SAILON 28.2

ABSTRACT

This manual describes the PDP-6/10 LISP 1.6 system developed by the Stanford Artificial Intelligence Project. The manual is not a tutorial on LISP but is intended to supplement existing LISP tutorials. Differences from other LISP systems and new functions and features are described in order to prepare one to understand and use this LISP system.

This manual supercedes and replaces SAILONS 1, 4, 28 and 28.1.

ACKNOWLEDGMENT

The Stanford AI LISP 1.6 system is an adaptation by John R. Allen and Lynn H. Quam of a LISP system developed at M.I.T. Sections of this document were contributed by John R. Allen and Anthony C. Hearn. This work was supported by the Advanced Research Projects Agency of the Office of the Secretary of Defense under Contract SD-183.

TABLE OF CONTENTS

CHAPTER 1

INTRODUCTION

This manual is intended to explain the interactive LISP 1.6 system which has been developed for the PDP-6/10 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. Users who are familiar with another system should see Appendix A: Differences from Standard LISP.

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-ouput channels, the ability to control the size of memory spaces, a standard relocating loader to load assembly language or compiled programs, etc.

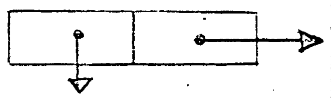
This system uses an interpreter; however there is also a compiler which produces machine code. Compiled functions are approximately ten times as fast and also take less memory space.

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.

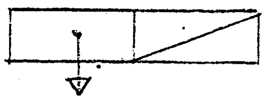
1.1 DOCUMENT CONVENTIONS

1.1.1 REPRESENTATION CONVENTIONS

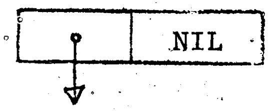
In the description of data structures, the following notational conventions will be used:

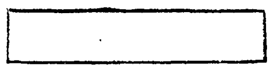


represents a 36-bit word in FREE STORAGE with 2 18-bit pointers.



means





represents a 36-bit word in FULL WORD SPACE.

1.1.2 SYNTAX CONVENTIONS

A modified BNF will be used to define syntax equations. Literal (terminal) strings will be explicitly quoted with " . Non-terminal syntax rules are not bracketed with < and > . Parentheses occur around optional constructs.

1.1.3 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.

1.1.4 OTHER CONVENTIONS

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

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 is equivalent to:

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

All examples at the top level assume this definition.

The following dialog shows how to log into the time-sharing system, start the LISP system, and interact with it at the top level. Underlined characters are typed by LISP. "↲" means carriage-return and line-feed. "$" means altmode.

```
                .L↲
login         - *1/FOO↲                        Give your project-programmer
                ↑C↲                               number.
starting        .R LISP↲                       Core size may be specified.
  LISP        - LISP 1.6 (date of system)↲        See note 1.
                ALLOC?↲                        Memory allocation can be
                *T$↲                              specified.  See note 2.
                T↲                             T and NIL always evaluate to
examples        *(QUOTE (A B C))$↲                themselves
  at            (A B C)↲                       You are talking to EVAL.
  top         - *(INC (INPUT SYS: SMILE))$↲    This READs the file SYS:SMILE
 level          SMILEFNS↲
                <a long sequence of output>    The output can be suppressed
                * (CONS 1 (QUOTE A))$↲            with ↑0.
                *(1 . A)↲
                *
```

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.

2,2 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 control characters should be useful in LISP.

teletype	III display	meaning
↑C	CALL	Stop the job and talk to the time sharing system.
↑O	δ	Suppress console printout until an input is requested.
↑U	∃	Delete the entire input line now being typed. (Only with (DDTIN NIL)).
↑G	π	Stop the LISP interpreter and return control to the top level of LISP. Only effective when LISP is asking for console input.
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:   ignored-character ::= carriage-return | line-feed

     delimiter        ::= "(" | ")" | "." | "/" | blank |
                              tab | altmode

     character        ::= (any ASCII character other than null and
                             rubout)

     digit            ::= "Ø" | "1" | ... | "9"

     letter           ::= (any character not a digit, not a delimiter,
                            and not an ignored-character)

     identifier       ::= letter

                      ::= identifier letter

                      ::= identifier digit

                      ::= "/" character

                      ::= identifier "/" character
```

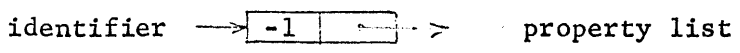
Semantics:

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.

```
Examples:   A
            APPLE
            FOObaz
            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.

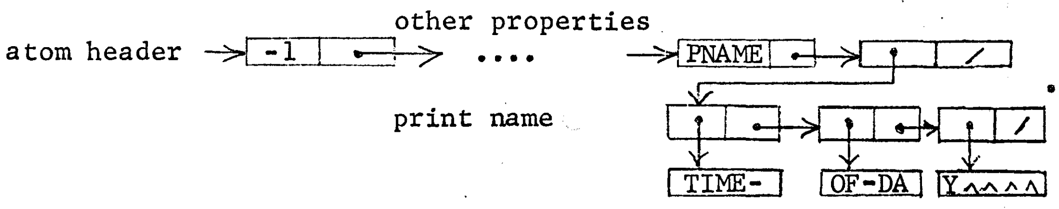
3.1 PROPERTY LISTS

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:



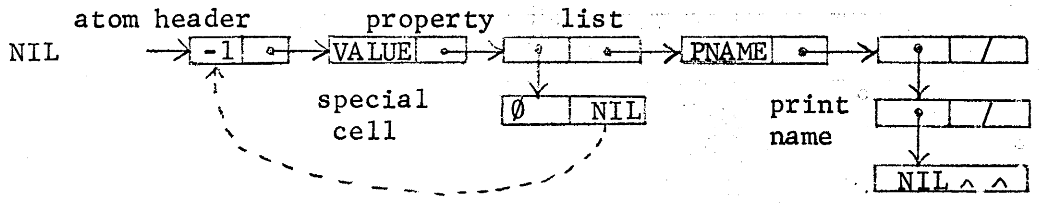
where ∧ means null or ASCII Ø.

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, 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.

CHAPTER 4. NUMBERS

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

4.1 INTEGERS

Syntax:

integer	::=	(sign) digits (".")
digits	::=	digit
	::=	digits digit
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 = 1∅ then integers will print with a following "." , unless the global variable *NOPOINT = T .

Examples with IBASE=8

input		meaning	
-13	=	-11.	$= -11_{1\emptyset}$
1∅∅∅	=	512.	$=+512_{1\emptyset}$
19	=	17.	$= +17_{1\emptyset}$

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	$\lvert n\rvert < K$	K is usually 2^{16}
FIXNUM	$K \leq \lvert n\rvert < 2^{35}$	
BIGNUM	$2^{35} \leq \lvert n\rvert$	

Representation of INUMs:

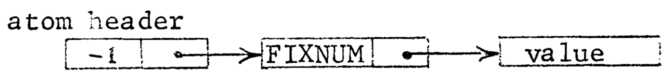
INUMs are represented by pointers outside of the normal LISP addressing space. INUMs are addresses in the range $2^{18}-2K+1$ to $2^{18}-1$.

Examples:

INUM	representation
-(K-1)	$2^{18}-K-(K-1)$
-1	$2^{18}-K-1$
Ø	$2^{18}-K$
+1	$2^{18}-K+1$
+(K-1)	$2^{18}-1$

Representation of FIXNUMs:

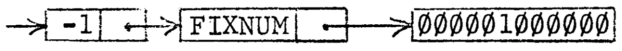
FIXNUMs are represented by list structure of the following form:



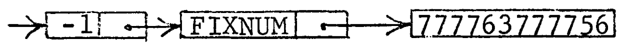
where value is the 2's complement representation of the fixed point number.

Examples:

$+1000000000_8$



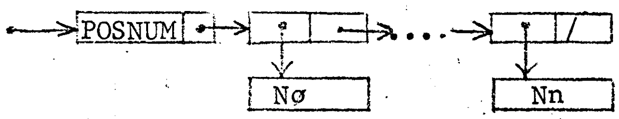
-1400000032_8



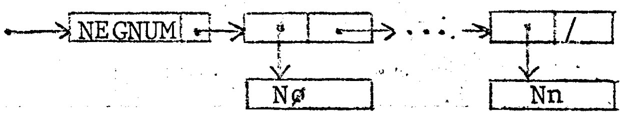
Representation of BIGNUMs:

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

$$\text{sign} \cdot \sum_{i=0}^{n} N_i \cdot (2^{35})^i .$$

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

4.2 REALS

Syntax:

real	::=	(sign) digits exponent
	::=	(sign) (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

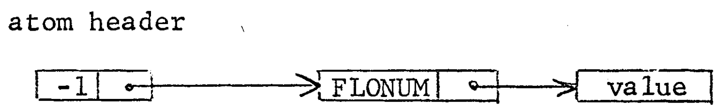
Restrictions:

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:



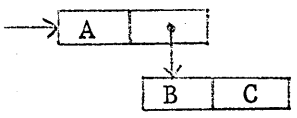
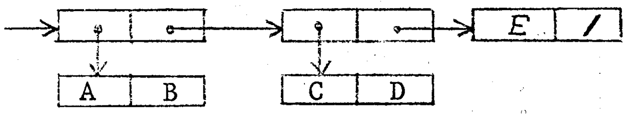
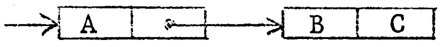
where value is in PDP-6/1Ø 2's complement floating point representation.

CHAPTER 5. S-EXPRESSIONS

Syntax:

```
atom              ::= identifier | number

S-expression           ::=  atom

                       ::=  "(" S-expression-list ")"

                       ::=  "(" ")"                        = NIL

S-expression-list          ::=  S-expression

                           ::=  S-expression  S-expression-list

                           ::=  S-expression list  "."  S-expression
```

Examples:

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.

(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.

(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

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.

Examples:

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

6.2 FEXPRs

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.

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

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.

(ARG N)

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

Example:

```
(DE MAX N
   (PROG (M)
         (SETQ M (ARG N))
   L     (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 -50) = 4
```

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

(QUOTE "E")

QUOTE returns the S-expression E without evaluating it

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 binding 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 LAMBDA 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 Y 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

The A-LIST which is used in some LISP systems does not exist here, but its effects are implemented through the SPECIAL pushdown list, and some special mechanisms. 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 some previous state of the SPECIAL push-down list which defines the context of variable binding in which to evaluate S-expressions.

If an FEXPR is defined with two arguments, then the second argument will be bound to the current state of the SPECIAL pushdown list, which is represented by a number specifying the value of the SPECIAL pushdown pointer.

Example:

```
    (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.

7.3 FUNCTIONAL ARGUMENTS

An argument to a function which is itself a function is called a functional argument. Because most arguments to functions are evaluated when they are passed, the special functions FUNCTION and *FUNCTION are used to control the passing of functional arguments.

```
(FUNCTION "FN") = (QUOTE "FN")
(*FUNCTION "FN")
```

*FUNCTION causes the current state of the special pushdown list to be passed in a list of the following form:

```
        (FUNARG FN . SPECPDL)
```

When APPLY sees a FUNARG list, APPLY performs evaluation of FN in the context of the SPECIAL pushdown list which was passed in the FUNARG list.

CHAPTER 8. CONDITIONAL EXPRESSIONS

A conditional expression has the following form:

$$\begin{array}{l}(\text{COND }(e_{1,1}\ e_{1,2}\ \cdots\ e_{1,n_1})\\ \qquad\quad (e_{2,1}\ e_{2,2}\ \cdots\ e_{2,n_2})\\ \qquad\qquad \cdots\\ \qquad\quad (e_{m,1}\ e_{m,2}\ \cdots\ e_{m,n_m}))\end{array}$$

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 $e_{k,2}$ $e_{k,3}$ ... 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 iff 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 iff 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 iff 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
(EQ (QUOTE (A B)) (QUOTE (A B)))     = NIL
```

(EQUAL X Y)

The value of EQUAL is T iff X and Y are identical S-expressions. EQUAL can also test for equality of numbers of the same type. To compare numbers of different types use: (ZEROP (DIFFERENCE X Y)). For non-numerical comparisons, EQUAL is equivalent to:

```
    (LAMBDA(X Y)(COND ((EQ X Y)T)
                      ((ATOM X) NIL)
                      ((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)   = NIL
(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 are not 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) (EQUAL X (PLUS X Ø.Ø)))
(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 B 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_2 ... 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)

*APPEND forms a list which is the second list appended to the first according to the following definition:

```
(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.

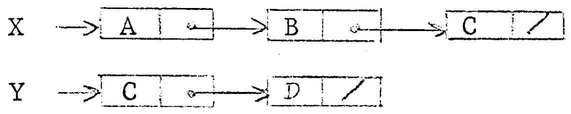
(RPLACA X Y)

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

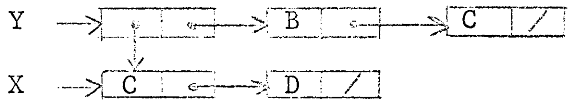
Example: (RPLACA (QUOTE (A B C)) (QUOTE (C D))) = ((C D) B C)

Representation:

BEFORE



AFTER



(RPLACD X Y)

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

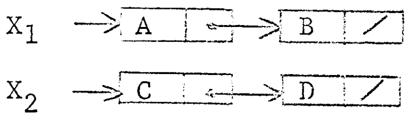
(NCONC X_1 X_2 ... X_n)

NCONC is similar in effect to APPEND, but NCONC does not copy list structure. 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.

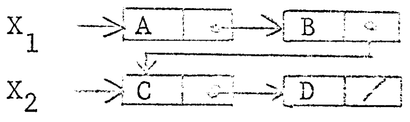
Examples: (NCONC) = NIL
(NCONC (QUOTE (A B)) (QUOTE (C D))) = (A B C D)

Representation:

BEFORE



AFTER



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 (see 7.3).

(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)))))
```

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

10.7 NON-STANDARD S-EXPRESSION FUNCTIONS

<u>(EXPLODE L)</u>

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

<u>Examples</u>:

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

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

<u>(EXPLODEC L)</u>

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

<u>Example</u>:

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

<u>(FLATSIZE L)</u> = (LENGTH (EXPLODE L))

<u>(MAKNAM L)</u>

<u>MAKNAM</u> 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.

<u>Examples</u>:

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

<u>(READLIST L)</u>

<u>READLIST</u> 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 erroreous 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:

```
(LAMBDA(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:

```
(LAMBDA (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 I. The value of PUTPROP is V.

```
Example:     (PUTPROP (QUOTE POSP) (QUOTE (LAMBDA (X) (GREATERP X Ø)))
                      (QUOTE EXPR))
```

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

DEFPROP is the same as PUTPROP except that it does not evaluate its arguments.

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

DE, DF and DM are useful for defining EXPR, FEXPRs, and MACROs.

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

```
= (PROG2 (PUTPROP ID (LIST (QUOTE LAMBDA) ARGS BODY) (QUOTE EXPR))
         ID)
```

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

```
= (PROG2 (PUTPROP ID (LIST(QUOTE LAMBDA) ARGS BODY)(QUOTE FEXPR))
         ID)
```

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

```
= (PROG2 (PUTPROP ID (LIST(QUOTE LAMBDA) ARGS BODY)(QUOTE MACRO))
         ID)
```

(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.

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 corresponding and returns a new identifier specified by the counter. The GENSYM counter is initialized to the identifier GØØØØ.

Example: 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 is described in Appendix B.

```
(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
```

Note: For integer arguments, *QUO and QUOTIENT give the number theoretic integer quotient.

```
(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)
```

FIX returns the truncated 2's complement integer value of X.

Examples:

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

Other arithmetic functions not defined in the LISP interpreter:

```
(FLOAT X) = (XPLUS X Ø.Ø)

(RECIP X) = (QUOTIENT 1 X)
```

(EXPT X Y) = X^Y

(SQRT X) = $\sqrt{X}$

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

(ENTIER X) = (SIGN X) * (FIX (ABS X))

(MIN X Y)  = (COND ((MINUSP (DIFFERENCE X Y)) X) (Y))

(MAX X Y)  = (COND ((MINUSP (DIFFERENCE 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
Ø	Ø	1Ø	$\overline{A_i} \wedge \overline{B_i}$
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 7Ø)     = 3Ø
(BOOLE 12 13 Ø)         = 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.

<u>(PROG "VARLIST" "BODY")</u>

<u>PROG</u> 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.

<u>(RETURN X)</u>

<u>RETURN</u> causes the PROG containing it to be exited with the value X.

<u>(GO "ID")</u>

<u>GO</u> 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 not evaluated. GO cannot transfer into or out of a PROG.

<u>Note</u>: Both RETURN and GO should 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.

<u>Example</u>: 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)))
```

<u>(PROG2 X_1 X_2 ... X_n)</u> , $n \leq 5$.

<u>PROG2</u> evaluates all expressions X_1 X_2 ... X_n, and returns the value of X_2.

13.1 SET and SETQ

<u>SET</u> and <u>SETQ</u> 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).

<u>(SET ID V)</u>

<u>SET</u> changes the value of the identifier ID to V and returns V. Both arguments are evaluated.

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

<u>(SETQ "ID" V)</u>

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

CHAPTER 14. INPUT/OUTPUT

14.1 DEVICE SELECTION AND CONTROL

The following functions select and control input/output devices and files.

14.1.1 FILE NAMES

File names are specified by a filename list of the following form:

Syntax:
```
                filename-list ::= device-name
                              ::= filename-list device-name
                              ::= filename-list filename

                device-name   ::= identifier ":"
                              ::= "(" atom atom ")"

                filename      ::= identifier
                              ::= "(" identifier "." identifier ")"
```

Semantics:

A device-name is either an identifier followed by 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.

Examples: The following examples show the correspondence between LISP and PIP filename specifications.

```
     LISP     (SYS: (SMILE .LSP))
     PIP      SYS:SMILE.LSP

     LISP     (DSK: FOO (1,FOO) (BAZ . ZAM) MAZ)
     PIP      DSK:FOO,DSK: [1,FOO]BAZ.ZAM,MAZ
```

14.1.2 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.

14.1.3 INPUT

(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 the optional argument ACTION is not specified or 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.

The input functions in 14.2 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 function 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.

14.1.4 OUTPUT

(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 14.3 transfer output to the selected output channel.

If the optional argument ACTION is unspecified, or 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.

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.

14.2 INPUT

(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.

(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)))
```

14.2.1 TELETYPE INPUT

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 last S-expression typed if the previous character was a space, right parens, tab, or comma, otherwise rubout will delete the last atom fragment typed or the last left parens. READ will terminate as soon as a complete S-expression is typed.

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 OUTPUT

(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).

(PRINC S)

PRINC is the same as PRIN1 except that no slashes are inserted.

(TERPRI)

TERPRI prints a carriage-return, line-feed pair and returns NIL.

(PRINT S)

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

(TYO N)

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

14.3.1 TELETYPE OUTPUT

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

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.

APPENDIX A

DIFFERENCES FROM STANDARD LISP

by Anthony C. Hearn

Standard LISP was developed to provide for the easy assembly of a given LISP program in more than one LISP system. Its syntactical form is described in detail in a Stanford AI Project Memo which a prospective user should consult for details. We shall not duplicate the content of this Memo here, but simply point out that the translation between a Standard LISP program and a given LISP system is achieved by a preprocessor which is defined for a given system and always loaded ahead of any Standard LISP program. This Appendix will therefore limit itself to pointing out the essential differences between Standard LISP and Stanford AI LISP 1.6 and the procedure for loading the preprocessor and running Standard LISP programs in this system.

A.1 WRITING STANDARD LISP PROGRAMS

The major differences between a program written in Standard LISP as opposed to one in Stanford AI LISP 1.6 are as follows:

(i) Programs are written in EVALQUOTE rather than EVAL format.

(ii) All atoms or character strings containing non-alphameric characters (alphameric characters in this context being the capitalized Roman letters A through Z and decimal digits 0 through 9) must be replaced by an alphameric atom. The particular value required for the atom in Stanford AI LISP 1.6 must then appear in a call to the preprocessor function NEWNAM described in the Standard LISP memo. It is advisable for the user to prepare a special file containing this call to NEWNAM, which can then be loaded with the preprocessor.

(iii) MACROS, LEXPRS and LSUBRS are not defined in Standard LISP.

(iv) COND expressions can have only one consequent, and must end with a pair (T FORM) unless the COND occurs within a PROG.

(v) The functions MAP, MAPCAR and MAPLIST are defined, but their arguments are in the conventional order <u>opposite</u> to LISP 1.6.

(vi) The following functions have alternative names or forms in Standard LISP. Users should consult the Standard LISP Memo for the particular definitions of the functions mentioned.

Stanford AI LISP 1.6 Function	Standard LISP function
APPEND	Same name, but only two arguments allowed.
APPLY	*APPLY
ARRAY	Same name, but different arguments.
CHRCT	Use POS instead.
ERR	Use ERROR instead.
ERRSET	ERRORST
EVAL (with one argument)	*EVAL
GENSYM	Use MKSYM (mksym[ ] = intern[gensym[ ]])
INC	Use RDS
INPUT	Same name, but different arguments.
LINELENGTH	OTLL
LSH	LEFTSHIFT
MAKNAM	Use COMPRESS instead.
OUTC	Use WRS
OUTPUT	Same name, but different arguments.
PUTPROP	Use PUT (put[u,v,w] = putprop [u,w,v])
READLIST	Use COMPRESS instead.
REMOB	Same name, but is a SUBR taking a single atom as argument.

/s

(vii) The following functions and value cells are not defined in Standard LISP. They may be included in a given program if the user can provide an equivalent definition in his pre-processor for another LISP system:

ALIST	DM	INTERN	TYI
ARG	ED	LAST	TYO
ASCII	EXAMINE	LOAD	XCONS
ASSOC	EXARRAY	MAKNUM	*AMAKE
BAKGAG	EXCISE	MAPC	*APPEND
BASE	EXPLODEC	MEMQ	*DIF
BOOLE	FLATSIZE	NCONS	*EVAL
BPEND	FORCE	NOUUO	*FUNCTION
BPORG	GC	NSTORE	*GREAT
CSYM	GCD	NUMVAL	*LCALL
DDTIN	GCGAG	OBLIST	*LESS
DDTOUT	GCTIME	PUTSYM	*NOPOINT
DE	GETL	SPEAK	*PLUS
DEFPROP	GETSYM	SPECIAL	*QUO
DEPOSIT	IBASE	STORE	*RSET
DF	INITFN	TIME	*TIMES

A.2 RUNNING STANDARD LISP PROGRAMS

To facilitate the running of Standard LISP programs in the Stanford AI LISP 1.6 system, a SYS: file PREP.LSP is available containing the Standard LISP preprocessor for this system. A LAP version of this file is also available on SYS: with filename PREP.LAP. To use the latter file, the user must have previously loaded LAP into his core partition.

Both PREP files are in Stanford AI LISP 1.6 format and can therefore be loaded using INC. In addition to the preprocessor, the files contain the following functions to help with the running of programs:

(SINC CHANNEL ACTION)

SINC is like INC, except that the files read in should be in Standard LISP format.

Example

```
(SINC (INPUT DSK: F1 (F2.. LSP))
```

will load the Standard LISP DSK: files F1 and F2.LSP.

(CMFILE "DEVICE-NAME"."FILENAME-LIST")

This function is used in conjunction with the LISP 1.6 compiler. CMFILE compiles all functions defined in the Standard LISP files appearing in FILENAME-LIST and outputs the LAP onto DEVICE-NAME as files in Stanford AI LISP 1.6 format with the same name as the input files and an extension LAP. Any EVALQUOTE pairs in the file which are not function definitions are also output after conversion to LISP 1.6 format.

The standard compiler messages are printed by CMFILE. In addition, a check is made for the existence of a function indicator on the property list of any function being compiled. If one is found, a message

(FUNCTION-NAME IN SYSTEM)

is printed. This is useful for detecting conflicts between system and user-program function names. Finally, any special variable declarations should be added to the user preprocessor file containing the call to NEWNAM.

Example

```
    (CMFILE DSK: DSK: F1 (F2 . LSP))
```

will produce LAP versions of the DSK: files F1 and F2.LSP as DSK: files with names F1.LAP and F2.LAP, respectively.

(GRINFILE "DEVICE-NAME"."FILENAME-LIST")

This function produces GRINDEF copies in Standard LISP format of all Standard LISP files declared in FILENAME-LIST. The files are produced with the same name and an extension NEW so that the disk can be used for both input and output. All the new files can have their extensions easily changed using PIP. To use GRINFILE, the SYS: file GRIN or the editor ALVINE must also be loaded.

Example

```
    (GRINFILE DSK: DSK: F1 (F2 . LSP))
```

will produce GRINDEF copies of the files F1 and F2.LSP as DSK: files with names F1.NEW and F2.NEW, respectively.

(GETDEF "DEVICE-NAME" "FILENAME" "FUNCTION-NAME" ... "FUNCTION-NAME")

GETDEF searches the Standard LISP file FILENAME for the function names specified and loads them when encountered. If any of the function names are not found, a message

(FUNCTION-NAME ... FUNCTION-NAME NOT FOUND)

is printed.

Example

```
    (GETDEF DSK: F1 CALL1 CALL2)
```

will load the functions CALL1 and CALL2 from the DSK: file F1.

APPENDIX B.

ERROR MESSAGES AND CONTROL

B.1 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.

<u>OUTPUT ERROR</u>

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

<u>OVERFLOW</u>

Some arithmetic function caused overflow - either fixed or floating.

<u>PDL OVERFLOW FROM GC - CANT CONTINUE</u>

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

<u>READ UNHAPPY - MAKNAM</u>

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

<u>REG PUSHDOWN CAPACITY EXCEEDED</u>
<u>SPEC PUSHDOWN CAPACITY EXCEEDED</u>

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

<u>TOO FEW ARGUMENTS SUPPLIED - APPLY</u>
<u>TOO MANY ARGUMENTS SUPPLIED - APPLY</u>

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

<u>X MADE ILLEGAL MEMORY REFERENCE</u>

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

<u>X NON-NUMERIC ARGUMENT</u>

Arithmetic functions require that their arguments be numbers.

<u>X PROGRAM TRAPPED FROM</u>

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.

B.2 FUNCTIONS FOR CONTROLLING ERRORS

(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 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.

(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)

*RSET sets a special flag in the interpreter to the value of X. Normally, (with (*RSET NIL)) when an error occurs, special variables are restored to their top level values from the special pushdown list. With (*RSET T), the special variables are not restored and still contain the values at the time of the error.

(BAKGAG X)

BAKGAG sets a special flag in the interpreter to the value of X. If the flag = T 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 call. 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 currently 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 of LISP to become EVALQUOTE instead of EVAL.

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 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	"
SPEC.PDL =	1000	"
REG.PDL =	1000	"
HASH =	77	"

Any remaining storage is divided between the spaces as follows:

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

HASH determines the number of buckets on the OBLIST.

C.2 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
.C 24
.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.3 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
.C 36
.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.

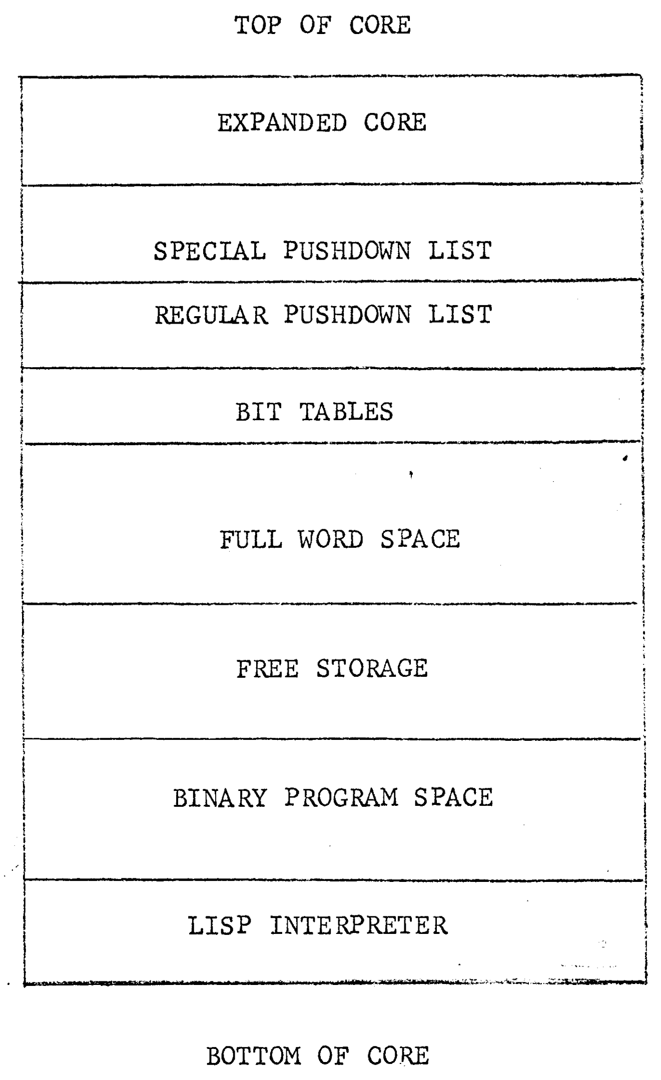
The following are the standard causes for expansion of core:

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

C.4 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 special variables.
EXPANDED CORE	Area for I/O buffers, ALVINE, LOADER, and any loaded programs.



Memory map for the LISP 1.6 system.

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 in 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)

GCGAG sets a special flag in the interpreter to the value of X. 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 radix BASE.

(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.

(TIME)

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

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	= ∅	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	
T1∅	= 1∅	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 = Ø 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)

NOUUO sets a special flag in the UUO calling mechanism to the value of x. When a CALL or JCALL to another compiled function is executed, if the flag = NIL then the CALL or JCALL is changed to a PUSHJ or JRST respectively. If the flag ≠ NIL then no change is made. CALLF and JCALLF are never changed.

NOTE: For debugging compiled functions, (NOUUO T) is recommended. For running debugged compiled functions, (NOUUO NIL) is more efficient.

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 - Operating Note No. 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.

   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:

   ```
   .R COMPLR

   *
   ```

   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:.

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)

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

Both the loader and the programs loaded are placed in expanded core. 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 "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 occurrence otherwise.

(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)) (NUMBRP (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:      Ø
BLT1:     BLOCK 2Ø

END
```

APPENDIX I

ALVINE

by John Allen

A new LISP editor, Alvine is now available. Significant improvements have been made in the command structure and speed of Alvine. The major addition to Alvine is a pointer which can be moved through the editor's string; the editing features affect only the area to the right of this pointer. One can insert and delete arbitrary character strings; and file and defile these strings on various I-O devices.

The data for Alvine are aribtrary strings of LISP atoms, numbers, parens and dots. The Alvine commands are designed to edit these strings into LISP S-expressions with a minimal amount of fuss. The editor is initially equipped to handle the special indicators, FEXPR, EXPR and VALUE. This list may be ammended by the programmer (see the description of "G").

Description of the Command Structure

Each command to Alvine consists of a single character 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.

The modification commands include insertion and deletion of material; "pointer string" refers to the string to the right of the pointer.

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	Delete the first n objects to the right of the pointer. If n is omitted, 1 is assumed.
E	Expunge	Expunge the first S-expression in the pointer buffer.
F x y: z	File	GRINDEFS the material referred to by "x" on device "y" using "z" as a file name. If "x" is a list then each element of "x" is filed under "z"; if "x" is an atom then "x" is assumed to be SETQed to a list of names to be filed.
Gx	Get	G will move the S-expression with name "x" into the Alvine buffer and initialize the pointer to the left-hand end of the buffer. If the indicator associated with "x" is "VALUE" then Alvine sees: (SETQ x (QUOTE . . .)) otherwise (DEFPROP x ( ) indicator). Alvine looks for the indicators on the list named "%%%L", which is initialized to "(FEXPR EXPR VALUE)". %%%L may be amended as needed. G also knows about traced functions and will edit them properly.
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.
M	Match	Move the pointer one s-expression to the right of the current pointer position; if there is no such s-expression the pointer is not moved and the bell is sounded.

P X	Put	Returns the editor string to X through EVAL. Thus, for example, DEFPROP and SETQ are handled by the editor.
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 than 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.
U x-y: z	Unfile	READS the functions specified by "x" from device "y" using "z" file name. If "x" is an atom then x is assumed to be SETQ to a list of names to be READ.
V	Vomit	Print the first balanced paren section to the right of the pointer 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 "Ø<" 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.
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 ∏, ␣ and $ respectively.

```
↑C↵

.R LISP 12↵
LISP 22-Aug-68
ALLOC? N␣↵
T␣↵
T ↵
( 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 J

BIGNUMS - ARBITRARY PRECISION INTEGERS

LISP numbers have always been second class citizens, in the sense that unlike strings (print names) numbers have had a maximum length. In the PDP-6/1Ø LISP system there is an optional arbitrary precision integer package which extends the length of LISP integers from 36-bits to any length.

To load the BIGNUM system, execute the following at the top level of LISP:

```
                    *(LOAD)*SYS:BIGNUM$
                       ...  <the loader types back>
                     *(INC(INPUT SYS: LAP (BIGNUM.LAP)))
                       ... <LISP types the names of the functions loaded>
                     *(APNINIT)
```

and then your core image will perform arbitrary precision integer operations using the standard LISP arithmetic functions which were redefined by APNINIT.

It is possible to load the BIGNUM package at any time unless you have already executed compiled functions with (NOUUO NIL), in which case you must reconstruct your core image.

INDEX

			Page
ABS	SUBR		12-1
ADD1	SUBR		12-1
ALIST			7-4
AND	FSUBR		9-3
APPEND	LSUBR	n arguments	10-1
APPLY	LSUBR		7-1
ARG	SUBR		6-3
ARRAY	FSUBR	different	15-1
ASCII	SUBR		11-3
ASSOC	SUBR	uses EQ	10-6
ATOM	SUBR		3-1,9-1
BAKGAG	SUBR		B-5
BASE	VALUE	initialized to 8	4-1
BIGNUM			J,4-2
BOOLE	LSUBR		12-3
BPEND	VALUE	initialized to top of binary program space.	C-3
BPORG	VALUE	initialized to bottom of binary program space.	C-3
CAR	SUBR		10-2
CDR	SUBR		10-2
.	.		.
.	.		.
.	.		.
CAAAAR	SUBR		10-2
CDDDDR	SUBR		10-2
CHRCT	SUBR		14-3
COND	FSUBR	allows more than one consequent	8-1
CONS	SUBR		10-1
CSYM	FSUBR		11-3
DDTIN	FSUBR		14-4
DDTOUT	FSUBR		14-5
DE	FSUBR		11-2
DEFPROP	FSUBR	n arguments	11-2
DEPOSIT	SUBR		15-3
DF	FSUBR		11-2
DIFFERENCE	LSUBR		12-1
DIVIDE	SUBR		12-1
DM	FSUBR		11-2
ED	SUBR		I-1
EQ	SUBR		9-1
EQUAL	SUBR	∅.∅ ≠ ∅	9-1
ERR	SUBR		B-5
ERRSET	FSUBR		B-5
EVAL	LSUBR	second argument allowed	7-1

			Page
MAKNUM	SUBR		E-2
MAP	SUBR	different argument order	10-5
MAPC	SUBR	" " "	10-5
MAPCARD	SUBR	" " "	10-6
MAPLIST	SUBR	" " "	10-5
MEMBER	SUBR	uses EQUAL	9-2
MEMQ	SUBR	uses EQ	9-2
MINUS	SUBR		12-1
MINUSP	SUBR		9-2
NCONC	LSUBR		10-3
NCONS	SUBR		10-1
NIL	VALUE	initialize to NIL	5-1
NOT	SUBR		9-3
NOUUO	SUBR		E-3
NSTORE	FSUBR		13-2
NULL	SUBR		9-2
NUMBER			4-1
NUMBERP	SUBR		9-2
NUMVAL	EXPR		E-2
OBLIST	VALUE		3-3
OR	FSUBR		9-3
OUTC	FSUBR		14-3
OUTPUT	FSUBR		14-3
PLUS	LSUBR	n arguments	12-1
PNAME			3-2
PRIN1	SUBR	allows non-atomic S-expressions	14-5
PRINC	SUBR		14-5
PRINT	SUBR	TERPRI first	14-5
PROG	FSUBR		13-1
PROG2	SUBR	allows up to 5 arguments	13-1
PROPERTY LIST			3-2
PUTPROP	SUBR	order of arguments different	11-1
PUTSYM	FSUBR		H-1
QUOTE	FSUBR		7-1
QUOTIENT	LSUBR	n arguments	12-1
READ	SUBR		14-3
READCH	SUBR		14-4
READLIST	SUBR		10-7
REMAINDER	SUBR		12-1
REMOB	FSUBR		11-2
RETURN	SUBR		13-1
REVERSE	SUBR		10-4
RPLACA	SUBR		10-3
RPLACD	SUBR		10-3
SASSOC	SUBR	uses EQ	10-6
SET	SUBR		13-2
SETQ	FSUBR		13-2
S-EXPRESSION			5-1