SAILON 28.3

STANFORD LISP 1.6 MANUAL

BY

LYNN H. QUAM

SEPTEMBER 1969

STANFORD ARTIFICIAL INTELLIGENCE PROJECT

PREFACE

This manual is an extensive rewrite of SAILON 28.2, and supercedes and replaces SAILONs 1, 4, 28, 28.1 and 28.2. Sections have been rewritten for both clarity and accuracy.

The major language changes since SAILON 28.2 are summarized as follows:

Sections 14.3.3, 3 and 3.3 - READ now has strings, super-parentheses, comments and QUOTEing.

Ch9 - EQUAL now compares integers with reals.

Ch 16 - *RSET and BAKGAG have been changed.

Acknowledgment

The STANFORD A.I. LISP 1.6 System was originally an adaptation of one developed by the Artificial Intelligence Project at M.I.T. Since 1966, that system has been largely rewritten by John Allen and the author.

John R. Allen implemented the storage reallocation system which make it possible for the user to change the sizes of the various memory spaces. He also designed and coded the editor ALVINE, wrote the first loader interface, and generally maintained and debugged the system. John Allen contributed the ALVINE documentation in Appendix A.

The author implemented the multichannel input-output system, the compatible loader interface, a faster arithmetic package which interfaces to the BIGNUM package, the BIGNUM package and various other smaller packages.

TABLE OF CONTENTS

CHAPTER Page

CHAPTER 1

INTRODUCTION

This manual is intended to explain the interactive LISP 1.6 system which has been developed for the PDP-10 at the Stanford University Artificial Intelligence Project. It is assumed that the reader is familiar with either some other LISP system or the LISP 1.5 PRIMER by Clark Weissman.[2]

The LISP 1.6 system described has as a subset most of the features and functions of other LISP 1.5 systems. In addition, there are several new features such as an arbitrary precision integer package, an S-expression editor, up to 14 active input-output channels, the ability to control the size of memory spaces, a standard relocating loader to load assembly language or compiled programs, etc.

This system uses an interpreter; however, there is also a compiler which produces machine code. Compiled functions are approximately ten times as fast and also take less memory space.

This manual is organized in a functional manner. First the basic data structures are described; then the functions for operating on them. The appendices present more detailed information on the system, its internal structure, the compiler, and several auxiliary packages.

1.1 Guide to the Novice:

The user who is not experienced with any LISP system is advised to follow the instructions below:

1) Become familiar with Weissman's LISP 1.6 Primer[2] or some equivalent introductory LISP Manual.

2) Learn the document conventions (1.4).

3) Become superficially familiar with LISP 1.6 identifiers, numbers and S-expressions (Chapters 3,4,and 5).

4) Understand the most useful functions: Those proceeded by exclamation marks " ! " in chapters 6 through 14.

5) Learn how to define functions (6.1).

6) Learn how to interact with LISP (Chapter 2).

7) Try some examples. Weissman[1] has some good problems.

8) Learn what other useful functions and features are available (1.3).

1.2 Guide to the User Experienced with Another LISP System

The user who has used another LISP system is advised to follow these instructions:

1) Learn the document conventions (1.4).

2) Learn top level of LISP 1.6 is EVAL, not EVALQUOTE.

3) Use DE,DF and DEFPROP for defining functions. (Section 11.1).

4) Many functions differ from those in other systems. Most of these are noted in the index.

5) The syntax of atoms is different from other systems (Chapters 3 and 4).

6) Learn how to interact with LISP (Chapter 2).

7) Try some examples.

8) Learn what other useful functions and features are available (1.3).

1.3 Guide to Useful Functions and Features

The following is a partial list of useful features and functions in LISP 1.6 and what they might be useful for.

1) ALVINE (Appendix A) is useful for editing functions and manipulating I/O files.

2) READ has some very useful control characters (Section 14.3).

3) Input/Output (Chapter 14) is very flexible.

4) One can control error messages (Chapter 16).

5) There is a LISP compiler (Appendix F) which generates code that runs approximately ten times as fast as interpreted functions.

6) There are auxiliary files on the disk which are often useful:

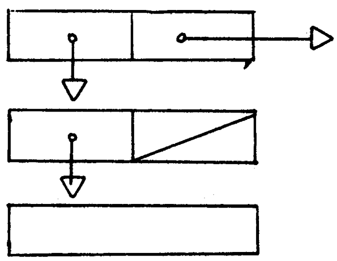
filename	use	document
SYS: SMILE	file manipulation	SAILON 41
SYS: TRACE	tracing function calls and SETQs	SAILON 41
SYS: (LISPDP.LSP)	III display functions	SAILON 41
SYS: (BIGNUM.LSP)	arbitrary precision integers	Appendix I
SYS: (DEBUG.LAP)	interactive debugging	SAILON 41
(SP DOC) LISP	corrections to this manual	
SYS: GRIN	Interpreted GRINDEF	Appendix A

7) One can load and link LISP to assembly language and Fortran compiled programs (Appendix H).

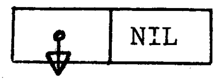
1.4 Document Conventions

1.4.1 Representation Conventions

In the description of data structures, the following notational conventions will be used.



represents a 36-bit word in FREE STORAGE with 2 18-bit pointers.

means 

represents a 36-bit word in FULL WORD SPACE.

1.4.2 Syntax Conventions

A slightly modified form of BNF is used to define syntax equations. Optional terms are surrounded by curly brackets { and }.

1.4.3 Calling Sequence Conventions

Calling sequences to LISP functions are presented in S-expression form, with the CAR of the S-expression being the name of the function. An argument to a function is evaluated unless that argument is surrounded by quotes (") in the calling sequence definition. Quotes mean that the function implicitly QUOTEs that argument.

Examples:	(SETQ "ID" V)	ID is not evaluated, but V is evaluated.
	(QUOTE "V")	V is not evaluated.

1.4.4 Other Conventions

The blank character (ASCII 40) is indicated by "␣" when appropriate for clarity.

A special notation in the left margin is used to indicate the degree of utility or difficulty of each section of this manual:

mark	meaning
!	basic
<no mark>	generally useful
*	useful but more sophisticated
#	not generally useful

CHAPTER 2

INTERACTIVE USE OF THE SYSTEM

This chapter attempts to explain how to use the LISP system in the interactive time-sharing environment of the PDP-6/10.

2.1 The Top Level

The top level of this system does not use EVALQUOTE as do many systems. However, EVALQUOTE may be defined as follows:

```
(DE EVALQUOTE NIL
    (PROG NIL
     L    (TERPRI)
          (PRINT (EVAL (CONS (READ) (MAPCAR
              (FUNCTION (LAMBDA (X) (LIST (QUOTE QUOTE) X)))
              (READ)))))
          (GO L)))
```

The top level of LISP 1.6 is equivalent to:

```
(PROG NIL
 L    (TERPRI)
      (PRINT (EVAL (READ)))
      (GO L))
```

All examples at the top level assume this definition.

2.2 Using the System

The following dialog shows how to log into the time-sharing system, start the LISP system, and interact with the top level of LISP. Lines beginning with period " . " are typed by the user to the time-sharing system, and the lines beginning with asterisk are typed to LISP. The symbol "↵" specifies carriage-return and line-feed, "$" means altmode, and ␣ means space typed by user.

Log in	.L #1/FOO↵ ↑C	Give your project-programmer number.
Starting LISP	.R LISP↵	Core size may be specified - Note 1.
	ALLOC?␣	Memory allocation can be specified - Note 2.
	AUXILIARY FILES?␣	Useful auxiliary files can be loaded - Note 3.
	DECIMAL?␣	Respond "Y" for decimal initialization of BASE and IBASE.

*T$	T and NIL always evaluate to themselves.
T	
*(QUOTE(A B C))	
(A B C)	Value of QUOTE
*(CONS 1 (QUOTE A))	Numbers always evaluate to themselves and thus need not be quoted.
(1 . A)	
*(PLUS 1 2)	
3	
*(INC (INPUT SYS: TRACE))	This READS the system file TRACE
NIL	
<a long sequence of output>	This output can be suppressed with ↑O.

Note 1. For limited use of the LISP system, type R LISP ↵.
If more core is needed, type R LISP n ↵ , where n is the desired number of 1024 word blocks.

Note 2. For limited use, type ↵ after ALLOC?. To allocate memory spaces type Y. The allocation procedure is explained in Appendix C.

Note 3. A response of Y to "AUXILIARY FILES" will read in auxiliary files of functions. Type Y for yes, blank for no after each question mark.

AUXILIARY FILES?Y

SMILE?	Functions for file handling - See SAILON 41.
ALVINE?	The LISP editor - see Appendix A.
TRACE?	Functions for debugging - See SAILON 41.
LAP?	The LISP assembler - see Appendix G.

2.3 Special Teletype Control Characters

The time-sharing system treats many control characters in special ways. For a complete discussion of control characters see the PDP-10 TIME SHARING MONITOR MANUAL. Briefly, the following special control character is used in LISP.

Teletype	III Display	Meaning
↑C	CALL	Stop the job and talk to time-sharing system.
↑O	5	Suppress console printout until an input is requested.
↑U	∃	Delete the entire input line now being typed. (Only with (DDTIN NIL)).
↑G (BELL)	π	Stop the LISP interpreter and return control to the top level of LISP. Only effective when LISP is asking for console input. See INITFN (16).
rubout	BS	Delete the last character typed. (For (DDTIN T) see 14.2.1).

CHAPTER 3

IDENTIFIERS

Identifiers are strings of characters which taken together represent a single atomic quantity.

Syntax:

<comments>::= <ASCII 176> <any sequence of characters not including line-feed> <line-feed>

<delimiter>::=(|) |[|]|•|@|/|"| <blank> | <altmode>|<carriage-return>|

<line-feed>|<tab>|<form-feed>

<character>::= <any extended ASCII character other than null and ASCII 176>

<digit>::= Ø|1|2|3|4|5|6|7|8|9

<letter>::=<any character not a digit and not a delimiter>

<identifier>::= <letter>

::= <identifier><letter>

::= <identifier><digit>

::= / <character>

::= <identifer>/<character>

Semantics:

Identifiers are normally strings of characters beginning with a letter and followed by letters and digits. It is sometimes convenient to create identifiers which contain delimiters or begin with digits. The use of the delimiter "/" (slash) causes the following character to be taken literally, and the slash itself is not part of the identifier. Thus, /AB is the same as AB is the same as /A/B.

Comments are useful for allowing descriptive text in files which will be completely ignored when read. Comments also make it possible to extend atoms (identifiers, strings and numbers) across line boundaries without any of the characters in the comment becoming part of the atom.

ASCII 176 cannot be typed directly into LISP. In STOPGAP, ?3 designates ASCII 176. On the line printer and III displays, ASCII 176 prints as tilde "~". ASCII 176 does not print on teletypes. (See CHRCT in 14.1.4.)

Examples:

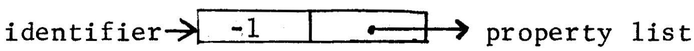
A
a
FOObaz

Examples (Continued)

TIME-OF-DAY
A1B2
/(
?
/13245
/.
LPT:

Representation:

An identifier is internally represented as a dotted pair of the following form:



which is called an atom header.

Thus CDR of an identifier gives the property list of the identifier, but CAR of an identifier gives the pointer 777777, which if used as an address will cause an illegal memory reference, and an error message. An identifier is referred to in symbolic computation by the address of its atom header.

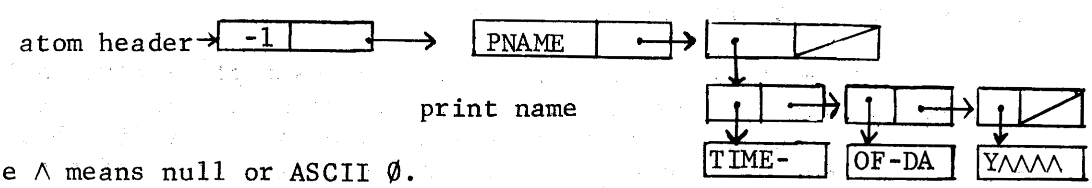
3.1 Property Lists

The property list of an identifier is a list of pairs: (property name, property value) associated with that identifier. The normal kinds of properties which are found in property lists are print names, values, and function definitions corresponding to identifiers.

3.1.1 Print Names

Every identifier has a print name (PNAME) on its property list. The print name of an identifier is a list of full words, each containing five ASCII characters.

Example: The identifier TIME-OF-DAY would be initially represented as follows:



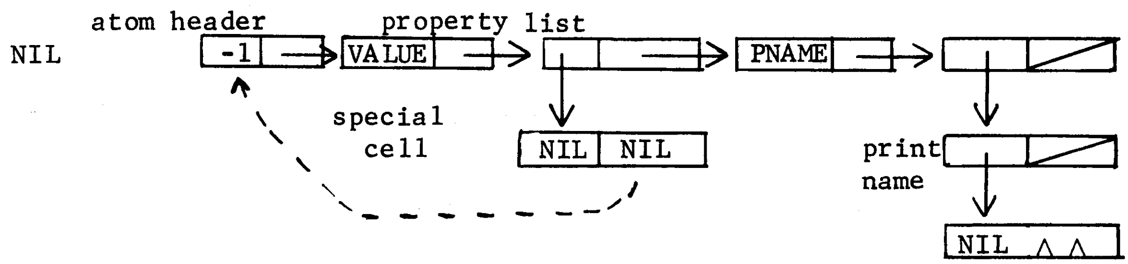
where ∧ means null or ASCII Ø.

3.1.2 Special Cells

When a value is assigned to an identifier, the property name VALUE is put on the identifier's property list with property value being a pointer to a special cell. The CDR of the special cell (sometimes called

VALUE cell) holds the value of the identifier, and the address of a special cell remains constant for that identifier unless REMPROPed (11.2), to enable compiled functions to directly reference the values of special variables. Global variables and all variables bound in interpreted functions store their values in special cells.

Example: The atom NIL has the following form:



3.2 The OBLIST

In order that occurrences of identifiers with the same print names have the same internal address (and hence value), a special list which is the VALUE of a global variable called OBLIST is used to remember all identifiers which READ and some other functions have seen. For the sake of searching efficiency, this list has two levels; the first level contains sequentially stored "buckets" which are "hashed" into as a function of the print name of the identifier. Each bucket is a list of all distinct identifiers which have hashed into that bucket. Thus, (CAR OBLIST) is the first bucket, and (CAAR OBLIST) is the first identifier of the first bucket.

3.3 Strings

Syntax:

string::= "<any sequence of characters not containing ">"

Semantics:

A string is an arbitrary sequence of characters surrounded by double quotes and not containing double quotes. Strings are represented identically to identifiers except that strings are not automatically INTERNed on the OBLIST. The double quotes surrounding strings actually become part of the PRINT NAME of the string unlike slashes in slashified identifiers.

Examples:

"I AM A STRING"
"1,3-X 5"

CHAPTER 4

NUMBERS

There are two syntactic types of numbers: integer and real.

<number> ::= <integer> | <real>

4.1 Integers

Syntax:

<integer> ::= {<sign>} <digits> {.}

<digits> ::= <digit>{<digits>}

<sign> ::= + | -

Semantics:

The global variable IBASE specifies the input radix for integers which are not followed by ".". Integers followed by "." are decimal integers. IBASE is initially = 8. Similarly, the global variable BASE controls output radix for integers. If BASE = 10 then integers will print with a following ".", unless the global variable *NOPOINT = T.

Examples with IBASE=8

input		meaning	
-13	=	-11.	$= -11_{10}$
1000	=	512.	$=+512_{10}$
19	=	17.	$= +17_{10}$

Representation:

There are three representations for integers depending on the numerical magnitude of the integer: INUM, FIXNUM, and BIGNUM. Their ranges are as follows:

INUM	$\|n\| < K$	K is usually 2^{16}
FIXNUM	$K \le \|n\| < 2^{35}$	
BIGNUM	$2^{35} \le \|n\|$	

Representation of INUMs:

INUMS are small integers represented by pointers outside of the normal LISP addressing space. INUMs are addresses in the range 2^{18}-2K to 2^{18}-2. The INUM representation for zero is $\alpha = 2^{18}-K-1$.

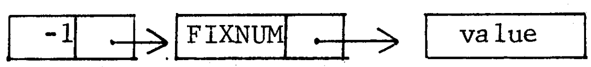
Examples:

INUM	representation
-(K-1)	α - (K-1)
-1	α - 1
Ø	$\alpha = 2^{18}$-K-1
1	α + 1
K-1	α + K-1

Representation of FIXNUMs:

FIXNUMs are represented by list structure of the following form:

atom header

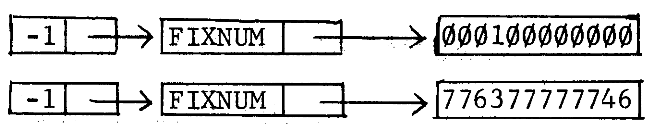


where value is the 2's complement representation of the fixed point number.

Examples:

+1ØØØØØØØØ$_8$

-14ØØØØØØ32$_8$

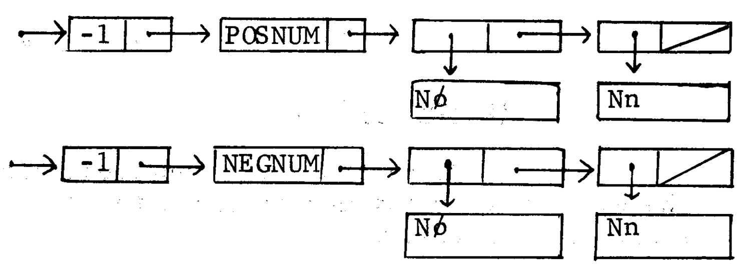


Representation of BIGNUMs

BIGNUMs are represented by list structure of the following forms:

Positive BIGNUMs

Negative BIGNUMs



where N_i are positive 36 bit integers ordered from least to most significant. The value of a BIGNUM is

$$\text{sign} \cdot \sum_{i=\emptyset}^{n} N_i \cdot (2^{35})^i .$$

Note: BIGNUMS are not normally a part of the interpreter. Appendix H describes the procedures for loading the BIGNUM package.

4.2 Reals

Syntax:

<real> ::= {<sign>} <digits> {.} <exponent>

::= {<sign>} {<digits>} . <digits> {<exponent>}

<exponent> ::= E {<sign>} <digits>

Examples:

	meaning
3.14159	+3.14159
+1E-3	+Ø.ØØ1
-196.37E4	-19637ØØ.Ø
Ø.3	+Ø.3
-Ø.3E+1	-3.Ø

Restrictions:

The radix for real numbers is always decimal. A real number x must be in the (approximate) range:

$$1Ø^{-38} < |x| < 1Ø^{+38} \quad \text{or} \quad x = Ø$$

A real number has approximately eight significant digits of accuracy.

Representation:

atom header

where value is in PDP-6/1Ø 2's complement floating point representation.

CHAPTER 5

S-EXPRESSIONS

Syntax:

<atom> ::= <identifier> | <string> | <number>

<S-expression> ::= <atom>

::= (<S-expression list>{.<S-expression>})

::= ( ) = NIL

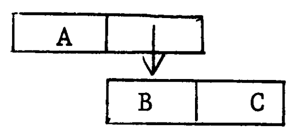
<S-expression list> ::= <S-expression>

::= <S-expression> <S-expression list>

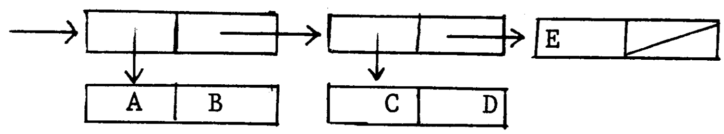
Representation:

S-expression / representation

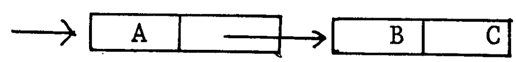
(A . (B . C))



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



(A B . C)



Exceptions:

The identifier NIL is the identifier which represents the empty list, i.e., () .

CHAPTER 6

LAMBDA EXPRESSIONS

LAMBDA expressions provide the means of constructing computational procedures (often called functions, subroutines, or procedures) which compute answers when values are assigned to their parameters. A LAMBDA expression can be bound to an identifier so that any reference to that identifier in functional context refers to the LAMBDA expression. In LISP 1.6 there are several types of function definition which determine how arguments are bound to the LAMBDA expression. The following is a LAMBDA expression:

! (LAMBDA "ARGUMENT-LIST" "BODY")

LAMBDA defines a function by specifying an ARGUMENT-LIST, which is a list of identifiers (except for LEXPRs, see 6.3) and a BODY, which is an S-expression. LAMBDA expressions may have no more than five arguments if they are to be compiled.

Examples: (LAMBDA NIL 1)

This LAMBDA expression of no arguments always evaluates to one.

(LAMBDA (X) (TIMES X X))

This LAMBDA expression computes the square of its argument, if x is a number. Otherwise an error will result.

(LABEL "ID" "LAMBDA-EXPR")

LABEL creates a temporary name ID for its LAMBDA expression. This makes it possible to construct recursive functions with temporary names.

Example:

```
(DE REVERSE (L)
    ((LABEL REVERSE1
            (LAMBDA (L M)
                    (COND ((ATOM L) M)
                          (T (REVERSE1 (CDR L) (CONS  (CAR L) M))))))
     L NIL))
```

LAMBDA expressions are evaluated by "binding" actual arguments to dummy variables of the LAMBDA expression, (see Chapter 14) then evaluating the body inside the LAMBDA expression with the current dummy variable bindings. However, actual arguments to LAMBDA expressions are handled in a variety of ways. Normally, there is a one-to-one correspondence between dummy variables and actual arguments, and the actual arguments are evaluated before they are bound. However, there are three special forms of function definition which differ in their handling of actual arguments.

! 6.1 EXPRs and SUBRs

An EXPR is an identifier which has a LAMBDA expression on its property list with property name EXPR. EXPRs are evaluated by binding the values of the actual arguments to their corresponding dummy variables. DE (see 11.1) is useful for defining EXPRs. The compiled form of an EXPR is a SUBR.

Examples:

```
(DE SQUARE (X) (TIMES X X))
(DE *MAX (X Y) (COND ((GREATERP X Y) X) (T Y)))
```

6.2 FEXPRs and FSUBRs

A FEXPR is an identifier which has a LAMBDA expression of one dummy variable on its property list with property name FEXPR. FEXPRs are evaluated by binding the actual argument list to the dummy variable without evaluating any arguments. DF (see 11.1) is useful for defining FEXPRs. The compiled form of an FEXPR is an FSUBR.

Examples:

```
(DF LISTQ (L) L)
        (LISTQ A (B) C) = (A (B) C)
        (LISTQ) = NIL
(DF DEFINE (L)
    (MAPC (FUNCTION (LAMBDA (X) (PUTPROP (CAR X)
                                          (CADR X)
                                          (QUOTE EXPR))))
          L))
       (DEFINE (LEQ (LAMBDA (X Y) (OR (LESSP X Y)
                                       (EQUAL X Y))))
               (GEQ (LAMBDA (X Y) (OR (GREATERP X Y)
                                       (EQUAL X Y)))))
```

6.3 LEXPRs and LSUBRs

An LEXPR is an EXPR whose LAMBDA expression has an atomic argument "list" of the form:

(LAMBDA "ID" "FORM")

LEXPRs may take an arbitrary number of actual arguments which are evaluated and referred to by the special function ARG. ID is bound to the number of arguments which are passed. The compiled form of an LEXPR is an LSUBR.

(ARG N)

ARG returns the value of the Nth argument to an LEXPR.

Example:

```
(DE MAX N
   (PROG (M)
         (SETQ M (ARG N))
         (SETQ N (SUB1 N))
         (COND ((ZEROP N) (RETURN M))
               ((GREATERP (ARG N) M) (SETQ M (ARG N))))
         (GO L)))
(MAX 1 1.2 4 3 -5∅) = 4
```

(SETARG N V)

SETARG sets the value of the Nth argument to V and returns V.

6.4 MACROs

A MACRO is an identifier which has a LAMBDA expression of one dummy variable on its property list with property name MACRO. MACROs are evaluated by binding the list containing the macro name and the actual argument list to the dummy variable. The body in the LAMBDA expression is evaluated and should result in another "expanded" form. In the interpreter, the expanded form is evaluated. In the compiler, the expanded form is compiled. DM (see 11.1) is useful for defining MACROs.

Examples:

1) We could define CONS of an arbitrary number of arguments by:

```
(DM CONSCONS (L)
   (COND ((NULL (CDDR L)) (CADR L))
         (T (LIST (QUOTE CONS)
                  (CADR L)
                  (CONS (QUOTE CONSCONS) (CDDR L))))))
```

(CONSCONS A B C) would call CONSCONS with L = (CONSCONS A B C). CONSCONS then forms the list (CONS A (CONSCONS B C)). Evaluating this will again call CONSCONS with L = (CONSCONS C). CONSCONS will finally return C.

The effect of (CONSCONS A B C) is then (CONS A (CONS B C)).

2) We could define a function EXPAND which is more generally useful for MACRO expansion:

```
(DE EXPAND (L FN)
    (COND ((NULL (CDR L)) (CAR L))
          (T (LIST FN (CAR L) (EXPAND (CDR L) FN)))))
```

Then we could define CONSCONS:

```
(DM CONSCONS (L) (EXPAND (CDR L) (QUOTE CONS)))
```

It should be noted that MACROs are more general than FEXPRs and LEXPRs. In fact the previous definitions can be replaced by the following MACROs:

```
(DM LISTQ (L) (LIST (QUOTE QUOTE) (CDR L)))
(DM MAX (L) (EXPAND (CDR L) (QUOTE *MAX)))
        (MAX A B C D) would expand to:
        (*MAX A (*MAX B (*MAX C D)))
```

3) (*EXPAND L FN)
   (*EXPAND1 L FN)

*EXPAND and *EXPAND1 are MACRO expanding functions used by PLUS, TIMES, etc. They are equivalent to:

```
(DE *EXPAND (L FN) (*EXPAND1 (REVERSE (CDR L)) FN))
(DE *EXPAND1 (L FN)
  (COND ((NULL (CDR L)) (CAR L))
    (T (LIST FN (*EXPAND1 (CDR L) FN) (CAR L)))))
```

Example:

With PLUS defined as

```
(DM PLUS (L) (*EXPAND L(QUOTE *PLUS)))
(PLUS A B C D) expands to:
   (*PLUS (*PLUS (*PLUS A B) C) D)
```

CHAPTER 7

EVALUATION OF S-EXPRESSIONS

This chapter describes the heart of the LISP interpreter, the **mechanism** for evaluating S-expressions.

! (*EVAL E)
(EVAL E)

*EVAL and EVAL (see 7.2) evaluate the value of the S-expression E.

Examples:

```
     (EVAL ( LIST (QUOTE ADD1) 3)) = 4
The top level of LISP is:
      (PROG NIL
       L    (PRINT (EVAL (READ))) (TERPRI) (GO L))
```

! (APPLY FN ARGS)

APPLY evaluates and binds each S-expression in ARGS to the corresponding arguments of the function FN, and returns the value of FN. See 7.2.

Example:

```
     (APPLY (FUNCTION APPEND)(QUOTE ((AB) (CD))))= (A B C D)
```

! (QUOTE "E")

QUOTE returns the S-expression E without evaluating it.

(FUNCTION "FN")

FUNCTION is the same as QUOTE in the interpreter. In the compiler, FUNCTION causes the S-expression FN to be compiled, but QUOTE generates an S-expression constant. See *FUNCTION in 7.2 for the special FUNARG feature.

The following function definitions lack some details but explain the essence of EVAL and APPLY. The A-LIST feature of these functions is not shown, but will be explained in 7.2.

```
     (DE EVAL (X)
         (PROG (Y)
          (RETURN
            (COND ((NUMBERP X) X)
                  ((ATOM X) (COND ((SETQ Y (GET X (QUOTE VALUE)))
                                   (CDR Y))
                                  (T (ERR (QUOTE (UNBOUND VARIABLE )))))
                  ((ATOM (CAR X))
                   (COND ((SETQ Y (GETL (CAR X) (QUOTE EXPR FEXPR MACRO))))
```

```
                                        (COND ((EQ (CAR Y) (QUOTE EXPR))
                                                (APPLY (CADR Y)
                                                       (MAPCAR (FUNCTION EVAL) (CDR X))))
                                              ((EQ (CAR Y) (QUOTE FEXPR))
                                                (APPLY (CADR Y) (LIST (CDR X))))
                                              (T (EVAL (APPLY (CADR Y) (LIST X))))))
                                      ((SETQ Y (GET (CAR X) (QUOTE VALUE)))
                                       (EVAL (CONS (CDR Y) (CDR X))))
                                      (T  (ERR (QUOTE (UNDEFINED FUNCTION ))))))
                                (T (APPLY (CAR X) (MAPCAR (FUNCTION EVAL) (CDR X))))))))

(DE APPLY (FN ARGS)
  (COND ((ATOM FN)
         (COND ((GET FN (QUOTE EXPR))
                 (APPLY (GET FN (QUOTE EXPR)) ARGS))
               (T (APPLY (EVAL FN) ARGS))))
        ((EQ (CAR FN)  (QUOTE LAMBDA))
         (PROG (Z)
               (BIND (CADR FN) ARGS)
               (SETQ Z (EVAL (CADDR FN)))
               (UNBIND (CADR FN))
               (RETURN Z)))
        (T (APPLY (EVAL FN) ARGS))))
```

The functions BIND and UNBIND implement variable bindings as described in the next section.

* 7.1 Variable Bindings

This section attempts to explain the different types of variable bindings and the difference between interpreter and compiler bindings.

* 7.1.1 Bound and Free Occurrences

An occurrence of a variable is a "bound occurrence" if the variable is a variable in any LAMBDA or PROG containing the occurrence so long as the occurrence is not contained in a FUNCTIONAL argument which is contained in the defining LAMBDA or PROG. The defining LAMDA or PROG is the innermost LAMBDA or PROG which contains the variable in its parameter list.

Examples:

```
(LAMBDA (X) (TIMES X Y))
     X has a bound occurrence.
     Y has a free occurrence.
(LAMBDA (Y Z) (MAPCAR (FUNCTION(LAMBDA(X) (CONS X Y)))Z)
     X and Z have only bound occurrences.
     Y has a free occurrence bound by the outer LAMBDA
```

* 7.1.2 Scope of Bindings

A variable bound in a LAMBDA or PROG is defined during the dynamic execution of the LAMBDA or PROG. Free occurrences of variables are defined if and only if either the variable is globally defined or the variable is bound in any LAMBDA or PROG which dynamically contains the free occurrence. A variable is globally defined if and only if it has a value at the top level of LISP. Variables can be globally defined by SETQ at the top level.

* 7.1.3 Special Variables

In compiled functions, any variable which is bound in a LAMBDA or PROG and has a free occurrence elsewhere must be declared SPECIAL (APPENDIX E).

Example:

```
(LAMBDA (A B)
        (MAPCAR (FUNCTION (LAMBDA (X) (CONS A X))) B))
```

The variable A which has a free occurrence must be declared SPECIAL if the outer LAMBDA expression is to be compiled.

* 7.1.4 Binding Mechanisms

All variables in interpreted functions, and SPECIAL variables in compiled functions store their values in SPECIAL (or VALUE) cells. These variables are bound at the entry to a LAMBDA or PROG by saving their previous values on the SPECIAL pushdown list and storing their new values in the SPECIAL cells. All references to these variables are directly to their SPECIAL cells. When the LAMBDA or PROG is exited, the old values are restored from the SPECIAL pushdown list.

In compiled functions, all variables not declared SPECIAL are stored on the REGULAR pushdown list, and the SPECIAL cells (if they exist) are not referenced.

7.2 The A-LIST and FUNARG Features

The A-LIST which is used in some LISP systems to implement recursive variable bindings does not exist here, but its effects are simulated through a special A-LIST feature. The functions EVAL and APPLY allow an extra last argument to be passed which is either a list of paired identifiers and values (like an A-LIST) or a "binding context pointer".

In the case of an A-LIST second argument, EVAL and APPLY will bind the SPECIAL cells of the variables in the A-LIST to their specified values, saving their previous bindings on the special pushdown list. When EVAL and APPLY return, the variable bindings are restored to their previous values.

A "binding context pointer" (BCP) is a pointer into the SPECIAL PUSHDOWN LIST designating a level in recursive variable binding. When EVAL and APPLY receive a BCP as their second argument, all SPECIAL (VALUE) CELLS are restored to the values they had at the time the BCP was generated. This then causes EVAL and APPLY to reference these variables in the binding context which existed at the time of BCP generation. This feature primarily is useful to prevent variable name conflicts when using EVAL, APPLY, and functional arguments. As with the A-LIST, when EVAL and APPLY exit, the previous bindings are restored.

There are two ways to generate a BCP:

If an FEXPR is defined with two arguments, then the second argument will be bound to the SPECIAL PUSHDOWN LIST level at the time the FEXPR is called.

The second way to generate a BCP is with *FUNCTION.

(*FUNCTION "FN")

*FUNCTION returns a list of the following form:

(FUNARG FN · <BCP>)

where BCP is the SPECIAL PUSHDOWN LIST level at the time *FUNCTION is called. Whenever such a functional form is used in functional context, all SPECIAL bindings are restored to the values they had at the time *FUNCTION was evaluated. When the functional argument has been APPLYed, the previous bindings are restored as with the A-LIST.

The use of FUNARGS is discussed further by Robert Saunders[3].

Example using the BCP feature:

```
(DF EXCHANGE (L SPECPDL)
    (PROG(Z) (SETQ Z(EVAL (CAR L) SPECPDL))
             (APPLY (FUNCTION SET)
                    (LIST (CAR L) (EVAL (CADR L) SPECPDL)
                    SPECPDL)
             (APPLY (FUNCTION SET)
                    (LIST (CADR L) Z
                    SPECPDL)))
```

In this example, the use of the extra argument SPECPDL has only one effect: to avoid conflicts between internal and external variables with names L and SPECPDL.

(EXCHANGE L M) will cause the values of L and M to be exchanged. The variable L in EXCHANGE is not referenced by the calls on SET.

CHAPTER 8

CONDITIONAL EXPRESSIONS

A conditional expression has the following form:

```
(COND (e1,1 e1,2 ... e1,n1)
      (e2,1 e2,2 ... e2,n2)
       ...
      (em,1 em,2 ... em,nm))
```

$$(\mathrm{COND}\ (e_{1,1}\ e_{1,2}\ \cdots\ e_{1,n_1})\ (e_{2,1}\ e_{2,2}\ \cdots\ e_{2,n_2})\ \cdots\ (e_{m,1}\ e_{m,2}\ \cdots\ e_{m,n_m}))$$

where the $e_{i,j}$'s are any S-expressions.

The $e_{i,1}$'s are considered to be predicates, i.e., evaluate to a truth value. The $e_{i,1}$'s are evaluated starting with $e_{1,1}$, $e_{2,1}$, etc., until the first $e_{k,1}$ is found whose value is not NIL. Then the corresponding $e_{k,2}\ e_{k,3} \cdots e_{k,n_k}$ are evaluated respectively and the value of e_{k,n_k} is returned as the value of COND. It is permissible for $n_k=1$, in which case the value of $e_{k,1}$ is the value of COND. If all $e_{i,1}$ evaluate to NIL, then NIL is the value of COND.

Examples:

```
(DE NOT (X) (COND (X NIL) (T)))
(DE AND (X Y) (COND (X (COND (Y T)))))
(DE OR (X Y) (COND (X T) (Y T)))
(DE IMPLIES (X Y) (COND (X (COND (Y T)))
                        (T)))
```

CHAPTER 9

PREDICATES

Predicates test S-expressions for particular values, forms, or ranges of values. All predicates described in this chapter return either NIL or T corresponding to the truth values false and true, unless otherwise noted. Some predicates cause error messages or undefined results when applied to S-expressions of the wrong type, such as (MINUSP (QUOTE FOO)).

! (ATOM X)

The value of ATOM is T if X is either an identifier or a number; NIL otherwise.

Examples:

```
(ATOM T)                  = T
(ATOM 1.23)               = T
(ATOM (QUOTE (X Y Z)))    = NIL
(ATOM (CDR (QUOTE (X))))  = T
```

! (EQ X Y)

The value of EQ is T if X and Y are the same pointer, i.e., the same internal address. Identifiers on the OBLIST have unique addresses and therefore EQ will be T if X and Y are the same identifier. EQ will also return T for equivalent INUMs, since they are represented as addresses. However, EQ will not compare equivalent numbers of any other kind. For non-atomic S-expressions, EQ is T if X and Y are the same pointer.

Examples:

```
(EQ T T)              = T
(EQ T NIL)            = NIL
(EQ (QUOTE A)         (QUOTE B))      = NIL
(EQ 1 1.Ø)            = NIL
(EQ 1 1)              = T
(EQ 1.Ø 1.Ø)          = NIL
```

(EQUAL X Y)

The value of EQUAL is T if X and Y are identical S-expressions. EQUAL can also test for equality of numbers of mixed types. EQUAL is equivalent to:

```
(LAMBDA(X Y) (COND ((EQ X Y) T)
                   ((AND (NUMBERP X) (NUMBERP Y))
                    (ZEROP (*DIF X Y)))
                   ((OR (ATOM X) (ATOM Y)) NIL)
                   ((EQUAL (CAR X) (CAR Y))
                    (EQUAL (CDR X) (CDR Y)))))
```

Examples:

```
(EQUAL T T)      = T
(EQUAL 1 1)      = T
(EQUAL 1 1.Ø)    = T
(EQUAL (QUOTE (A B)) (QUOTE (A B)))     = T
(EQUAL (QUOTE (T)) T)     = NIL
```

9.1 S-Expression Predicates

(NULL L) = T iff L is NIL.

(MEMBER L1 L2) = T iff L1 is EQUAL to a top level element of L2.

MEMBER is equivalent to:

```
(LAMBDA(L1 L2)(COND ((ATOM L2) NIL)
                    ((EQUAL L1 (CAR L2))T)
                    (T(MEMBER L1 (CDR L2)))))
```

Examples:

```
(MEMBER (QUOTE (C D))(QUOTE ((A B)(C D)E)))     = T
(MEMBER (QUOTE C)(QUOTE ((C)))) = NIL
```

(MEMQ L1 L2) = T iff L1 is EQ to a top level element of L2.

MEMQ is equivalent to:

```
(LAMBDA(L1 L2)(COND ((ATOM L2) NIL)
                    ((EQ L1 (CAR L2))T)
                    (T(MEMQ L1 (CDR L2)))))
```

Examples:

```
(MEMQ (QUOTE (C D))(QUOTE ((A B)(C D) E))) = NIL
(MEMQ (QUOTE A) (QUOTE (Q A B)))  = T
```

9.2 Numerical Predicates

(NUMBERP X)
= T if X is a number of any type.
= NIL otherwise

(ZEROP X)
= T if X is zero of any numerical type
= error if X is a non-numerical quantity
= NIL otherwise

(MINUSP X)
= T if X is a negative number of any type
= error if X is a non-numerical quantity
= NIL otherwise

(*GREAT X Y) = T if X and Y are numbers of any type and X > Y.
= error if either X or Y is not a number
= NIL otherwise

(*LESS X Y) = (*GREAT Y X)

(GREATERP X_1 X_2 ... X_n) = T if (*GREAT X_1 X_2) and
(*GREAT X_2 X_3) and ...
(*GREAT X_{n-1} X_n)
= error if any X_i is a non-numerical quantity
= NIL otherwise

(LESSP X_1 X_2 ... X_n) = (GREATERP X_n X_{n-1} ... X_1)

Other numerical predicates may be defined as follows:

```
(DE FLOATP (X) (COND ((EQ X (PLUS X Ø))NIL)
                     ((EQ (CADR X) (QUOTE FLONUM)) T)
                     (T NIL)))
(DE FIXP (X) (NOT(FLOATP X)))
(DE ONEP (X) (ZEROP (DIFFERENCE X 1)))
(DE EVENP (X) (ZEROP (REMAINDER X 2)))
```

9.3 Boolean Predicates

The Boolean predicates perform logical operations on the truth values NIL and T. A non-NIL value is considered equal to T.

(NOT X) = T if X is NIL
= NIL otherwise

(AND X_1 X_2 ... X_n) = T if all X_i are non-NIL
= NIL otherwise

Note: (AND)=T. AND evaluates its arguments from left to right until either NIL is found in which case the remaining arguments are not evaluated, or until the last argument is evaluated.

(OR X_1 X_2 ... X_n) = T if any X_i is non-NIL
= NIL otherwise

Note: (OR) = NIL. OR evaluates its arguments from left to right until either non-NIL is found in which case the remaining arguments are not evaluated, or until the last argument is evaluated.

CHAPTER 10

FUNCTIONS ON S-EXPRESSIONS

This chapter describes functions for building, fragmenting, modifying, transforming, mapping, and searching S-expressions, as well as some non-standard functions on S-expressions.

! 10.1 S-Expression Building Functions

(CONS X Y)

The value of CONS of two S-expressions is the dotted pair of those S-expressions.

Example: (CONS (QUOTE A) (QUOTE B)) = (A . B)

Note: See Appendix D for information on functions associated with CONSing, such as SPEAK, GCGAG, and GC.

(XCONS X Y) = (CONS Y X)

(NCONS X) = (CONS X NIL)

(LIST X_1 ... X_n) = (CONS X_1 (CONS X_2 ... (CONS X_n NIL)...))

List evaluates all of its arguments and returns a list of their values.

Examples:
```
(LIST) = NIL
(LIST (QUOTE A)) = A
(LIST (QUOTE A) (QUOTE B)) = (A B)
```

(*APPEND X Y)

```
(DE *APPEND (X Y)
    (COND ((NULL X) Y)
          (T (CONS (CAR X) (*APPEND (CDR X) Y)))))
```

(APPEND X_1 X_2 ... X_n) = (*APPEND X_1 (*APPEND X_2 ... (*APPEND X_n NIL)...))

Example:
```
(APPEND) = NIL
(APPEND (QUOTE (A B)) (QUOTE (C D)) (QUOTE (E F))) = (A B C D E F)
```

! 10.2 S-Expression Fragmenting Functions

(CAR L)

The CAR of a non-atomic S-expression is the first element of that dotted pair. CAR of an atom is undefined and will usually cause an illegal memory reference.

(CDR L)

The CDR of a non-atomic S-expression is the second (and last) element of that dotted pair. The CDR of an identifier is its property list. The CDR of an INUM causes an illegal memory reference. The CDR of any other number is the list structure representation of that number.

Examples:
```
(CAR (QUOTE (A B C))) = A
(CAR (QUOTE A)) is illegal
(CDR (QUOTE (A B C))) = (B C)
(CDR (QUOTE A)) is the property list of A
(CDR (QUOTE (A))) = NIL
```

CAAR, CADR,..., CDDDDR

All of the composite CAR-CDR functions with up to four A's and D's are available.

Examples:
```
(CADR X) = (CAR (CDR X))
(CAADDR X) = (CAR (CAR (CDR (CDR X))))
```

(LAST L)

LAST returns the last part of a list according to the following definition:

```
(DE LAST (L)
    (COND ((ATOM (CDR L)) L)
          (T (LAST (CDR L)))))
```

Examples:
```
(LAST (QUOTE (A B C))) = (C) = (C . NIL)
(LAST (QUOTE (A B . C))) = (B . C)
```

* 10.3 S-Expression Modifying Functions

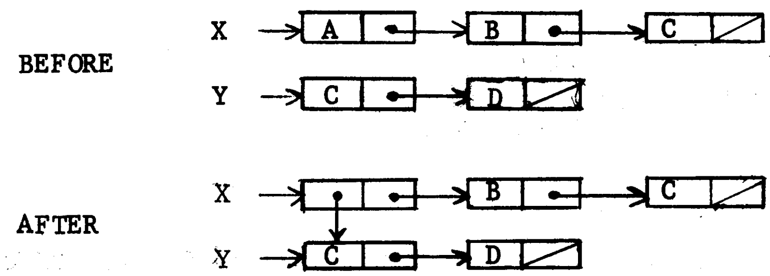
The following functions for manipulating S-expressions differ from all others in that they actually modify existing list structure rather than constructing new list structure. These functions should be used with caution since it is easy to create structures which will confuse or destroy the interpreter.

(RPLACA X Y)

Replaces the CAR of X by Y. The value of RPLACA is the modified S-expression X.

Example: (RPLACA (QUOTE (A B C)) (QUOTE (C D))) = ((C D) B C)

Representation:



(RPLACD X Y)

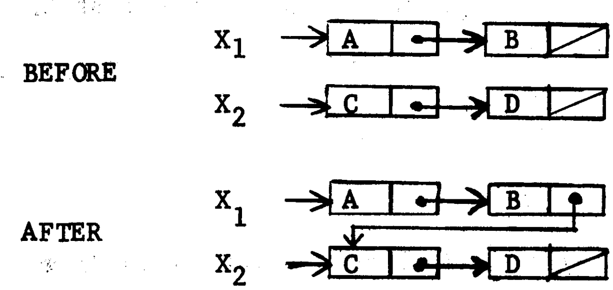
RPLACD replaces the CDR of X by Y. The value of RPLACD is the modified S-expression X.

(NCONC X_1 X_2 ... X_n)

NCONC is similar in effect to APPEND, but NCONC does not copy list structures. NCONC modifies list structures by replacing the last element of X_1 by a pointer to X_2, the last element of X_2 by a pointer to X_3, etc. The value of NCONC is the modified list X_1, which is the concatenation of X_1, X_2, ..., X_n.

Examples: (NCONC) = NIL
(NCONC (QUOTE (A B)) (QUOTE (C D))) = (A B C D)

Representation:



10.4 S-Expression Transforming Functions

The following functions transform S-expressions from one form to another.

(LENGTH L)

LENGTH returns the number of top-level elements of the list L. LENGTH is equivalent to:

```
(DE LENGTH (L)
    (COND ((ATOM L) Ø)
          (T (ADD1 (LENGTH (CDR L))))))
```

(REVERSE L)

REVERSE returns the reverse of the top level of list L. REVERSE is equivalent to:

```
(DE REVERSE (L) (REVERSE1 L NIL))
(DE REVERSE1 (L M)
    (COND ((ATOM L) M)
          (T (REVERSE1 (CDR L) (CONS (CAR L) M)))))
```

(SUBST X Y S)

SUBST substitutes S-expression X for all EQUAL occurrences of S-expression Y in S-expression S. SUBST is equivalent to:

```
(DE SUBST (X Y S)
    (COND ((EQUAL Y S) X)
          ((ATOM S) S)
          (T (CONS (SUBST X Y (CAR S))
                   (SUBST X Y (CDR S))))))
```

Note: (SUBST Ø Ø X) is useful for creating a copy of the list X.

Example: (SUBST 5 (QUOTE FIVE) (QUOTE (FIVE PLUS FIVE IS TEN)))
= (5 PLUS 5 IS TEN)

10.5 S-Expression Mapping Functions

The following functions perform mappings of lists according to the functional arguments supplied.

(MAP FN L)

MAP applies the function FN of one argument to list L and to successive CDRs of L until L is reduced to NIL. The value of MAP is NIL. MAP is equivalent to:

```
(DE MAP (FN L)
    (PROG NIL
     L1   (COND ((NULL L)(RETURN NIL)))
          (FN L)
          (SETQ L (CDR L))
          (GO L1)))
```

Example: (MAP (FUNCTION PRINT) (QUOTE (X Y Z))) =

```
PRINT:      (X Y Z)
PRINT:      (Y Z)
PRINT:      (Z)
RETURN:     NIL
```

(MAPC FN L)

MAPC is identical to MAP except that MAPC applies function FN to the CAR of the remaining list at each step. MAPC is equivalent to:

```
(DE MAPC (FN L)
    (PROG NIL
     L1   (COND ((NULL L) (RETURN NIL)))
                (FN (CAR L))
                (SETQ L (CDR L))
                (GO L1)))
```

Example: (MAPC (FUNCTION PRINT) (QUOTE (X Y Z))) =

```
PRINT:      X
PRINT:      Y
PRINT:      Z
RETURN:     NIL
```

(MAPLIST FN L)

MAPLIST applies the function FN of one argument to list L and to successive CDRs of L until L is reduced to NIL. The value of MAPLIST is the list of values returned by FN. MAPLIST is equivalent to:

```
(DE MAPLIST (FN L)
    (COND ((NULL L) NIL)
    (T (CONS (FN L) (MAPLIST FN (CDR L))))))
```

Examples:

```
(MAPLIST (FUNCTION CAR) (QUOTE (A B C D))) = (A B C D)
(MAPLIST (FUNCTION REVERSE) (QUOTE (A B C D))) =
      ((D C B A) (D C B) (D C) (D))
```

(MAPCAR FN L)

MAPCAR is identical to MAPLIST except that MAPCAR applies FN to the CAR of the remaining list at each step. MAPCAR is equivalent to:

```
(DE MAPCAR (FN L)
    (COND ((NULL L) NIL)
          (T (CONS (FN (CAR L)) (MAPCAR FN (CDR L))))))
```

Examples:

```
(MAPCAR (FUNCTION NCONS) (QUOTE (A B C D))) = ((A) (B) (C) (D))
(MAPCAR (FUNCTION ATOM) (QUOTE ((X) Y (Z)))) = (NIL T NIL)
```

10.6. S-Expression Searching Functions

(ASSOC X L)

ASSOC searches the list of dotted pairs L for a pair whose CAR is EQ to X. If such a pair is found it is returned as the value of ASSOC, otherwise NIL is returned. ASSOC is equivalent to:

```
(DE ASSOC (X L)
    (COND ((NULL L) NIL)
          ((EQ X (CAAR L)) (CAR L))
          (T (ASSOC X (CDR L)))))
```

Example:

```
(ASSOC 1 (QUOTE ((1 . ONE) (2 . TWO)))) = (1 . ONE)
```

(SASSOC X L FN)

SASSOC searches the list of dotted pairs L for a pair whose CAR is EQ to X. If such a pair is found it is returned as the value of ASSOC, otherwise the value of FN, a function of no arguments, is returned.

```
(DE SASSOC (X L FN)
    (COND ((NULL L) (FN))
          ((EQ X (CAAR L)) (CAR L))
          (T (SASSOC X (CDR L) FN))))
```

Example:

```
(SASSOC Ø (QUOTE ((1 . ONE) (2 . TWO)))
          (FUNCTION (LAMBDA NIL (QUOTE LOSE)))) = LOSE
```

10.7 Character List Transforming Functions

(EXPLODE L)

EXPLODE transforms an S-expression into a list of single character identifiers identical to the sequence of characters which would be produced by PRIN1.

Examples:

```
(EXPLODE (QUOTE (DX␣/-␣DY)))
= (/( D X /␣ //  /-  /␣ D Y /))

(EXPLODE (QUOTE APPLE))
= (A P P L E)
```

(EXPLODEC L)

EXPLODEC transforms an S-expression into a list of single character identifiers identical to the sequence of characters which would be produced by PRINC.

Example:

```
(EXPLODEC (QUOTE (DX␣/-␣DY)))
= (/( D X /␣ /- /␣ D Y /))
```

(FLATSIZE L) = (LENGTH (EXPLODE L))

(MAKNAM L)

MAKNAM transforms a list of single character identifiers (actually takes the first character of each identifier) into a S-expression identical to that which would be produced by READing those characters. MAKNAM however does not INTERN any of the identifiers in the S-expression it produces.

Examples:

```
(MAKNAM (QUOTE (A P P L E))) = APPLE
(MAKNAM (QUOTE (//␣/)))) = /)
```

(READLIST L)

READLIST is identical to MAKNAM except that READLIST INTERNs all identifiers in the S-expression it produces. READLIST is the logical inverse of EXPLODE, i.e.,

```
(READLIST (EXPLODE L)) = L
(EXPLODE (READLIST L)) = L
```

CHAPTER 11

FUNCTIONS ON IDENTIFIERS

There are three basic types of functions on identifiers: those which manipulate their property lists, those which create new identifiers, and those which control their membership in the OBLIST.

NOTE: All functions described in this chapter which expect an identifier as one (or more) of its arguments will give either erroneous results, or an error condition if any S-expression other than an identifier is supplied.

11.1 Property List Functions

(GET I P)

GET is a function which searches the property list of the identifier I looking for the property name which is EQ to P. If such a property name is found, the value associated with it is returned as the value of GET, otherwise NIL is returned. Note that confusion exists if the property is found, but its value is NIL. GET is equivalent to:

```
(DE GET (I P) (COND((NULL (CDR I)) NIL)
                   ((EQ (CADR I) P) (CADDR I))
                   (T (GET (CDDR I) P))))
```

(GETL I L)

GETL is another function which searches property lists. GETL searches the property list of the identifier I looking for the first property which is a member (MEMQ) of the list L. GETL returns the remaining property list, including the property name if any such property was found, NIL otherwise. GETL is equivalent to:

```
(DE GETL (I L) (COND ((NULL (CDR I)) NIL)
                     ((MEMQ (CADR I) L) (CDR I))
                     (T (GETL (CDDR I) L))))
```

(PUTPROP I V P)

PUTPROP is a function which enters the property name P with property value V into the property list of identifier J. If the property name P is already in the property list, the old property value is replaced by the new one; otherwise the new property name P and its value V are placed on the beginning of the property list. PUTPROP returns V.

(REMPROP I P)

REMPROP removes the property P from the property list of identifier I. REMPROP returns T if there was such a property, NIL otherwise.

SET and SETQ are used to change the values of variables which are bound by either LAMBDA or PROG, or variables which are bound globally. (See 7.1).

* (SET E V)

SET changes the value of the identifier specified by the expression E to V and returns to V. Both arguments are evaluated.

Note: In compiled functions, SET can be used only on globally bound and special variables.

! (SETQ "ID" V)

SETQ changes the value of ID to V and returns V. SETQ evaluates V, but does not evaluate ID.

(DEFPROP "I" "V" "P") = (PROG2 (PUTPROP (QUOTE I) (QUOTE V) (QUOTE P))(QUOTE I))

DEFPROP is the same as PUTPROP except that it does not evaluate its arguments, and DEFPROP returns I.

Example: (DEFPROP POSP (LAMBDA (X) (GREATERP X Ø)) EXPR)

(DE "ID" "ARGS" "BODY")

DE places the form (LAMBDA ARGS BODY) on the property list of ID under property EXPR. If ID previously had any of the properties EXPR, FEXPR, SUBR, FSUBR, LSUBR, or MACRO, then DE will return the list (ID REDEFINED). Otherwise, DE returns ID.

(DF "ID" "ARGS" "BODY")

Same as DE except defines a function with FEXPR property.

(DM "ID" "ARGS" "BODY")

Same as DE except defines a MACRO.

11.2 OBLIST Functions

(INTERN I)

INTERN puts the identifier I in the appropriate bucket of OBLIST. If the identifier is already a member of the OBLIST, then INTERN returns a pointer to the identifier already there. Otherwise, INTERN returns I.

Note: INTERN is only necessary when an identifier which was created by GENSYM, MAKNAM, or ASCII needs to be uniquely stored.

(REMOB "X_1" "X_2" ... "X_n")

REMOB removes all of the identifiers X_1, X_2, ... , X_n from the OBLIST and returns NIL. None of the X_i's are evaluated.

Example: (REMOB FOO BAZ)

11.3 Identifier Creating Functions

The following functions create new identifiers but do not INTERN them onto the OBLIST.

(GENSYM)

GENSYM increments the generated symbol counter and returns a new identifier specified by the counter. The GENSYM counter is initialized to the identifier GØØØØ. Successive executions of (GENSYM) will return.

GØØØ1, GØØØ2, GØØØ3, ...

(CSYM "I")

CSYM initializes generated symbol counter to the identifier I, and returns I. CSYM does not evaluate its argument.

```
Example:    (CSYM ARYØØ)  = ARYØØ
            (GENSYM)      = ARYØ1
            (GENSYM)      = ARYØ2
               etc.
```

(ASCII N)

ASCII creates a single character identifier whose ASCII print name equals N.

Example: (ASCII 1Ø1) is an identifier with print name "A".

CHAPTER 12

FUNCTIONS ON NUMBERS

There are two types of functions which operate on numbers to create new numbers: arithmetic and logical.

! 12.1 Arithmetic Functions

Unless otherwise noted, the following arithmetic functions are defined for both integer, real and mixed combinations of arguments, and evaluate all their arguments. The result is real if any argument is real, and integer if all arguments are integer. Most arithmetic functions may cause overflow which produces an error message.

(MINUS X)	= -X
(*PLUS X Y)	= X + Y
(PLUS X1 X2 ... Xn)	= X1 + X2 + ... + Xn
(*DIF X Y)	= X - Y
(DIFFERENCE X1 X2 ... Xn)	= X1 - X2 - ... - Xn
(*TIMES X Y)	= X * Y
(TIMES X1 X2 ... Xn)	= X1 * X2 * ... * Xn
(*QUO X Y)	= X / Y
(QUOTIENT X1 X2 ... Xn)	= X1 / X2 / ... / Xn

For integer arguments, *QUO and QUOTIENT give the integer part of the real quotient of the arguments.

Examples: (*QUO 5 2) = 2
(*QUO -5 2) = -2

(REMAINDER X Y) ... X - (X / Y) * Y

Note: Remainder is not defined for real arguments.

(DIVIDE X Y) = (CONS (QUOTIENT X Y) (REMAINDER X Y))

(GCD X Y) GCD returns the greatest common divisor of the integers X and Y.

(ADD1 X) = X + 1
(SUB1 X) = X - 1
(ABS X) = | X |

(FIX X) returns the largest integer not greater than X.

Examples:

```
(FIX 1) = 1
(FIX 1.1) = 1
(FIX -1.1) = -2        not -1
```

Other arithmetic functions not defined in the LISP interpreter can be defined as follows:

```
(FLOAT X) = (*PLUS X Ø.Ø)

(RECIP X) = (QUOTIENT 1 X)

(SIGN X) = (COND ((ZEROP X) Ø)
                 ((MINUSP X) -1)
                 (T 1))

(ROUND X) = (TIMES (SIGN X) (FIX (PLUS (ABS X) Ø.5)))
```

Examples:

```
(ROUND .5) = 1
(ROUND .49) = Ø
(ROUND -.49) = Ø
(ROUND -35.1) = -35
```

```
(MIN X Y) = (COND((LESSP X Y)  X) (Y))

(MAX X Y) = (COND((LESSP X Y)  Y)(X))
```

Examples:

```
(MINUS 1)             = -1
(MINUS -1.2)          = 1.2
(PLUS 1 2 3.1)        = 6.1
(PLUS 6 3 -2)         = 7
(DIFFERENCE 6 3 1)    = 2
(TIMES -2 2.Ø)        = -4.Ø
(QUOTIENT 5 2)        = 2
(QUOTIENT 5.Ø 2)      = 2.5
(QUOTIENT -5 2)       = 2
(REMAINDER 5 2)       = 1
(REMAINDER -5 2)      = -1
(REMAINDER 5.Ø 2)     = undefined.
(ABS -32.5)           = 32.5
(FIX 32.5)            = 32.
(FIX -32.5)           = -33.
```

12.2 Logical Functions.

The following functions are intended to operate on INUM and FIXNUM arguments, but their results are not defined for BIGNUM or FLONUM (real) arguments.

(BOOLE N X1 X2 ... Xn)

BOOLE causes a 36 bit Boolean operation to be performed on its arguments. The value of N specifies which of 16 Boolean operations to perform.

For n=2, each bit_i in (BOOLE N A B) is defined:

N	result	N	result
0	0	10	$\overline{A}_i \wedge \overline{B}_i$
1	$A_i \wedge B_i$	11	$A_i \equiv B_i$
2	$\overline{A}_i \wedge B_i$	12	$\overline{A}_i$
3	B_i	13	$\overline{A}_i \vee B_i$
4	$A_i \wedge \overline{B}_i$	14	$\overline{B}_i$
5	A_i	15	$A_i \vee \overline{B}_i$
6	$A_i \neq B_i$	16	$\overline{A}_i \vee B_i$
7	$A_i \vee B_i$	17	1

For n > 2, BOOLE is defined:

(BOOLE N ... (BOOLE N (BOOLE N X1 X2) X3) ... Xn)

(LSH X N)

LSH performs a logical left shift of N places on X. If n is negative, X will be shifted right. In both cases, vacated bits are filled with zeros.

Examples with IBASE = 8

```
(BOOLE 1 76 133)        = 32
(BOOLE 1 76 133 70)     = 30
(BOOLE 12 13 0)         = 777777777764
(BOOLE 7 7 12)          = 17
(LSH 15 2)              = 64
(LSH 15 -2)             = 3
(LSH -1 -2)             = 177777777777
```

CHAPTER 13

PROGRAMS

The "program feature" allows one to write ALGOL-like sequences of statements with program variables and labels.

(PROG "VARLIST" "BODY")

PROG is a function which takes as arguments VARLIST, a list of program variables which are initialized to NIL when the PROG is entered (see 7.1), and a BODY which is a list of labels (which are identifiers) and statements which are non-atomic S-expressions. PROG evaluates its statements in sequence until either a RETURN or GO is evaluated, or the list of statements is exhausted, in which case the value of PROG is NIL.

(RETURN X)

RETURN causes the PROG containing it to be exited with the value X.

(GO "ID")

GO causes the sequence of control within a PROG to be transferred to the next statement following the label ID. In interpreted PROGs, if ID is non-atomic, it is repeatedly evaluated until an atomic value is found. However, in compiled PROGs, ID is evaluated only once. GO cannot transfer into or out of a PROG.

Note: Both RETURN and GO should only occur either at the top level of a PROG, or in compositions of COND, AND, OR, and NOT which are at the top level of a PROG.

Example: The function LENGTH may be defined as follows:

```
(DE LENGTH (L)
    (PROG (N)
          (SETQ N Ø)
     L1   (COND ((ATOM L) (RETURN N)))
          (SETQ N (ADD1 N))
          (SETQ L (CDR L))
          (GO L1)))
```

(PROG2 X_1 X_2 ... X_n) , $n \leq 5$.

PROG2 evaluates all expressions X_1 X_2 ... X_n, and returns the value of X_2.

CHAPTER 14

INPUT/OUTPUT

14.1 File Names

Syntax:

<filename - list>	::=	<device-name>
	::=	<filename - list> <device-name>
	::=	<filename - list> <file - name>
<device - name>	::=	<identifier> :
	::=	(<atom> <atom>)
<filename>	::=	<identifier>
	::=	(<identifier> · <identifier>)

Semantics:

A device-name is either an identifier ending with colon (:) which is the name of some input or output device, or a list containing a project-programmer number which implicitly specifies the disk.

A filename is either an identifier which specifies a filename with a blank extension, or a dotted pair of filename and extension. In both cases the filename applies to the most recently (to the left) specified device-name.

14.2 Channel Names

Channel names can optionally be assigned to files selected by the functions INPUT and OUTPUT. A channel name is any identifier which is not followed by a colon. If no channel name is specified to INPUT or OUTPUT then the channel name T is assumed. The channel name NIL specifies the teletype in the functions INC and OUTC. Up to 14 channels may be active at any time.

14.3 Input

14.3.1 Selection and Control

(INPUT "CHANNEL" . "FILENAME-LIST")

INPUT releases any file previously initialized on the channel, and initializes for input the first file specified by the filename-list. INPUT returns the channel if one was specified, T otherwise. INPUT does not evaluate its arguments.

(INC CHANNEL ACTION)

INC selects the specified channel for input. The channel NIL selects the teletype. If ACTION = NIL then the previously selected input file is not released, but only deselected. If ACTION = T then that file is released, making the previously selected channel available. At the top level, ACTION need not be specified.

The input functions in 14.3.3 receive input from the selected input channel. When a file on the selected channel is exhausted, then the next file in the filename-list for the channel is initialized and input, until the filename-list is exhausted. Then the teletype is automatically selected for input and (ERR (QUOTE EOF))is called. The use of ERRSET around any functions which accept input therefore makes it possible to detect end of file. If no ERRSET is used, control returns to the top level of LISP. INC evaluates its arguments, and returns the previously selected channel name.

In order to READ from multiple input sources, separate channels should be initialized by INPUT, and INC can then select the appropriate channel to READ from.

Examples: (At the top level)

(INC (INPUT SYS: (SMILE . LSP)))

will READ the file SYS: SMILE . LSP on channel T and reselect the teletype when the file is ended.

(INC (INPUT FOO DSK: BAZ ZAB))

will READ the files DSK: BAZ and DSK: ZAB on channel FOO and reselect the teletype after both files are exhausted.

(PGLINE)

When reading an input file, it is sometimes desirable to know the page and line being read from. PGLINE returns the dotted pair (page number . line number) for the selected input file. The page number is applicable only to STOPGAP files. If the file has no line numbers, PGLINE will always return (1 . Ø).

14.3.2 Teletype Input Control

When input is from the teletype, READ is terminated by either an entire S-expression or by an incomplete S-expression followed by altmode. Altmode has the effect of typing a space followed by the appropriate number of right parens to complete the S-expression. This feature is particularly useful when an unknown number of right parens are needed or when in (DDTIN NIL) mode.

* (DDTIN X)

DDTIN is a function which selects teletype input mode. With (DDTIN NIL), and typing to READ, READCH, or TYI, a rubout will delete the last character typed, and control U (↑U) will delete the entire last line typed. Input is not seen by LISP until either altmode or carriage return is typed.

With (DDTIN T) and typing to READ, a rubout will delete the entire S-expression being read and start reading again.

Note: (DDTIN T) is not recommended when the time-sharing system is swapping, since the program is reactivated (and hence swapped into core) after every character typed.

14.3.3 Input Transfer

! (READ)

Read causes the next S-expression to be read from the selected input device, and returns the internal representation of the S-expression. READ uses INTERN to guarantee that references to the same identifier are EQ.

READ will accept any S-expression which conforms to the following syntax:

Syntax:

```
<readable S-expr>  ::=  <atom>
                   ::=  @<readable S-expr>
                   ::=  (<readable S-expr list> {. <readable S-expr>})
                   ::=  [<unbalanced S-expression list>]
                   ::=  ( )                       = NIL

<readable S-expr list>  ::= <readable S-expr>
                        ::= <readable S-expr> <readable S-expr list>
```

Semantics:

The delimiter "@" designates that the following readable S-expr is to be quoted.

Examples:

```
@A          means (QUOTE A)
@(@A B)     means (QUOTE ((QUOTE A) B))
@@A         means (QUOTE (QUOTE A)))
```

The delimiters "[" and "]" operate as "super-parentheses". A right bracket "]" will close all open left parentheses "(" up to the matching left bracket "[". If there is no matching left bracket, it will close the entire S-expression as does altmode. No syntax is given for unbalanced-S-expression-list, but it is intended to mean an S-expression-list which is lacking one or more right parentheses.

Example:

```
(COND [(ATOM X) (REVERSE(CDR Y][T(APPEND Y Z])
(COND ((ATOM X) (REVERSE(CDR Y)))(T(APPEND Y Z)))
```

(READCH)

READCH causes the next character to be read from the selected input device and returns the corresponding single character identifier. READCH also uses INTERN.

(TYI)

TYI causes the next character to be read from the selected input device and returns the ASCII code for that character.

A function TEREAD which ignores all characters until a line-feed is seen can be defined:

```
(DE TEREAD NIL
    (PROG NIL
     L    (COND ((EQ (TYI) 12) (RETURN NIL)))
          (GO L)))
```

14.4 Output

14.4.1 Selection and Control

(OUTPUT "CHANNEL" . "FILENAME-LIST")

OUTPUT initializes for output on the specified channel the single file specified by the filename-list. OUTPUT does not evaluate its arguments, and returns the channel name if specified, T otherwise.

(OUTC CHANNEL ACTION)

OUTC selects the specified channel for output. The channel NIL selects the teletype. The output functions in 14.4.3 transfer output to the selected output channel.

If ACTION = NIL, then the previously selected output file is not closed, but only deselected. If ACTION = T then that file is closed, i.e., an end of file is written. OUTC evaluates its arguments and returns the previously selected channel name. At the top level, ACTION need not be specified.

Examples: (At the top level)

```
(OUTC (OUTPUT LPT:) T)
(OUTC NIL T)
(OUTPUT FOO DSK: BAZ)
(OUTC (QUOTE FOO) NIL)
```

(LINELENGTH N)

LINELENGTH is used to examine or change the maximum output linelength on the selected output channel. If N = NIL then the current linelength is returned unchanged, otherwise the linelength is changed to the value of N which is returned and must be an integer.

(CHRCT)

CHRCT returns the number of character positions remaining on the output line of the selected output channel.

When characters are output, if CHRCT is made negative, an ASCII 176 followed by a carriage-return and a line-feed are output. These characters are completely ignored on input. (See Chapter 3).

14.4.2 Teletype Output Control

Output to the teletype is accumulated in a buffer until some condition causes the buffer to be printed (FORCE). The buffer is always printed when a teletype input is requested or when the buffer is full. The following functions determine other conditions for printing the buffer.

* (DDTOUT X)

DDTOUT selects the teletype output mode. (DDTOUT T) causes the teletype output buffer to be printed after every character. (DDTOUT NIL) is the normal mode. DDTOUT returns T or NIL according to the previously selected mode.

* (FORCE)

FORCE is sometimes useful for output to the teletype when in (DDTOUT NIL) mode. FORCE causes the teletype output buffer to be printed. This allows one to see output during long computations which would otherwise be buffered until the computation was finished or until the buffer was full.

14.4.3 Output Transfer

! (PRIN1 S)

PRIN1 causes the S-expression S to be printed on the selected output device with no preceding or following spaces. PRIN1 also inserts slashes ("/") before any characters in identifiers which would be syntactically incorrect otherwise (see Chapter 3). Double quotes around strings are printed.

(PRINC S)

PRINC is the same as PRIN1 except that no slashes are inserted and double quotes around strings are not printed.

(TERPRI X)

TERPRI prints a carriage-return and line-feed and returns the value of X. X may be omitted if the value of TERPRI is not used.

Example: (PRINC(TERPRI X))

is the same as

```
(PROG2 (TERPRI) (PRINC X))
```

! (PRINT S)

```
= (PROG2 (TERPRI)
         (PRIN1 S)
         (PRINC (QUOTE /␣)))
```

(TYO N)

TYO prints the character whose ASCII value is N, and returns N.

CHAPTER 15

ARRAYS

(ARRAY "ID" TYPE B_1 B_2 ... B_n) $n \leq 5$.

ARRAY is a function which declares an array with name ID, and places an array referencing function on the property list of ID. TYPE determines the type of an array as follows:

TYPE	INITIAL VALUE	ARRAY ELEMENT
T	NIL	LISP S-expressions stored as pointers 2 per word
NIL	Ø.Ø	REAL numbers stored one per word in PDP-6/1Ø floating point representation.
36.	Ø	36 bit 2/s complement integers stored 1 per word.
Ø<n<36.	Ø	n bit positive integers packed [36./n] per word.

B_1 B_2 ... B_n are array subscript bounds which should evaluate to either positive integers S_i, or to dotted pairs of integers (L_i . U_i) where $L_i \leq U_i$, which specify lower and upper subscript bounds as follows:

B_i	LOWER BOUND	UPPER BOUND	LENGTH
S_i	Ø	S_i-1	S_i
(L_i . U_i)	L_i	U_i	U_i-L_i+1

The elements of an array are referenced by:

(<array name> i_1 i_2 ... i_n) where $L_j \leq i_j \leq U_j$.

The ARRAY subscripts i_j must be integers. References to memory locations outside of the area reserved for the array are prohibited and will cause an illegal memory reference message. Array elements are stored in BINARY PROGRAM SPACE.

Examples:

1) To declare a 1 dimensional array CHARS of 7 bit characters and with subscripts 1 to 50:

```
(ARRAY CHARS 7 (QUOTE (1 . 50)))
```

The first element of CHARS is referenced:

(CHARS 1)

2) To declare a 2-dimensional array A of REAL numbers and with subscripts $\emptyset \leq i < N$, $\emptyset \leq j < M$:

(ARRAY A NIL N M)

3) To declare a 1-dimensional array FOO of S-expressions and with subscripts $-K \leq i \leq K$:

(ARRAY FOO T (CONS (MINUS K) K))

* (EXARRAY "ID" TYPE B_1 $B_2 \ldots B_n$) $n \leq 5$.

EXARRAY is identical to ARRAY except that array elements are stored in the body of a subroutine loaded by the LOADER (see Appendix H), and exarray elements are not initialized. The array referencing subroutine is stored in BINARY PROGRAM SPACE as with ARRAY. EXARRAY searches symbol tables as does GETSYM (see Appendix H).

Note: Both ARRAY and EXARRAY consume BINARY PROGRAM SPACE. If there is insufficient room there (see Appendix C) the error message "BINARY PROGRAM SPACE EXCEEDED" will result.

(STORE ("ID" i_1 i_2 ... i_n) value)

STORE changes the value of the specified array element to value, and returns value.

Note: STORE evaluates its second argument first.

Examples: With the arrays declared previously:

```
(STORE (FOO ∅) (QUOTE (A B)))
(STORE (FOO (BAZ L)) 1)
(STORE (A I J) (A J I))
(STORE (CHARS 1) 17)
```

15.1 Examine and Deposit

(EXAMINE N)

EXAMINE returns as an integer the contents of memory location N.

(DEPOSIT N V)

DEPOSIT stores the integer V in memory location N and returns V.

CHAPTER 16

OTHER FUNCTIONS

(TIME)

TIME returns the number of milliseconds your job has computed since you logged into the system.

(ERRSET E "F")

ERRSET evaluates the S-expression E and if no error occurs during its evaluation, ERRSET returns (LIST E). If an error occurs, then the error message will be suppressed if and only if F ≠ NIL, and NIL is returned as the value of ERRSET. If the function ERR is called during evaluation, then no message is printed and ERRSET returns the value returned by ERR.

The following example shows the use of ERRSET to keep trying to initialize the line printer until it is available:

```
(DE LPTGRAB NIL
    (PROG NIL
    L (COND((ATOM(ERRSET (OUTPUT LPT:) T))
            (WAIT) (GO L)))))
```

where WAIT is some function (such as the time-sharing sleep UUO) which causes a delay.

(ERR E)

ERR returns the value of E to the most recent ERRSET, or to the top level of LISP if there is no ERRSET.

(*RSET X) flag = NIL initially

*RSET sets a special flag in the interpreter to the value of X, and returns the previous value of the flag. Normally, with (*RSET NIL), when an error occurs, special variables are restored to their top level values from the special pushdown list, and the top level READ-EVAL-PRINT loop is entered.

With (*RSET T), specials are not restored, neither pushdown list is changed, and the READ-EVAL-PRINT loop is entered. This makes it possible to examine the variable bindings immediately after an error message has been printed. To restore special bindings to their top level values and return to the top level, type a bell (π), or evaluate (ERR).

(BAKGAG X)

BAKGAG sets a special flag in the interpreter to the value of X and returns the previous setting of the flag. Only if the flag ≠ NIL when an error occurs, then a backtrace is printed as a series of function calls, determined from the regular pushdown list, starting from the most recent function call. If X is an integer, then X specifies the number of regular pushdown list words to include the backtrace. If X is T then the entire regular pushdown list is backtraced to the most recent ERRSET. The format for printing is:

printout	meaning
fn1-fn2	Function 1 called function 2.
fn1 - EVALARGS	The arguments to fn1 are being evaluated before entering function 1.
fn1 - ENTER	The function 1 is entered.
? - fn1	Some internal LISP function called function 1.

Note:

The BACKTRACE printout is often confused by compiled function calls of the form (RETURN (FOO X)) which is compiled as (JCALL (E FOO)) which can be changed to (JRST entrance to FOO), which will not show up in the BACKTRACE.

(INITFN FN)

INITFN selects the function of no arguments FN as an initialization function which is evaluated after a LISP error return to the top level has occurred or whenever a BELL is typed. INITFN returns the previously selected initialization function.

Initialization functions are useful when it is desirable to change the top level of LISP. For instance,

(INITFN (FUNCTION EVALQUOTE))

causes the top level to become EVALQUOTE instead of EVAL.

APPENDIX A

ALVINE

by John Allen

ALVINE is a LISP editor which is very convenient for interactive debugging. ALVINE allows one to edit both function definitions and S-expression values. ALVINE is characterized by the simplicity with which one can correct parenthesis mismatch and make context searches and replacements. This simplicity arises from the data structure ALVINE uses to represent a S-expression. All S-expressions are flattened into a list of atoms including the atoms %LP, %RP and %D which represent "(",")" and "." . Because of this representation, ALVINE looks more like a string type text editor with the smallest unit of resolution being a single atom or S-expression delimiter (%LP %RP or %D).

ALVINE has a pointer which can move through the string being edited. The editing functions affect only the string to the right of the pointer.

ALVINE also contains functions for manipulating input-output files, and GRINDEF which is useful for printing function definitions.

ALVINE is not ordinarily a resident part of the LISP system, but is automatically loaded whenever the function ED is called.

(ED X)

ED loads ALVINE if it is not already loaded. If X = NIL then the editor is entered. IF X = T, the editor is not entered. This is useful to load GRINDEF without entering the editor.

From the top level of LISP, (ED) is the same as (ED NIL).

(SPRINT X Y 1)

SPRINT prints S-expression X in a special format which automatically indents according to parenthesis level. Whenever any sub-S-expression of X cannot fit entirely on the same printing line then its sub-S-expressions are printed on separate lines with matching indentation. The parameter Y specifies the initial left hand column indentation. SRRINT uses CHRCT and LINELENGTH to determine the number of characters remaining on the print line.

(GRINDEF "F1" "F2" ... "Fn")

GRINDEF is used to print the definitions of functions and values in DEFPROP format. GRINDEF uses SPRINT to print function definitions in a highly readable format. GRINDEF prints all properties of the identifiers F1, F2, ... Fn which are in the list %%%L which is initialized to EXPR, FEXPR, VALUE, MACRO and SPECIAL.

Example: (GRINDEF PLUS)

```
(DEFPROP PLUS
 (LAMBDA (L) (*EXPAND L (QUOTE *PLUS)))
MACRO)
```

Description of the Command Structure

Each command to ALVINE consists of a single character (possibly preceeded by a number) followed by a string of arguments. These commands modify the text string presently occupying ALVINE's buffer. When text is introduced to ALVINE a pointer is attached preceeding the first object in the buffer. ALVINE'S commands allow the user to move this pointer through the buffer. ALVINE'S text modifying commands only affect the string to the right of this pointer.

In the following command descriptions, "pointer string" will mean the string to the right of the pointer, and "$" means an altmode. All of the commands which allow a repetition argument n assume 1 if n is omitted.

COMMAND	MEANING	DESCRIPTION
A	All	Print the buffer string. No attempt is made to make the output pretty.
B	Balanced?	Examines the number of parens in the buffer string. Returns the count of left and right parens if unbalanced; otherwise replies "BAL".
nC	Count	For readability, the commands "D", "M", ">", "<", "S", and "W", will print an initial segment of the pointer buffer. "nC" sets the length of this printing segment to n objects.
nD	Delete	Deletes the first n "objects" to the right of the pointer.
nE	Expunge	Deletes the first n S-expressions to the right of the pointer.
F x y:z	File	GRINDEFS the identifiers referred to by x on device y using file name z. If X is a list of identifiers then each element of x is GRINDEFed. If x is an atom, then the value of x is used as a list of atoms to GRINDEF.

COMMAND	MEANING	DESCRIPTION
Gx	Get	G will convert an S-expression with name x into ALVINE form, move it into the ALVINE buffer and initialize the pointer to the left hand end of the buffer. GET looks on the property list of x for the first property in the list %%%L which is initialized to (FEXPR EXPR VALUE MACRO SPECIAL). %%%L may be SETQed globally as desired. G also knows about TRACEd functions and will handle them properly. ALVINE format was described earlier as a single level list of atoms including the special atoms %LP, %RP, %D.
I	Insert	Insert comes in two flavors: 1. Ix: insert "x" immediately to the right of the pointer. 2. Ixy: insert "y" after the first occurrence (in the pointer string) of the string "x". "x" may be a complete string or described by ellipsis as "w ... z". If "x" is % then "y" is introduced to the editor as the current string.
nM	Match	Move the pointer n S-expressions to the right of the current pointer position. If n is negative, move n S-expressions left. If there is no such S-expression the pointer is not moved and the bell is sounded.
Px	Put	Converts the editor string from ALVINE format to an S-expression and puts it back on the property list of x with the appropriate property.
Qx	<follows P>	Same as P except no function name need be specified. Q puts the S-expression in the editor buffer back on the property list of the identifier the last G specified.

COMMENT	MEANING	DESCRIPTION
Rxy	Replace	Replace the first occurrence of "x" by "y". As with "I", "x" may be described elliptically; and if "y" is %, the first occurrence of "x" is deleted.
nSx$	Search	Search for the n^{th} occurrence of the string "x" (in the pointer string). If found, the pointer is moved to the beginning of the string following that occurrence. If less the n occurrences are located, the pointer is positioned after the last such occurrence. If none are found the pointer is not moved. If "x" is not given, i.e., "nS$", then the last given search-string is used.
Uxy:z	Unfile	READs and defines the functions specified by the list x from device y using z as a file name. If x is an atom, then the value of x is used as a list function names to READ. U prints the names of the functions as it defines them. The specified file must be in GRINDEF format.
V	Vomit	Print the first balanced paren section to the right of the point in pseudo GRINDEF format.
W	Where?	Prints the beginning of the pointer string.
n> and n<		These commands are dual; they move the string pointer "n" objects to the right or left respectively. If "n" is such that either the left or right end of the string would be exceeded, the pointer is set to that extreme and "bell" is typed. To reset to the extreme left of the string "ϕ<" may be used.
↑		This command returns control to LISP. ALVINE'S buffer is left intact, and returning to ALVINE, the user will find the pointer at the left hand end of the old string.

COMMAND	MEANING	DESCRIPTION
Gx	Get	G will convert an S-expression with name x into ALVINE form, move it into the ALVINE buffer and initialize the pointer to the left hand end of the buffer. GET looks on the property list of x for the first property in the list %%%L which is initialized to (FEXPR EXPR VALUE MACRO SPECIAL). %%%L may be SETQed globally as desired. G also knows about TRACEd functions and will handle them properly. ALVINE format was described earlier as a single level list of atoms including the special atoms %LP, %RP, %D.
I	Insert	Insert comes in two flavors: 1. Ix: insert "x" immediately to the right of the pointer. 2. Ixy: insert "y" after the first occurrence (in the pointer string) of the string "x". "x" may be a complete string or described by ellipsis as "w ... z". If "x" is % then "y" is introduced to the editor as the current string.
nM	Match	Move the pointer n S-expressions to the right of the current pointer position. If n is negative, move n S-expressions left. If there is no such S-expression the pointer is not moved and the bell is sounded.
Px	Put	Converts the editor string from ALVINE format to an S-expression and puts it back on the property list of x with the appropriate property.
Qx	<follows P>	Same as P except no function name need be specified. Q puts the S-expression in the editor buffer back on the property list of the identifier the last G specified.

COMMENT	MEANING	DESCRIPTION
Rxy	Replace	Replace the first occurrence of "x" by "y". As with "I", "x" may be described elliptically; and if "y" is %, the first occurrence of "x" is deleted.
nSx$	Search	Search for the n^{th} occurrence of the string "x" (in the pointer string). If found, the pointer is moved to the beginning of the string following that occurrence. If less the n occurrences are located, the pointer is positioned after the last such occurrence. If none are found the pointer is not moved. If "x" is not given, i.e., "nS$", then the last given search-string is used.
Uxy:z	Unfile	READs and defines the functions specified by the list x from device y using z as a file name. If x is an atom, then the value of x is used as a list function names to READ. U prints the names of the functions as it defines them. The specified file must be in GRINDEF format.
V	Vomit	Print the first balanced paren section to the right of the point in pseudo GRINDEF format.
W	Where?	Prints the beginning of the pointer string.
n> and n<		These commands are dual; they move the string pointer "n" objects to the right or left respectively. If "n" is such that either the left or right end of the string would be exceeded, the pointer is set to that extreme and "bell" is typed. To reset to the extreme left of the string "ϕ<" may be used.
↑		This command returns control to LISP. ALVINE'S buffer is left intact, and returning to ALVINE, the user will find the pointer at the left hand end of the old string.

COMMAND	MEANING	DESCRIPTION
Bell		Bell may be used during any command to return control to ALVINE'S command-listen-loop.

AN EXAMPLE OF ALVINE

Note: 1. All typeout is underlined.

2. Bell, space and alt-mode are represented by π, ␣ and $ respectively.

```
(ED)

*
I % $(DEFPROP TEST(LAMBDA $; the string bounded by "$" is introduced
                                 to ALVINE


*
A$
(DEFPROP TEST(LAMBDA ; print the entire ALVINE buffer
*
B$
2LPS
Ø RPS
*
I LAMBDA $ (X) (CAR Y) EXPR) $; append the string bounded by "$" to
                                 the buffer


*
B$
4LPS
3 RPS
*
I CAR Y $)$ ; add the deficient right paren
*
                                  ; the following commands would also
                                    have the same effect:
                                  ; 1.  "11<", "I $)$"
                                  ; 2.  "SY$", I $)$

B$
BAL ;
*
V$
(DEFPROP TEST (LAMBDA (X) (CAR Y))EXPR)
*
P TEST$; convert ALVINE string to LISP function
*
```

```
↑  ;  exit ALVINE
T$ ;  now talking to LISP
T
(TEST (QUOTE(A B)))
Y
UNBOUND VARIABLE-EVAL  ; LOSE
(ED)       ; reenter ALVINE,
*
W $
(DEFPROP TEST ; "G" need not be executed since the buffer is always
*                                       left intact.
R X $ Y$
*
P TSπ  ; flush incorrect "put" command by typing bell. (π)
*
PTEST $  ; redefine TEST
*
↑
(TEST (QUOTE(A B))) $ ; try again
A                     ; win
(ED)
*
5> $
(Y)
*
5C $ ; change print count
*
M $
(CAR Y))
*
E $
*
A $
(DEFPROP TEST (LAMBDA (Y))EXPR)
*
W $
)EXPR)
*
0< $
π(DEFPROP TEST(
*
R TEST ...) $%$
*
A $
(DEFPROP)EXPR)  ; same effect by :
                  1.  "SDEFPROP $", "6D"
*
```

APPENDIX B.

ERROR MESSAGES

The LISP interpreter checks for some error conditions and prints messages accordingly. Many erroneous conditions are not tested and result in either the wrong error message at some later time, or no error message at all. In the latter case the system has screwed you (or itself) without complaining.

When error messages are printed, it is usually difficult to determine the function which caused the error and the functions which called it. In this situation, (BAKGAG T) will turn on the BACKTRACE flag which causes the hierarchy of function calls to be printed as described in the next section.

The following is an alphabetical listing of error messages, their cause, and in some cases, their remedy. Some error messages print two lines, such as:

FOO
UNBOUND VARIABLE - EVAL

These messages are described last in the listing, and are of the form:

X <message>

<u>BINARY PROGRAM SPACE EXCEEDED</u>

ARRAY, EXARRAY, or LAP has exceeded BINARY PROGRAM SPACE. ALLOCATE more BPS next time.

<u>CANT EXPAND CORE</u>

INPUT, OUTPUT, LOAD, or ED failed to expand core. Your job is too large.

<u>CANT FIND FILE - INPUT</u>

The input file was not found. You probably forgot to give the file name extension, or a legal file name list.

<u>DEVICE NOT AVAILABLE</u>

INPUT or OUTPUT found the specified device unavailable. Some other job is probably using it.

<u>DIRECTORY FULL</u>

The directory of the output device is full.

<u>DOT CONTEXT ERROR</u>

READ does not like dots adjacent to parens or other dots.

<u>FILE IS WRITE-PROTECTED</u>

OUTPUT found that the specified file is write-protected.

<u>FIRST ARGUMENT NON-ATOMIC - PUTPROP</u>

An attempt was made to PUTPROP onto a non-identifier.

<u>GARBAGED OBLIST</u>

Some member of the OBLIST has been garbaged. You are in trouble.

<u>ILLEGAL DEVICE</u>

INPUT or OUTPUT was attempted to either a non-existant device or to a device of the wrong type. I.e., INPUT from the lineprinter.

<u>ILLEGAL OBJECT - READ</u>

READ objects to syntactically incorrect S-expressions.

<u>INPUT ERROR</u>

Bad data was read from the selected device.

MORE THAN ONE S-EXPRESSION - MAKNAM

MAKNAM and READLIST object to a list which constitutes the characters for more than one S-expression.

NO FREE STG LEFT

All free storage is bound to the OBLIST and protected cells (such as list ARRAY cells), and bound variables on either the REGULAR or SPECIAL pushdown list. Unbinding to the top level will usually release the storage. If you are in a bind for more free storage, try to REALLOC as described in APPENDIX C.

NO FULL WORDS LEFT

All full words are being used for print names and numbers. The problem and its solution are similar to FREE STG.

NO I/O CHANNELS LEFT

INPUT or OUTPUT failed to find a free I/O channel. There is a maximum of 14 active I/O channels.

NO INPUT - INC

An attempt was made to select a channel for input with INC which was not initialized with INPUT.

NO LIST - MAKNAM

MAKNAM and READLIST object to an empty list.

NO OUTPUT - OUTC

An attempt was made to select a channel for output with OUTC which was not initialized with OUTPUT.

NO PRINT NAME - INTERN

INTERN found a member of the OBLIST which has no print name. You are in trouble.

OUTPUT ERROR

Data was improperly written on the selected output device. Possibly a write-locked DECTAPE.

OVERFLOW

Some arithmetic function caused overflow - either fixed or floating.

PDL OVERFLOW FROM GC - CANT CONTINUE

There is not enough regular pushdown list to finish garbage collection. You lose. Try to REALLOC as described in APPENDIX C.

READ UNHAPPY - MAKNAM

MAKNAM and READLIST object to a list which is not an entire S-expression.

REG PUSHDOWN CAPACITY EXCEEDED
SPEC PUSHDOWN CAPACITY EXCEEDED

A pushdown list has overflowed. This is usually caused by non-termination of recursion. Sometimes you need to ALLOCATE or REALLOC more pushdown list.

TOO FEW ARGUMENTS SUPPLIED - APPLY
TOO MANY ARGUMENTS SUPPLIED - APPLY

APPLY checks all calls on interpreted functions for the proper number of arguments.

X MADE ILLEGAL MEMORY REFERENCE

The function X referred to an illegal address. Usually caused by taking the CAR or CDR of an atom or number.

X NON-NUMERIC ARGUMENT

Arithmetic functions require that their arguments be numbers.

X PROGRAM TRAPPED FROM

An illegal instruction was executed in function X.

X UNBOUND VARIABLE - EVAL

EVAL tried to evaluate an identifier and found that it had no value. You probably forgot to QUOTE some atom or to initialize it.

X UNDEFINED COMPUTED GO TAG IN

A GO in some compiled function had an undefined label.

X UNDEFINED FUNCTION
X UNDEFINED FUNCTION - APPLY

The function X is not defined.

X UNDEFINED PROG TAG - GO

A GO in some interpreted function had an undefined label.

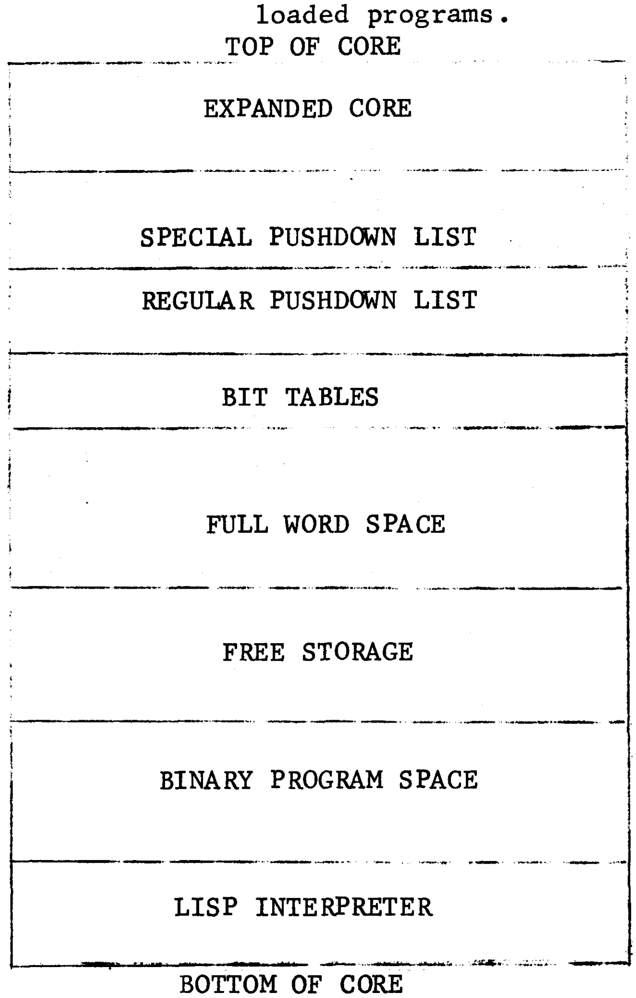
APPENDIX C

MEMORY ALLOCATION

The LISP 1.6 system has many different areas of memory for storing data which can independently vary in size. Some LISP applications demand larger allocations for these areas than others. To allow users to adjust the sizes of these areas to their own needs, a memory allocation procedure exists.

C.1 Summary of Storage Allocation Areas

BINARY PROGRAM SPACE	Area for compiled functions and arrays.
FREE STORAGE	Area for LISP nodes.
FULL WORD SPACE	Area for print names and numbers.
BIT TABLES	Area for the garbage collector.
REGULAR PUSHDOWN LIST	Area for all function calls and non-special variables in compiled functions.
SPECIAL PUSHDOWN LIST	Area for interpreted variables and special variables.
EXPANDED CORE	Area for I/O buffers, ALVINE, LOADER, and any loaded programs.



Memory map for the LISP 1.6 system.

C.2 ALLOC

When the LISP system is initially started, it types "ALLOC?". If you type "N" or space (for no) then the system uses the standard allocations. If you type "Y" (for yes) then the system allows you to specify for each area either an octal number designating the number of words for that area, or a space designating the standard allocation for that area. While typing an octal number, rubout will delete the entire number typed.

	Standard allocation	Alternative
ALLOC? Y		type Y or space
FULL WORDS =	400	octal number or space
BIN.PROG.SP =	2000	octal number or space
SPEC.PDL =	1000	octal number or space
REG.PDL =	1000	octal number or space
HASH =	77	octal number or space

Any remaining storage is divided between the spaces as follows:

1/16 for full word space,
1/64 for each pushdown list,
the remainder to free storage and bit tables.

HASH determines the number of buckets on the OBLIST.

C.3 REALLOC

If you have an existing LISP core image but have exhausted one of the storage areas, it is possible to increase the size of that area using the reallocation procedure. First, expand core with the time sharing system command CORE (C) and then reenter the LISP core image with the REE command. For example, if the original core size was 20K, you could increase it by 4K as follows:

```
↑C
.C24
.REE
*
```

When you reenter a core image, all additional core is allocated as follows:

1/4 for full word space
1/64 for each pushdown list,
the remainder to free storage and bit tables.

C.4 Binary Program Space

The reallocation procedure does not increase the size of binary program space. However, it is possible to increase binary program space by expanding core with the CORE (C) command and setting BPORG and BPEND to the beginning and end of the expanded area of core. For example, if you now have 32K of core and want 4K more BPS, do the following:

```
↑C
.C36
.S
*(SETQ BPORG (TIMES 32. 1Ø24.))
*(SETQ BPEND (PLUS BPORG 4Ø95.))
```

Note: If you use the reallocation procedure after having expanded core for any purpose, it will reallocate this additional core for its own purposes, thus destroying the contents of the expanded core.

The following are the standard causes for expansion of core:

1) using I/O channels.
2) using the LOADER - (LOAD).
3) expanding core for more binary program space.
4) using (ED).

APPENDIX D

GARBAGE COLLECTION

All LISP systems have a function known as the garbage collector. This function analyzes the entire state of list structure which is pointed to by either the OBLIST, the regular pushdown list, the special pushdown list, list arrays, and a few other special cells. By recursively marking all words on free and full word spaces which are pointed to in this manner, it is possible to determine which words are not pointed to and are therefore garbage. Such words are collected together on their respective free storage lists.

(GC)

GC causes a garbage collection to occur and returns NIL. Normally, a garbage collection occurs only when either free or full word space has been exhausted.

(GCGAG X) flag = NIL initially.

GCGAG sets a special flag in the interpreter to the value of X, and returns the previous setting of the flag. When any garbage collection occurs, if the flag ≠ NIL, then the following is printed:

either	FREE STORAGE EXHAUSTED
or	FULL WORD SPACE EXHAUSTED
or	nothing
followed by	x FREE STORAGE, y FULL WORDS AVAILABLE

where x and y are numbers in octal.

(SPEAK)

SPEAK returns the total number of CONSes which have been executed in this LISP core image.

(GCTIME)

GCTIME returns the number of milliseconds LISP has spent garbage collecting in this core image.

It is possible to determine the lengths of the free and full word free storage lists by:

(LENGTH (NUMVAL 15_8)) = length of free storage list
(LENGTH (NUMVAL 16_8)) = length of full word list

APPENDIX E

COMPILED FUNCTION LINKAGE AND ACCUMULATOR USAGE

This appendix is intended to explain the structure of compiled functions, function calls, and accumulator usage. This discussion is relevant only if one intends to interface hand coded functions or possibly functions generated by another system (such as FORTRAN) with the LISP system. In such a case, it is highly recommended that one examine the LAP code generated by the LISP compiler for some familiar functions.

ACCUMULATOR USAGE TABLE

s means "sacred" to the interpreter

p means "protected" during garbage collection

NIL	= Ø	s,p	Header for the atom NIL.
A	= 1	p	Results from functions, 1st arg to functions
B	= 2	p	2nd arg
C	= 3	p	3rd arg
AR1	= 4	p	4th arg
AR2A	= 5	p	5th arg
T	= 6	p	used for LSUBR linkage
TT	= 7	p	
T1Ø	= 1Ø	p	rarely used in the interpreter
S	= 11		rarely used in the interpreter
D	= 12		
R	= 13		
P	= 14	s,p	regular pushdown list pointer
F	= 15	s,p	free storage list pointer
FF	= 16	s,p	full word list pointer
SP	= 17	s,p	special pushdown list pointer.

TEMPORARY STORAGE

Whenever a LISP function is called from a compiled function, it is assumed that all accumulators from 2 through 13 are destroyed by the function unless it is otherwise known. Therefore, local variables and parameters in a compiled function should be saved in some protected cells such as the regular pushdown list. The PUSH and POP instructions are convenient for this purpose.

SPECIAL VARIABLE BINDINGS

Special variables in compiled functions are bound to special cells by:

```
PUSHJ P,SPECBIND
Ø n1,var1
Ø n2,var2
   ...
start of function code.
```

SPECBIND saves the previous values of var_i on the special pushdown list and binds the contents of accumulator n_i to each var_i. The var_i must be pointers to special cells of identifiers. Any n_i=Ø causes the var_i to be bound to NIL.

Special variables are restored to their previous values by:

```
PUSHJ P,SPECSTR
```

which stores the values previously saved on the special pushdown list in the appropriate special cells.

NUMBERS

To convert the number in A from its LISP representation to machine representation use:

```
PUSHJ P,NUMVAL
```

which returns the value of the number in A, and its type (either FIXNUM or FLONUM) in B.

To convert the number in A from its machine representation to LISP representation use either:

	PUSHJ P,FIX1A	for FIXNUMS
or	PUSHJ P,MAKNUM	with type in B.

Both of the above functions return the LISP number in A.

FUNCTION CALLING UUOS

To allow ease in linking, debugging, and modificating of compiled functions, all compiled functions call other functions with special opcodes called UUOs. Several categories of function calls are distinguished:

1) Calls of the form (RETURN (FOO X)) are called terminal calls and essentially "jump" to FOO.

2) Calls of the form (F X) where F is a computed function name or functional argument is called a functional call.

The function calling UUOs are:

	non-terminal	terminal
non-functional	CALL n,f	JCALL n,f
functional	CALLF n,f	JCALLF n,f

where f is either the address of a compiled function or a pointer to the identifier for the function, and n specifies the type of function being called as follows:

n = ∅ to 5	specifies a SUBR call with n arguments
n = 16	specifies an LSUBR call
n = 17	specifies an FSUBR call.

The function calling UUOs are defined in MACRO by:

```
OPDEF CALL [34B8]
OPDEF JCALL [35B8]
OPDEF CALLF [36B8]
OPDEF JCALLF [37B8]
```

(NOUUO X) flag = T initially

NOUUO sets a special flag in the compiled function calling mechanism to the value of X and returns the previous setting of the flag. Compiled functions initially call other functions with function calling UUOs which "trap" into the UUO mechanism of the interpreter. Ordinarily, such function calls involve searching the property list of the function being called for the functional property, and then (depending on whether the function is compiled or an S-expression) the function is called.

If the NOUUO flag is set to NIL, then the overhead in calling a compiled function from a compiled function can be eliminated by replacing the CALL by PUSHJ and JCALL by JRST. CALLF and JCALLF are never changed.

However, there are several dangers and restrictions when using (NOUUO NIL). Once the UUO's have been replaced by PUSHJ's then it is not possible to redefine or TRACE the function being called. It is therefore recommended that compiled functions be debugged with (NOUUO T).

SUBR LINKAGE

SUBRs are compiled EXPRs which are the most common type of function. Consequently, considerable effort has been made to make linkage to SUBRs efficient.

Arguments to SUBRs are supplied in accumulators 1 through n, the first argument in 1. There is a maximum of 5 arguments to SUBRs.

To call a SUBR from compiled code, use call n,FUNC, where n is the number of arguments, and call is the appropriate UUO.

The result from a SUBR is returned in A(= 1).

FSUBR LINKAGE

FSUBRs receive one argument in A and return their result in A. FSUBRs which use the A-LIST feature call:

```
PUSHJ P,*AMAKE
```

which generates in B a number encoding the state of the special pushdown pointer. To call an FSUBR, use call 17, FUNC, here call is the appropriate UUO.

LSUBR LINKAGE

LSUBRs are similar to SUBRs except that they allow an arbitrary number of arguments to be passed. To call an LSUBR, the following sequence is used:

```
        PUSH P, [ret]           ;return address
        PUSH P,arg1             ;1st argument
            ...
        PUSH P,argn             ;nth and last argument
        MOVNI T,n               ;minus number of arguments
        call 16,func            ;the appropriate UUO
ret:                            ;the LSUBR returns here
```

When an LSUBR is entered, it executes:

```
JSP 3,*LCALL
```

which initializes the LSUBR. A will contain n. The ith argument can be referenced by:

```
MOVE A,-i-1(P)
```

Exit from an LSUBR with

```
POPJ P,
```

which returns to *LCALL to restore the stack.

APPENDIX F

THE LISP COMPILER

The LISP compiler is a LISP program which transforms LISP functions defined by S-expressions into LAP code. This code can be loaded into binary program space by LAP which produces actual machine code.

Compiled functions are approximately ten times as fast as interpreted functions. Compiled functions also take less memory space and relieve the garbage collector from marking function definitions. In a very large system of functions, this last point is particularly significant.

To use the LISP compiler, the following procedure is recommended:

1. Prepare your functions in an I/O file (disk, dectape, etc.) in DEFPROP format such as produced by GRINDEF. (See DSKOUT and GRINL in SMILE - SAILON-41).

   a. It is also permitted for this file to contain global variable definitions, MACROs, and SPECIAL variable definitions.

   b. SPECIAL variable definitions must occur before the functions which bind these variables. (DEFPROP FOO T SPECIAL) will declare the variable FOO to be SPECIAL. Variables which are used in a functional context must be declared SPECIAL or else the compiler will mistake them for undefined EXPRs. Use (SPECIAL FOO1 FOO2...FOOn) for several SPECIAL declarations.

   c. FEXPR definitions should occur before functions which call them. If this cannot be arranged, a FEXPR forward reference can be declared to the compiler by (DEFPROP FOO T *FEXPR) where FOO is the name of the FEXPR. The compiler assumes that undefined functions are EXPRs unless otherwise declared.

   d. MACROs must occur before the functions which use them.

   e. Global variable definitions are required to be in DEFPROP format.

2. START the LISP compiler by typing to the system:

   <u>.</u>R COMPLR

   <u>*</u>

a. Declare any FEXPR forward references, MACROs, or SPECIAL variables which are not defined in your I/O file.

b. The global variables IFL and OFL designate to the compiler the names of the input and output devices for compilation. These are both initialized to DSK:.

The global variable LISTING designates a file and device for output of compiler messages such as (FOOBAZ UNDECLARED). For example, if it is desired to output this listing on file DSK: LISTING, evaluate (SETQ LISTING (QUOTE (DSK: LISTING))). LISTING is initialized to NIL which designates the teletype.

c. Compile your function definition files with:

(COMPL fn1 fn2 fn3 ... fnn)

where each fn_i designates a file name on device IFL. Each fn_i is either an atom designating a file name, or a dotted pair designating file name and extension. COMPL produces LAP output on device OFL on files with the same file names but with LAP extensions. COMPL also transfers through unaltered any DEFPROPs with properties other than EXPR, FEXPR, MACRO, and SPECIAL.

d. COMPL will type out:

(x UNDEFINED) for undefined function references. The compiler assumes that x is an EXPR. If x is actually an FEXPR, you must recompile and declare x as an FEXPR by (DEFPROP x T *FEXPR).

(x UNDECLARED) for undeclared global variable references. You need not worry about this message unless x is SPECIAL and you forgot to declare it.

e. When COMPL is done, it returns:

(n PROGRAM BREAK)

where n is the length of the LAP code produced.

3. Load LAP into your core image, then load the compiled functions. For example: (INC (INPUT SYS: LAP DSK: (FOO . LAP))) Be sure to allocate sufficient binary program space for the functions. The proper size is the sum of the program breaks plus the length of LAP which is about 400_8 words.

APPENDIX G

THE LISP ASSEMBLER - LAP

LAP is a primitive assembler designed to load the output of the compiler. Normally, it is not necessary to use LAP for any other purpose.

The format of a compiled function in LAP is:

(LAP name type)

<sequence of LAP instructions>

NIL

where name is the name of the function, and type is either SUBR, LSUBR, or FSUBR.

A LAP instruction is either:

1. A label which is a non-NIL identifier.

2. A list of the form

   (OPCODE AC ADDR INDEX)

   a. The index field is optional.

   b. The opcode is either a PDP-6/10 instruction which is defined to LAP and optionally suffixed by @ which designates indirect addressing, or a number which specifies a numerical opcode.

   c. The AC and INDEX fields should contain a number from Ø to 17, or P which designates register 14.

   d. The ADDR field may be a number, a label, or a list of one of the following forms:

(QUOTE S-expression)	to reference list structure.
(SPECIAL x)	to reference the value of identifier x.
(E f)	to reference the function f.
(C OPCODE AC ADDR INDEX)	to reference a literal constant.

For example, the function ABS could be defined:

```
(LAP ABS SUBR)
(CALL 1 (E NUMVAL))
(MOVMS Ø 1)
(JCALL 2(E MAKNUM))
NIL
```

APPENDIX H

THE LOADER

A modified version of the standard PDP-6/10 MACRO-FAIL-FORTRAN loader is available for use in LISP. One can call the loader into a LISP core image at any time by executing:

(LOAD X)

When a * is typed, you are in the loader, and the loader command strings are expected. As soon as an altmode is typed, the loader finishes and exits back to LISP.

The loader is placed in expanded core. If X = NIL then loaded programs are placed in expanded core, otherwise (if X ≠ NIL) they are placed in BINARY PROGRAM SPACE.

The loader removes itself and contracts core when it is finished. In the following discussion a "RELOC" program will refer to any program which is suitable for loading with the loader. The output of FORTRAN or MACRO is a RELOC program.

(EXCISE)

EXCISE unexpands core to its length after ALLOC or the last REE. This removes I/O buffers, ALVINE, and all RELOC programs.

(*GETSYM S)

*GETSYM searches the DDT symbol table for the symbol S and if found returns its value, otherwise it returns NIL.

(GETSYM "P" "S_1" "S_2" ... "S_n")

GETSYM searches the DDT symbol table for each of the symbols S_i and places the value on the property list of S_i under property P.

Example: (GETSYM SUBR DDT)

This causes DDT to be defined as a SUBR located at the value of the symbol DDT.

Note: In order to load the symbol table, either /S or /D must be typed to the loader. Symbols which are declared INTERNAL are always in the symbol table without the /S or /D. In the case of multiply defined symbols, i.e., a symbol used in more than one RELOC program, a symbol declared INTERNAL takes precedence, the last symbol otherwise.

(*PUTSYM S V)

*PUTSYM enters the symbol S into the DDT symbol table with value V.

(PUTSYM "X_1" "X_2" ... "X_n")

PUTSYM is used to place symbols in the DDT symbol table. If X_i is an atom then the symbol X_i is placed in the symbol table with its value pointing to the atom X_i. If X_i is a list, the symbol in (CAR X_i) is placed in the symbol table with its value (EVAL (CADR X_i)). PUTSYM is useful for making LISP atoms, functions, and variables available to RELOC programs. Symbols must be defined with PUTSYM before the LOADER is used.

Examples: (PUTSYM BPORG (VBPORG (GET (QUOTE BPORG)(QUOTE VALUE))))

defines the identifier BPORG and its value cell VBPORG. A RELOC program can reference the value of BPORG by:

```
MOVE X,VBPORG
```

```
(PUTSYM (MAPLST (QUOTE MAPLIST)) (NUMBERP (QUOTE NUMBERP)))
(PUTSYM (MEMQ (GET(QUOTE MEMQ) (QUOTE SUBR))))
```

A RELOC program would call these functions as follows:

```
CALL 2,MAPLST
CALL 1,NUMBRP
PUSHJ P,MEMQ   or   CALL 2,MEMQ
```

An example of a simple LISP compatible MACRO program to compute square roots using the FORTRAN library.

```
    TITLE TEST
    P=14
    A=1
    B=2

EXTERN MAKNUM,NUMVAL,SQRT,FLONUM

LSQRT:  CALL 1, NUMVAL
        MOVEM A,AR1
        MOVE A,[XWD Ø,BLT1]; SAVE THE AC'S
        BLT A,BLT1+17
        JSA 16,SQRT
        JUMP 2,AR1 ;SOP TO FORTRAN
        MOVE Ø,AR1
        MOVE A,[XWD BLT1 ,Ø]
        BLT A,17
        MOVE A,AR1
        MOVEI B,FLONUM
        JCALL 2, MAKNUM
```

```
AR1:      ∅
BLT1:     BLOCK 2∅

END
```

APPENDIX J

References

1. John McCarthy, et al, LISP 1.5 Programmer's Manual (Cambridge, Mass., MIT Press, 1962).

2. Clark Weissman, LISP 1.5 Primer, (Dickenson Publishing Co., 1967).

3. Robert A. Saunders, "LISP - On the Programming System", in Edmond C. Berkley and Daniel G. Bobrow (eds.), The Programming Language LISP: Its Operation and Applications, 2nd ed., (Cambridge, Mass., The MIT Press, 1966), p54.

APPENDIX K

INDEX

(I HATE NUMERICAL ANALYSIS)