list of the elements of L1 and the elements of L2. No duplicate elements will appear in the list returned, i.e., no two elements will be EQ. The order of the elements in the resulting list will be L1 followed by elements of L2 not in L1. Note: (UNION L NIL) may be used to generate a top-level copy of the list L, but all duplicate EQ entries will be deleted.

```
(UNION '(A B C) '(A B C D)) = (A B C D)
```

(INTERSECT L1 L2) Returns a list of the elements of L1 which are EQ to elements of L2. No duplicate elements will appear in the list returned. The order of the resulting list will be the same as that of L2.

```
(INTERSECT '(A B B A) '(A C)) = (A)
```

(EXCLUDE L1 L2) Returns a list of the elements of L2 which are not EQ to elements of L1. No duplicate elements will appear in the list returned. The order of the resulting list will be the same as that of L2.

```
(EXCLUDE '(A B B A) '(A C)) = (C)
```

(GENSYM <A>) Returns a unique atom. If no argument is given, GENSYM creates atoms G1, G2, ..., etc. Each 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. This will be 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 reset 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.

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

(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.

(IMPLODE LA) Returns an atom whose PNAME is the concatenation of the PNAMEs of the elements of LA. The atom returned is not on the OBJECT LIST.

Functions That Modify Existing LISP Structures

(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.

```
(SET 'X 'A 'Y '(B C)) = (B C),
```

The VALUE of X is set to A, the VALUE of Y to (B C).

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

```
(RPLACA '(A B C) '(E F)) = ((E F) B C)
```

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

```
(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 CONC, APPEND, 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 cost a great deal before the user discovers the program is in an infinite loop. For example, if L = (A B), then (RPLACA L L) creates the structure:

(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.

```
(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 equivalent to taking the CDR of L, but DELETEing any other element will cause an actual change in the list structure.

If multiple occurrences of an element are DELETEd from a list, it is as though multiple DELETE operations had been performed, each one on the result of the previous one. Thus, if the VALUE of L is (A A A B), then (DELETE 'A L) = (B). Note that the VALUE of L remains (A A A B), since nothing has occurred to alter the list structure (A A A B).

(GRAFT 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. The value of GRAFT will be the (modified) list L1.

For example, if the VALUE of X is (A B), the VALUE of Y is (C D), and the VALUE of Z is (E F), then (GRAFT X Y Z) = (A B C D E F), and the VALUE of Z is (E F), the VALUE of Y is

(C D E F), and the VALUE of X is (A B C D E F).

Note: The same changes in structures occur in GRAFT as in RPLACA and RPLACD. Thus, the warnings given in the note above apply to GRAFT as well.

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). Examples of these operations follow each description.

(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. 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.

```
(PUT '(A B) 'INCL 'X) = X
```

The property INCL with property-value X is put on the PLIST of A and B.

(GET A IND) Returns the property-value associated with the indicator IND on the PLIST of A. If A does not have a property EQUAL to IND, GET returns NIL.

```
(PUT 'A 'INCL '(X Y)) = (X Y)
(GET 'A 'INCL) = (X Y)
(GET 'A 'NOTON) = NIL
```

Assumes NOTON is not on the PLIST of A.

(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.

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

(GETL A L) Finds the first indicator on the PLIST of A which is a member of the list L. Returns the sublist of the PLIST of A, starting with the indicator which was found. If no such indicator is found, GETL returns NIL.

For example, 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)) = (TOPIC MATH)
(GETL 'BOOK '(WEIGHT)) = NIL
```

(ADDPROP <A,LA> IND <PVAL>)
Operates like PUT with the exception that a new instance of IND is always placed on the PLIST of A, or on the PLIST of each of the atoms in LA. Thus, by 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.

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

Basic Numeric Predicates

(GREATER N1...NN) Returns T if N1...NN is a strictly decreasing sequence; otherwise, returns NIL.

(LESS N1...NN) Returns T if N1...NN is a strictly increasing sequence; otherwise, returns NIL.

(ZERO N) Returns T if N=0; otherwise, returns NIL.

(EVEN N) Returns T if N is an even integer; otherwise, returns NIL.

(INTEGER N) Returns T if N is an integer; otherwise returns NIL. N must be a numeric atom.

Basic Numeric Operations

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

(PLEN A)	Returns the length of the PNAME of the atom A. A must be a literal atom or IOARG.
(ADD1 N)	Returns N+1. N must be an integer.
(SUB1 N)	Returns N-1. N must be an integer.
(MINUS N)	Returns -N. N must be an integer.
(ABS N)	Returns the absolute value of N. N must be an integer.
(FIX N)	Returns the integral (truncated) part of N.
(FLOAT N)	Returns the floating-point equivalent of N.
(MAX N1...NN)	Returns the algebraic maximum of N1...NN.
(MIN N1...NN)	Returns the algebraic minimum of N1...NN.
(ADD N1...NN)	Returns the sum of N1...NN.
(SUB N1 N2)	Returns N1-N2.
(TIMES N1...NN)	Returns the product of N1...NN.
(DIVIDE N1 N2)	Returns the quotient of N1 and N2. Floating-point division is performed.
(REMAIN N1 N2)	Returns the remainder of N1/N2. N1 and N2 must be integers.
(IDIVIDE N1 N2)	Returns the integer quotient of N1 and N2. N1 and N2 must be integers.
(ADDRESS S)	Returns a numeric atom equal to the address of the LISP structure S.
(SHIFT N1 N2)	Returns the number N1, shifted N2 bits to the left. N1 and N2 must be integers. If N2 is negative, the effect is a shift to the right. (SHIFT 32 -1) = 16 (SHIFT 3 2)= 12
(LAND N1 N2)	Returns the result of a bitwise logical AND of N1 and N2. N1 and N2 must be integers. (LAND 3 5)=1
(LOR N1 N2)	Returns the result of a bitwise logical OR of N1 and N2. N1 and N2 must be integers.

(LOR 3 5)=7

(LXOR N1 N2) Returns the result of a bitwise logical EXCLUSIVE-OR of N1 and N2. N1 and N2 must be integers.

(LXOR 3 5)=6
(LXOR -1 3) = -4

Basic Control Functions

This section includes the functionals, which take as their arguments definitions of functions to be invoked. Also included are EVAL and PROGN, which control the evaluation of forms in LISP. Examples of these functions follow each description.

(EVAL S) Evaluates S and returns the result.

For example, if the VALUE of X is (A B C), and the VALUE of A is VALA, then (EVAL (CAR X)) = VALA.

(PROGN S1...SN) Evaluates S1 through SN and returns the value of SN, its last argument. This function is useful when the user wants to do several different things in one step, and wants only the last result returned. 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.

(PROGN S1...SN 'DONE) = DONE

(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 0, REPEAT does not evaluate S, but only returns NIL.

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

(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.

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

(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.

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

(MAP FN L1...LN) Causes the function FN to be called, with L1...LN as its arguments, and then 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.

(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.

(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.

(MAPCAR FN L1...LN) Works like MAPLIST except 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.

(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.

(MAPCAN FN L1...LN) Works 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.

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 GRAFT 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 GRAFT, RPLACA, etc.

Let the VALUE of X be ((D 7) (A 6) (N 5)), then

```
(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)
```

N-Type Functions in LISP

There are a number of LISP functions which have the effect of automatically QUOTEing their argument-designators. That is, the arguments which are passed to the function by the LISP system are the argument-designators themselves, rather than their values. The function called can then evaluate the arguments selectively by calling EVAL. This is the mechanism used by LISP to implement conditional execution, or conditional evaluation.

(SETQ A1 S1...AN SN) Sets arguments A1...AN to the values of arguments S1...SN, respectively.

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

Note: If the VALUE of X is VALX, then (SET 'X '(B C) 'Y X) = VALX, and X is set to (BC), and 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 (BC), and Y is set to (B C), since the SETQ performs an EVAL-SET-EVAL-SET... loop.

(UNCONS L A) Returns the CAR of the VALUE of L, and also sets A to the CDR of L.

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

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

(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.

(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.

(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 no Si has a value of NIL, AND returns the value of SN.

(AND (CAR Z) (SETQ Z A) (SETQ X 'DONE)) has the following effect: if (CAR Z) is NIL, AND merely returns NIL; otherwise, Z is set to the VALUE of A, and if the VALUE of A is NIL, then AND returns NIL; otherwise, X is set to DONE, and DONE is returned.

(OR S1...SN) Evaluates the arguments S1...SN until a value which is not NIL is found. OR then returns that value. If all of the arguments evaluate to NIL, then OR returns NIL.

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

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

This 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 non-NIL, COND will

evaluate S1...SN successively, and the value returned from COND will be the value of SN. If no Si is given, COND only returns the value of P.

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.

It is seen that the functions AND and OR are merely subcases of COND.

```
(AND S1...SN) = (COND
                   ((NOT S1) NIL)
                   ((NOT S2) NIL)
                          .
                          .
                   ((NOT S(N-1)) NIL)
                   ((SN)))

(AND S1...SN) = (COND
                   (S1 (COND
                   (S2 (COND...
                          .
                          .
                   (COND(S(N-1)SN))...))))))

(OR S1...SN) = (COND
                   (S1)
                   (S2)
                     .
                     .
                   (SN))
```

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

This function 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 is given, then SELECT only returns the value of E1.

If no Ei matches 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.

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

(PROG LA S1...SN)
(GO A)

The PROG function 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 from anything 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.

S1...SN is 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 transferred 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 PROG.

If at any point within a PROG, a form (RETURN S) is evaluated, then PROG immediately exits, and returns the value of S.

Note: GO is, like PROG , an N-type function. Thus, (GO A) will cause a branch to the form labeled by the atom A. However, if GO is given a nonatomic argument, it will EVAL this argument, and then attempt to branch to the result.

(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.

(RETURN S <STACKID>) The RETURN function can be used in two modes. When (RETURN S) is evaluated and no second argument is given, it will cause an immediate return from the dominating PROG, and the value of the PROG will be S. If a return of this type is evaluated, but there is no dominating PROG, an error will be generated.

(RETURN S STACKID) allows the user to return from any dominating LISP evaluation, and the value returned will be S. If STACKID is a positive integer N, the return will be made from the EVAL level N deep, starting at the top level of LISP. Thus, (RETURN 'A 1) will always cause a return to the top level of LISP, and A will be printed. If STACKID is a negative integer -N, the return will be made from the Nth previous EVAL level, starting with -1 at the level before the call to RETURN. Thus, (CAR (CDR (CAR (CDR (RETURN '(A B) -3))))) = A.

If STACKID is not a number, then all outstanding EVAL forms will be examined, from most recent to least recent, and the return will be made from the first form found whose CAR is EQUAL to STACKID. Thus, (RETURN S 'PROG) has the same effect as (RETURN S).

If STACKID does not identify an existing EVAL form, either because it is too large a number or because it does not match the CAR of any outstanding form, an error will be generated.

The following example program searches a list for atoms and prints out each atom, along with its syntactic depth of occurrence. The

function (READ) is called, and returns one S-expression which it reads from the system input device, SCARDS.

Program Description (in an algebraic-type programming language)

```
RESTLIST = ((READ-IN-ARGUMENT . 0));
DO WHILE RESTLIST ¬= NIL;
LIST = (CAAR RESTLIST);
DEPTH = (CDAR RESTLIST);
RESTLIST = (CDR RESTLIST);
IF (LIST IS ATOM) THEN DO;
PRINT "ATOM" ":" LIST ", DEPTH" DEPTH;
END;
ELSE DO;
TEMP = (CAR LIST);
LIST = (CDR LIST);
IF LIST ¬= NIL THEN
RESTLIST = (CONS (CONS LIST DEPTH) RESTLIST);
ELSE;
RESTLIST = (CONS (CONS TEMP DEPTH+1) RESTLIST);
END;
END;
```

Program:

```
(PROG (LIST DEPTH TEMP RESTLIST)
(SETQ RESTLIST (LIST (CONS (READ) 0)) )
A     (COND
((NOT RESTLIST) (RETURN 'DONE))
(T (SETQ LIST (UNCONS (UNCONS RESTLIST
      RESTLIST ) DEPTH))
(COND ((ATOM LIST)
(MAPC 'PRIN1 (LIST 'ATOM ': LIST '"," 'DEPTH DEPTH))
(TERPRI))
(T (SETQ TEMP (UNCONS LIST LIST))
(COND (LIST
(SETQ RESTLIST (CONS(CONS LIST DEPTH) RESTLIST))))
(SETQ RESTLIST (CONS (CONS TEMP
      (ADD1 DEPTH)) RESTLIST))
))))
(GO A))
```

Recursive Implementation of the Same Program:

```
(PROG NIL (
(LABEL ATOMPRINT (LAMBDA (RESTLIST)
(COND ((NOT RESTLIST) (RETURN 'DONE))
((ATOM (CAAR RESTLIST)) (MAPC 'PRIN1
      (LIST 'ATOM ': (CAAR RESTLIST)
            '"," 'DEPTH (CDAR RESTLIST)))
(TERPRI)
(ATOMPRINT (CDR RESTLIST)))
( T (ATOMPRINT (GRAFT
```

```
(LIST (CONS (CAAAR RESTLIST) (ADD1 (CDAR RESTLIST))))
(AND (CDAAR RESTLIST) (LIST (CONS (CDAAR RESTLIST)
     (CDAR RESTLIST))))
          (CDR RESTLIST)))))))
(LIST (CONS (READ) 0))))
```

Output from Program with Input:

```
(A (B C) (D (E F (G H (I) J K) L))))))
```

```
ATOM : A , DEPTH 1
ATOM : B , DEPTH 2
ATOM : C , DEPTH 2
ATOM : D , DEPTH 2
ATOM : E , DEPTH 3
ATOM : F , DEPTH 3
ATOM : G , DEPTH 4
ATOM : H , DEPTH 4
ATOM : I , DEPTH 5
ATOM : J , DEPTH 4
ATOM : K , DEPTH 4
ATOM : L , DEPTH 3
DONE
```

MORE ABOUT FUNCTIONS

LAMBDA-Expressions

As we noted earlier, the CAR of a form being EVALed is interpreted as a function specification and an atomic CAR 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; in turn, S1...SN are EVALed. 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 whenever a function specification is required, for example, as the first argument of APPLY, MAP, MAPLIST, etc. For example,

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

Note: When it is stated that an atom is bound to some value, this means 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 be NIL, in which case the function takes no arguments, but it may not be omitted.

The No-Spread Form of a LAMBDA

Another form of LAMBDA-expression which takes an indefinite number of arguments may be defined. 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 given number of arguments 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.

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

```
(LAMBDA 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))
```

Other Forms of LAMBDA-Expressions

There are two other 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. The only exception is that the argument-designators themselves, rather than their values, are used as the arguments to the NLAMBDA. For example,

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

The second alternate 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. In the case of a spread-type FLAMBDA, this single argument 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. For example,

```
((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 or mapping the three types of functions. The argument-designators to APPLY and APPLY1 are always EVALed before being passed to these functions, and will not be evaluated again. Thus, for the purposes of APPLY, APPLY1, MAP, etc., the differences between LAMBDA and NLAMBDA-functions disappear. However, for FLAMBDA-type functions, the arguments given are made into a list when used with APPLY1, or left in their list form in the case of APPLY. Thus, when these functions are APPLYed they always 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.

Named LAMBDA-Expressions (LABEL-Expressions)

LISP provides a special syntax for writing LAMBDA-expressions which are capable of calling themselves. This is the LABEL-expression. The basic form of a LABEL-expression is:

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

where NAME may be any atom. NAME is first bound to the LAMBDA-expression which is the second argument of the LABEL-expression. The evaluation continues as though the LAMBDA-expression had been given. The effect is that NAME is temporarily defined 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. For example,

```
((LABEL COUNT (LAMBDA (L N)
        (COND ((NOT L) N)
              ((COUNT (CDR L) (ADD1 N))))))
     '(A B C D E) 0) = 5
```

This LABEL-expression temporarily defines a function COUNT, which will return the sum of its second argument and the number of elements in its first argument.

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 without evaluation as do 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 two 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 prevent 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 just as any other list. This may often be a good method of making corrections to a user-defined function. However, modifying or destroying the links to built-in LISP functions should be done carefully, if at all.

Defining New Functions in LISP

DEFUN and DEFINE are two functions for defining new functions in LISP.

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. The basic form of DEFUN is:

(DEFUN NAME <TYPE> ARGLIST S1...SN)

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 places the created LAMBDA-expression on the property-list of NAME under the indicator EXPR. The value returned from DEFUN is the atom NAME. If TYPE is omitted, then ARGLIST may not be EXPR, NEXPR, or FEXPR. For example,

```
(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, or no arguments.

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

DEFINE is the basic function for defining and naming new LISP functions. The basic form of DEFINE is:

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

DEFINE is an N-type function which takes an indefinite number of definitions, DEFN, as arguments. NAME is always an atom, which is the name of the entity being defined. TYPE may be EXPR, BUG, ARRAY, SUBR, NSUBR, or FSUBR, 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 next subsection.

The ARRAY and SUBR definitions require special parameters which respectively define LISP arrays and create linkage to external subroutines. The form and meaning of these definitions is explained in the subsections "Arrays" and "Calling External Routines from LISP."

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. For example,

```
(DEFINE (TEST EXPR (LAMBDA (Y) (PRINT Y)))) = TEST
```

This defines a function TEST which merely prints its argument.

Note: DEFUN and DEFINE, which place 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 already on the PLIST; 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, the new property is merely placed in front of it. This process guarantees that the most recent function definition of an atom will be the active one.

June 1976

BUG

In order to facilitate the writing of debugging routines in LISP, a new facility called a BUG has been added to LISP/MTS. A BUG is a pseudofunction definition which can be placed on the property-list of an atom already defined as a function. BUG will cause an interception of the function upon entry and upon exit. The user can display the arguments sent to the function or any other LISP structures, can test entry conditions, and can display the value 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 ability to alter the arguments sent to A (except, of course, by physical modification of the argument structures), but it can abort the call entirely. (See the subsection "Error Recovery and Debugging Procedures.")

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.

Notes:

(1) When a BUG is placed on the property-list of an atom, and, subsequently, a new function definition is placed on the same property-list, the BUG will be ignored when the function is called. In other words, BUGs must precede other system indicators 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.

(2) One or more BUGs appearing on the property-list of an atom A, which has no function definition, will generate an error if A is invoked as a function.

(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, and finally the output BUG functions, from last to first.

(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)
                        (PRIN1 'ARGUMENTS:)
                             (PRIN1 ARGS) (TERPRI))
                    .(LAMBDA (RET)
                        (PRINT 'EXIT-FROM-COUNT)
                        (PRIN1 'VALUE:) (PRIN1 RET)
                             (TERPRI)))))
```

Arrays

The basic form of a LISP array definition is

(DEFINE (A ARRAY (A1...AN) <FILL>))

This is the basic form of a LISP array definition. In this example, A will be defined as an N-dimensional array with subscript bounds A1...AN. A1...AN must be atoms whose VALUEs are integers. If a fourth argument, FILL, is given, the initial value of each element in the array will be set to that structure. Otherwise, each element in the array is initially set to NIL.

An array definition associated with an atom operates like a function definition for that atom. A pointer to the appropriate access code is stored on the property-list of the atom, under the indicator SUBR. The value of an array element is obtained by invoking the "function", with the appropriate subscripts as arguments. For example,

(A 1 (ADD 3 5) (CADR '(B 3)))

will evaluate to the array element A(1,8,3). It should be noted that since an atom may be defined as a function in only one way at a time, defining A as an array will negate any other function definition A may have had, and defining A as a function will negate the definition of A as an array.

To set the value of an array element, the SETA function is used:

(SETA (A 2 (ADD 2 2) 2) '(B C D))= (B C D)

and the array element A(2,4,2) will be set to the list (B C D).

The value of an array element may be any LISP structure.

Note: The user may conveniently define an array and use his own hash-coding algorithm to compute the subscripts as follows:

(DEFINE (A ARRAY (8000)))

To obtain an element of A:

(A (HASHFN CODE1...CODEN))

To set an element of A:

(SETA (A (HASHFN CODE1...CODEN)) VALUE)

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

Calling External Routines from LISP

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

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

(DEFINE (FN <SUBR,NSUBR,FSUBR> (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 (see the loader description in MTS Volume 5 for a description of entry point determination). If no ENTRY-NAME is given for a library file, an error will be generated. If the ENTRY-NAME given is already in memory, nothing will be loaded, and the LISP function FN will be defined to be the ENTRY-NAME function. N specifies the type of calling conventions to be used, and must be set at 0, 1, 2, 3, or 4.

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 system 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 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 external 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 fullword integer. (Note that the number represented by the atom is passed, and not the atomic structure.)

Upon return from the function, floating-point register 0 will be treated as a single-precision value of the function, and a numeric atom with that value will be returned. No changes to any arguments will occur.

N=2 signifies FORTRAN function calling conventions, with an integer value. Any number of arguments may be given, and their interpretation will be the same as for N=1.

Upon return from the function, general register 0 will be treated as an integer return value from the function, and a numeric atom with that value will be returned. No changes to any arguments will occur.

N=3 signifies FORTRAN subroutine calling conventions. Any number of numeric arguments may be given, and their interpretation will be the same as for N=1 or N=2.

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 nonzero, then the value returned from the LISP function will be NIL. If the return code is zero, then a list of the argument values (which may have possibly been modified) 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 that some new numeric atoms are returned as part of the 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.

N=4 signifies the same calling conventions as N=0; however, if the system is CHECKPOINTed, the external code will also be CHECKPOINTed. (Normally, an external routine is reloaded after each RESTORE.)

Calls to the MTS functions STOP and ERROR while executing an external routine are trapped by LISP. STOP causes a return from the external function with a value of NIL. ERROR generates a call to the LISP function ERR.

A Note on Recursion in Function Specification

The CAR of a form being EVALed has a unique status in LISP. It has been previously stated that the CAR is interpreted as a function specification, and we have given some examples of typical function specifications. If the CAR is a LAMBDA-expression, the LAMBDA-function will be applied to the rest of the form being EVALed. Some of the other possibilities will now be described in more detail.

If the CAR of the form being evaluated is an atom, then EVAL looks for one of the special system indicators on the property-list of the atom. If one is found, it will either be one of the built-in system function indicators, in which case LISP goes off to execute that function, or it will be one of the user-defined function indicators. If it is one of the latter, then the property-value associated with the indicator is merely substituted for the atom, and the evaluation process continues.

```
(DEFINE (NEWCAR EXPR CAR))
(NEWCAR '(A B C)) = A
```

Assume, for a moment, that the CAR of the form being EVALed is an atom which has no system indicator on its property-list. In this case, EVAL searches for a system indicator, fails to find one, and substitutes the VALUE of that atom for the atom itself, and the process continues. In this manner, an atom NAME can have a function definition temporarily associated with it during execution of a LABEL-function. However, the VALUE need not be a LAMBDA-expression, but may be another atom, or any other "function specification."

Finally, if the CAR of the form being EVALed is not an atom and not a LAMBDA-expression, it is then interpreted as an "indirect" function specification. It is itself EVALed, its value is substituted for itself, and the process continues.

```
(SETQ A '(CONS))
(SETQ B 'CAR)
((B A) 'X 'Y) = (X . Y)
```

INPUT/OUTPUT IN LISP

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, rather than physical records, constituting the divisions between I/O operations. Thus, several S-expressions may be read from one input line; 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. Contents of the buffer will be printed, and the remainder of the current expression will be continued as a new buffer.

(READ) Calling READ with no arguments causes one S-expression to be read from the system input device. The structure represented by the S-expression will be returned.

(READCH) Gets the next character from the current input stream. Blanks, periods, parentheses, etc., as well as alphanumeric characters, are returned as one-character atoms.

(READLINE) Causes a new line to be read from the system input device into the system input buffer. The previous contents of the buffer are destroyed, and the next READ will begin with the new line. The value returned from READLINE is a pointer to the buffer, which is described in the next subsection.

(PRIN1 S) Causes the list or S-expression form of the LISP structure S to be "printed" into the system output buffer. Each S-expression printed will be preceded by a blank. When the buffer is full, its contents will be printed on the system output device, and printing of the remainder of the print name of S will be continued at the beginning of the empty buffer.

The value returned from PRIN1 is S.

(TERPRI) Terminates the system output buffer, causing its current contents (if any) to be printed, and the buffer to be cleared. If there is nothing in the buffer, TERPRI has no effect. The value of TERPRI is NIL.

(PRINT S) Operates like (PROGN (TERPRI) (PRIN1 S) (TERPRI)), except that the value returned from PRINT is S.

(TAB N) Causes the pointer for the system output buffer to be moved to position N. (Position 1 is the first position.) Any spaces skipped are filled with blanks, and buffer contents skipped on a TAB to the left are destroyed. The value returned from TAB is the current buffer.

(SKIP N) Causes the pointer for the system output buffer to be moved N spaces to the right. If N is negative, the pointer moves to the left. Any spaces skipped in a shift to the right are filled with blanks, and buffer contents skipped over in a shift to the left are destroyed. The value returned from SKIP is the current buffer.

Note: TAB and SKIP cannot move a buffer pointer outside the buffer range. If a TAB or SKIP indicates a move too far to the right or left, an error will be generated.

Example:

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

The following line will be printed:

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

I/O Data Types

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

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 are 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 buffer rather than the system input buffer; if the buffer is used up, a new line is read from the specified file. On output, data is printed into the specified bufer rather than the system output buffer; if an overflow occurs (or the operation is a PRINT), data is printed on the specified 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, e.g., EXPLODE, etc. If either component of an IOARG is NIL, then the appropriate system buffer or file is 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.

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.

Instead of the buffer itself, the IOARG whose CAR is the buffer is always passed as an argument to a function. For example, functions such as EXPLODE, which forms a list of one-character atoms from the characters in a buffer, or GENSYM, which creates an atom whose PNAME begins with the current contents of the buffer, expect an IOARG to be passed rather than the buffer itself. The FILE portion of the IOARG is 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 a PRINT or PRIN1. For printing purposes, a buffer is treated like any other literal atom, and its PNAME is inserted into the output buffer.

For example, if (PRIN1 (CAR LISPIN) BUF1) appears as an input line under normal conditions of operation, the character string " (PRIN1 (CAR LISPIN) BUF1)" is placed in the buffer associated with I/O destination atom BUF1.

Files

The FILE is an atomic 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 or device name, logical I/O unit name, or logical I/O unit number.

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. However, if the buffer overflows, the file for that I/O operation is 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 is interleaved in the file.

Buffer and File Prefix Characters

Any LISP buffer may have a prefix of up to 255 characters, which may be set or reset by calling the STATUS function. The purpose of the buffer prefix is to allow prefix strings to precede output lines. All PRINT operations, including TAB and SKIP, treat a buffer with an active prefix as though it begins after the prefix. Prefix characters use up character positions at the beginning of the buffer, and are included in the buffer size limit of 255 characters. Since READ operations do not recognize buffer prefixes, a physical read operation into a buffer with a prefix destroys or replaces the prefix.

A file prefix character may be attached to any LISP file by calling the STATUS function. This has the effect of calling the MTS subroutine SETPFX which causes any input from or output to the terminal to be prefixed by the prefix character. For example, the following is a sample run in which a buffer is created, given a prefix, the prefix is used, and is then disabled. Lines which are not indented are entered by the user. Lines which are indented are responses from LISP.

(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 content.

     NIL
(PRINT 'PRINT2 ABUF)
     THIS IS A TEST PRINT2
     PRINT2
(STATUS (10 ABUF NIL))        Disables the prefix.

     14
(PRINT 'PRINT3 ABUF)
     THIS IS A TEST           The first TERPRI prints  buffer  con-
                              tents (no longer a prefix).
     PRINT3
     PRINT3
(PRINT 'PRINT2 ABUF)
     PRINT2
     PRINT2
```

Buffer Overflow Interception

The user may, on an I/O call, specify a read or print intercept function as an optional argument. The intercept must be a function which takes one argument. If an intercept function is specified in a call to READ, READLINE, or READCH, on any attempt to do a physical read into the buffer, the intercept function will be called first. The IOARG for that READ will be passed to the intercept function as its argument.

If an intercept function is specified 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.

End-of-File Processing

Each LISP 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 the STATUS function.

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 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* in conversational mode will cause the user to be prompted to continue. 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 should be taken on return from an EOF function is determined by the value returned from the function. If the value returned 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, then the READ will be tried again.

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, which will be bound to the current IOARG when the function is called. An atom is defined as a READMACRO or PRINTMACRO by calling the STATUS function with the appropriate arguments.

READMACRO Atoms

(STATUS (2 HIT T)) defines the atom HIT as a READMACRO. If HIT is encountered in the input stream during a READ operation, the function associated with HIT will be invoked immediately.

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

(1) 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.

(2) 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. For example,

```
(DEFUN HIT (X) (COND ((ATOM (SETQ X (READ)))
(LIST (LIST X 'HIT)))
```

```
          ((LIST (MAPCAN '(LAMBDA (A)
                 (LIST A 'HIT)) X)))))
(STATUS (2 HIT T))

(READ)
(A B C HIT (D E F) G)
```

will return

```
(A B C (D HIT E HIT F HIT) G)
```

and

```
(READ)
(A B C HIT D E F)
```

will return

```
(A B C (D HIT) E F)
```

PRINTMACRO Atoms

(STATUS (4 HIT T)) defines HIT as a PRINTMACRO atom. Whenever an attempt is made to print the atom HIT, the HIT function will be called instead. The value returned from the HIT function is ignored, since the HIT function itself has complete access to the current buffer. After return from the HIT function, the rest of the PRINT operation will be completed.

The READMACRO Character Characteristic

A single-character READMACRO atom may be given the additional characteristic of a READMACRO character by altering its disposition in the READ scan table. A READMACRO character need not occur as an atom, but may occur at the beginning 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 as a macro (unless STATUS was used to alter the system READ tables--see codes 22-24 in subsection "The STATUS Function").

For example, redefine 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))
```

Description of Optional I/O Parameters

IOARG

The IOARG parameter, if present in an I/O call, defines the buffer-file pair to be used. It must be the value of an I/O destination atom created by OPEN, one of the system I/O destination atoms (LISPIN, LISPOUT, ERRIN, ERROUT), or NIL.

If either the buffer or file portion of the IOARG is NIL, then the appropriate system default buffer will be used. Thus, a user who wishes to specify the IOSW or FN parameters but not the IOARG can specify a NIL IOARG, and the system defaults will be used.

IOSW

The IOSW parameter is used as a switch to control system I/O parameters. The values below can be added together to specify several nondefault options on one I/O call. The default for all codes is 0.

Code	Meaning if Nonzero
01	Disable all I/O macro processing for this operation.
02	Suppress insertions of a blank character before each S-expression on output (meaningful only for PRINT and PRIN1).
04	If an atom to be printed contains any break characters (such as blanks, primes, etc.), then insert a quote character before and after the atom in the output buffer. This option allows the user to produce file output which can be read in at a later time (meaningful only for PRINT and PRIN1).
08	Print in "terse mode," that is, print only the first line of the S-expression which is to be printed (meaningful only for PRINT AND PRIN1).
16	Double-space the first output line of this print.

FN

The FN parameter, if specified, is the intercept function for the operation. It must be a LISP function specification. The operation of the intercept function is described above in the subsection "Buffer Overflow Interception."

Input/Output Functions

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

This function establishes any number of new I/O destination atoms. IODA must be a literal atom; its VALUE will be set to the new buffer-file pair which is created. BUFFER must be an integer between 1 and 255, 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 BUFFER 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. If FILE is a list of a single atom, then that atom is interpreted as a logical I/O unit number or name. 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 an N-type function which takes its arguments unevaluated. The value returned from OPEN is NIL.

(EOF IOARG)

This function closes the file associated with IOARG and reassigns it to *MSOURCE*. An end-of-file on *MSOURCE* will cause a "CONTINUE?" prompt in conversational mode, and termination of execution in batch mode.

(READ <IOARG <IOSW <FN>>>)

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.

(READCH <IOARG <IOSW <FN>>>)

READCH is similar to 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, quotes, blanks, and other special characters are treated like any other characters, and simply formed into single-character atoms.

READCH, like READ, automatically reads a new input line if it runs out of characters. The user may, however, supply an intercept function (FN) and use RETURN to abort the READCH.

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. This can be suppressed by the IOSW parameter. Similarly, multiple-character READ macros cannot be activated by READCH.

(READLINE <IOARG <IOSW <FN>>>)

READLINE causes a new line to be read into the current buffer. The previous contents of the buffer are destroyed. No LISP structures are created.

If an intercept form (FN) is supplied, it will always be called before the line is read.

The value of READLINE is the buffer containing the line that was read, or NIL, if an end-of-file occurs.

(PRINT S <IOARG <IOSW <FN>>>)

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

(PRIN1 S <IOARG <IOSW <FN>>>)

PRIN1 places the print-name of S in the current buffer, following 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.

(TERPRI <IOARG <IOSW <FN>>>)

TERPRI causes the contents (if any) of the current buffer to be printed in the current file. If the buffer is empty, TERPRI does nothing. The value of TERPRI is NIL.

If an intercept function (FN) is supplied, it will be called whenever the buffer is printed.

(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) will clear a buffer without printing it.) If the buffer has a prefix, TAB operates relative to the prefix. If N is nonpositive, 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, 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.

(SKIP N <IOARG <FILL>>)

SKIP causes a skip operation to be performed 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.

ERROR RECOVERY AND DEBUGGING PROCEDURES

Error Atoms, Error Forms, and Error Expressions

There are a number of 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 restarted 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 function will associate an error number with a given atom. After this, whenever that error type occurs, the VALUE of that atom will be used as the error form.

At present, there are three predefined 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 nonnumeric program interrupt occurs. The atom *ERR* is the error atom for all other errors.

ATTN, *PGNT*, and *ERR* are initially set to the form (DUMP 7). See the description of the dump program later in this subsection.

Note: When certain errors occur from which the system cannot recover, the message "ABORT N" is printed and the MTS subroutine ERROR is called. The abort codes which may be printed have the following significance:

1 - bad parameter on $RUN command (batch only)
2 - registers demolished
3 - stack expand failure
4 - freespace expand failure
5 - input line longer than 255 characters
6 - end-of-file from *MSOURCE* (batch only)
7 - BREAK called (batch only)
8 - program interrupt when a lock is set
9 - garbage collection disabled (see STATUS - code 49)

System Error IOARGs

It has been stated that there are initially two buffers maintained by the LISP system, the system input and output buffers. 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; 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 operations which do not specify their own IOARGs.

(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., simply enter 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.

A call to BREAK in batch mode causes execution to terminate.

(RES <N>)

RES is the LISP internal restart function, and may be called to continue the current evaluation after an attention interrupt, a timer interrupt, or a STEP error (see the description of the STEP function later in this subsection). 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 evaluated.

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 execution 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).

(DUMP <N <SW>>)

DUMP is the LISP system dumping and traceback program. DUMP can be called in two modes. The first mode occurs when no second argument is given. In this mode, the value of N indicates what 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 description specify the relative order of performance of the various actions. The default value of N is 7.

Code	Action
1	Print current error message and expression which generated the error (1).
2	Print a traceback of EVAL forms. The number of levels to be printed is determined by the system traceback number (5).
4	Call BREAK (6).
8	Print PSW and contents of general registers (2).
16	Dump 32 bytes of memory starting 16 bytes before PSW location (3).
32	Dump LISP stack data (4).

(DUMP), the default form for all errors, causes the error message and error-generating form to be printed, a traceback to be given, and a break loop to be entered.

There are three internal parameters controlling the traceback produced by DUMP. These may be altered by calling STATUS. The first parameter is the terse printout switch. Ordinarily, only the first output line of each expression is printed by dump, in order to eliminate long tracebacks. By calling STATUS the user may reset this parameter and receive full traceback printout. The second parameter controls the printing of arguments. Ordinarily the CAR function specification and CDR argument list of each form in the traceback will be printed. The user may, by calling STATUS, suppress the printing of the argument list and receive a traceback of function specifications only. The third parameter controls how many forms will be traced. This defaults to 3, but may be set to any number.

(DUMP 0) is a special code which causes a traceback of all outstanding EVAL forms 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 be used only 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 form (SW is rounded to a multiple of 16). If SW is not a number, then N is assumed to be the address of a LISP structure, and that structure is printed.

DUMP always returns NIL.

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.

(UNEVAL STACKID <S>)

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 is to be EVALed, plus all relevant information needed to restart at that level.

When the first argument to UNEVAL is a negative integer, it refers to the Nth previous EVAL block on the stack. When the first argument to UNEVAL is a positive integer, it refers to the Nth EVAL block on the stack, beginning with the top-level form.

For example, if the program is in a break loop, and the user enters (UNEVAL -1), the last form sent to EVAL will be returned. This will be (BREAK) if the program 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 an expression S which is not an integer, then the argument refers to the most recent call to EVAL for which the CAR of the form to be evaluated was equal to S. For example, if (UNEVAL 'ASSOC) is evaluated, 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, an error is generated.

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 restarted at that level. Thus, if (UNEVAL 'ASSOC T) is evaluated, the system will exit from its current level, unbind all bindings back to the point where ASSOC was called, and restart the call to ASSOC.

If the second argument to UNEVAL is anything other than T, then execution is restarted 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 the user evaluates (UNEVAL -4 '(APPEND X Y)), the system will unbind to the fourth 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 restores only the values of variables bound in LAMBDA or PROG

expressions, and will not restore altered data structures, property-lists, or VALUEs of free variables changed via SET or SETQ.

(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 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 be 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.

If FN is an atom which is currently defined as an EXPR but the PVAL associated with the EXPR property is not a LAMBDA-expression, then the PVAL will be substituted for FN and the search will continue.

If FN is an atom without a SUBR or EXPR type function definition, or if FN is any other S-expression, then FN is EVALed, its value is substituted for itself, and the search continues.

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

For example,

```
(GETFN '(LAMBDA (X) X) (LAMBDA (X) X)
(GETFN 'CAR) = * .
(DEFUN EX (X Y) (CONS X Y)) = EX
(SETQ A EX) = EX
(GETFN 'A) = (LAMBDA (X Y) (CONS X Y))
```

The SUBR atom will be printed as an asterisk, but it may be dumped, compared to other addresses, or transferred to the PLISTs of other atoms. This example assumes A has no function definition on its PLIST.

(DISPLAY STACKID <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:

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

(2) If the EVAL block is a COND, PROG, SELECT, AND, OR, or 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 subform of the LAMBDA to be EVALed.

(3) The level of the EVAL block (starting with depth 1 for the top-level form).

The first argument to DISPLAY has the same significance as the first argument of UNEVAL. If it is an integer, it refers to the Nth or Nth-previous EVAL block. If it is not an integer, it refers to the most recent EVAL block which has STACKID as its CAR. As in UNEVAL, a negative integer references the Nth previous form. If the EVAL block referenced does not exist, an error will be generated.

The second argument to DISPLAY is: B for binding (option 1 above), F for form (option 2 above), and L for level (option 3 above).

The third argument to DISPLAY is given whenever the second argument is B. 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.)

Note: In DISPLAY mode F, it is possible to find a COND, SELECT, PROG, AND, OR, or a LAMBDA block on the stack which is not yet being executed. This will occur if the user issues an attention interrupt 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 will be generated.

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

(MODIFY STACKID <B,F> <A> S)

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

The argument of MODIFY have the same significance as those of DISPLAY, except that S 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, S will be EVALed and its VALUE will be used as the replacement binding or expression.

The VALUE returned from MODIFY is the value of S.

(ERR S)

This function generates a type 15 error (see the later subsection "Error Codes"), with S treated as the expression which generated the error (error expression). In addition, the atom ERR is set to S.

(STEP N1 <N2>)

The STEP function causes subsequent entries to EVAL, exits from EVAL, or both, to be counted. When this count reaches N1, an error is generated. The error message "STEP DONE - IN" or "STEP DONE - OUT" will be printed. The error form will be the argument to EVAL for "IN", and the value being returned from EVAL for "OUT".

Under standard error processing, a traceback will be printed and a break loop entered. The user may, however, substitute other actions by defining his own error functions for the STEP errors.

If N2 is 1, STEP will count only the number of times EVAL is entered; is N2 is 2, it will count only the number of times EVAL is exited. If N2 is 3, STEP will count both entries to and exits from EVAL. N2 defaults to 1. Any error which occurs under step control will terminate the counting process.

(STEP NIL) causes step control to terminate.

(TRACE A1...AN) and (UNTRACE A1...AN)

The TRACE function allows the user to put an internal trace indicator on an atom. Whenever that atom is called as a function, the atom and its arguments will be printed on entry, and the VALUE returned will be printed on exit. UNTRACE removes the internal trace indicator.

System tracing can be disabled globally by calling UNTRACE with no arguments. Any call to TRACE will cause system tracing to be in effect again, e.g., (TRACE).

By putting an internal trace indicator on the atom T, i.e., (TRACE T), the user can cause all trace output to be in terse mode; that is, for each entry to or exit from a traced routine, only one line of trace output will be printed. Removing the trace indicator from T, i.e., (UNTRACE T), causes tracing to revert to normal mode.

TRACE and UNTRACE are both N-type functions, and their argument-designators are not EVALed.

| The following functions may not be traced or BUGged: LAMBDA, NLAMBDA, FLAMBDA, LABEL, and all arrays.

Error Codes

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 7), which includes a printout of the current error message and error expression, these will normally be printed every time an error occurs. Error types 0, 1, 3, and 4 use NIL for their error expression. Other errors use as an error expression the argument which caused the error, unless otherwise noted.

Code	Meaning
0	Program interrupt. Likely to be caused by a CDR operation on a numeric atom. For this error only, an attention interrupt which occurs during the printing of the error message will cause an immediate return to MTS.
1	Attention interrupt.
2	Timer interrupt.
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 too few or too many subscripts.
7	An atom used as a function specification had a SUBR, NSUBR, or FSUBR property on its PLIST, but the PVAL was not a LISP subroutine.

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 An atom is undefined or a function definition is missing.

17 Infinite EVAL loop--the function specification in process by EVAL is an atom which has no system function definition on its property-list, and which has itself as its VALUE.

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 nonpositive or greater than 255 characters.

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.

23 Attempt to reset a buffer to a size less than its current contents.

24 The number of steps specified in a STEP call have been completed.

25 A dummy argument of a PROG or LAMBDA, or the function name of a LABEL, was not an atom.

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, where the second argument is not the name of any outstanding no-spread dummy argument.

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

29 An attempt to define an external SUBR with an illegal type specification.

30 LISP could not find or could not 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 nonpositive or exceeded the limits of that subscript position.

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

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

34 A call to CHECKPOINT or RESTORE which did not specify a line file, or a call to RESTORE which specified a file which was not produced by the current CHECKPOINT.

35 The number of steps specified in a STEP call have been completed.

36 A call to RETURN, 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 Unbalanced angle brackets were encountered.

SPECIAL FEATURES

The STATUS Function

The STATUS function is used for two purposes--to get and to set the values of system switches and parameters. There are two types of status calls. The first interrogates the system and returns the value of a

system parameter, and the second supplies a value which is to replace the system parameter.

The various system parameters are identified by status codes. Type I status codes are used to get and set parameters associated with buffers, files, arrays, and atoms. To get one of these parameter values, the argument to STATUS will be of the form:

(STATUS-CODE NAME)

where NAME is the appropriate I/O destination atom, array, or atom.

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

(STATUS-CODE NAME VALUE)

where VALUE is the new value for the parameter.

Type II status codes are used for general system switches and parameters. To get and set these parameters, the argument to STATUS will be of the form:

STATUS-CODE	Gets parameter values.
(STATUS-CODE VALUE)	Sets parameter values.

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

Note: In a call to STATUS, the status code parameter may be any atom, and its value (which must be a legal status code) will be used as the actual status code. This allows mnemonic definitions to be given to status codes, e.g.,

(STATUS (SETPFX ABUF NIL))

where the VALUE of SETPFX is 10.

In addition, where a status code is set to some numeric value, the VALUE it is set to may also be any atom, and the VALUE of the atom will be used. For example,

(STATUS (SETGC GCBIG))

where the VALUE of SETGC is 45, and the VALUE of GCBIG is a numeric atom.

Type I Status Codes--Buffer, File, Array, and Atom Characteristics

Code	Meaning
1	This status code is used to get or set the error atom associated with a particular error number. (See the subsection "Error Recovery and Debugging Procedures" above 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)), which will set A to 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 code is used to get or set the 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 a READMACRO. If the switch is non-NIL, then whenever the atom appears as part of an S-expression which is read in, it will be treated as a READMACRO as described in the subsection "READMACRO and PRINTMACRO Functions." The initial value of this parameter for all atoms is NIL.
3	(Reserved.)
4	This status code 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 enabled, whenever the atom is printed into a buffer, the atom will be treated as a PRINTMACRO as described in the subsection "READMACRO and PRINTMACRO Functions."
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 characters (e.g., blanks). 4 Left parenthesis "(". 8 Right parenthesis ")". 12 End-of-line (or semicolon). 16 Period. Signifies dotted-pair or number. 20 Plus sign "+". Signifies beginning of a number. 24 Minus sign "-". Signifies 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). 44 Quote character. Special processing. 48 Left angle bracket "<". 52 Right angle bracket ">".

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 literal 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 code is used to get the number of dimensions of an array. Its argument must be an array name.

9 This status code is used to get or set the size of a buffer (i.e., the right margin). The buffer size includes the buffer prefix (if any), and may not exceed 255 characters.

10 This status code is used to get or set the length of the buffer prefix for a buffer. Evaluating the function (STATUS (10 IODA T)) freezes the current contents of the buffer associated with IODA as a prefix, and returns the length of the previous prefix. Evaluating the function (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 code 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 reset to zero whenever a TERPRI or a physical write operation is performed. A TAB or SKIP to a smaller number will reset the pointer to the smaller number.

12 This status code is used to get or set the 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. For a description of the form of the EOF function and the significance of the value returned from it, see the subsection "End-of-File Processing."

The initial value of this parameter for all files is the system function EOF.

13 This status code 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 I/O to or from the file will be echoed on *MSINK*. If the value is NIL, echoing will not occur. The global echo switch overrides the individual file switches if the global switch is NIL. Otherwise, the individual file switches control echoing.

The initial value of this parameter for all files is NIL.

14 This status code 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 code 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 code 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 section, "Files and Devices," in MTS Volume 1, for a description of the significance of modifier values.

The initial value of this parameter for all files is 0.

17 This status code is used to get the maximum output line length for the MTS file or device attached to a LISP file. The argument given must be an I/O destination atom.

18-19 (Reserved).

Type II STATUS Codes--System Switches and Parameters

Code	Meaning
20	Default standard input IOARG. Initially set to the dotted-pair of the system input buffer (size 255 characters) and SCARDS.
21	Default standard output IOARG. Initially set to the dotted-pair of the system output buffer (size 70 characters) and SPRINT.
22	Default 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 characters) and GUSER.
23	System error output IOARG. Used in break loops in place of standard output IOARG. Initially set to the dotted-pair of the system error output buffer (size 70 characters) and SERCOM. Note: These initial IOARGs may be obtained by calling STATUS. They are also the initial values of the atoms LISPIN, LISPOUT, ERRIN, and ERROUT (for the user's convenience).
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 traceback. (0 = none, -1 = all). Defaults to 3.
27	Traceback argument switch. Zero specifies only function specifications (i.e., CAR of EVAL form) are to be printed on EVAL form traceback. Any number greater than zero specifies both CAR and CDR of form (i.e., both function specification and arguments) are to be printed. The switch is initially set to 1.
28	Most recent expression which generated an error (get only).
29	Error number of most recent error (get only).
30	Terse traceback switch. Zero specifies traceback output in terse mode should be printed (only one line is printed for each expression.) Any number greater than zero specifies that a full traceback should be printed. Initially 0.
31	Global switch for echoing input lines on *MSINK*. T specifies echo, and NIL specifies no echo. Initially NIL.

32 System message switch.

0 No messages.
1 Print garbage collection messages (see the later subsection "The Garbage Collector").
2 Print "CHECKPOINT DONE" messages (see the later subsection "(CHECKPOINT A <S> and RESTORE").
4 Print "FREE SPACE EXPAND" messages (see the later subsection "The Parameter List").

Initially 7.

33 Batch/terminal switch. 4 specifies batch, 0 specifies terminal.

34 Interrupt trap switch. Initially 0 (all traps on).

1 Disable program interrupt trap.
2 Disable attention interrupt trap.
4 Alternate error atom control (used for writing debugging packages).

35 Step count. The value of this parameter is the number of steps remaining before a "STEP DONE" error will occur. It is meaningful only when running under STEP control.

36 Value of GENSYM counter.

37 Initialization call for TIME (get form only).

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). A timer unit is about 13.3 microseconds.

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 CHECKPOINT switch. 0 specifies exit after CHECKPOINT. 1 specifies automatic restore after CHECKPOINT. Initially 1.

44 This status code is used to get or set the current value of the FCS parameter.

45 This status code is used to get or set the current value of the GC# parameter.

46 This status code is used to get the current number of pages of freespace in use by the system.

47 This status code is used to get or set the value of the internal integer array. The value of the second parameter must be an array atom, where the elements or the array form an increasing sequence of integer atoms. When this status number is set, the first and last array elements are used to reset the limits of fast number access.

48 This status code is used to control the =FL linking option of the compiler (see the later subsection "The LISP Compiler: (COMPILE A1...AN)"). Initially T.

49 This status code enables or disables the garbage collector. Initially NIL (GC allowed).

The OBJECT LIST

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 a literal atom is read, READ compares the atom with the atoms on the OBJECT LIST. If they match, then the pointer which was created references the atom 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 to be created with the same PNAME, and the original atom will not be referenced.

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

(OBLIST)

The function (OBLIST) with no arguments returns a (long) list of all the atoms which are on the OBJECT LIST.

(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. REMOB is an N-type function and its arguments are not EVALed.

(PUTOB A1...AN)

The function PUTOB places literal atoms on the OBJECT LIST. If PUTOB finds an atom on the OBJECT LIST with the same PNAME as one of its arguments, an error will be generated.

The Parameter List

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. The keyword parameters recognized by LISP, and their significance, are described below.

FCS=N — N specifies the number of pages of initial free-space. If space is needed beyond this amount, a garbage collection will be performed. The default value is 25 pages. Increasing the value of this parameter to the maximum space needed will eliminate the necessity for garbage collection.

GC#=N — After a garbage collection, the system will get more space unless N LISP cells are available. Setting N to a large number will tend to increase the amount of memory used by the system and decrease the frequency of garbage collections. The default value is 4000.

ERR=N — N indicates the initial status of interrupt traps (see status code 34). The default value is 0 (all traps on).

OBJ=N — N indicates the number of hash buckets for the literal atom OBJECT LIST. The greater the number of buckets, the faster resolution of atomic references should be. An odd number is recommended. The default value is 69.

INT=N1[,N2] — If the form INT=N1 is specified, all integer atoms from 0 to N1 will be stored in an internal array where they can be accessed quickly. Alternatively, if the form INT=N1,N2 is specified, all integer atoms from N1 to N2 will be stored in an internal array where they can be accessed quickly.

The user can access the INT array by calling STATUS. The INT array can also be changed by creating a new array containing any set of consecutive integer atoms, and calling STATUS with that array as an argument.

The TIMER Function: (TIMER ID SW)

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

The significance of the SW argument is as follows:

SW	ID	Meaning
0 < N < 1001	Any non-NIL structure	Set up an interrupt structure 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 from TIMER is ID.
1000 < N	Any non-NIL structure	Set up an interrupt structure 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 from TIMER is ID.
T	Any non-NIL structure	If there is an outstanding structure TIMER request with an ID which is EQ to ID, then TIMER returns the clock time remaining in that request. Otherwise TIMER returns NIL.
NIL	NIL	Cancel all outstanding TIMER requests. The value of TIMER is NIL.
NIL	Any non-NIL structure	Cancels the pending structure interrupt request, if any, associated with ID. The value of TIMER is the remaining clock time in that request.

Examples:

```
(TIMER 'X 1.E6) = X
```

A timer interrupt is set up with the ID X for one second of CPU time.

```
(TIMER T 20) = T
```

A timer interrupt is set up with the ID T. The interrupt will occur after 20 seconds of elapsed time.

```
(TIMER T NIL) = XX
```

The interrupt is canceled, and the remaining time is returned.

```
DEFUN TCOUNT (X Y) (TIMER T Y) (EVAL X)
      (SUB Y (TIMER T NIL)))
```

Here a function TCOUNT is defined. TCOUNT takes a form X to be evaluated, and a number Y which is the maximum time allowed to it. TCOUNT will either generate a TIMER error, or return the time it took to EVAL the form (plus a small amount of overhead).

The Garbage Collector

This section only briefly describes the garbage collection routine in the LISP system. This routine is activated when a job runs out of space needed to create new LISP structures. The garbage collector reuses space which is occupied by unreferenced structures, allocates more space if necessary, and notifies the user if the maximum allowable space is exceeded.

The user may optionally receive a message at the end of each garbage collection (see the STATUS function) indicating that the garbage collection has occurred.

Two attention interrupts issued during garbage collection will cause an immediate return to MTS. A restart from MTS will return to the garbage collector and continue execution.

(CHECKPOINT A <S>) and (RESTORE A)

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. In addition, checkpointing often is much less expensive than garbage-collecting a large program if the collection can be anticipated.

| (CHECKPOINT A) saves the current system in the MTS file A. Both
| sequential and line files are supported; however, line files are more
| efficient for storing many checkpoints.

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

(CHECKPOINT A S) checkpoints only the LISP structure S. On a restore of the file A, the system will be augmented by structure S.

Note: The arguments to CHECKPOINT and RESTORE are not IOARGs. They are actual MTS file names. The user should not attempt to open a file for the purpose of CHECKPOINT and RESTORE.

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 number of pages of memory used by his program. In addition, (CHECKPOINT A), which destroys freespace, immediately initiates a restore of the system which it just checkpointed. However, if the appropriate status code is set, CHECKPOINT will not initiate a
| restore, but will terminate execution. Upon termination of a RESTORE, a
| message is printed on *MSINK* indicating what was restored and when it
| was last checkpointed.

CHECKPOINT and RESTORE are N-type functions which do not have their arguments evaluated.

Notes:

(1) On a restore of a specific structure S, it may be the case that an atom A occurs in the structure to be restored, and there is already an atom A on the system OBJECT LIST. In general, the VALUE and PLIST of the existing atom A will be modified to the VALUE and PLIST of A at the time of the checkpoint, and this atom will be referenced by the structure being restored. Thus, structures which referred to A before the restore was performed will find that the same atom has been given a new VALUE and PLIST by RESTORE. However, the user may reverse this priority by setting the PLIST of an atom to *UNDEF* before he performs the checkpoint. In that case, when the restore takes place, if there is no atom A on the OBJECT LIST, then A will be created, and both its VALUE and PLIST will be *UNDEF*. If there is an atom A, however, the checkpointed structure will reference it, but its VALUE and PLIST will not be altered.

(2) After a total system checkpoint file is restored, the system will begin reading from the current input buffer (usually 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).

(3) A call to CHECKPOINT with a specific structure S will not do an automatic restore operation after the checkpoint is completed, but rather will always terminate execution. This is necessary because there may be references to the checkpointed structure which no longer exists.

(4) Two attention interrupts occurring during a checkpoint or restore will cause an immediate return to MTS. A restart from MTS will return to CHECKPOINT or RESTORE and continue execution.

(5) The user of CHECKPOINT and RESTORE should be aware that if the LISP I/O units have been modified before a checkpoint was performed, then the same modifications will be in effect immediately after a restore is performed.

Automatic Restoration of LISP Functions

Since the VALUE of an atom which has no function indicator on its PLIST will be EVALed when the atom is used as a function specification, setting the VALUE of an atom to a RESTORE form can have the effect of making that atom a "load-on-call" function.

For example, if the atom FN is set to the form (RESTORE FN), and FN is a proper checkpoint file (but not a total system checkpoint file), then the structures in FN will be automatically restored the first time FN is called. The newly restored function property of the atom FN will be found and used, making the restore process transparent to the user.

The following functions have values initially set to RESTORE forms which will cause large packages of functions and structures to be loaded from the public file *LISPLIB:

COMPILE	SET2
EDIT	SETA2
DEBUG	GRAFT2
NEWWORLD	DELETE2
GETWORLD	PUT2
REALWORLD	REM2
RPLACA2	ADDPROP2
RPLACD2	UNCONS2
SETQ2	

Creating a LISP Library

A special feature of LISP allows the user to create a library of checkpointed structures in a single MTS file. The operations to accomplish this take the following form:

(CHECKPOINT (FILENAME . ENTRYNAME) <S>)
(RESTORE (FILENAME . ENTRYNAME))

Whenever a simple (CHECKPOINT FILE) or (RESTORE FILE) is executed, ENTRYNAME is given a default value of LISPSTD. Thus, (RESTORE A) = (RESTORE (A . LISPSTD)).

Note: An attempt to recheckpoint an ENTRYNAME into a file where that ENTRYNAME was already checkpointed will cause the original version of ENTRYNAME to be replaced; however, if the file is a sequential file, any other checkpoint entries which occur after ENTRYNAME in the file will be destroyed. The use of line files is recommended.

Direct Memory Modification: (STATUS (0 N A))

This special status code permits the user to alter up to seven consecutive bytes of memory to any value. Obviously, this is done at the user's own risk.

N must be an atom whose VALUE is a numeric atom representing the first address which will be modified. A is an I/O destination atom whose associated buffer contains the data to be inserted in memory, starting at address N. The first character in the buffer must be the character X. This must be followed by an even number of hexadecimal digits, up to a maximum of 14, representing half the number of bytes to be modified. Alternately, A may be a literal atom whose PNAME has the same form as the buffer contents described above.

Example:

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

If the buffer TBUF contains the characters "X00000000", then the VALUE of the atom ZAP will be set to 0. An attempt to evaluate (CAR ZAP) will generate a program interrupt.

(LTR S SW)

The LTR function may occasionally be useful for altering the normal process of evaluating lists, e.g., argument lists. Its effects may be confusing, and its use is recommended only for advanced users. 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, PROG, or the "S1...SN" portion of a COND. It may also be invoked during evaluation of a sequence 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.

Value	Meaning
SW = NIL	Do not 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.

Example:

```
(REM (READ) (LTR (READ) X) (READ))
```

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

LTR stands for "list terminate or redirect."

(MTS <A>)

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.

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

MTS always returns NIL.

The Transport System

LISP 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 the state of all LISP structures at a particular moment is considered 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. A system of hypothetical worlds can be pictured 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 a LISP structure. When the user returns to a dominating world, the alterations he has performed are undone, 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.

Creating Hypothetical Worlds:

All reversible alterations of a LISP structure must be made using special functions defined for that purpose. These functions are part of the transport system RESTORE package. When any function that is part of the transport system is called, the entire system is restored from *LISPLIB.

The functions listed below work exactly like the corresponding LISP system functions except that the alterations they make to a LISP structure are reversible.

The functions are:

RPLACA2	SETQ2
RPLACD2	SET2
GRAFT2	SETA2
DELETE2	PUT2
ADDPROP2	REM2
UNCONS2	

Note: The functions MAPCAN and MAPCON do not have transport system duplicates, even though they alter LISP structures.

(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. For example, (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. NEWWORLD does not cause a transfer to any other world. Its purpose is to create tickets.

(GETWORLD S)

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

(GETWORLD EARTH)

(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.

The LISP Compiler: (COMPILE A1...AN)

For each atom Ai, the LISP compiler finds the EXPR property on its PLIST and, if it is a LAMBDA-expression, translates it into machine code to perform the same computation. This machine code program, in the form of a LISP "SUBR atom," is put on the PLIST of the atom Ai under the appropriate indicator (SUBR for LAMBDAs, NSUBR for NLAMBDAs, FSUBR for FLAMBDAs), and the EXPR property is removed.

COMPILE is an FLAMBDA function which takes its arguments unEVALed.

When COMPILE is called, the compiler will be loaded automatically from *LISPLIB. Evaluating the form (=EXCISE) will remove it.

Compiler Features

Declarations--(DECLARE <A,LA> IND <PVAL>)

The function DECLARE, an NSUBR, is used to control the various compiler options described below. The arguments to declare take the form:

(DECLARE <A,LA> IND <PVAL>)

where the property IND with property value PVAL (or T if PVAL is omitted) will be put on the PLIST of the atom A or the atoms in LA. For example, (DECLARE (A B) =SPECIAL)

Declarations may be made before compiling a function, or may be made during the process of compilation by inserting a (DECLARE...) as a top level form of a LAMBDA or PROG. The declaration will not cause any code to be produced, and will be in effect until it is overridden by another declaration.

Variable Types

Normally, a variable which occurs as an argument, a PROG variable, or a LAMBDA-variable within a compiled function is assumed to be local to that function. Its value is set when the variable is first bound, and may be changed using SETQ or UNCONS, but the variable has no relation to the LISP atom of the same name. However, if such a variable is declared to be special: (DECLARE X =SPECIAL) all references to it are assumed to refer to the LISP atom of that name, and the value of the atom will be updated accordingly.

References to variables which are not arguments, PROG variables, or LAMBDA-variables (i.e., FREE variables), are assumed to refer to LISP atoms in the usual way. Warning: All references to atoms which occur in QUOTEd expressions, including the arguments to N-type and F-type functions that are implicitly QUOTEd, will refer to LISP atoms and not to local variables of the same name. A user who wants to pass a local variable to an FSUBR must APPLY the FSUBR, or declare the variable to be special.

Setting the variable =SPECIAL to T causes all variable references to be compiled as if the variable had been declared =SPECIAL.

Function Types

All external functions called by compiled programs are assumed to be SUBR or LAMBDA-type EXPRs unless they are declared otherwise. The function-type declarations which are available are:

(DECLARE F =TYPE FLAMBDA)

for FSUBRS or FLAMBDA-type EXPRs,

(DECLARE G =TYPE NLAMBDA)

for NSUBRS or NLAMBDA-type EXPRs, and

(DECLARE H =TYPE =ARRAY)

for arrays.

An atom may also be declared to be an array name with the following declaration:

```
(DECLARE H =ARRAY)
```

Interpreter functions which are N-type or F-type functions are already declared correctly within the compiler.

Number Types

The LISP compiler assumes that no floating point numbers will ever be generated, and therefore compiles all numeric calculations to perform integer arithmetic.

The user can change this assumption with the declaration:

```
(DECLARE =SYSFLAG =INTEGERS NIL)
```

in which case all functions that allow floating-point arguments will be compiled as calls to the interpreter.

Block Compiling

Within a compiled program, any internal LAMBDA-expression is defined to be a separate "block," and each compiled program consists of one primary block and zero or more secondary blocks.

In addition, external functions which are defined as LAMBDA-expressions may be declared to be "macros," and the compiler will replace the function name with the LAMBDA definition, and compile the LAMBDA-expression as an internal block.

The important property of an internal block is that it has access to the local variables defined in the block which called it (and all higher blocks).

For example, suppose function A calls function B, and

```
(DECLARE B =MACRO)
```

was in effect when A was compiled. Then the SUBR code for A contains a compiled "copy" of B. If X is a local variable within A, all free references to X within the copy of B will refer to that local variable.

The number of blocks compiled and the length (in bytes) of code produced for each block may be obtained via a printed map on LISPOUT by setting the variable =MAP to T before compiling.

The Fixed-Link Option

The linkage from a compiled function to another compiled function that it calls, or an array that it accesses, may be declared to be "fixed-linked" (=FL), provided that the definition of the called function or array will not change after the first time it is accessed by the calling function.

The first time the =FL call is executed, the pointer to the atom that is the =FL function or array name is replaced by a pointer to the actual SUBR code or array area. This replacement eliminates the property-list search of the function or array name on all subsequent executions of the call.

A function or array is declared to be fixed-linked as follows:

```
(DECLARE F =FL)
```

If, on the first execution of a =FL linkage, the system cannot find a SUBR property, or if STATUS code 48 is set to NIL, the system uses a normal linkage, and the =FL declaration is ignored for that call.

Setting the variable =FL to T causes all function and array references to be compiled as fixed-linked, unless the function or array has been declared =SL as follows:

```
(DECLARE F =SL)
```

Note: Only compiled functions and arrays may be declared =FL.

Note: It is virtually impossible to debug a large LISP system that is heavily fixed-linked, since =FL calls do not generate any evidence on the stack, or in the trace-back. When an error occurs, the user has no way to determine which of his =FL functions was being executed. Therefore, when compiling a large system, the following procedure is recommended:

(1) Compile all functions with =FL declarations in their final form.
(2) Excise the compiler.
(3) CHECKPOINT the system before performing any executions. The =FL linkages will not yet be resolved.
(4) Set STATUS code 48 to NIL to suppress completion of fixed links.
(5) Test the system thoroughly.
(6) RESTORE the original unexecuted system, set STATUS code 48 to T, and proceed.

The =CHECK Option

If a function has been declared to be a =CHECK function as follows

(DECLARE F =CHECK)

then when it is compiled, the generated code will include a test to check that the correct number of arguments was passed to the function, and various other error checks (for valid atomic, list, or numeric arguments, etc.) will be made.

Normally, little or no error checking is performed by compiled programs.

If the variable =CHECK is set to T, all functions will be compiled as if they had been declared =CHECK.

Functions Known to the Compiler

The following functions have fixed definitions within compiled code:

ABS ADD ADD1 AND APPLY APPLY1 ARG ASSOC ATOM C...R COND CONS IDIVIDE EQ EQUAL EVAL EVEN GET GO GRAFT GREATER LAND LESS LENGTH LIST LOR LXOR MAP MAPC MAPCAN MAPCAR MAPCON MAPLIST MAX MEMBER MIN MINUS NOT NTH NUMBER OR PROG PROGN PUT QUOTE REMAIN REPEAT RETURN REVERSE SELECT SET SETA SETQ SHIFT SUB SUB1 TIMES UNCONS ZERO

The following functions are declared to be fixed-link functions:

| CHECKPOINT, COPY, DECLARE, DEFUN, DISPLAY, LABEL, MODIFY,
| NEWWORLD, OBLIST, OPEN, REALWORLD, REM, REMOB, SETA2, SETQ2,
| STATUS, TRACE, UNTRACE,

Limitations and Warnings

(1) General Warning:

Compiled functions do a minimum of error checking, so users are advised to debug their programs before compiling them. Unless the =CHECK option is used, the normal checks for undefined variables, bad atomic, list, and numeric arguments, array dimensions exceeding legal limits, etc., do not occur in compiled code.

(2) Since fixed-link functions are not executed under control of EVAL, debugging features such as TRACE, BUG, and STEP cannot be used with them.

(3) An APPLY of a no-spread macro or LAMBDA-expression cannot be compiled. (Use APPLY1 instead.)

(4) Functions used as macros may not be recursive. An attempt to compile a recursive MACRO will generate an error.

(5) Each internal block must be less than one page in length. If the compiler finds that a block is too long, it will terminate with an error message. The user can then break the program downinto smaller blocks and recompile. Note: Experience indicates that about 50-75 lines of LISP code (uninterrupted by internal blocks) will compile into one page.

(6) The total number of blocks in one compiled routine cannot exceed 150.

(7) The number of bytes in the first block of a program, plus the number of special variables, times four, must be less than 4096.

(8) The number of local variables defined at one time cannot exceed 1020.

(9) The compiler occupies 40 pages of memory, and compilation of even a small program is likely to increase storage to 50 pages. After EXCISing (using the =EXCISE function described above) the compiler, the user may want to compress his core usage by performing a CHECKPOINT.

(10) Functions which depend on calls to EVAL, such as UNEVAL and RETURN with a second argument, may not operate in the same manner when called from a compiled program, since many function calls are compiled directly and do not generate calls to EVAL.

(11) The compiler does not process the LABEL function, hence the use of LABELs in compiled programs is not recommended. If a LABEL is encountered, the compiler generates a call to the interpreter.

Other Special Features

The functions EDIT and DEBUG provide access to a LISP editor and debugging package. These features are documented in the sections "The LISP Editor" and "LISP Debugging Facilities" in this volume.

Page Revised January 1983 June 1976

June 1976

THE LISP EDITOR

INTRODUCTION

The LISP editor is a LISP program designed to examine and modify LISP data structures, especially function definitions, as they exist within the LISP interpreter. It does not edit MTS files, although there are LISP editor commands which will read and write files on request. In the following section, the term "expression" refers to any LISP data structure (which is also known as an S-expression, denoting its external representation).

The LISP editor has been checkpointed into the public file *LISPLIB. It is automatically restored and invoked, using a command of the form

```
(EDIT fn)
```

The value of EDIT is NIL. The argument "fn" determines the expression to edited as follows:

(1) If "fn" is an atom with an EXPR on its property list, then the value of the EXPR property (normally a LAMBDA expression) is the expression to be edited. This is the normal method of editing function definitions.
(2) Otherwise, "fn" itself is the expression to be edited. Since EDIT is an FSUBR, "fn" is not evaluated. Thus, the above form is not normally useful for editing arbitrary structures. APPLY or APPLY1 should be used instead, e.g.,

```
(APPLY1 'EDIT list)
```

where the value of "list" is the expression to be edited.

The editor has its own command language; one or more commands may be entered on each line, separated by blanks. There are several variable length commands which must be separated from any subsequent commands on the same line by a colon (:). These are INSERT, DELETE, EXTRACT, ML, MR, BI, and BO, and are described in the subsection "Commands that Modify the Current Expression." Commands are read from *SOURCE* using the prefix character period (.), and output is written on *SINK*, using the prefix colon.

All editor commands operate on a subexpression of the original argument to EDIT; this subexpression is called the current expression. There are several commands available for specifying the current expression. They are described in later in this section.

Ignoring dotted pairs for the moment, any proper subexpression of a LISP expression (hence any current expression) must be one of the following:

(1) An atom.
(2) A list which is an element of some higher level list.
(3) A proper sublist, which will be referred to as the tail of a list.

For example, given the expression (A (B C) D), the atoms are A, B, C, and D; the list (B C) is an element of the top-level list; the tails of (A (B C) D) are ((B C) D) and (D); and the tail of (B C) is (C).

Dotted pairs (expressions whose CDR are atoms other than NIL) have not been adequately described, and indeed, the behavior of the editor is erratic at best when it encounters a dotted pair.

For the purpose of this description, editor commands are divided into five major groups: printing commands, commands specifying the current expression, commands modifying the current expression, commands undoing previous modifications, and miscellaneous commands.

COMMANDS THAT PRINT THE CURRENT EXPRESSION

Command: P [n]

The P command prints the current expression up to level "n". The optional argument n defaults to 2. To avoid excessive output, lists at level n are printed as the character ampersand (&). A current expression which is a tail of some list is indicated by ellipsis marks (...) preceding the expression.

A P command is assumed at the end of every line unless a P, ?, or PP command is the last command on the line.

Command: ?

The ? command is equivalent to a P 1000 command; it effectively prints the entire current expression. This command is used in most of the examples in this section.

Command: PP

The PP command also prints the entire current expression, but in an indented format which makes the structure more clearly visible. The PP command does not print ellipsis marks for tails of lists.

Examples:

```
.?
:(LAMBDA (X) (COND ((NULL X) NIL) (T (CONS X X))))
.P
:(LAMBDA (X) (COND & &))
.P 3
:LAMBDA (X) (COND (& NIL) (T &)))

.PP
:(LAMBDA (X)
:         (COND ((NULL X) NIL)
:               (T (CONS X X))))
```

COMMANDS THAT SPECIFY THE CURRENT EXPRESSION

Command: [±]n

A positive integer "n" or "+n" selects the nth element of the current expression, counting from the left, and makes it the current expression. A negative integer "-n" operates in the same manner except that it counts from the right. Users should note that zero is a separate command, described later in this section.

Examples:

```
.?
:(LAMBDA (X) (COND ((NULL X) NIL) (T (CONS X X))))
.3 ?
:(COND ((NULL X) NIL) (T (CONS X X)))
.2 1 ?
:(NULL X)
```

Command: UP

If the current expression is an element of a higher-level list, the UP command specifies the tail of the higher-level list beginning with the current expression, as the new current expression. Otherwise, the UP command has no effect. Note that if the current expression is the first element of a higher-level list, the UP command produces the entire list, rather than just a tail of the list.

Examples:

```
.?
:(LAMBDA (X) (COND ((NULL X) NIL) (T (CONS X X ))))
.3 ?
:(COND ((NULL X) NIL) (T (CONS X X)))
```

```
.UP ?
:... (COND ((NULL X) NIL) (T (CONS X X))))

.?
:... (COND ((NULL X) NIL) (T (CONS X X))))
.1 ?
:(COND ((NULL X) NIL) (T (CONS X X)))
.1 UP
:(COND ((NULL X) NIL) (T (CONS X X)))
```

Command: !0

The !0 command is similar to the UP command, except that it produces the entire higher-level list, rather than a tail of that list. In addition, if the current expression is a tail of some list (in which case the UP command has no effect), the !0 command specifies the entire list as the new current expression.

Examples:

```
.?
:(LAMBDA (X) (COND ((NULL X) NIL (T (CONS X X))))
.3 UP ?
:... (COND ((NULL X) NIL) (T (CONS X X))))
.!0 ?
:(LAMBDA (X) (COND ((NULL X) NIL)(T (CONS X X))))
.3 !0 ?
:(LAMBDA (X) (COND ((NULL X) NIL)(T (CONS X X))))
```

Command: 0

The 0 (zero) command has an effect somewhere between UP and !0. Since its precise effect defies description, its use is not recommended.

Commands: NX, BK

The NX command specifies the next element to the right of the current expression as the new current expression. It is equivalent to the command sequence UP 2. The BK command is the inverse of NX; it specifies, as the current expression, the next element to the left.

Examples:

```
.?
:(LAMBDA (X) (COND ((NULL X) NIL) (T (CONS X X))))
.1 ?
:LAMBDA
.NX ?
:(X)
.NX ?
:(COND ((NULL X) NIL) (T (CONS X X)))
```

```
.1 ?
:COND
.NX ?
:((NULL X) NIL)
.NX ?
(T (CONS X X))
.BK ?
:((NULL X) NIL)
.BK ?
:COND
```

Command: !NX

The NX command fails if the current expression is the last element of a list. The !NX command is similar to the NX command, but it moves to a higher level if necessary to reach an element which is not the last in a list.

Example:

```
.?
:(LAMBDA (X) (COND ((NULL X) NIL) (T (CONS X X))))
.3 2 1 ?
:(NULL X)
.NX ?
:NIL
.NX ?
:YOU'RE AT THE END
.!NX ?
:(T (CONS X X))
```

Command: #

The # command specifies the original argument to EDIT as the new current expression.

Command: ¬P

The ¬P command restores the current expression to what it was at the next-to-last P command. It is useful for switching back and forth between two expressions.

Example:

```
.P
:(LAMBDA (X) (COND & &))
.F NULL P ¬P
:(NULL X)
:(LAMBDA (X) (COND & &))
```

Command: F s

The F command finds the first occurrence, in print order, of an element in the current expression which is EQUAL to the S-expression "s". This element becomes the current expression. If s is an atom, the F command is implicitly followed by an UP command.

Examples:

```
.?
:(LAMBDA (X) (COND ((NULL X) NIL) (T (CONS X X))))
.F X ?
:(X)
.# F NULL ?
:(NULL X)
.# F COND ? F X ?
:(COND ((NULL X) NIL) (T (CONS X X)))
:... X)
.!0 ?
:(NULL X)
.UP UP ? F CONS ?
:... ((NULL X)NIL) (T (CONS X X)))
:(CONS X X)
.# F (NULL X) ?
:(NULL X)
```

COMMANDS THAT MODIFY THE CURRENT EXPRESSION

All of these structure-modifying commands require a location specification to determine the element(s) to be modified. This location specification must be a sequence of S-expressions, where each S-expression is either:

(1) Any command (except F) from the above subsection, or
(2) Anything else, in which case an F command is implied with the S-expression as argument.

These location specification "commands" are executed in the order specified, beginning with the current expression, to determine the expression to be modified. However, the location specification does not determine a new current expression. A location specification is indicated in the following subsection by "p1...pn" or "q1...qn".

Since all of the commands in this subsection, except EMBED, are of variable length, they must be followed by a colon if further commands are entered on the same line.

Command: INSERT s1...sn {BEFORE|AFTER|FOR} p1...pn

The INSERT command, which may be abbreviated to I, inserts the sequence of S-expressions "s1...sn" before, after, or in place of, the element determined by the location specification "p1...pn".

Examples:

```
.?
:(LAMBDA (X) (COND ((NULL X) NIL) (T (CONS X X))))
.INSERT (A B C) AFTER X : ?
:(LAMBDA (X) (A B C) (COND ((NULL X) NIL) (T (CONS X X))))

.INSERT B BEFORE X 1 : ?
:(LAMBDA (B X) (A B C) (COND ((NULL X) NIL) (T (CONS X X))))

.F NULL ? INSERT B FOR X : ?
:(NULL X)
:(NULL B)
.INSERT ABC D AFTER B : ?
:(NULL B ABC D)
```

Command: DELETE p1...pn

The DELETE command removes the element designated by "p1...pn" from the structure. This command may be abbreviated to D.

Examples:

```
.?
:(LAMBDA (Q X) (A B C) (COND ((NULL X) NIL) (T (CONS X X))))
.DELETE Q 1 : ?
:(LAMBDA (X) (A B C) (COND ((NULL X) NIL) (T (CONS X X))))

.DELETE (A B C) : ?
:(LAMBDA (X) (COND ((NULL X) NIL) (T CONS X X))))

.DELETE NIL ?
:(LAMBDA (X) (COND ((NULL X) (T (CONS X X))))
```

Command: EMBED p1...pn IN s

The EMBED command replaces every occurrence of the atom "*" in the S-expression "s" with the element designated by "p1...pn", and then replaces the element designated by "p1...pn" with the result. The EMBED command may be abbreviated to EM.

Examples:

```
.?
:(COND ((NULL X) NIL) (T (CONS X X)))
.EMBED NULL IN (PRINT *) ?
:(COND ((PRINT (NULL X)) NIL) (T (CONS X X)))
```

Command: EXTRACT p1...pn FROM q1...qn

The EXTRACT command replaces the element designated by "q1...qn" with the element designated by "p1...pn". The element designated by "p1...pn" must be a substructure of that designated by "q1...qn UP". EXTRACT may be abbreviated to EX.

Examples:

```
.?
:(COND ((NULL (CAR X)) NIL) (T (CONS  (CAR X)(CADR X))))
.EXTRACT CAR FROM NULL : ?
:(COND ((CAR X) NIL) (T (CONS (CAR X) CADR X))))
.EXTRACT CONS FROM COND : !0 ?
:(LAMBDA (X) (CONS (CAR X) (CADR X)))
```

The remaining commands in this section move elements to higher or lower levels in the structure. In the printed form, this appears to have the effect of adding, removing, or shifting parentheses.

Commands: ML ±n p1...pn
MR ±n p1...pn

The ML command moves the left parenthesis of the element designated by "p1...pn" ±n elements to the right. The MR command moves the right parenthesis in the same way. "p1...pn" must designate a nonatomic element.

Examples:

```
.?
:(A B C (D E F) G H I)
.ML -1 4 : ?
:(A B (C D E F) G H I)

.ML 3 C : ?
:(A B C D E (F) G H I)

.MR 2 F : ?
:(A B C D E (F G H) I)

.MR -1 F : ?
:A B C D E (F G) H I)
```

Command: BI p1...pn THRU q1...qn

The BI command (mnemonic for Both In) adds parentheses around the sequence of elements beginning with the element designated by "p1...pn" and ending with the element designated by "q1...qn". The designated elements must be members of the same list, and "p1...pn" must precede "q1...qn".

Examples:

```
.?
:(A B C D E F)
.BI B THRU E : ?
:(A (B C D E) F)
.BI C THRU D : ?
:(A (B (C D) E) F)
.BI 1 NX THRU -1 BK : ?
:(A (( B (C D) E)) F)
```

Command: BO p1..pn

The BO command (mnemonic for Both Out) removes parentheses from the element designated by "p1...pn". This element must be nonatomic.

Examples:

```
.?
:(A ((B (C D) E)) F)
.BO C : ?
:(A ((B C D E)) F)
.BO 2 : ?
:(A (B C D E) F)
.BO 1 NX : ?
:(A B C D E F)
```

COMMANDS FOR ERROR RECOVERY

Each time a structural change is made using a command that modifies the current expression, the LISP system keeps a record of the changes made so that they can later be reversed. It is always possible to reverse any changes made since the invocation of the LISP editor. This feature allows the user to experiment without fear of permanent damage; such experimentation is encouraged.

Command: ??

The ?? command prints a list of the structure modification commands which have been executed, in inverse order of their execution.

Command: UNDO [command]

The UNDO command reverses the last structure-modifying command. If the optional argument "command" is given, it is compared with the name of the command to be reversed, and reversal takes place only if they are the same.

UNDO is itself a structure-modifying command, and may be UNDOne, but UNDO commands are normally skipped by subsequent UNDO commands unless a reversal is requested explicitly by UNDO UNDO.

Examples:

```
.?
:(A B C D E F)
.DELETE C : DELETE D : ?
:(A B E F)
.UNDO
:DELETE UNDONE
.?
:(A B D E F)
.UNDO
:DELETE UNDONE
.?
:(A B C D E F)
.UNDO
:NOTHING SAVED
.??
:UNDO UNDO

.UNDO UNDO
:UNDO UNDONE
.?
:(A B D E F)
```

Commands: TEST, !UNDO, UNBLOCK

These three commands can be used to make a set of tentative changes to a structure. All of these changes can subsequently be reversed at once. The TEST command places a block in the list of structure-modifying commands. A block stops the execution of the !UNDO command, which reverses all structural changes made since the block (i.e., since the TEST command). If there is no block in the list the !UNDO command reverses all changes made since the invocation of the LISP editor. The UNBLOCK command removes the most recent block from the list.

Note: A block also blocks the UNDO command. The UNDO command skips the !UNDO command unless it is explicitly requested, but !UNDO reverses all commands, including UNDO and !UNDO.

MISCELLANEOUS COMMANDS

Command: OK

The OK command causes the EDIT function to return to its caller.

Command: E form

The E command evaluates the S-expression "form". If an error occurs during evaluation, the editor may be reentered by evaluating (UNEVAL 'EDITONE NIL) in the break loop.

Command: S atom

The S command saves the current expression on the property list of "atom". It may be retrieved by using "## atom" in the "s1...sn" sequence of the INSERT command.

Example:

```
.?
:(LAMBDA (X) (COND ((NULL X) NIL) (T (CONS X X))))
.F NULL P S QREG # ?
:(NULL X)
:(LAMBDA (X) (COND ((NULL X) NIL) (T (CONS X X))))
.INSERT ## QREG FOR CONS : ?
:(LAMBDA (X) (COND ((NULL X) NIL) (T (NULL X))))
```

Command: EXCISE

The EXCISE command removes all editor-related atoms from the object list, thereby making the editor available for garbage collection.

Command: DSKIN FDname

The DSKIN command reads S-expressions from the MTS file or device "FDname". Each S-expression is saved on the property list of FDname for later use in the DSKOUT command, and then is evaluated. The result of the evaluation is ignored.

Command: DSKOUT FDname

The action of the DSKOUT command depends on "FDname" as follows:

(1) If "FDname" has been used previously in the DSKIN command, the list of expressions read by that command is retrieved from the property list of "FDname", and the user is prompted for another "FDname", on which this list of (possibly changed) expressions are written. This second "FDname" may be the same as the first "FDname" given in the command, but in any case must be a file name, since it is

emptied before writing.

(2) If "FDname" has not been used previously in the DSKIN command, the user is prompted for an S-expression designating the items to be written on "FDname". This S-expression is evaluated, and returns a list of items which are treated as follows:

(a) If an item is an atom, it is assumed to be a function name, and must have a LAMBDA expression as an EXPR on its property list. The appropriate DEFUN form is written to "FDname".
(b) If an item is nonatomic, it is written to "FDname" as is.

"FDname" is emptied before it is written.

June 1976

LISP DEBUGGING FACILITIES

INTRODUCTION

Debugging a collection of LISP functions involves isolating problems within particular functions and/or determining when and where incorrect data are being generated and transmitted. There are three facilities available which augment the facilities of the interpreter for monitoring a LISP program. One of these is the error package, which takes control whenever an error occurs in a program, and which allows the user to examine the environment at the time of the error. The other two facilities, BREAKF and TRACEF, allow the user to modify selected function definitions temporarily so that the flow of control in the program may be followed. All of these facilities use the same LISP function, BREAKFUNCTION, as the user interface.

BREAKF and TRACEF together are called the Break Package.

BREAKF modifies the definition of a function "fn", so that if a break condition (defined by the user) is satisfied, the evaluation is halted temporarily on a call to "fn". The user can then interrogate the state of the world, perform any computations, and continue or return from the call. For a more complete description of BREAKF, see the subsection "Break Package."

TRACEF modifies a definition of a function "fn" so that whenever "fn" is called, its arguments (or some other values specified by the user) are printed. When the value of "fn" is computed it is printed. For a more complete description of TRACEF, see the subsection "Break Package."

BREAKFUNCTION

BREAKF and TRACEF redefine functions in terms of BREAKFUNCTION. When an error occurs control is passed to BREAKFUNCTION.

Whenever LISP types a message of the form:

n: DEPTH= m DEBUGGING fn

the user is "talking to" BREAKFUNCTION, and is said to be in a break. "n" is the number of active calls to BREAKFUNCTION, "m" is the number of function calls on the pushdown stack, and "fn" is the function last called before BREAKFUNCTION (normally the function for which BREAKF was issued). BREAKFUNCTION allows the user to interrogate the state of the

world and affect the course of the computation. It uses the prompting character (=) to indicate it is ready to accept input for evaluation, in the same way as the top level of LISP uses "*". The user may type in an expression for evaluation and the value will be printed out, followed by another (=). Or the user can type in one of the commands described below which are specifically recognized by BREAKFUNCTION. Since BREAKFUNCTION puts all of the power of LISP at the user's command, anything that can be done at the top level of LISP can be done with BREAKFUNCTION. For example, one can define new functions or edit existing ones, set breaks, or trace functions. The user may evaluate an expression, see that the value was incorrect, call the editor, change a function, and evaluate the expression again, all without leaving the break.

It is important to emphasize that once a break occurs, the user is in complete control of the flow of the computation, and the computation will not proceed without specific instruction. Only if the user gives one of the commands that exits from the break (GO, OK, RETURN, FROM) will the computation continue. The computation can also be aborted (using # or ##, which are defined later in this section).

Note that BREAKFUNCTION is just another LISP function, not a special system feature like the interpreter or the garbage collector. It has arguments and returns a value, like any other function. A call to BREAKFUNCTION has the form

(BREAKFUNCTION BREAKEXPR BREAKWHEN BREAKFN BREAKCMDS BREAKTYPE)

BREAKWHEN — This argument is a LISP function which is evaluated to determine if a break will occur.

BREAKEXPR — BREAKEXPR is a form to be evaluated by BREAKFUNCTION. If BREAKWHEN returns NIL, BREAKEXPR is evaluated and returned as the value of the break. If BREAKWHEN returns any other value, a break occurs. After a break occurs, the commands GO, OK, and EVAL (see command descriptions) cause BREAKEXPR to be evaluated.

BREAKFN — This argument is the name of the function being broken. BREAKFN is used to print the above message when a break occurs.

BREAKCMDS — This argument is a list of command lines which are executed immediately, in the event of a break. The command lines on BREAKCMDS are executed before commands are accepted from the terminal, so that if one of the commands on BREAKCMDS causes a return, a break occurs without the need for user interaction.

BREAKTYPE — This argument identifies the type of break. It is used primarily by the Error Package. In all cases the user can use BREAK for this argument.

The value returned by BREAKFUNCTION is called "the value of the break." The user can specify this value explicitly by using the RETURN command described below. In most cases, however, the value of the break is given implicitly, via a GO or OK command, and is the result of evaluating "the break expression," BREAKEXPR.

BREAKEXPR is, in general, an expression equivalent to the computation that would have taken place had no break occurred. In other words, one can think of BREAKFUNCTION as a fancy EVAL, which permits interaction before and after evaluation. The break expression then corresponds to the argument to EVAL. For BREAKF and TRACEF, BREAKEXPR is a form equivalent to that of the function being traced or broken. For errors, BREAKEXPR is the form which causes the error.

Break Commands

Once in a break, in addition to evaluating expressions, the user can ask BREAKFUNCTION to perform certain useful actions by issuing atomic items as "break commands." The following commands can be entered directly by the user or may be put on the BREAKCMDS list.

GO	This command releases the break and allows the computation to proceed. BREAKFUNCTION evaluates BREAKEXPR, prints and returns the value, as the value of the break.
OK	This command operates in the same manner as GO except that the value of BREAKEXPR is not printed.
EVAL	This command causes BREAKEXPR to be evaluated. The break is maintained and the value of the evaluation is printed and bound on the variable BREAKVALUE. Neither GO nor OK will cause reevaluation of BREAKEXPR following EVAL, but another EVAL will. EVAL is a useful command when the user is not sure whether the break will produce the correct value and wants to be able to correct it if it is wrong.
RETURN form	The "form" is evaluated and its value is returned as the value of the break. For example, one might RETURN (REVERSE BREAKVALUE).
FROM form	This permits the user to release the break and return to a previous context with "form" to be evaluated. For details, see the subsection "Context Commands."
USE expr	For use either with UNDEFINED ATOM error or UNDEFINED FUNCTION error. USE replaces the expression (using RPLACA, the change is permanent) containing the error with expr (not the value of "expr") e.g.,

```
+    ***16 UNDEFINED ATOM
+     Q
+
* 1 : CAR BROKEN
USE XX
```

changes Q to XX in the form (CAR Q), which caused the error.

`#` This aborts the break. This is a useful way to unwind to a higher-level break. All other errors, including those encountered while executing the GO, OK, EVAL and RETURN commands, maintain the break.

`##` This returns control directly to the top level of LISP.

`ARGS` This prints the names and the current values of the arguments of the function at BREAKPOINTER. In most cases, these are the arguments of the broken function.

`<FORM>` This is EVALed if not a break command.

Context Commands

All information pertaining to the evaluation of forms in LISP is kept on the push-down stack. Whenever a form is evaluated, the form is placed on the push-down stack. Whenever a variable is bound, the old binding is saved on the push-down stack. The context (the bindings of free variables) of a function is determined by its position in the stack. When a break occurs, it is often useful to explore the contexts of other functions on the stack. BREAKFUNCTION allows this by means of BREAKPOINTER, which is a context pointer into the push-down stack. BREAKFUNCTION commands move the context pointer and evaluate atoms or expressions relative to their positions in the stack. For the purpose of this document, when moving through the stack, "backward" is considered to be toward the top level or, equivalently, towards the older function calls on the stack.

`F arg1 arg2 ... argN`

This command resets the variable BREAKPOINTER, which establishes a context for the commands USE, ARGS, AT, FROM and the backtrace commands described below. BREAKPOINTER is the position of a function call on the push-down list. It is initialized to the function just before the call to BREAKFUNCTION.

F takes the rest of the input line as its list of arguments. Each argument may be either a function

name, in which case the stack is searched for the most recent occurrence of the function preceding BREAKPOINTER, or a number [±]n. If negative, the number "n" specified causes BREAKPOINTER to move back (i.e., towards the top level) the appropriate number of calls. If positive, the number "n" specified causes BREAKPOINTER to move forward.

For example, if the push-down stack consists of:

```
BREAKFUNCTION       (13)
FOO                 (12)
SETQ                (11)
COND                (10)
PROG                 (9)
FIE                  (8)
COND                 (7)
FIE                  (6)
COND                 (5)
FIE                  (4)
COND                 (3)
PROG                 (2)
FUM                  (1)

then

F FIE COND          sets BREAKPOINTER to (7)
F COND              sets BREAKPOINTER to (5)
F -2                moves BREAKPOINTER to (3)
TOP                 resets BREAKPOINTER to (12)
```

F can be used on BREAKCMDS. In that case, the next element of the list is treated as the list of arguments to F, e.g., (F (FOO FIE FOO)).

TOP — TOP repositions BREAKPOINTER to a stack position just before BREAKFUNCTION.

EDIT arg1 arg2 ... argn

EDIT uses its arguments to reset BREAKPOINTER in the same manner as the F command. The form at BREAKPOINTER is then given to EDIT. This command can often save the user the trouble of calling EDIT and finding the expression that he needs to edit.

AT arg1 arg2 ... argn

This command is used to display the values of variables at position BREAKPOINTER. If the user types:

```
AT X (CAR Y)
```

the value of X and the value of (CAR Y) are printed. The difference between using AT and entering X and (CAR Y) directly into BREAKFUNCTION is that AT evaluates its input as of BREAKPOINTER. This provides a way of examining variables or forms at a particular point on the stack. For example,

```
F FOO -1 FOO
AT X
```

allows the user to examine the value of X in an earlier call to FOO.

AT can also be used on the BREAKCMDS list. The next element of the BREAKCMDS list is then treated as the list of arguments. For example, if BREAKCMDS is (EVAL AT (X (CAR Y) GO), BREAKEXPR will be evaluated, values of X and (CAR Y) will be printed, and the function will be exited after its value has been printed.

FROM [form] FROM exits the break by undoing the push-down stack back to BREAKPOINTER. If "form" is not specified, reevaluation continues with the form on the push-down stack at BREAKPOINTER. If "form" is specified, the function call on the push-down stack at BREAKPOINTER is replaced by "form", and evaluation continues with "form" which is evaluated in the context of BREAKPOINTER. There is no way of recovering the break because the push-down stack has been undone. FROM allows the user to return a particular value for any function call on the stack. To return 1 as the value of the previous call to FOO:

```
:F FOO
:FROM 1
```

Backtrace Commands

The backtrace commands print information about function calls on the push-down list. The information is printed in reverse order to that in which calls were made. All backtraces start at BREAKPOINTER.

BKF BKF gives a backtrace of the CARs of forms that are still pending.

BKE BKE gives a backtrace of the expressions which called functions still pending (i.e., it prints the function calls themselves instead of only the names, as in BKF).

BK — BK gives a full backtrace of all expressions still pending. It evaluates the form (DUMP 0). Output is in the form:

```
function name
list of arguments
function name
list of arguments
(etc.)
```

BKF and BKE may be followed by an integer. If the integer is included, it specifies how many blocks are to be printed. The limiting point of a block is a function call. This form is useful when working on an IBM 3270-type terminal.

By specifying an integer with BKF or BKE and issuing an F command, the user can display any contiguous part of the backtrace.

BREAK PACKAGE

How to Set a Break

In order to access any of the following functions, the user must be sure the DEBUG package has been loaded. This may be accomplished by calling the DEBUG function (see the subsection "Error Package").

The following functions are useful for setting and unsetting BREAKs and TRACEFs.

Both BREAKF and TRACEF use a function BREAKO to do the actual modification of function definitions.

BREAKF — BREAKF is an FLAMBDA. For each atomic argument, BREAKF breaks the function named each time it is called. For each argument that is a list of the form (FN1 IN FN2), it breaks only those occurrences of FN1 which appear in FN2. This feature is very useful for breaking a function that is called from many locations, but where one is only interested in the call from a specific function, e.g., (RPLACA IN FOO), (PRINT IN FIE), etc. For each argument that is neither atomic nor a list in the above form, BREAKF assumes that the CAR is a function to be broken; the CADR is the break condition. When the function is called, the break condition is evaluated. If it returns a non-NIL value, the break occurs. Otherwise, computation continues without break and the CADDR is a list of command lines to be performed before an interactive break is made (see BREAKWHEN and BREAKCMDS of BREAKFUNCTION) or NIL. For example,

(BREAKF FOO1 (FOO2 (GREATERP N 5) (ARGS)))

breaks all calls to FOO1 and all calls to FOO2 when N is greater than 5 after first printing the arguments of FOO2.

(BREAKF ((FOO4 IN FOO5) (MINUSP X) NIL))

breaks all calls to FOO4 made from FOO5 when X is negative.

Examples:

(BREAKF FOO)

(BREAKF ((GET IN FOO) T (GO)))

TRACEF — TRACEF is an FLAMBDA. For each atomic argument, it TRACEFs the function named each time it is called. For each list in the form (FN1 IN FN2), it TRACEFs only those calls to FN1 that occur within FN2.

For example, (TRACEF FOO1 (SETQ IN FOO3) causes both FOO1 and SETQ in FOO3 to be traced.

Note: The user can always call BREAKO himself to obtain combinations of options of BREAKFUNCTION not directly available with BREAKF and TRACEF (see section on BREAKO below). These functions merely provide convenient ways of calling BREAKO, and will serve for most uses.

UNBREAK — UNBREAK is an FLAMBDA. It takes a list of functions modified by BREAKF or TRACEF and restores them to their original state. Its value is NIL. (UNBREAK T) will unbreak the function most recently broken. (UNBREAK) will unbreak all of the functions currently broken.

If one of the functions, say FN, is not broken, UNBREAK prints "FN NOT BROKEN" for that function and no changes are made to FN.

UNTRACEF — UNTRACEF is an FLAMBDA. It is the similar to UNBREAK.

BREAK0 [fn when coms]

BREAK0 is an EXPR. It sets up a break on the function "fn" by redefining "fn" as a call to BREAKFUNCTION with BREAKEXPR a form equivalent to the definition of "fn", and "when", "fn", and "coms" as BREAKWHEN, BREAKFN, and BREAKCMDS, respectively (see BREAKFUNC-

TION). BREAK0 also adds "fn" to the front of the list BROKENFNS. Its value is "fn".

If "fn" is nonatomic and of the form (fn1 IN fn2), BREAK0 first calls a function which changes the name of "fn1" wherever it appears inside of "fn2" to that of a new function, fn1-IN-fn2, which is initially defined as "fn1". Then BREAK0 proceeds to break on fn1-IN-fn2 exactly as described above. This procedure is useful for breaking on a function that is called from many places, but where one is only interested in the call from a specific function, e.g., (RPLACA IN FOO), (PRINT IN FIE), etc. This only works in interpreted functions.

ERROR PACKAGE

| The error package is enabled by EVALing (DEBUG T). When an error occurs during the evaluation of a LISP expression, control is turned over to the error package. The idea behind the error package is that it may be possible to "patch up" the form in which the error occurred and
| continue. Or, at least, the user may find the cause of the error more easily if he can examine the state of the world at the time of the error. Basically, what the error package does is call BREAKFUNCTION with BREAKEXPR set to the form in which the error occurred. This puts the user "in a break" around the form in which the error occurred. BREAKFUNCTION acts just like the top level of the interpreter with some added commands (see the section on BREAKFUNCTION). The main difference
| when the error package is enabled is that the variable bindings that were in effect when the error occurred are still in effect. Furthermore, the expressions that were in the process of evaluation are still
| pending. While the error package is enabled, variables may be examined or changed, and functions may be defined or edited just as if the user were at the top level. In addition, there are several ways in which the user can abort or continue from the point of error. In particular, if the error can be patched up, entering "OK" will cause the program to continue. If the error can't be fixed, # will cause the program to exit from the break. When the error package is being used, the prompt character is (=); this is preceded by a level number. Note: If for some reason, the error package is not to be invoked, it can be disabled
| by evaluating (DEBUG NIL).

June 1976

SLIP

INTRODUCTION AND HISTORICAL NOTES

This description is a user's guide for the double-precision version of SLIP installed in MTS. This version of SLIP is compatible with FORTRAN IV and can be used in conjunction with programs that are able to call FORTRAN routines. Most of this version of SLIP is written in FORTRAN with a small portion, the SLIP primitives, written in 360/370-assembler language. Complete citation for the references noted are included in the last subsection "References."

The definitive paper on SLIP by Weizenbaum (3) was published in 1963. That paper is not a user's guide, but achieves a general description of SLIP by defining the available data structuring functions together with implementational details. The paper is novel in that it includes a listing of the FORTRAN code. Two letters to the Communications of ACM (4,5) add information for SLIP implementors and users. Subsequently a book by Findler, Pfaltz, and Bernstein (6) that is a readable and useful reference for users was published. Another book by Waite (7) offers constructive criticisms, some of which are employed in this implementation. However, these are directed mainly to the student of high-level list-processing systems for FORTRAN IV and thus perpetuate the policy of Weizenbaum's paper. User functions, together with implementational details, are presented.

This description attempts to minimize emphasis on implementational details and concentrates instead on the user functions.

The literature indicates that many versions of SLIP, for a variety of machines, exist. Here, every attempt has been made to maintain the spirit of the original version of SLIP and the names and assignments of its functions. This attempt has not been entirely successful. New or alternate functions were required for this implementation. Care has been exercised to state where deviation from the original version was necessary. Essentially deviations arise from limitations obtained from the IBM System/360/370 32-bit word size.

Basic Concepts of List Processing

The following are the nine basic operations that can be performed on a list consisting of "n" elements (synonymous terms for element are node or item).

(1) Access the kth element (1<k<n).
(2) Insert a new element before or after the kth.
(3) Delete the kth element.
(4) Concatenate (i.e., chain together) two or more simple lists.
(5) Split a list into two or more lists.
(6) Make a copy of a list in terms of its contents and structure.
(7) Count the number of elements on a list.
(8) Sort the elements according to some criterion; for example, in ascending order of the integers in a given field of the elements.
(9) Search for an element that has a particular value.

Special reference is usually made either to the first (top) or the last (bottom) element (i.e., when k=1 or k=n). There is even a particular terminology dealing with list processing: inserting a new element to the top is called pushing down the list, and deleting an element from the top is called popping up the list. Although these terms are completely acceptable in automata theory, they may be misleading in programming, as we will see later, since the elements below the top element do not in fact change location.

To achieve ease and flexibility in the above-mentioned nine basic operations, the usual sequential structure of the memory, as in FORTRAN, must be given up. In other words, elements that are consecutive logically on the list may not be consecutive geometrically in the computer hardware memory. Nothing is novel about this. For example, when we store the elements of a two-dimensional array column-wise, the row-wise neighboring elements are no longer adjacent in the essentially one-dimensional memory. However, if we know the location of the first element, the dimensions of the array, and the so-called mapping function (i.e., the algorithm of storing the array), we can easily determine the geometrical location of any element. Some similar arrangement is needed in the case of lists, and the solution is provided by links or, using another term, by pointers. A part of the space representing the list element is occupied by the name of the next element. This name field links with, or points to, the neighboring element, which in fact may be in any part of memory.

Let us now consider one more concept, that of the Available Space List for free storage, before seeing how the nine basic operations are carried out. The Available Space List is essentially a reservoir of free space from which we draw when a new cell is needed and to which we return cells no longer needed, in order not to run out of free space too soon. Similarly, the SLIP system can automatically draw cells from or return cells to this reservoir. This dynamic memory allocation enables us to omit specifying in advance how long a list is going to be. In fact, we can create, change the size of, and erase lists during execution time.

The importance of the Available Space List cannot be overemphasized, particularly in nonnumeric computations.

Generally, dynamic memory management becomes indispensable whenever data are interrelated in a complex manner, and especially when processing causes data to be dynamically reorganized in an unpredictable way as part of the solution to the problem.

Many of the problems that require these facilities are partly or wholly symbol-manipulating in nature, such as projects in artificial intelligence, analysis and synthesis of natural languages, computer graphics, simulation of cognitive processes, information retrieval, etc. However, the study of many numerically oriented problem areas is also greatly helped by list-processing techniques. The representation and manipulation of sparse matrices is an obvious example of this case.

The initial objection to using list-processing languages was based on the apparent waste of memory space occupied by pointers. As we can see, in the problem of sparse matrices, just the opposite is often true. There are, of course, other arguments in favor of list-processing languages.

Let us now turn back to the basic list processes. Operations 2 to 5 amount to managing the relative linkings of certain cells. The name of the top cell is always contained in a special word, the symbolic name of which is FAVSLC. We can say the word points to the top of Available Space List.

We can now discuss the rest of the basic operations.

(1) Accessing the kth element consists of going along the links and increasing a counter of the elements passed by until k is reached.
(2) Described above.
(3) Described above.
(4) Concatenating two lists amounts to simply overwriting the end-of-list symbol of the first list with the name of the top cell of the second list.
(5) Splitting a list into two is the opposite of the above process. The symbolic name of the new list is identified by SLIP with the name of its top cell and is output in a standard manner.
(6) Copying a list is straightforward. The name of the new list is usually output according to some convention. The name is output also when a new empty list is created. A single cell with an empty symbol field and no link is considered an empty list.
(7) Counting the elements of a list consists of going down the list via the linking pointers, always adding one to a counter until the end-of-list symbol is encountered.
(8) Sorting consists of systematically comparing "keys" and manipulating the link fields whenever necessary.
(9) Searching for a special element on a list is accomplished again by a sequence of comparisons. The result can be either a simple yes/no answer concerning the success of the search, or the name of the matching element when found. In case of failure, a special symbol, such as zero, is output.

Data types more complex than simple lists are also needed. A list structure consists of a main list and a hierarchy of sublists. The elements on the main list, and also on its sublists, are either data or names of sublists. In the cases of self-referencing by a list or cross-referencing between two lists, special care must be taken, of course, to avoid an infinite loop in processing.

Rings and ring structures are analogous to lists and list structures, respectively. However, in rings and ring structures a link connects the last and first cells. This is why the term circular list is also used. The first cell of every ring must be marked; otherwise, the processing of elements would again go on indefinitely. The information structures in SLIP are of this type. Distinction can be made as to whether one- or two-directional pointers connect neighboring elements. The latter arrangement, also used by SLIP, offers certain processing advantages at the price of an extra link field in every node.

Data representations present problems of external (user's) and internal (inside the computer) representation. The solution to the problems of external representation is, of course, idiosyncratic to the language and will not be discussed here. Higher-level and more recent languages tend to be more convenient for the user.

Depending on the amount of information stored in each cell and on the word length of the machine on which the list processing language is implemented one, two, or sometimes a higher varying number of computer words are used for a cell. The lists need not contain the actual information but may contain only links that point to a centralized data table. The referencing to data can be direct or manifold indirect (pointer to pointer to ... pointer to data). If a list contains data of different modes, special markers are needed to indicate the type of information in each cell.

Description Lists contribute to the power of SLIP. These lists consist of a sequence of attribute-value pairs and can provide further information about lists. Suppose, for example, a list structure describes the current position on a chessboard. It has 64 sublists, one for each square. Each sublist has a Description List with attributes: occupancy status (the possible values are the 32 men and "empty"), your own men defending the square (the value is a list of men, possibly empty), opponent's men attacking the square (the value is as before), etc. As we can see, in general the value can be symbolic or numeric, a single unit of data or a list of data. Also, a Description List itself can have a Description List and so on.

If a list or list structure is copied or erased (i.e., returned to Available Space List), its Description Lists undergo the same treatment.

An elegant and often very important list-processing technique involves the use of recursive computations. A recursive subroutine is one that calls itself. The input parameters of the subroutine called later normally depend on the intermediate results of the same subroutine

called earlier. Since these parameters all can contribute to the overall final result, they have to be preserved on push-down stacks.

An example should clarify the idea. Instead of referring to the hackneyed recursive computation of the factorial function N!, let us consider the following.

Suppose there is a subroutine R, which simplifies symbolic formulae. It carries out the addition/subtraction of identical terms; performs the multiplication/division of numeric coefficients; reduces A^0 to 1, A+0 to A, A*0 to 0, A*1 to A; gives warning messages in cases of 0/0 and infinity/infinity, etc. We call R for a complex, heavily parenthesized expression. (For the sake of explanation, let us forget that the computer representation of formulae is usually in a parenthesis-free, so-called Polish notation. The technique, however, is basically the same in any form of representation.) So, R first simplifies the expression as if one single symbol were inside the outermost pair of parentheses. R then turns to treat the expression therein by calling itself again. As it goes further and further inside, peeling off pairs of parentheses, it calls itself again and again. Finally, the parentheses disappear and further simplifications may have to be done. R has to substitute the last result into the next to last, this result into the one before that, and so on until it arrives back at the highest level of expression. The intermediate results or, rather, pointers to the intermediate results, are popped off a stack whenever they are needed.

Although most, if not all, recursive computations can be transformed into iterative ones, recursion is always elegant and often more efficient.

The last topic to be discussed in this section is the problem of memory management. We saw earlier that the dynamic memory allocation scheme enables the system to use only the currently necessary amount of storage area. It is obvious, however, that during the execution of the program many cells, lists, and list structures may no longer be needed.

We would soon run out of even the largest memories available today if somehow the storage areas occupied by unnecessary information were not returned to Available Space List. This process is called garbage collection and is of extreme importance.

We can distinguish between three methods of memory management as far as garbage collection is concerned. First, it can be completely under the programmer's control, for example, IPL. See Newell, et al. (1) and Sammet (2).

The SLIP System uses a second type of garbage collection. In SLIP one list may have several superlists. There is, therefore, a so-called reference counter at the head of every list that indicates the number of times that list is referred to by other "live" data structures. Every time a list is erased the reference counters of its sublists are decreased by one. If a reference counter reaches zero, that sublist is also returned to Available Space List. Self-referencing and cross-

referencing lists obviously require special attention; otherwise they would never be erased, or infinite processing loops could develop. The programmer may, therefore, override the control role of the counters and return to Available Space List, or keep alive, any list.

The third technique is completely automatic, relieving the programmer of all housekeeping duties. The LISP system lets the program run until almost all free space is exhausted. Two phases of garbage collection are then called into action. In the first phase, all those lists are marked that can be accessed from (i.e., named by) other lists. The nonaccessible data are returned to Available Space List in the second phase, and the marks are eliminated so that another cycle of garbage collection can take place at a later stage.

Obviously, this is a very time-consuming method. It excludes the possibility of using the language in "real time." With most large-scale projects, the frequency of garbage collection goes up rapidly as the program progresses, and the number of liberated cells per action diminishes at the same time. To stop nonsensical oscillations of this kind, the programmer can prespecify in some systems the termination of the run if a garbage collection cycle results in less than a certain number of cells being returned to Available Space List.

Conventions

The basic element of SLIP is a SLIP-cell. Each such cell is divided into two parts. The first part, the linkword, is occupied with linking and bookkeeping information useful in linking each cell with its relatives as required by SLIP. The second part occupying the second half of the cell is a datum. This datum contains the user's information. A datum may be any bit configuration such as integer, real, character strings, etc. A datum may also have the value of a SLIP-name which is the name of a list. It is in this manner that a list becomes a sublist of another list.

Every list has one cell, the Header, which is equivalent to the "name of that list." This Header cell's space is completely devoted to information regarding the state of the list. No space in this cell is available for the user. However the user, through SLIP functions, has access to this information; one may inquire if the list is empty, i.e., there is a Header but no other cells are attached to the Header.

Each SLIP cell has as a name an INTEGER value specifying its location in the SLIP memory. In this description, CADR refers to this name or cell address.

Even in the original version of SLIP it was necessary to provide two names for certain functions to be compatible with FORTRAN and to prevent undesirable conversions. This problem is further aggravated in this

version and consequently additional functions are provided to shelter the user from this nonproductive concern.

It is preferred, as a matter of style, to speak of cell names rather than machine addresses of cells. The particular conversion from cell name to machine address ought not to be a user's concern. However, we are at the mercy of our wish to remain as compatible as possible with the tradition of SLIP and thus we retain previously defined mnemonics emphasizing the machine address of a word or cell. We have, however, added functions to make the user's lot comfortable.

However, now particular attention must be paid to the mode declaration of functions to insure compatibility with FORTRAN IV and its implicit mode conversion policies. To assist the user, a complete tabulation of the policy is provided as is a complete tabulation of the proper mode declaration for functions; see the subsection "Summary of 360 SLIP Functions and Subroutines."

FUNDAMENTAL SLIP OPERATIONS

For a readable introduction to the principles of list processing, and SLIP in particular, see Findler, et al. (4).

The data structures of the Symmetric List Processor (SLIP) consist of bidirectional rings. Each element is connected by pointers to both its left and right neighbors. (Sometimes we will use the synonymous terms above and below instead of left and right respectively.) Also, the last cell points back to the beginning of a ring.

In the following discussion we will use the more customary terms list and list structure rather than ring and ring structure, since no misunderstanding can arise from this substitution.

SLIP Data Elements

The first word of a SLIP cell, the linkword, contains the list-linking information and is not directly needed by the user. The second word of a SLIP-cell contains some datum.

The DATUM field is one word or 32 bits long. It may contain anything which can be represented by 32 bits.

The cell name of the Header is called the name of the list. If more than one FORTRAN word has the list name, the list is said to have several aliases. If two aliases are established for a list that does not imply that it has been referenced twice.

Before we consider the various operations in SLIP, several points must be discussed.

Programming Conventions (SLIP with FORTRAN IV)

A FORTRAN subroutine-type subprogram does not return values other than those of the variables, either in COMMON or in the list of the subroutine arguments. A function-type subprogram, on the other hand, returns, in addition, a single value, that of its name, as if the name were a variable. This represents the main programming advantage of functions: they can be nested in arbitrary depth in the argument list of other subprograms (no recursive calls). Further, a function can be CALLed like a subroutine, in which case the returned value is, of course, lost. Finally, the results of the processes in a subroutine can be obtained by including the subroutine name in a FORTRAN arithmetic statement as if it were a function. A function must always have at least one dummy argument lest it be mistaken for a variable.

The FORTRAN mode conventions need to be explained. Many SLIP subprograms disregard the mode of the subprogram arguments but they are concerned with addressing conventions. To avoid unwanted conversions, the functions REALS and INTGER may be used for the output of ill-named functions. For example, as we will see later, the subprogram TOP(LST) retrieves the datum of the top cell of the list with alias name LST. The cell's datum may contain a list alias. If TOP is called as a function, it delivers the contents of the top cell as its value. If this value is an integer, we do not want it to be converted into a floating-point number. To avoid this we use assignment statement

```
I=INTGER(TOP(LST))
```

to obtain the integer stored in the datum. Similarly, the function REALS (REAL-short) delivers the single-precision REAL value without conversion.

Let us consider a significant difference between FORTRAN and SLIP. In FORTRAN, we manipulate contents of words that have symbolic names. For example, the instruction

```
A = B+C
```

adds the contents of the words with symbolic addresses B and C and places the sum into a word with symbolic address A. In SLIP, both the contents and the addresses of words can be operands.

The following notation will be used uniformly in the arguments of SLIP subprograms.

Symbol	Meaning	Remark
X	FORTRAN variable	Symbolic address of a computer word of any contents, mode, or format.
LST	Alias name of a list	Symbolic address of a DOUBLE PRECISION computer word that contains, in repeated form the cell name of the first word of the Header cell.
CADR or MADR	FORTRAN variable	Symbolic address of a computer word or that contains, in integer format, the name of a SLIP cell. CADR is an INTEGER variable. Note that a name, either in list name format or as an integer in the left-word can be offered as a MADR, a DOUBLE PRECISION variable. [Note a MADR can always be offered where a CADR is expected. The converse is not true.]
KADR	FORTRAN variable	Symbolic address of a computer word that contains, in integer format, the machine address of another word.
NRD	Alias name of a reader	Symbolic address of a computer word that contains, in integer format, the name of the top cell of a Reader stack.

A list notation which will often be convenient to represent list structures is the so-called string format. For example, the following two sets of strings are equivalent. On one hand

```
List A:  (S3,Q1,(R2,R4),T1,(),(S1,T2))
List B:  (U2,(V3,V1,(W0,Q2)),S2)
```

and on the other

```
List A:  (S3,Q1,List C,T1,List D,List E)
List B:  (U2,List F,S2)
List C:  (R2,R4)
List D:  ()
List E:  (S1,T2)
List F:  (V3,V1,List G)
List G:  (W0,Q2)
```

As shown above, parentheses delimit lists and sublists; commas (or, according to another notation, spaces) separate elements on a list, which are written horizontally in a sequential manner. Direct input of lists and list structures is similar to this in format (see RDLSTA and PRLSTS).

Initialization. SLIP requires that the program first call the subroutine INITAS which sets up 10 public lists with the symbolic aliases W(1), W(2), ..., W(10). The subroutine has two arguments, INITAS (SPACE,NDIM), that refer to the name and the dimension of the one-dimensional array of free space. If we intend to use these public lists then we must first declare[1]

COMMON/PUBLIC/W(10)

The variables SPACE and W must be declared DOUBLE PRECISION.

The argument SPACE is the name of a DOUBLE PRECISION linear array of dimension NDIM. In our system the user need not make of this space assignment if he sets NDIM to a negative integer; the magnitude of this integer will be used as an estimate of the number of pages of memory initially assigned. The integer is rounded up, modulo 4096. In any event space is acquired as needed. Under these conditions SPACE does not have to be dimensioned but must be declared DOUBLE PRECISION.

Alternately, if SPACE is dimensioned NDIM, and NDIM is positive, then this will be the assigned space for the available space list of NDIM/2 SLIP cells. Should all the available space be exhausted SLIP will abruptly terminate execution with announcements when additional space is sought.

Another point to be noted here, but which has no effect on the user, is that in our SLIP system the available space list is a linked list of one-directional pointers whose starting and ending names are stored in FAVSLC and LAVSLC respectively.

Processes Affecting the Available Space

A basic operation, creating a list, is done by the function

LIST(LST)

Both its returned value and the value of its argument are the aliases of the newly created empty list. We can, for example, write

DM2 = LIST(STACK)

and refer to the same list by the names DM2 and STACK. We are, of course, also permitted to put

---

[1]Normally AVSL and W were maintained in unlabeled COMMON; we have elected to use FAVSLC and LAVSLC for AVSL and labeled common PUBLIC, a deviation from the early SLIP implementation. Also only 10 public lists are created in this version of SLIP; early SLIP implementations use 100.

```
DN4 = LIST(DN4)
```

Further, if the argument is the numeral 9, a "local" sublist is created with a reference counter 0 rather than the usual 1. If the name of this list is put on another list, the sublist is automatically erased at the time the superlist is returned to the available space list. (Sometimes this kind of list is called temporary, as opposed to permanent lists that are created with non-9 aliases.)

Another function

```
J = IRALST(LST)
```

returns the list with alias LST to the available space list. This return takes place only if the list's reference counter is 1; otherwise the counter is decremented by 1. J is always the value of the reference counter after the operation. If it is 0, the list has been erased. The functions

```
DELETE(CADR)
```

or

```
DELETE(MADR)
```

should be employed to erase a list cell. They erase any list cell except the Header and should be employed to return a cell no longer needed to the available space list. These functions insure that the link fields are properly rewritten. The value of the functions is the datum of the cell returned. When accidentally a Header cell is named by CADR or MADR, the value is 0, a warning message is given, and no action is taken.

Another function of direct help to the programmer is

```
MTLIST(LST)
```

The function returns to the available space list all cells of the list LST except its Header and its Description List, if any. Its returned value is the alias of the just-emptied list.

It is noteworthy that, because of the postponed clean-up operation in the returned cells, the time required by MTLIST is completely independent of the length of the list in question. The operation affects the same number of link fields, i.e., those in the boundary cells, each time.

The function

```
LPURGE(LST)
```

deletes all recursive references to lists from a list structure with the alias LST. Its returned value is the number of times names were

deleted. The purpose of this function is to eliminate any circularity within a list structure, such as cross-referencing and self-referencing. If the programmer does not wish to thus modify a list structure, but still wishes to process each element on it, he can make a copy of the list structure and subject the copy to LPURGE before processing. A similar technique employs marking of processed sublists and final unmarking at the end. See Findler, Pfaltz and Bernstein (4).

Adding Cells and Data to Lists

Most of the subprograms here appear in pairs because of the symmetric nature of SLIP lists.

```
NXTLFT(X,CADR)
```

or

```
NXTLFT(X,MADR)
```

and

```
NXTRGT(X,CADR)
```

or

```
NXTRGT(X,MADR)
```

are functions that insert a new cell to the left and right, respectively, of the cell with name CADR or MADR. The contents of X are then placed into the datum word of the inserted cell. When these routines are called as functions, the delivered value is the name of the newly inserted cell, an integer.

For example

```
IDUM=NXTRGT(LIST(9),LIST(L1))
```

creates a list with alias L1 and puts on it the name of an empty sublist. The functions

```
NEWTOP(X,LST)
```

and

```
NEWBOT(X,LST)
```

add a new cell to the top and bottom, respectively, of the list with alias LST and put the contents of X into the datum of the new cell. The remarks made concerning the previous two routines also apply here. These functions perform the push-down operation with the piece of datum

X at either end of the named list. As can be seen, none of the original cells are shifted in the memory. Therefore, the terms push-down and, similarly, pop-up must be interpreted accordingly.

The datum of the top and bottom cells, respectively, can be replaced (as opposed to the preserving push-down operation) by the functions

SUBSTP(X,LST)

and

SUBSBT(X,LST)

The remarks made above concerning NXTLFT and NXTRGT are valid here, too, except that the returned value, when SUBSTP or SUBSBT is called as a function, is the old contents of the cell being overwritten with X.

The function

SUBST(X,CADR)

replaces the contents of the datum of the SLIP-cell, of cell name CADR, with the contents of X. The returned value is the old contents of the cell.

Another REAL function, STRDAT, has a similar purpose; it is invoked by

RDUM=STRDAT(X,CADR)

and RDUM will contain the value of X upon return.

Block insertions are performed by the functions

INLSTR(LST,CADR)

and

INLSTL(LST,CADR)

They decapitate the list LST (i.e., leave its Header as an empty list) and insert the rest of the list to the right and left, respectively, of the cell with name CADR.

Retrieving Data from Lists

Two types of operations fall into this category. Data can be accessed either without changing the list or coupled with destroying (i.e., returning to the Available Space List) the cell that contains the information being retrieved. Accordingly, the functions

TOP(LST)

and

BOT(LST)

deliver the datum of the top and bottom cells, respectively, of the list with alias LST without changing the list. The programmer should first test whether the list is empty. If it is, the returned value is the contents of the datum word of the Header, essentially garbage.

The equivalent functions, that also destroy the cell involved, are

POPTOP(LST)

and

POPBOT(LST)

In case LST refers to an empty list, 0 is returned and a warning message is given.

More general retrieval routines, affecting the inner part of lists, are discussed in connection with the Reader mechanism (see the subsection "The Reader Mechanism and Advance Functions").

Retrieving Data from SLIP-Cells

The twin functions

DATUM(CADR)

or

DATUM(MADR)

and

IDATUM(CADR)

or

IDATUM(MADR)

have similar roles; they return the contents of the datum part of the cell whose name is in CADR or MADR. Again the duplication is provided to overcome the FORTRAN mode-conversions in unwanted cases. Specifically DATUM returns the full double word contents of the datum in DOUBLE PRECISION mode while IDATUM returns only the first word of the datum in INTEGER format. Should that first word be wanted in REAL format then one can employ

```
REALS(DATUM(CADR))
```

Note that

```
IDATUM(CADR)
```

is equivalent to

```
INTGER(DATUM(CADR))
```

More Routines Concerning List Cells

The following two functions split lists in two.

```
NULSTR(CADR, LST)
```

and

```
NULSTL(CADR, LST)
```

create new lists. The new list is formed of a block of cells taken from the original list with alias LST. The block consists of the cell with name CADR and all cells to its right or left, respectively. These cells are, of course, removed from the original list. The schematic diagram in Figure 1 makes these operations clear. The returned value of the functions is the name of the new list.

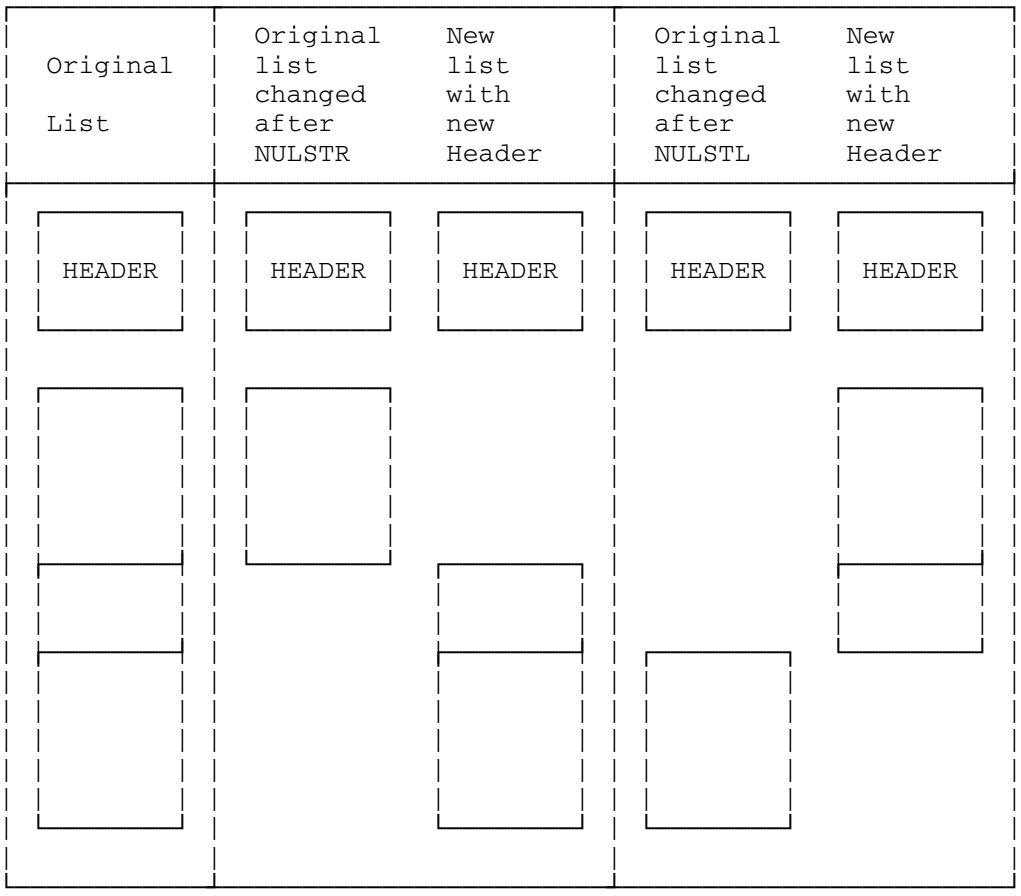


Figure 1: The Result of NULSTR and NULSTL Operations

The function

LSTEQL(LST1, LST2)

compares two list structures with aliases LST1 and LST2. If the two are identical in structure and content, the returned value is 0; otherwise it is -1. Description Lists and List Marks (see the subsection "List Marks and Description Lists") are not compared.

The function

LSSCPY(LST)

creates a copy of the list structure with alias LST. Its returned value is the name of the new list structure. Description Lists and List Marks are not copied.

The function

LISTMT(LST)

tests whether the list with alias LST is empty. If so, it delivers 0 value; otherwise it delivers -1. Note that an empty list can have a nonempty Description List.

The following functions return cell names.

MADLFT(CADR)

and

MADRGT(CADR)

yield the names of the cell to the left and right, respectively, of the one specified in the argument. The value is returned in INTEGER mode. If it refers to the Header the SLIP name format is used. If the integer input argument is the cell name of a Header of a list, the names of the bottom and top cells, respectively, are obtained.

For this version of SLIP two functions CADLFT and CADRGT whose purpose is similar to the above have been added.

CADLFT(CADR,IFLAG)

and

CADRGT(CADR,IFLAG)

yield the names of the cell to the left and right, respectively, of the one specified in the argument. The value is always in INTEGER mode. If the value refers to the Header, IFLAG is -1, otherwise IFLAG is 0. Therefore, for example

```
DAT = DATUM(MADRGT(LST))
```

or

```
DAT = DATUM(CADRGT(LST,IFLAG))
```

and

```
DAT = TOP(LST)
```

are equivalent.

MADNTP(LST,N)

and

MADNBT(LST,N)

return the cell name of the Nth cell from the top and bottom, respectively. If fewer than N cells are on the list, the counting is circular. The Header is included in the count, but the first cell (N = 1) from the top is the top cell. Therefore, if the list contains M cells, including the Header, we can obtain the cell name of the first cell by writing either

```
MADHD = MADNTP(LST, M + 1)
```

or

```
MADHD = MADNBT(LST,M+1)
```

The datum, say an integer, can be retrieved from the Ith cell by

```
INT = INTGER(DATUM(MADNTP(LST,I)))
```

or

```
INT = IDATUM(MADNTP(LST,I))
```

HOW TO MAKE COMMENTS ON LISTS

List Marks and Description Lists

It is often necessary to attach additional information to lists. If this information is restricted to designating four different classes of lists, list marks are used. More extensive comments require Description Lists.

The function

```
MRKLST(N,LST)
```

places the integer N(=0,1,2, or 3) as a List Mark for the list with alias LST. The returned value of the function is the list name (in list name format).

The function

```
MRKLSS(N,LST)
```

performs the same operation for a whole list structure. The returned value is the list name (in list name format).

The function

```
LSTMRK(LST)
```

delivers as its value the List Mark found in the Header cell of the list with alias LST.

The processing of a complex list structure with interwoven auto- and cross-referencing is greatly enhanced by the List Mark facility.

Description Lists are "associated" with the corresponding lists, as opposed to the "subordination" of sublists. However, a particular list can be both a Description List and a sublist of several lists--if it makes sense. This close relation betweeen a list and its Description List also means that erasing a list automatically erases its Description List as well.

To erase the Description List only, we can use

```
IDUM=IRALST(NAMEDL(LST))
```

where the function

```
NAMEDL(LST)
```

delivers as its value the alias of the Description List associated with list LST.

Each Description List contains an even number of SLIP-cells, representing attribute-value pairs. An attribute can be any type of characteristic of the list, in a format determined by the programmer. The value of the attribute occupies the second cell and can be a number, characters, or even the alias of a value list.

Several subprograms facilitate the use of Description Lists.

The function

```
LISTAV(LST)
```

creates an empty Description List for the list with alias LST. If a Description List already exists for this list, it is not erased but is replaced with the new empty Description List. The function returns as its value the alias of the new Description List.

The function

```
MAKEDL(LST1,LST2)
```

causes the list with alias LST1 to become the Description List of list LST2. If the latter already has a Description List, it is first erased. The function returns the value LST2.

The function

```
LDATVL(AT,VAL,LST)
```

either adds to an existing Description List of the list LST or creates one and places onto it the attribute-value pair AT and VAL. It does not check whether the given attribute is already on an existing Description List. The returned value is the alias of the Description List.

The function

NEWVAL(AT,VAL,LST)

on the other hand, searches the Description List of list LST for the attribute AT. If the attribute is found, the function assigns the new VAL to it and returns the old one. Otherwise both AT and VAL are added to the Description List. If the Description List does not exist, an empty one is created first. When either AT is not found or there is no Description List, the value of the function is set to 0. Of course, if distinction must be made between "no previous attribute AT" or "0 previous value of AT," a corresponding test should precede the use of NEWVAL. Also, note the difference in the operation and in the returned values between LDATVL and NEWVAL.

The function

NOATVL(AT,LST)

removes the attribute AT and its associated value from the Description List of LST. The deleted value is the value returned by the function. If either LST has no Description List, or the attribute AT is not on it, the returned value of NOATVAL is 0, and no action is taken. Again, a distinction must be made between "no previous attribute AT" and "0 previous value of AT."

The function

ITSVAL(AT,LST)

delivers as its value, the value associated with the attribute AT. If the latter is not found, 0 is returned.

The function

MTDLST(LST)

empties the Description List of list LST if there is one; otherwise, no action is taken. It returns the alias, LST, of the list.

The function

MADATR(AT,LST)

returns the name of the cell on the Description List that contains the attribute AT. If the latter is not found, the function delivers -1.

The above retrieval routines call the subroutine

DERROR(LST)

if they fail to find the required Description List. The routine prints the name of the Header cell and a warning message.

The Reader Mechanism and the Advance Functions

The Reader is a one-directional stack, the elements of which refer to the branch points at which descents were made into sublists.

The following subprograms are useful when working with Readers.

The function

LRDROV(LST)

appoints a Reader for the list with alias LST. It is a single SLIP-cell. A fatal error message is sent if LST is not a list alias. The cell name of the Reader is the returned value.

The function

IRARDR(NRD)

does the opposite--it erases the whole Reader stack whose name is, in integer format, in NRD. The returned value indicates the current depth of the Reader's descent into the associated list structure.

The following three routines unpack the information in the top cell of the Reader stack.

The function

LPNTR(NRD)

returns the name of the cell, CADR, to which the Reader currently points.

The function

LOFRDR(NRD)

delivers the name of the list to which the Reader is appointed.

Finally, the function

LCNTR(NRD)

yields the depth to which the Reader has descended into the associated list structure.

Other routines include the function

REED(NRD)

which returns as value the contents of the SLIP-cell to which the Reader NRD points. It can be considered a null advance function compared to the following three routines.

The function

LVLRVT(NRD)

causes the Reader NRD to ascend back to the main list from any current position in a list structure. After the execution of this routine, the Reader points to that SLIP-cell on the main list from which the descent originated. If the Reader initially points to a main-list element no action is taken. When LVLRVT is used as a function the returned value is that of the argument. Thus, LVLRVT can be nested within another function that also requires the NRD argument.

The function

LVLRV1(NRD)

has a similar role, but it makes the Reader ascend only one level. Again, no action is taken if the Reader initially points to the main list.

The function

INITRD(NRD)

goes linearly along the list it is currently pointing to until it hits the Header cell.

A Reader can be initialized from any state by

IDUM=INITRD(LVLRVT(NRD))

Here LVLRVT brings the Reader to the "surface" (i.e., to the main list), and INITRD makes it point to the Header of that list.

There is no theoretical limitation concerning the number of Readers appointed to and traversing concurrently the same list structure. A copy of the Reader can be made by using the function

LRDRCP(NRD)

The name of the new Reader is delivered as its value.

June 1976

The function

LSTPRO(LST,NRD)

searches the Reader stack with alias NRD until it finds a cell that is associated with, or appointed to, the list LST. The returned value is 0 if the search is successful; otherwise it is -1.

This function is needed when a sublist references itself or a higher-level list. Clearly, in this case, structural advancing would never terminate. The function LSTPRO discovers such a situation. Note that the advance functions described below do not use LSTPRO and, therefore, fail in the case of recursive list structures.

The best schema to define the advance functions is

```
          |E|
      |L| | | |L|
ADV   | | |N| | |  (NRD,IFLAG)
      |S| | | |R|
          |W|
```

One letter from each column must be chosen to derive an advance function. ADVLWR and ADVSWL are representative of the twelve possible cases. In each case, the Reader designated by NRD, which was set up beforehand, is made to point to the next unit. The advancement is linear (L) or structural (S); the next unit is a datum element (E), a sublist name (N), or a word containing either of these two types of elements (W); finally, the direction is to the left (L) or right (R).

The returned value of these functions is the datum of the SLIP-cell to which the advanced Reader points. The INTEGER parameter IFLAG stays 0 as long as the advancement does not reach a Header. With linear advancement, when a Header is reached, IFLAG assumes the value -1. With structural advancement, when a Header is reached, the Reader ascends into a superlist if it can. When the Reader already points to the main list, no further ascent is possible; and IFLAG becomes -1. Thus, advancement is not terminated unless the whole list (linear case) or the whole list structure (structural case) has been systematically and selectively traversed. When IFLAG = -1, the returned values are the contents of the datum word of the Header cell and, therefore, are essentially garbage.

The following program segment searches a list structure with alias LST determining whether any of its elements are identical with the FORTRAN word SYMBOL.

```
      DOUBLE PRECISION ADVSER,DATUM,SYMBOL
      LRDR=LRDROV(LST)
10    CONTINUE
      DATUM=ADVSER (LRDR,IFLG)
      IF (IFLG.NE.0) GO TO 20
      IF (DATUM.NE.SYMBOL) GO TO 10
```

```
      WRITE (6, 100) SYMBOL
100   FORMAT (5X,A8,15H IS IN LIST LST)
      GO TO 30
20    CONTINUE
      WRITE (6,110) SYMBOL
110   FORMAT (5X,A8,19H IS NOT IN LIST LST)
30    CONTINUE
      CALL IRARDR (LRDR)
```

As can be seen, the use of the advance functions must be preceded by the creation of the Reader. Similarly, after the task has been accomplished, the programmer should erase the Reader.

Suppose we wish to write a subroutine that searches a list structure with alias LST for all occurrences of the symbol S. If found, S is removed; otherwise nothing happens.

```
      SUBROUTINE SEARCH (LST,S)
      DOUBLE PRECISION ADVSWL,ADVSWR,X,S,LST
      LSTRDR = LRDROV (LST)
100   CONTINUE
      X = ADVSWR (LSTRDR,IFLG)
      IF(IFLG.NE.0) GO TO 200
      IF(X.NE.S) GO TO 100
      IPOINT = LPNTR (LSTRDR)
      CALL ADVSWL (LSTRDR,IFLG)
      CALL DELETE (IPOINT)
      GO TO 100
200   CONTINUE
      CALL IRARDR(LSTRDR)
      RETURN
      END
```

Note that the second call to the advance function in the opposite, left, direction is necessary because of the DELETE operation.

The following three examples illustrate the use of these functions.

```
WHATS = ADVLWR(KADR1,IFLAG1)
```

modifies the LPNTR field of the Reader with alias KADR1 to point to the cell immediately to the right of the one it pointed to before execution of the function. If this new cell is the Header of the list involved, IFLAG1 is set to -1, and WHATS is essentially garbage. Otherwise, IFLAG1 is 0, and WHATS contains the datum of the SLIP-cell currently pointed to by the Reader. Note that no descent is made to a sublist even if a reference to it is encountered.

The statement

```
IUP = INTGER(ADVSNL(NRD2,IFLAG2))
```

assigns to IUP the next accessible alias, if available, on the list structure with Reader NRD2. If the Reader originally points to one, a descent is made. If the original level corresponds to a terminal list with no more sublists, an ascent is made. IFLAG2 stays 0 as long as the assignment is successful. When the Reader finally points to the Header of the main list, IFLAG2 becomes -1. The advances are made in the left direction.

Finally,

```
THERIN = ADVSER(NRD3,IFLAG3)
```

produces the next nonname datum to the right. Descents and ascents are made whenever needed. IFLAG3 is 0 until the Reader no longer points to the Header of the main list. It then becomes -1.

In addition to the Reader and the advance functions, another simpler, less time-consuming, and in some applications completely satisfactory, technique is available. The sequence operations, the basis of this technique, use a Sequence Reader, which is a single FORTRAN word, rather than a stack of special SLIP-cells. Single-level traversal and descent to arbitrary depth are possible; but since no historical record of the descents is kept, no ascent can take place.

The function SEQRDR via the code

```
SRDR = SEQRDR(LST)
```

sets up a single INTEGER computer word SRDR as a Sequence Reader for the list LST. This Sequence Reader contains the cell name of the SLIP-cell currently pointed to by the Reader. A sequence advance operation indicates the cell name to which the advanced Sequence Reader points next.

The returned value of the sequence functions is again the datum of the SLIP-cell being examined. The linear sequence functions

```
SEQLR(SRDR,IFLAG)
```

and

```
SEQLL(SRDR,IFLAG)
```

are analogous to the linear advance functions ADVLWR and ADVLWL, respectively. SRDR is the Sequence Reader. IFLAG is set to -1, 0, or +1 if the retrieved SLIP-cell contains a datum, contains a sublist name, or is a Header cell, respectively.

The structural sequence functions

```
SEQSR(SRDR,IFLAG)
```

and

SEQSL(SRDR,IFLAG)

resemble ADVSER and ADVSEL, respectively. They never terminate on a name cell; rather, they descend into the sublist. When they arrive at a terminal sublist (i.e., one that has no more sublists), they assume the role of the above two linear sequence functions, SEQLR and SEQLL, at that level. Therefore, IFLAG can only be -1 for datum cells and +1 for Header cells.

Recursion

Recursion is one of the most powerful techniques in list processing. SLIP also has this facility although its level of elegance is well below that of LISP.

Let us first consider a few subprograms.

The subroutine

PRESRV(N)

preserves (i.e., pushes down and duplicates) the top cell of the first N(≤10) public lists, the Ws, which are described in the subsection "Programming Conventions (SLIP with FORTRAN IV)."

The subroutine

RESTOR(N)

does the opposite. It deletes the top cell from each of the first N(≤10) public lists.

The function PARMT has two similar forms of implementation. In our version of SLIP

PARMT2(A,B)

places A and B on the top of the first two public lists, W(1) and W(2) respectively.

However

PARMT(N,ARG)

puts the first N(≤10) arguments, ARG(1)...ARG(N), in the top cell of the first N public lists. In both versions of PARMT, the returned value is the new contents of W(1). If an attempt is made to put values on more than the allowable public list an error comment is printed and the program is terminated.

Now we can return to the problem of recursion. Obviously, if a FORTRAN routine calls itself before control is yielded back to its superroutine, the linkages pointing to the superroutine are overwritten and errors arise. In a similar manner, the values of the arguments to be transferred are also lost.

If we could, however, store the return locations and the argument values in push-down stacks, we could solve our problem. Every new call of a routine would add new top layers to the appropriate stacks (linkages and argument values), and a return to a superroutine (or higher-level use of the same routine) would be associated with popping off the top of the stacks. These push-down and pop-off operations are performed by the function VISIT and TERM, respectively.

Actually, SLIP provides the facility for using a recursive loop within a subroutine rather than the more general recursive call of routines. The function VISIT transfers to the first statement of the loop and provides the arguments for it. The subroutine TERM terminates the loop and returns control to the last VISIT function executed. It is possible to go through loops within loops. The format for this is

```
VISIT(INSNAM)
```

and

```
TERM(Y)
```

where INSNAM is an instruction name to which a statement number was previously assigned by a FORTRAN ASSIGN statement. Control is transferred to this statement by VISIT. The return linkage stack, internal to VISIT, is also pushed down.

The value, Y, delivered by VISIT is actually provided by TERM after the calculation in the loop is completed.

This action completes VISIT, and control is transferred to the next statement.

A relatively simple example, using the calculation of N factorial, gives the best explanation. Using its recursive definition,

```
N!  = N* (N - 1)!
0!  = 1
```

we can code it as follows.

```
      FUNCTION IFACT(N)
      COMMON/PUBLICE/ W(100)
      DOUBLE PRECISION AVSL,W
      DOUBLE PRECISION TOP,NEWTOP,REALL
      IF(N.GT.0) GO TO 10
      IFACT = 1
      RETURN
```

```
10    CONTINUE
      ASSIGN 20 TO LOOP
      M = N
      IFACT = INTGER(VISIT(LOOP,NEWTOP(REALL(M),W(1))))
      RETURN
C
C     recursive loop
C
20    CONTINUE
      M = M-1
      IF(M.EQ.0) GO TO 40
      IFACT = INTGER(VISIT(LOOP,NEWTOP(REALL(M),W(1))))
      IFACT = IFACT*INTGER(TOP(W(1)))
30    CONTINUE
      CALL TERM (REALL(IFACT),RESTOR(1))
40    CONTINUE
      IFACT = 1
      GO TO 30
      END
```

The first VISIT function enters the loop at statement 20. The decreasing values of N are pushed down on the argument stack by consecutive executions of the second VISIT function. When finally, with N = 0, TERM passes the control back to the first VISIT function, the stacks have been popped off and all housekeeping duties accomplished.

Another illustrative example of recursion is the SLIP function LSTEQL(LA,LB). In the following sample program, it checks whether two list structures are identical. The function PARMT2(A,B) is used. It places A and B on the top of public lists W(1) and W(2) and delivers the value A.

```
      FUNCTION LSTEQL(LISTA,LISTB)
      DOUBLE PRECISION PARMT2,TOP,ADVLWR,REALL
      DOUBLE PRECISION A,W,LISTA,LISTB,DATUMA,DATUMB
      INTEGER VISIT,READLA,READLB
      COMMON /PUBLIC/ W
      DIMENSION W(100)
C
C...  Recoded by B. Herzog, May 1974
C
C
C      The two input parameters to this function are both
C      names of list structures. The objective of this
C      function is to determine whether or not these
C      list structures are equal. If they are, the value
C      of the function is zero, otherwise it is nonzero.
C      The two list structures are equal if they have
C      identical structures, i.e., sublist names
C      appearing in corresponding places within both
C      structures, and if corresponding elements
C      appearing in both structures are identical.
C
```

```
      LSTEQL=0
      ASSIGN 100 TO L100
      LSTEQL=VISIT(L100,PARMT2(LRDROV(LISTA),LRDROV(LISTB)))
      RETURN
100   CONTINUE
         READLA=INTGER(TOP(W(1)))
         READLB=INTGER(TOP(W(2)))
C...
200   CONTINUE
      IF (LSTEQL.NE.0) GO TO 800
C...     Advance the readers on both lists.
         DATUMA=ADVLWR(READLA,IFLAGA)
         DATUMB=ADVLWR(READLB,IFLAGB)
C...     Inquire if the structures are the same.
C...     IFLAGA or IFLAGB will be nonzero if a HEADER
C...     is encountered on the corresponding list.
         IF (IFLAGA.NE.IFLAGB) GO TO 600
C...        So the structures are the same. Have the readers
C...        returned to the HEADER?
            IF (IFLAGA.NE.0) GO TO 500
C...           No. Now examine the DATUM returned from each list.
C...           Are they list names?
               NAMTST yields zero if Datum is name of a list.
               IF ((NAMTST(DATUMA).EQ.0)
                  .AND.(NAMTST(DATUMB).EQ.0)) GO TO 300
C...              Not list names! But does DATUMA.EQ.DATUMB?
                  IF (DATUMA.NE.DATUMB) LSTEQL=-1
                  GO TO 400
300            CONTINUE
C...              So both are list names; keep advancing.
                  CALL PARMT2(LRDROV(DATUMA),LRDROV(DATUMB))
                  LSTEQL=VISIT(L100)
                  IF (LSTEQL.EQ.0) GO TO 100
                  GO TO 400
400            CONTINUE
500         CONTINUE
            GO TO 700
600         CONTINUE
C...        Arrive here if the list structures are not equal.
            LSTEQL=-1
            GO TO 700
700      CONTINUE
800   CONTINUE
      IF ((LSTEQL.EQ.0).AND.(IFLAGA.EQ.0)) GO TO 900
C....
         CALL IRARDR(TOP(W(1)))
         CALL IRARDR(TOP(W(2)))
         CALL TERM(REALL(LSTEQL),
RESTOR(2))
900      CONTINUE
      GO TO 200
      END
```

Input/Output Operations

In addition to the powerful FORTRAN I/O facilities, the following routines are available in the SLIP system.

Instead of generating a list structure by program, a somewhat cumbersome operation, we can read it in from cards.

The function

RDLSTA(DUMMY)

has a dummy argument, DUMMY, and returns the name of the list or list structure generated. Lists and sublists are delimited by parentheses; elements (always in Hollerith format) are separated by a comma or left parenthesis delineating a sublist. All columns are read up to the current though arbitrary limit of 72. A list structure can be punched on several consecutive cards. It starts with an open parenthesis and is terminated after the last matching closed parenthesis or a terminating asterisk (*). Blanks can be used freely to improve readability, but they will be squeezed out of the string read in, even from within elements. Characters are stored and may be printed in A-format.

If an element is longer than four characters, it is truncated to four characters. If it is shorter than four characters, it is left justified and filled with blanks. All blank elements are ignored.

The subroutine

PRLSTS(LST,M)

prints out the datum of every nonname SLIP-cell of a list structure with alias LST. M can assume three values according to the following table.

M	Mode	Format
1	Integer	5X,I14
2	Real	F19.8
3	Double Precision	D19.8
4	Alphanumeric	A8
5	Hexadecimal	3X,Z16

The subroutine prints text lines to mark the beginning and end of the main list and sublists (empty or nonempty) and a line for each element, in one of the above formats. Description Lists are not printed. Recursive list structures must be avoided since no testing in that regard is included.

For example, if the list structure to be printed M1 is in the string format

(1,(2,3,(4,())),(5,6,7),8,9,10)

the use of

CALL PRLSTS(M1,1)

produces the following.

```
BEGIN LIST
                1
  BEGIN  SUBLIST
                2
                3
  BEGIN  SUBLIST
                4
  BEGIN  SUBLIST
   EMPTY SUBLIST
  END    SUBLIST
  END    SUBLIST
  END    SUBLIST
  BEGIN  SUBLIST
                5
                6
                7
  END    SUBLIST
                8
                9
               10
END LIST
```

TYPES OF SLIP FUNCTIONS

As noted before, many SLIP quantities and functions should be declared DOUBLE PRECISION. Several functions have values which are normally used as integer numbers, so these functions must be declared INTEGER. Several other functions may be declared DOUBLE PRECISION or INTEGER and are usually used as integers. These include XMASK, all partial word functions. Several other functions may be used as integers but should normally be used as double-precision quantities.

Flags are returned by many routines, including SEQLL, ADVLWR, etc., and are returned as integers.

A complete listing of functions and their modes and the modes of their arguments appears in the subsection "Summary of 360 SLIP Functions and Subroutines."

New SLIP Functions

Several functions have been added to the SLIP repertoire for the IBM SYSTEM/360/370 version:

DATUM(A) and IDATUM(A)

These functions are identical; the two names making it convenient for INTEGER/REAL conventions. DATUM should normally be declared DOUBLE PRECISION and IDATUM should be declared INTEGER. The value of DATUM is the datum contained in the SLIP cell whose cell name is A. The value of IDATUM(A) is the contents of the leftmost word of the datum in the SLIP cell whose cell name is A.

REALL(Y)

This function is similar in purpose to INTGER. Y is assumed to be a fullword quantity (integer or real). The value of REALL(Y) is a doubleword consisting of Y in the left word and 0s in the right word.

REALS(D)

This function is similar in purpose to INTGER and REALL. The argument is assumed to be DOUBLE PRECISION (but could be REAL or INTEGER). The mode of the function is REAL and it returns the leftmost word of the argument without conversion.

SLPDMP

This subroutine gives a dump of all SLIP storage in a format oriented toward the SLIP structure. It is considerably easier to use than a regular memory dump.

SETRAC

This function, whose arguments are the same as those of INITAS, sets the SLIP system to give a dump (via SLPDMP) if an error requiring program termination is detected (e.g., exhausting the available SLIP storage).

F4TRBK

This subroutine should be used as an error exit. A function and subroutine traceback and a SLIP dump may be obtained.

MRK(D)

This function retrieves the MRK field of the doubleword D.

SETMRK(I,A)

This function sets the mark I in the MRK field of the SLIP cell whose name is A.

A series of functions, called partial word functions, which obtain parts of words have been implemented. The values of these functions may be treated as integer (the normal situation) or double-precision quantities. In all cases, the value returned is right-justified.

C1(A) and CHR1(A)

These functions are identical and return the contents of the first character (first byte or 8 bits) of the word A. A may be a fullword or doubleword quantity.

C2,CHR2,C3,CHR3,C4,CHR4

These are similarly defined.

C5,CHR5,...,C8,CHR8

These are similarly defined except they must have a doubleword argument.

Q1(A) and QTR1(A)

These functions are identical and return the first (leftmost) quarter (two bytes or 16 bits or half-word) of A.

Q2,QTR2,Q3,QTR3,Q4,QTR4

These are appropriately defined. A must be a doubleword for Q3,QTR3,Q4 and QTR4.

H1(A) and HLF1(A)

These functions are identical and return the leftmost word of A.

H2(A) and HLF2(A)

These functions are identical and return the rightmost word of the (doubleword) quantity A.

For example, consider the number X = 0102030405060708 (in hexadecimal):

	Fullword	Doubleword
C1(x) is	00000001	0000000000000001
C3(X) is	00000003	0000000000000003
Q3(X) is	00000506	0000000000000506

See Figure 2 below.

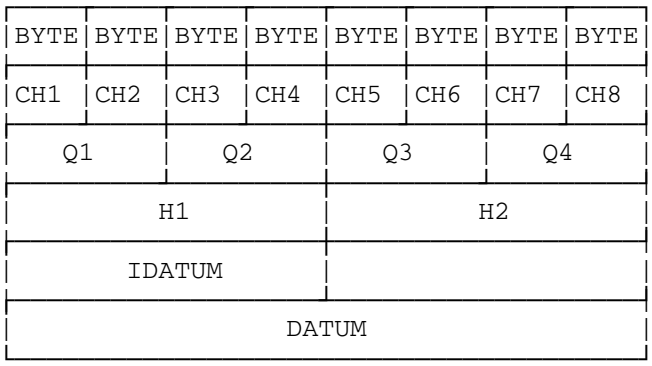


Figure 2.

SUMMARY OF 360 SLIP FUNCTIONS AND SUBROUTINES

The list below gives only those SLIP functions for application programs, what they expect for arguments, and what they return as values. Omitted from this list are those SLIP functions that manipulate the link word of the cells.

Symbols beginning with	represent
A	Cell names, INTEGER (compare to CADR). Note that whenever an A is expected as an argument a DA may be offered.
AT	Attribute of description list, DOUBLE PRECISION
D	Any datum, DOUBLE PRECISION
DA	Listname or cell name in name or cell name format but DOUBLE PRECISION (compare to MADR). Note that whenever a DA is expected as an argument it is not proper to offer an A.
F	Flag, INTEGER (-1, 0, or 1)
L	List name, DOUBLE PRECISION (compare to LST)
I	INTEGER
LR	List reader, INTEGER
R	A fullword, REAL
S	Sequence reader (sequencer), INTEGER
V	Value of an attribute, DOUBLE PRECISION
X	Any value
Y	Any fullword (INTEGER or REAL) value

The following table lists the 360 SLIP functions and subroutines and their values.

Value	Routine and Arguments
D	ADVLEL(LR,F)
D	ADVLER(LR,F)
D	ADVLNL(LR,F)
D	ADVLNR(LR,F)
D	ADVLWL(LR,F)
D	ADVLWR(LR,F)
D	ADVSEL(LR,F)
D	ADVSER(LR,F)
D	ADVSNL(LR,F)
D	ADVSNR(LR,F)
D	ADVSWL(LR,F)
D	ADVSWR(LR,F)
D	BOT(L)
A	CADLFT (A,F)
A	CADRGT (A,F)
I or D	C1(X)
I or D	C2(X)
I or D	C3(X)
I or D	C4(X)
I or D	C5(D)
I or D	C6(D)
I or D	C7(D)
I or D	C8(D)
I or D	CHR1(X)
I or D	CHR2(X)
I or D	CHR3(X)
I or D	CHR4(X)
I or D	CHR5(D)
I or D	CHR6(D)
I or D	CHR7(D)
I or D	CHR8(D)
D	DATUM(A or DA)
D	DELETE(A)
-	DERROR(L)
-	F4TRBK
I or D	H1(X)
I or D	H2(D)
I or D	HLF1(X)
I or D	HLF2(D)
I	IDATUM(A or DA)
-	INITAS(D,I)
LR	INITRD(LR)
L	INLSTL(L1,A)
L	INLSTR(L1,A)
I	INTGER(X)
I	IRALST(L)
I	IRARDR(LR)
V	ITSVAL(AT,L)

Value	Routine and Arguments
V	LCNTR(LR)
L	LDATVL(AT,V,L)
L	LIST(L1 or 9)
L	LISTAV(L1)
L	LISTMT(L)
L	LOFRDR(LR)
A	LPNTR(LR)
I	LPURGE(L)
LR	LRDRCP(LR1)
LR	LRDROV(L)
L	LSSCPY(L1)
F	LSTEQL(L1,L2)
I	LSTMRK(L)
F	LSTPRO(L,A)
LR	LVLRV1(LR)
LR	LVLRVT(LR)
A	MADATR(AT,L)
DA	MADLFT(A)
A	MADNBT(L,I)
A	MADNTP(L,I)
DA	MADRGT(A)
L	MAKEDL(L1,L2)
I	MRK(D)
L	MRKLSS(I,L1)
L	MRKLST(I,L1)
L	MTDLST(L1)
L	MTLIST(L1)
L	NAMEDL(L)
A	NEWBOT(D,L)
A	NEWTOP(D,L)
D	NEWVAL(AT,V,L)
A	NXTLFT(D,A or DA)
A	NXTRGT(D,A or DA)
AT	NOATVL(AT,L)
L	NULSTL(A,L1)
L	NULSTR(A,L1)
-	PARMT2(X1,X2)
-	PARMTn(X1,X2,...,Xn)
D	POPBOT(L)
D	POPTOP(L)
-	PRESRV(I)
-	PRLSTS(L,I)
I or D	Q1(X)
I or D	Q2(X)
I or D	Q3(D)
I or D	Q4(D)
I or D	QTR1(X)
I or D	QTR2(X)
I or D	QTR3(D)
I or D	QTR4(D)

Value	Routine and Arguments
L	RDLSTA(-)
D	REALL(X)
R	REALS(X)
D	REED(LR)
-	RESTOR(I)
D	SEQLL(S,F)
D	SEQLR(S,F)
S	SEQRDR(L)
D	SEQSL(S,F)
D	SEQSR(S,F)
-	SETMRK(I,A)
-	SLPDMP(D,I)
-	STRDAT(D,A)
D	SUBSBT(D1,L)
D	SUBST(D1,A)
D	SUBSTP(D1,L)
-	TERM(0.0D0 or 0 or D)
D	TOP(L)
-	VISIT(0 or I)

HOW TO USE SLIP

All error comments and the SLIP dump, obtained via DMPSVM or SLPDMP are dispatched to device 16. If an error dump is to be sent to a terminal printer then the user should note that the current format requires 92 printing columns. It is, therefore, best to assign 16 to a temporary file which can be examined with the editor and perhaps eventually printed by copying to *PRINT*. Alternately 16 can be assigned to *PRINT* directly.

Note: In the near future, the device for error comments will be different from the one for dumps.

Thus, typical $RUN commands would be:

```
$RUN yourprogram+*SLIP 16=*PRINT* ...
```

or

```
$RUN yourprogram+*SLIP 16=-BUG ...
```

or it may be useful, for debugging runs from a terminal, to source a file containing the following:

```
$CREATE -BUG
$EMPTY -BUG
$DEBUG yourprogram+*SLIP 16=-BUG ...
$CONTINUE WITH *MSOURCE*
```

Note: These instructions are subject to change--please contact the Computing Center staff if you encounter any difficulties.

REFERENCES

(1) Newell, A., et al., eds., Information Processing Language-V Manual, 2nd ed., Englewood Cliffs: Prentice-Hall, 1965.

(2) Sammet, J. E., Programming Languages: History and Fundamentals, Chapter 6, Engelwood Cliffs: Prentice-Hall, 1969.

(3) Weizenbaum, J., "Symmetric List Processor," Communications of the ACM, Volume 6, No. 9, pp. 524-544, September 1963.

(4) Russel, D. B., Letter to the Editor, Communications of the ACM, Volume 8, No. 5, p. 263, May 1965.

(5) Weizenbaum, J., Letter to the Editor, Communications of the ACM, Volume 8, No. 5, pp. 263-264, May 1965.

(6) Findler, N. V., Pfaltz, J. L., and Bernstein, H. J., Four High-Level Extensions of FORTRAN IV: SLIP, AMPOL-II, TREETRAN, SYMBOLANG, New York: Spartan Books, 1972.

(7) Waite, William M., Implementing Software for Non-Numeric Applications, Series in Automatic Computation, New York: Prentice-Hall, 1973.

INDEX

Reader's Comment Form

LISP and SLIP in MTS
Volume 8
June 1976
(January 1983 Reprint)

Errors noted in publication:

Suggestions for improvement:

Your comments will be much appreciated. The completed form may be sent to the Computing Center by Campus Mail or U.S. Mail, or dropped in the Suggestion Box at the Computing Center, NUBS, or BSAD.

Date _____

Name _____

Address _____

_____

_____

Publications
Computing Center
University of Michigan
Ann Arbor, Michigan 48109

Update Request Form

LISP and SLIP in MTS
Volume 8
June 1976
(January 1983 Reprint)

Updates to this manual will be issued periodically as errors are noted or as changes are made to MTS. If you desire to have these updates mailed to you, please submit this form.

Updates are also available in the memo files at both the Computing Center and NUBS; there you may obtain any updates to this volume that may have been issued before the Computing Center receives your form. Please indicate below if you desire to have the Computing Center mail to you any previously issued updates.

Name _____

Address _____

_____

_____

Previous updates needed (if applicable):_____

The completed form may be sent to the Computing Center by Campus Mail or U.S. Mail, or dropped in the Suggestion Box at the Computing Center, NUBS, or BSAD. Campus Mail addresses should be given for local users.

Publications
Computing Center
The University of Michigan
Ann Arbor, Michigan 48109

Users associated with other MTS installations (except the University of British Columbia) should return this form to their respective installations. Addresses are given on the reverse side.

Addresses of other MTS installations:

Publications Clerk
352 General Services Bldg.
Computing Services
The University of Alberta
Edmonton, Alberta
Canada T6G 2H1

Information Officer, NUMAC
Computing Laboratory
The University of Newcastle upon Tyne
Newcastle upon Tyne
England NE1 7RU

Rensselaer Polytechnic Institute
Documentation Librarian
310 Voorhees Computing Center
Troy, New York 12181

Simon Fraser University
Computing Centre
User Services Information Group
Burnaby, British Columbia
Canada V5A 1S6

Wayne State University
Computing Services Center
Academic Services Documentation Librarian
5950 Cass Ave.
Detroit, Michigan 48202