DECUS PROGRAM LIBRARY

DECUS NO.	11-304
TITLE	LISP-11
AUTHOR	Jeffrey Kodosky
COMPANY	Applied Research Laboratories University of Texas at Austin Austin, TX 78712
DATE	6 January 1977
SOURCE LANGUAGE	MACRO

GENERAL INFORMATION

Object Computer(s)	PDP-11	Source Computer (if different)	
File Name	LISP	Version No.	
Title	LISP-11		
Author	Jeffrey Kodosky		
Submitter (if other than author)			
Affiliation	Applied Research Laboratories, University of Texas at Austin		
Address	P.O. Box 8029		
	Austin, TX 78712	Country	USA
Monitor/Operating System	RT-11	DEC No.	
Core Storage Required	16K	Starting Address	
Peripherals Required			
Other Software Required		DEC or DECUS No.	
Source Language	MACRO	Category	
Restrictions, Deficiencies, Problems	relatively bug free		
Date of Planned or Possible Future Revisions			

TAPES AVAILABLE

Paper Tapes   Object Binary ☐   Object ASCII ☐   Source ☐   Other ______

DECtape ☐   LINCtape ☐   Format ______   Magtape: 7 Track ☒   9 Track ☐   BPI 800

Object Files ☐   Source Files ☒   Documentation Files ☐   Other ______

ABSTRACT

LISP-11 is an interpreter for the LISP language which runs in the backroung under RT-11. There are 125 LISP functions implemented with provision to conditionally assemble out as many as 60 in order to maximimize free space.

NOTE: Users who believe they have found LISP system bugs, omissions or errors in the documentation should mail these with printed output (where possible) to the author at the above address.

TABLE OF CONTENTS

TABLE OF CONTENTS (Cont'd)

1. INTRODUCTION

This manual documents the details of how this program implements the LISP language on the PDP-11 series computers with the RT-11 operating system.

This implementation is an interpretive system which is available to users in both batch and conversational modes of operation.

Familiarity with the LISP language is assumed throughout the remainder of this manual. The following are suggested references:

Friedman, D. P., THE LITTLE LISPER, (Science Research Associates, Menlo Park, California, 1974).

Siklossy, L., LET'S TALK LISP, (Prentice-Hall, Inc., Englewood Cliffs, New Jersey, 1975).

Weissman, C., LISP 1.5 PRIMER, (Dickenson Publishing Co., Inc., Belmont, California, 1967).

McCarthy, J., LISP 1.5 PROGRAMMER'S MANUAL, (M.I.T. Press, Cambridge, Massachusetts, 1969).

2. USING LISP

2.1 RUNNING LISP

LISP is executable only as a background program under RT-11 and is invoked by the monitor RUN command: RUN LISP. The LISP interpreter responds with '*' signaling that it expects a standard format command string containing at most one output specification and multiple input specifications. Input specifications other than the first are considered to be LISP subsystems and are loaded without any echo to the output file starting with the highest channel. The default output and first input device is TT:. Any combination of the switches listed in the following table may occur after any specification, but if a switch appears more than once the earliest occurrence overrides the others. After the command line is successfully entered control is transferred to mainloop which identifies itself by printing 'Eval:'. At this point LISP expressions may be input as described in Section 2.2.

Those switches which may be dynamically altered by the user program have the corresponding atom's print name in the last column.

Switch	Meaning	Default	Atom
/A:NNNN	Allocate NNNN words for array space	0	
/G	Print garbage collection message	OFF	%G
/I	Inhibit recognition of C...R atoms	OFF	%I
/L:N	Listing control switch	1	%L

Switch	Meaning	Default	Atom
	Value=0 Only user output is printed 1 Input is echoed to output file 2 Input is echoed in internal format 3 Input is echoed with parenthesis count		
/M	Maximize usable core space by forcing the USR to swap if possible	OFF	
/O	Enable octal format for integer printout	OFF	%O
/R	Output special constructions in LISP readable format	OFF	%R
/S:NNNN	Allocate NNNN words for LISP stacks	1000.	
/T	Print EVAL timing message	OFF	%T
/W	Enable wide line (132 col.) output instead of TTY (80 col.)	OFF	%W
/X	Expert switch: allows primitive operations on atoms (CAR, CDR)	OFF	%X
/Z	Allow generation and legal usage of infinity	OFF	%Z

Bad switches and extra output specifications are ignored. Default file extensions for input and output are .LSP and .LST, respectively.

2.2 ENTERING PROGRAMS

2.2.1 INPUTTING LISP EXPRESSIONS IN CONVERSATIONAL MODE

Rub out any CTRL/U work as usual to delete the previous character and current line, respectively. In addition, since S-expressions may extend over several lines any character in lexical class 8 (e.g., CTRL/X) may be used to delete the entire S-expression.

Note that atom names and numbers may not extend over line boundaries, whereas strings may.

2.2.2 LISP SUBSYSTEMS

A LISP subsystem is a canned set of function and constant definitions which constitutes an extension of the facilities provided by LISP, e.g., a subsystem may contain a set of application oriented primitives which can be utilized by the user to construct programs in some particular application area.

A subsystem is physically a file containing text in normal LISP format. It is specified in the command string or can be read in using the SYSIN function. When specified in the command string, it is read in, one S-expression at a time, and evaluated. No echo or printing of results is done. If it is read in under SYSIN, the printing is controlled by the state of the %L switch at the time SYSIN is called.

2.3 SYNTAX SUMMARY

2.3.1 LEXICAL CLASS DEFINITIONS

Each ASCII character belongs to a lexical class and that class determines the significance of the character. The following table lists the classes and their members. A function (CHLEX) is provided which changes the lexical class of a character but it should be used with care. Unpredictable results with respect to number representation will be generated if classes 2, 4, 6, and 13 are altered indiscriminately.

Lexical Class	Members	Comments
0	<CR> <LF> <FF> <VT>	End of line tokens
1	Letters except E and Q	Components of literal atoms
2	Digits 0-9	Components of literal atoms and numbers
3	! # $ % & * / : < = > ? @ \ _	Components of non-standard literal atoms
4	E	Decimal exponent indicator and component of literal atoms
5	Q	Octal radix indicator and component of literal atoms
6	+ -	Numeric sign and component of non-standard literal atoms
7	↑	Universal unary operator
8	ASCII codes ≥ 140, codes < 40 which are not in class 0 or 15	All characters not in another class
9	(	
10	)	
11	[	
12	]	
13	.	Decimal point or dotted pair indicator
14	,	Delimiter
15	<space> <tab>	Delimiters
16	"	QUOTE token
17	'	String token
18	;	Comment token
19	<empty>	

Class 8 is used only as an editing aid when input is from TT:. Typing any character from lexical class 8 followed by a lexical class 0 character causes the S-expression currently being input to be deleted.

Class 0 characters delimit atom print names and numbers but have no other significance.

Class 11 is equivalent to class 9 except that it "marks" the open parenthesis for matching by a class 12 character.

Class 12 characters are equivalent to an arbitrary number of class 10 characters in order to close all parentheses up to and including the next level "marked" open parenthesis, e.g., the following groups are equivalent S-expressions:

```
a)   (X) (X] [X) [X]
b)   (((X] (((X))) (((X)] [((X] ([X])
c)   (([X]] ((((X] ((((X))))
```

Class 15 characters delimit atom print names and numbers the same as class 14; however, whereas multiple class 14 characters may not occur any number of class 15 characters may occur (with or without a single class 14 character).

Class 16 is provided as a shorthand for quoting S-expressions on input. "X is equivalent to (QUOTE X) where X represents any S-expression.

Class 17 is used to delimit the beginning and end of a string. All classes with the exception of 7 (and 17) lose their significance when they occur within the string field.

Class 18 is used to mark the start of a comment field which extends to the end of the line. All classes (except 0) lose their significance within a comment field. (Comments are set up just as in MACRO).

Class 19 is a special class which is initially empty. Its members act like class 1 members except that they are also delimiters; e.g., if ! is put in class 19 then the following S-expression (HELP!) would be represented as a list of two atoms, the first being HELP and the second !. If ! were not in class 19 then the list would consist of only one atom, HELP!.

2.3.2 PARTIAL SYNTAX DEFINITION

2.3.2.1 ATOMS

There are three types of atoms in this implementation: literal atoms, numerical atoms, and string atoms. Literal atoms may contain an arbitrary number of characters (the print name must be input on a single line) from classes 1 to 6 except that they must not begin with a class 2 character. Non-standard literal atoms are formed the same way as literal atoms but any character in classes 7-19 may be included in the name if it is preceded by a class 7 character.

Numerical atoms are either integer or floating point. Integers are formed with class 2 characters (octal integers are formed with digits 0-7 terminated by a Q) optionally preceded by a class 6 character. If an integer is larger than 32767 (or smaller than -32768) it is automatically converted to floating point (octal integers may not be more than 6 digits--the least significant 16 bits are kept). Floating point atoms are formed with classes 2, 4, 6, and 13 in the standard manner. Any numbers too large to be represented in the normal floating point format are represented as infinity (infinity prints as 1E999) if that option is selected. Whenever an ambiguity arises with respect to the interpretation of "." a numerical interpretation is attempted first;

e.g., (A.5) is equivalent to (A 0.5) not (A . 5)

(+.B) is a dotted pair, (+.0) is a list of one floating point atom whose value is 0.

Numbers may not extend over line boundaries.

Integers are terminated by classes 0, 1, 3, 6-12, 14-19. (Class 5 terminates the number if a non-octal digit was scanned; else any character following the class 5 character terminates the octal number field.) Floating point numbers are terminated by any character which would violate the format rules (e.g., a second decimal point, a second sign, a second E, etc.).

String atoms are formed by a class 17 character followed by an arbitrary number of characters from classes 1-6, 8-16, 18, 19. Line boundaries are ignored within a string field. Characters from classes 7 and 17 may be included if preceded by a class 7 character.

Examples: Literal Atoms

legal constructions	illegal constructions
ANYNUMBEROFCHARS	9CANTSTARTLITERALATOM
↑31SOK prints as 31SOK	
%$!?	
↑A is the same atom as A	
↑1234 prints as 1234	
+E5	+5E

Numerical Atoms

legal constructions	illegal constructions
773100Q (truncates to 173100(8))	9Q
-347Q	1234567Q
+74932 (converted to floating point)	+5E
-3.7E+4	-3.7E 4 (two atoms)
-1E1	-E1

All literal atoms are represented uniquely in memory. All references to a given literal atom refer to the same memory location. System atoms (indicators, APVALS, and functions) should not be used for anything but their intended purpose (e.g., do not use T as a program variable).

2.3.2.2 S-EXPRESSIONS

The domain of LISP is the set of S-expressions. The most primitive S-expressions are atoms. Atoms may be combined into more complex structures, the "unit" of "structure" being the "dotted pair". The dotted pair consists of a "CAR" part and a "CDR" part. It is represented on input and output by a character from class 9 or 11, followed by the CAR S-expression, followed by a character from class 13, followed by the CDR S-expression, followed finally by a character from class 10 or 12. (Note that this constitutes a recursive definition for non-atomic S-expressions, i.e., an S-expression is either an atom or a dotted pair of two S-expressions.) An arbitrary number of characters from class 15 or 0 may optionally separate the components of a dotted pair on input. (A single space is used on output.)

The simplest dotted pair is one in which the CAR and CDR parts are atomic, e.g.,

Dotted Pairs

legal constructions	illegal constructions
(A.B)	(.A)
(X.NIL)	(B.)
(B.((A.A).B))	.A
(1 . 2)	(1.2) (this is a list of one numeric atom)
(A. (B.C))	(A.B.C)

Dotted pairs whose only atomic CDR part is the atom NIL occur frequently and thus have a shorthand representation as "lists". The syntax for lists is shown below along with the dotted pair equivalents:

(A.NIL) is equivalent to the list (A)

(A.(B.NIL)) is equivalent to the list (A B) or (A,B)

(A.(B.(C.NIL))) is equivalent to the list (A B C) or (A,B,C), etc.

((A.B).((C.D).NIL)) is equivalent to the list ((A.B) (C.D))

((A.NIL).((B.NIL).NIL)) is equivalent to the list ((A) (B))

The S-expressions in a list may be separated by an arbitrary number of characters from classes 0 or 15 and at most one character from class 14.

S-expressions are in general not stored uniquely; i.e., if the dotted pair (A.B) is found several times each will be stored in a different memory location.

A hybrid shorthand format is also available for dotted pairs as shown below (useful for dotted pairs which are "almost" lists)

(A.(B.(C.D))) may also be input as (A B C.D)

or (A,B,C.D)

where A B C D stand for any S-expressions.

The semantics of LISP functions often require lists of the form (QUOTE S), S being any S-expression. A special shorthand is available to input lists of this form. A character from class 16 followed by any S-expression is equivalent to a list whose CAR is QUOTE and whose CDR is a list containing the S-expression.

```
e.g., "(ANYTHING) is converted on input to (QUOTE (ANYTHING))
      "B          is converted on input to (QUOTE B)
```

Output of composite S-expressions is always in list or hybrid format. A new line is started if the current line is full or if the next atom print name won't fit on it.

3. FUNCTION DEFINITIONS

3.1 FUNCTION TYPES

There are four function types in LISP identified by the indicators EXPR, SUBR, FEXPR, and FSUBR. Functions of type EXPR and FEXPR are stored in memory as list structures representing LAMBDA expressions to be executed in an interpretive manner. Functions of type SUBR and FSUBR are stored as machine code and executed directly by the computer when called. User functions are either type EXPR or FEXPR while system functions are type SUBR or FSUBR. Functions of type EXPR and SUBR take a specified number of arguments and an error will result if an incorrect number is supplied. Functions of type FEXPR and FSUBR, sometimes called "special forms", take an arbitrary number of arguments. Furthermore, these arguments are unevaluated when passed to the function. When writing FEXPR functions, the user should be aware that two arguments are actually supplied; the first is a list of unevaluated arguments for the function; the second is the association list (ALIST) as it exists at the time of the function call.

All LISP functions return exactly one value. The range of this value depends upon the classification of the function as a normal function, pseudofunction, or predicate. Normal functions calculate a result which depends on the arguments which were input. Pseudofunctions perform a side effect operation (i.e., changing some internal data structures of the LISP system, printing, etc.) and return a value which may or may not bear any relation to the arguments given it. Predicates are functions which return a truth value that depends on some relationship which holds among the arguments. LISP represents false by the atom NIL, while truth is represented by anything other than NIL. Unless otherwise specified LISP predicates return the specific atom *T* as their value for truth.

Some pseudofunctions modify the structure of existing S-expressions and should be used with care. In many cases a portion of one S-expression may be shared by several other S-expressions. Thus modifying that portion will change all the S-expressions which share that portion.

3.2 NOTATION

Throughout this section function descriptions will be given using EVAL notation. Angle brackets are used to enclose the mnemonic for the generic argument type which is expected by the function. For SUBRS the argument class refers to the evaluated arguments that are passed to the function. For FSUBRS the argument class refers to the unevaluated arguments as passed to the function. Repeated arguments or an indefinite number of arguments will be represented by an ellipsis (...). If an argument supplied to a function does not belong to the generic type expected by the function unexpected results may occur or a LISP error may be generated. The behavior of the functions for unexpected argument types is also described for those functions which can detect the error. (Some functions have no way to verify the argument type but assume it is well formed. An ill-formed argument is then usually detected only after several nested calls to additional functions.)

Generic mnemonic	Description
<ATOM>, <A>, <A1>, etc.	The argument must be an atom, either a literal atom, numeric atom, or string atom.
<BOOL>	The argument may be any S-expression but will be interpreted as a truth value: NIL is false, anything else is true.
<EXP>, <E>, <E1>, etc.	The argument must be some LISP expression which can be evaluated by EVAL.
<FCTN>, <FCT>, <F1>, etc.	The argument must be a LISP function, e.g., LAMBDA expression or function name.
<LAT>	The argument must be a list of literal atoms.
<LIST>	The argument must be a list (or NIL).
<LITATM>	The argument must be a single literal atom.
<NAT>	The argument must be some non-atomic S-expression.
<NUMBER>, <N>, <N1>, etc.	The argument must be a numeric atom.
<SEXPR>, <S>, <S1>, etc.	The argument may be any arbitrary S-expression.

The function descriptions appear in groupings according to complexity and usage and logical relation. An alphabetic index of system atoms (which includes all system functions) appears in Section 4.

The LAMBDA expressions accompanying many of the function descriptions should not be taken too literally. In most cases recursion is suggested, whereas function evaluation is actually iterative.

3.3. FUNCTIONS

3.3.1 ELEMENTARY FUNCTIONS AND PREDICATES

(ALPHAP <ATOM1> <ATOM2>)

PREDICATE; SUBR

ALPHAP compares the print names of its arguments and returns true if the first argument alphabetically precedes the second argument. It returns false if the second precedes the first or if they are the same. The ASCII code sequence is used to determine the alphabetization. ALPHAP can be used to compare literal atoms and string atoms. False is returned if either argument is numeric or non-atomic.

(ATOM <S>)

PREDICATE; SUBR

ATOM returns true if its argument is either a literal atom, numeric atom, or string atom. It returns false if the argument is any other S-expression.

(CAR <NATS>)

NORMAL; SUBR

CAR returns the left part of the dotted pair argument (the contents of the CAR field). In list terms it returns the first element of the list. If the argument is atomic an error is generated unless the "expert" switch %X is on. (In the case of a literal atom, a string atom is returned whose value is the atom's print name. Numeric or string atoms are returned identically.)

(CDR <NATS>)

NORMAL; SUBR

CDR returns the right part of the dotted pair argument (the contents of the CDR field). In list terms it returns the remainder of the list after the first element is deleted. If the argument is atomic an error is generated unless the "expert" switch %X is on. (In the case of a literal atom, the property list (without the atom's print name) of the atom is returned. In the case of a numeric or string atom NIL is returned.)

(CAAR <NATS>)

(CADR <NATS>)

(CDAR <NATS>)

(CDDR <NATS>)

(CAAAR <NATS>)

⋮

NORMAL; SUBR

Multiple CAR-CDR functions are allowed and may contain an arbitrary number of A's and D's (subject to the usual restriction that the print name must be contained on a single input line). Evaluation is from right to left, i.e., (CADDR <NATS>) = (CAR (CDR (CDR <NATS>))).

(One SUBR is used to implement all variations of multiple CAR-CDR operations. This SUBR actually scans the currently associated print name to perform the CAR-CDR operations. The atoms CAAR, CADR, CDDR, ..., do not actually exist in the system until they are read in and recognized by READ.)

(CONS <S1> <S2>)

NORMAL; SUBR

CONS builds a new composite S-expression. It constructs the dotted pair (<S1> . <S2>) by obtaining a new cell from the available free cell list and placing a pointer to <S1> in the CAR field and a pointer to <S2> in the CDR field.

(Note: (CAR (CONS S1 S2)) = S1 and (CDR (CONS S1 S2)) = S2 and although (CONS (CAR S) (CDR S)) is "EQUAL" to S it is not "EQ" to S. It is a new cell at a different address.)

(EQ <S1> <S2>)

PREDICATE; SUBR

EQ returns true if its two arguments share the same memory location and false otherwise. Literal atoms are stored uniquely in LISP so that their equality may be determined by simple comparison of machine addresses. Numbers, strings, and lists are not stored uniquely so EQ will return false unless the two arguments are physically the same.

(EQN <S1> <S2>)

PREDICATE; SUBR

EQN is similar to EQ except that it also works for numeric or string atoms as well. EQN will return true if its arguments are EQ or if they are both numeric atoms which have the same value. (They may be numbers of different types; for mixed types the integer is converted to real before comparison. Comparison of floating point numbers has the same dubious merit as in FORTRAN.) In addition EQN will return true if both arguments are strings containing the same characters, or if one argument is a literal atom and the other is a string atom containing the same characters as in the literal atom's print name. In all other cases EQN returns false.

```
(EQUAL <S1> <S2>)             (LAMBDA (S1 S2)     (COND
                                      ((ATOM S1) (EQN S1 S2))
PREDICATE; SUBR                       ((ATOM S2) F)
                                      ((EQUAL (CAR S1)(CAR S2))
                                                 (EQUAL (CDR S1)(CDR S2)))
                                      (T F)      ))
```

EQUAL is used to do similar comparisons involving general S-expressions. EQUAL returns true if both arguments are equivalent S-expressions. S-expressions are equivalent if they are both atoms and are EQN, or if they are composed of the same atoms in the same corresponding positions.

(GRADP <S1> <S2>)

PREDICATE; SUBR

GRADP returns true if its first argument resides at a lower memory address than its second argument. It returns false if the first argument resides at the same or higher address than the second argument. This function can be used as an arbitrary but consistent ordering predicate for literal atoms since a given literal atom will always reside at the same address during the course of a LISP run (unless it is removed from the OBLIST via REMOB).

(LIST <E1> <E2> ... <E-N>)

NORMAL; FSUBR

LIST takes an arbitrary number of arguments and constructs a new list such that (EVAL <E1>) is the first element (EVAL <E2>) the second, etc. If no arguments are given the result is NIL.

(Note: As an FSUBR, LIST receives unevaluated arguments. It evaluates them and returns a list of the results. Contrast this with (EVLIS (<E1> <E2> ...)). EVLIS receives one argument which has already been evaluated. EVLIS then evaluates each element of its list type argument and returns a list of the results, e.g.,

```
EVAL:                                     EVAL:
   (LIST A B C D) is equivalent to           (EVLIS "(A B C D))
                  and also equivalent to  EVAL:
                                             (CONS A (CONS B
                                                (CONS C (CONS D NIL))))
```

```
(MEMBER <S> <LIST>)          (LAMBDA (S L)      (COND
                                     ((NULL L) NIL)
NORMAL; SUBR                         ((EQUAL S (CAR L)) L)
                                     (T (MEMBER S (CDR L)))   ))
```

MEMBER searches the top level of <LIST> for the first occurrence of an element EQUAL to <S>. If such an element is not found the value of MEMBER is NIL. If it is found the value of MEMBER is the remainder of the list beginning with <S>.

(MEMBER uses internal calls to CAR and CDR to search the list. If the second argument is not a list and no element is equal to <S>, an error will be generated when trying to take the CAR of the final CDR atom.)

(MEMQ <S> <LIST>)

NORMAL; SUBR

MEMQ is identical to MEMBER in all respects except that it uses EQ rather than EQUAL to test for the equality of <S> with an element of <LIST>.

(NOT <BOOL>)

(NULL <S>)

PREDICATE; SUBR

NULL returns true if its argument is the atom NIL. It returns false for all other S-expressions. (NULL is the same function as the logical negation function NOT.)

(NUMBERP <S>)

PREDICATE; SUBR

NUMBERP returns true if its argument is a number (fixed point or floating point). It returns false if its argument is any other S-expression.

(QUOTE <S>)

NORMAL; FSUBR

The value of QUOTE is <S>. It is essentially a do-nothing function designed to prevent evaluation of its argument. In LISP functions, atoms are usually evaluated as variables and lists as function calls. When QUOTE precedes an S-expression, it is effectively a signal to the EVAL interpreter that the S-expression represents actual data to the function and is not to be evaluated. Like any FSUBR the arguments are not evaluated when passed to QUOTE. QUOTE merely returns the unevaluated first argument as its value.

(Since QUOTE occurs frequently and its function is so trivial, it is not actually implemented as an FSUBR. The atom QUOTE is used as an indicator and specifically recognized by EVAL. In particular, there is no FSUBR indicator or value on the property list of QUOTE.)

(RPLACA <NATS> <S>)

PSEUDOFUNCTION; SUBR

RPLACA replaces the CAR field of its first argument with a pointer to its second argument. The value of RPLACA is its first argument which has been physically modified. (Contrast this with (CONS <S> (CDR <NATS>)) which forms an equivalent S-expression result without modifying <NATS>. Note, however, that RPLACA can be used to create circular structures. These circular structures could cause infinite loops in certain processes such as printing.) If the first argument is atomic the value of RPLACA is its first argument unchanged (i.e., the call is ignored).

(RPLACD <NATS> <S>)

PSEUDOFUNCTION; SUBR

RPLACD replaces the CDR field of its first argument with a pointer to its second argument. The value of RPLACD is its first argument which has been physically modified. (Contrast this with (CONS (CAR <NATS>) <S>) which forms an equivalent S-expression result without modifying <NATS>. Note, however, that RPLACD can be used to create circular lists. Circular lists should be used with extreme caution since they could cause infinite loops in certain processes such as printing.) If the first argument is atomic the value of RPLACD is its first argument unchanged (i.e., the call is ignored).

```
(SET <LITATM> <S>)       (LAMBDA (L S)      (COND
                                 ((SASSOC L "ALIST NIL)
PSEUDOFUNCTION; SUBR                     (RPLACD (SASSOC L "ALIST NIL) S))
                                 (T (PUT L "APVAL S))    ))
```

SET is the principal value assignment function. The value of SET is its second argument. If <LITATM> has an associated value on the ALIST it is changed to <S>. If it is not on the ALIST it is made a global by putting <S> on its property list with the indicator APVAL. (When evaluating a literal atom, EVAL first checks for a global value and if none exists it then checks for a binding on the ALIST.)

A special construction exists for altering the evaluation context: (SET (QUOTE (ALIST)) <S>) makes <S> the new association list. (Refer to MAINLOOP, EVAL, APPLY, and ALIST.) The user should be aware that no protection is provided that would prevent changing the value of the system globals T, F, OBLIST, NIL, *T* (such changes would be catastrophic).

(SETQ <LITATM> <EXP>)

PSEUDOFUNCTION; FSUBR

SETQ is similar to SET except the first argument is not evaluated.

```
e.g.,      EVAL:                                  EVAL:

               (SETQ A S) is equivalent to           (SET "A S)
```

(STRINGP <S>)

PREDICATE; SUBR

STRINGP returns true if its argument is a string. It returns false if it is any other S-expression.

3.3.2 LIST MANIPULATION FUNCTIONS

```
(APPEND <LIST> <S>)    (LAMBDA (L S)      (COND
                               ((NULL L) S)
NORMAL; SUBR                   (T (CONS (CAR L) (APPEND (CDR L) S)))   ))
```

APPEND is usually used to concatenate two lists. The first argument is copied and its terminal NIL is replaced with a pointer to <S>. (APPEND uses internal calls to CAR and CDR. If the first argument is not a list, then an error will be generated when trying to take the CAR of the final CDR atom.)

(CONC <EXP1> <EXP2> ...)

PSEUDOFUNCTION; FSUBR

CONC is similar to NCONC in that it concatenates its arguments into one list. However, CONC accepts an arbitrary number of (unevaluated) arguments, evaluates them, and then concatenates them. The value of CONC is its evaluated first argument with its final CDR modified.

```
       EVAL:                                      EVAL:

          (CONC A B C)    is equivalent to        (NCONC A (NCONC B C))
```

(Refer to the cautions under NCONC and RPLACD.)

(COPY <S>)

NORMAL; SUBR

COPY returns a new S-expression which is equivalent to its argument but which uses different cells. Dotted pairs, numbers, and strings are copied completely. Literal atoms are not copied (they are stored uniquely and always reside in the same memory location).

```
(EFFACE <S> <LIST>)        (LAMBDA (S L)      (COND
                                   ((NULL L) NIL)
PSEUDOFUNCTION; SUBR               ((EQUAL (CAR L) S) (CDR L))
                                   (T (RPLACD L (EFFACE S (CDR L))))   ))
```

EFFACE removes the first occurrence of <S> as an element of <LIST> and returns its modified second argument.

Contrast EFFACE with the following function which produces an equivalent list result without modifying the existing S-expressions.

```
     (DELETE (LAMBDA (S L)      (COND
                     ((NULL L) NIL)
                     ((EQUAL (CAR L) S) (CDR L))
                     (T (CONS (CAR L) (DELETE S (CDR L))))    ))
```

(EFFACE uses internal calls to CAR and CDR to search the list. If the second argument is not a list and no element is equal to <S>, an error will be generated when trying to take the CAR of the final CDR atom.)

```
(LENGTH <S>)                  (LAMBDA (S)      (COND
                                      ((ATOM S) 0)
NORMAL; SUBR                          (T (ADD1 (LENGTH (CDR S))))    ))
```

LENGTH returns a fixed point numeric atom whose value is equal to the number of CAR elements in its argument. If <S> is atomic it returns 0. If <S> is a list it returns the number of elements in the list. If <S> is not a list then length ignores the final CDR atom and counts the elements as if it were a list.

```
(NCONC <LIST> <S>)            (LAMBDA (L S)      (COND
                                      ((NULL L) S)
PSEUDOFUNCTION; SUBR                  ((NULL (CDR L))(RPLACD L S))
                                      (T (RPLACD L (NCONC (CDR L) S)))    ))
```

NCONC joins its two arguments into a single S-expression by modifying its first argument. The value of NCONC is its first argument with the terminal NIL replaced by a pointer to <S>. Contrast NCONC with APPEND, which yields an equivalent result without modifying existing S-expressions. (Refer to the cautions under RPLACD.)

(NCONC uses internal calls to CAR and CDR. If the first argument is not a list then an error will result.)

```
(PAIR <LIST1> <LIST2>)       (LAMBDA (L1 L2)     (COND
                                     ((OR (NULL L1)(NULL L2)) NIL)
NORMAL; SUBR                         (T (CONS (CONS (CAR L1)(CAR L2))
                                        (PAIR (CDR L1)(CDR L2))))   ))
```

PAIR creates a new list whose length is equal to the length of the shorter of its two arguments lists. The value of PAIR is a list of dotted pairs, the CAR of each dotted pair being an element of <LIST1> and the CDR being the corresponding element of <LIST2>.

```
EVAL:

   (PAIR "(A B C)"(D E F G))

VALUE IS ...

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

(PAIR uses internal calls to CAR and CDR. If either argument is not a list then an error will result.)

```
(REVERSE <LIST>)   (LAMBDA (L)     (COND
                           ((NULL L) NIL)
NORMAL; SUBR               (T (NCONC (REVERSE (CDR L)) (CONS(CAR L) NIL)))   ))
```

REVERSE returns a new list whose elements are the same as in its argument list, except that the top level order is reversed. Sublists are not reversed.

```
EVAL:

   (REVERSE "(A (B C)(D E) F))

VALUE IS ...

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

(REVERSE uses internal calls to CAR and CDR. If its argument is not a list an error will result.)

(REVERSIP <LIST>)

PSEUDOFUNCTION; SUBR

REVERSIP returns an S-expression equivalent to that returned by REVERSE except that <LIST> is modified to effect the reversal.

(REVERSIP uses internal calls to CDR. If its argument is not a list an error will result.)

(SIZE <S>)

NORMAL; SUBR

SIZE returns a fixed point numeric atom which gives the number of actual memory cells which are needed to represent <S> exclusive of those required to store the atoms themselves.

e.g., (SIZE <ATOM>) = 0

(SIZE <LAT>) = (LENGTH <LAT>)

(SIZE <S>) $\leqslant$ (SIZE (COPY <S>)) since copy does not make use of common sub S-expressions.

(SUBLIS <LIST> <S>)

NORMAL; SUBR

```
(LAMBDA (L S)    (COND
        ((NULL L) S)
        (T (SUBLIS (CDR L)
           (SUBST (CDAR L)(CAAR L) S)))   ))
```

SUBLIS receives a list of dotted pairs (such as the output of PAIR) as its first argument. The value of SUBLIS is a new S-expression obtained from <S> by substituting the right part of each dotted pair for every occurrence of the left part (the original <S> is not modified).

```
EVAL:

   (SUBLIS "((A. 1)(B. XTRA)(C Y Z)) "(A (B A C) C))

VALUE IS ...

   (1 (XTRA 1 (Y Z)) (Y Z))
```

(SUBLIS uses internal calls to CAR and CDR. If its first argument is not a list of dotted pairs an error will result.)

```
(SUBST <S1> <S2> <S3>)      (LAMBDA (S1 S2 S3)      (COND
                                    ((EQUAL S2 S3) S1)
NORMAL; SUBR                        ((ATOM S3) S3)
                                    (T (CONS (SUBST S1 S2 (CAR S3))
                                             (SUBST S1 S2 (CDR S3))))   ))
```

SUBST returns a new S-expression obtained from <S3> by substituting <S1> for each occurrence of <S2> within <S3>. (The original <S3> is not modified.)

```
    EVAL:
       (SUBST "NEW "OLD "(OLD SHOES ((MY OLD (OLD) HAT)) NEW NOSE))
    VALUE IS ...
       (NEW SHOES ((MY NEW (NEW) HAT)) NEW NOSE)
```

3.3.3 EVALUATION SEQUENCE CONTROL FUNCTIONS

(AND <EXP1> <EXP2> ...)

PREDICATE; FSUBR

AND evaluates its arguments one at a time from left to right until it encounters a null result or the end of the argument list. If all the arguments evaluate to non-null results the value of AND is true. If any argument evaluates to NIL no further arguments are evaluated and the value of AND is NIL. If no arguments are supplied to AND the result is true.

(APPLY <FCTN> <LIST>)

NORMAL; SUBR

The value of APPLY is the result of applying the function <FCTN> to the list of arguments <LIST>. The first argument to apply must be either a LAMBDA or LABEL expression or the name of a function of type EXPR or SUBR.

(Refer to appendix A for an outline of the APPLY implementation.)

(COND <LIST1> <LIST2> ...)

NORMAL; FSUBR

Each argument to COND is a list of one or more expressions. COND proceeds by evaluating the first expression of the first list. If the first expression evaluates to NIL then COND proceeds to the next list (leaving the remaining expressions of the first list unevaluated) and evaluates its first expression. COND continues in this fashion until the first expression of one of the argument lists evaluates to a non-null result. In this case COND evaluates the remaining expressions in the list and returns the value of the last one. (If none follow the first, the value of COND is NIL.) No further arguments of COND are evaluated. If no first expressions evaluate to non-null results or if no arguments are given to COND the value returned by COND is NIL.
(COND uses internal calls to CAR, CDR, and PROGN to search and evaluate the lists of expressions. If the selected <LIST> is not actually a list, an error will result within PROGN.)

(EVAL <EXP>)

NORMAL; SUBR

EVAL returns the result obtained by evaluating the evaluated argument passed to it.

```
                    EVAL:
                       (EVAL (QUOTE F))
                    VALUE IS ...
                       NIL
```

The top level function EVAL evaluates (QUOTE F) to F and passes it to the SUBR EVAL which evaluates F to NIL. Evaluation takes place within the current context (as defined by the association list, ALIST). (Refer to appendix A for an outline of the EVAL implementation.)

```
(EVALQUOTE <FCTN> <LIST>)    (LAMBDA (FN ARGS)     (COND
                                     ((OR (GET FN "FEXPR) (GET FN "FSUBR))
NORMAL; SUBR                              (EVAL (CONS FN ARGS)))
                                     (T (APPLY FN ARGS))    ))
```

EVALQUOTE is similar to APPLY in that its value is the result obtained by applying the function <FCTN> to the list of arguments <LIST>. However, EVALQUOTE will also properly handle functions of type FEXPR or FSUBR.

```
(EVLIS <LIST>)               (LAMBDA (L)      (COND
                                     ((NULL L) NIL)
NORMAL; SUBR                         (T (CONS (EVAL (CAR L))
                                              (EVLIS (CDR L))))    ))
```

EVLIS receives a list of expressions and returns a new list formed by evaluating each successive expression.

```
EVAL:
   (EVLIS "("A (CDR "(B C)) T))
VALUE IS ...
   (A (C) *T*)
```

(FUNCTION <FCTN>)

PSEUDOFUNCTION; FSUBR

The argument of FUNCTION is expected to be a function name or a LAMBDA expression. FUNCTION is required in certain very special cases to prepare an environment for a functional argument being passed to another function. Its value is the so-called FUNARG expression. In most cases when passing a functional argument QUOTE may be used interchangeably with FUNCTION. However, if any free variables exist within the function definition, then FUNCTION essentially causes the bindings at the time FUNCTION is entered to be used at the time the argument function is evaluated (unless a "non-local" SET was performed).

(Refer to appendix A for an outline of the implementation of FUNCTION with respect to APPLY and EVAL.)

(There is no actual FSUBR implemented for FUNCTION. The atom is used as an indicator and specifically recognized by EVAL.)

(GO <LITATOM>)

PSEUDOFUNCTION; FSUBR

GO is used to control the execution sequence within a PROG expression. The argument must be one of the atoms which are used as labels within the most recent PROG expression.

(Refer to the example under PROG.)

(OR <EXP1> <EXP2> ...)

PREDICATE; FSUBR

OR evaluates its arguments one at a time from left to right until it encounters a non-null result or the end of the argument list. If all the arguments evaluate to NIL the value of OR is NIL. If any argument evaluates to a non-null result no further arguments are evaluated and the value of OR is true. If no arguments are supplied to OR the result is NIL.

(PROG <LAT> <S1> <S2> <S3> ...)

PSEUDOFUNCTION; FSUBR

The PROG function is used to form iterative functions as opposed to recursive functions. The first argument is a list of atoms which serve as local variables within the scope of the PROG expression. (No explicit check is made that <LAT> is well-formed.) These variables are initialized to NIL by dotting each with NIL and appending them to the association list. The remaining arguments are either literal atoms which serve as labels or else expressions which serve as statements of a sequential programming language. The expressions are evaluated in order until a GO or RETURN function is encountered or the end of the argument list is reached. In the latter case the value of PROG is NIL. If RETURN is encountered then PROG returns as its value the result of the RETURN expression. If GO is encountered the sequential execution is continued starting with the expression following the atom (label argument) which is the argument to GO. GO and RETURN may be executed in other functions which are called from within a PROG expression.

e.g., the following is an iterative program to generate the reverse of a list (compare with the LAMBDA expression provided for REVERSE).

```
(LAMBDA (L) (PROG (U V)
            (SETQ U L)
     LABL   (COND ((NULL U) (RETURN V)))
            (SETQ V (CONS (CAR U) V))
            (SETQ U (CDR U))
            (GO LABL)   ))
```

(PROGN <EXP1> <EXP2> ...)

PSEUDOFUNCTION; FSUBR

PROGN is used to perform a simple sequence of operations where LISP normally allows only one. PROGN evaluates each of its arguments in turn and returns the value of the last one as the value of PROGN. If no arguments are given to PROGN the value returned is NIL.

(RETURN <S>)

PSEUDOFUNCTION; SUBR

RETURN causes LISP to exit the most recent PROG expression with <S> as the value of PROG.

(Refer to the example under PROG.)

(SELECT <EXP1> <LIST1> <LIST2> ... <LIST-N> <EXP2>)

NORMAL; FSUBR

SELECT is similar to COND in that it selectively evaluates its arguments until a test is satisfied at which time it terminates. Each argument to SELECT except the first and last are assumed to be lists of expressions. SELECT proceeds as follows: <EXP1> is evaluated. Then the first expression of the next <LIST> argument is evaluated. If the result is

not EQUAL to the value of <EXP1>, then none of the remaining expressions in the <LIST> are evaluated and the first expression of the next <LIST> is evaluated. SELECT continues in this way until a result is found equal to the value of <EXP1> or the end of the <LIST>s occurs. In the latter case the value of SELECT is the result of evaluating <EXP2>. In the former case the remaining expressions of the <LIST> are evaluated and the value of the last one is the result of SELECT. If none follow the first then the value of <EXP1>, which is equal to the value of the first expression in the list, is the result of SELECT.
(SELECT uses internal calls to CAR, CDR, PROGN, and EVAL to search and evaluate its arguments. If the selected <LIST> is not in fact a list an error in CAR or CDR will occur within PROGN.)

3.3.4 PROPERTY LIST MANIPULATION FUNCTIONS

Every literal atom in LISP has a property list associated with it. The property list contains items of information about, or properties of, the atom. Each property is characterized by an indicator (which is usually a literal atom itself, although this is not necessary) and an associated S-expression. The property list looks as follows:

```
        (<INDICATOR1> <PROPERTY1> <INDICATOR2> <PROPERTY2> ...)
```

The indicators EXPR, SUBR, FESPR, FSUBR, and APVAL have special significance within LISP and should be used with care.
(Every literal atom has an implied property, its print name (PNAME). The PNAME cannot be obtained or changed using GET, PROP, or PUT but must be handled by special functions.)

```
(DEFINE  LIST )               (LAMBDA (L) (DEFLIST L "EXPR))
PSEUDOFUNCTION; SUBR
```

DEFINE is used to put EXPR function definitions on the property lists of atoms (using the indicator EXPR). The argument is assumed to be a list of lists each of which is length two. The first element of the sublist is a literal atom which is the function name. The second element is the definition. The value of DEFINE is a list of the function names.

(See notes under DEFLIST.)

```
e.g.,    EVAL:

             (DEFINE "(

                      (FUNCTIONONE (LAMBDA (ETC) ETC))

                      (FUNCTIONTWO (LAMBDA (ETC) ETC))    ))

         VALUE IS ...

             (FUNCTIONONE FUNCTIONTWO)
```

```
(DEFLIST <LIST> <LITATM>)    (LAMBDA (L I)      (COND
                                     ((NULL L) NIL)
PSEUDOFUNCTION; SUBR                 (T (CONS (PUT (CAAR L) I (CADAR L))
                                               (DEFLIST (CDR L) I)))    ))
```

DEFLIST is used to simultaneously assign a new property to a number of different atoms. The first argument is a list of lists each of which is length two. The first element of the sublist is a literal atom to which the property (the second element of the sublist) is assigned with the indicator <LITATM> (the second argument). The value of DEFLIST is a list of all the atoms which have had a property assigned.

(DEFLIST uses internal calls to CAR, CDR, and PUT. If the first argument is ill-formed an error will most likely result within CAR or CDR.)

(Refer to notes under PUT.)

```
(GET <LITATM> <S>)           (LAMBDA (L I)      (COND
                                     ((PROP L I NIL) (CADR (PROP L I NIL)))
NORMAL; SUBR                         (T F)    ))
```

GET is the inverse of PUT in that it retrieves the value associated with the indicator <S> on the property list of <LITATM>. If <S> is not on the property list the value of GET is NIL.

(Refer to notes under PUT and PROP.)

(No check is made for <LITATM>. If it is not in fact atomic NIL will most likely be the result of GET.)

(PROP <LITATM> <S> <FCTN>)

NORMAL; SUBR

PROP is similar to GET in that it searches the property list of <LITATM> for the indicator <S>. However, if it is found PROP returns the entire property list beginning with <S>. If it is not found then <FCTN>, which must be a function of no arguments, is applied and the result is the value of PROP.

(PUT <LITATM> <S1> <S2>)

PSEUDOFUNCTION; SUBR

PUT searches the property list of <LITATM> for the indicator <S1>; if it is found the associated property value is changed to <S2>; if it is not found a list of the indicator <S1> followed by the property value <S2> is appended to the property list of <LITATM> (in this way an indicator appears at most once on any given property list). The value of PUT is <LITATM>.

(Note: PUT modifies the existing property list structure if the indicator is found.)

(REMPROP <LITATM> <S>)

PSEUDOFUNCTION; SUBR

REMPROP removes the property whose indicator is <S> from the property list of <LITATM>. (The indicator and associated value are removed from the property list by physically modifying the list structure.) The value of REMPROP is true if the indicator was found and deleted, and false if the indicator was not present to start with.

3.3.5 FUNCTIONALS

```
(MAP <LIST> <FCTN>)           (LAMBDA (L FN)     (COND
                                      ((NULL L) NIL)
PSEUDOFUNCTION; SUBR                  (T (PROGN (APPLY FN (LIST L))
                                                (MAP (CDR L) FN)))   ))
```

MAP applies the function <FCTN> to <LIST> and successive non-null CDRs of <LIST>. <FCTN> must describe a function of only one argument. The value of MAP is always NIL. MAP is always used to perform some side effect operation.

(If the first argument to MAP, MAPC, MAPCAR, MAPLIST, MAPCON is not a list the final CDR is ignored.)

```
(MAPC <LIST> <FCTN>)          (LAMBDA (L FN)      (COND
                                      ((NULL L) NIL)
PSEUDOFUNCTION; SUBR                  (T (PROGN (APPLY FN (LIST (CAR L)))
                                         (MAPC (CDR L) FN)))   ))
```

MAPC is similar to MAP except that <FCTN> is applied to the CAR of successive CDRs of <LIST>. The result of MAPC is always NIL.

```
(MAPCAR <LIST> <FCTN>)        (LAMBDA (L FN)      (COND
                                      ((NULL L) NIL)
NORMAL; SUBR                          (T (CONS (APPLY FN (LIST (CAR L)))
                                          (MAPCAR (CDR L) FN)))   ))
```

MAPCAR is similar to MAPC except that the result of MAPCAR is a list of all the results of applying <FCTN> to the CAR of successive CDRs of <LIST>.

```
e.g.,     EVAL:

            (MAPCAR "(A B C D) "(LAMBDA (X) (CONS X NIL)))

          VALUE IS ...

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

```
(MAPCON <LIST> <FCTN>)       (LAMBDA (L FN)    (COND
                                     ((NULL L) NIL)
PSEUDOFUNCTION; SUBR                 (T (NCONC (APPLY FN (LIST L))
                                        (MAPCON (CDR L) FN)))    ))
```

MAPCON is similar to MAP except that the result of MAPCON is a list formed by concatenating all the results of applying <FCTN> to successive CDRs of <LIST>. The result of each application of <FCTN> must therefore itself be a list. (Note that the list structure is modified by the concatenation.)

```
e.g.,     EVAL:

            (MAPCON "(A B C D) "(LAMBDA (X) (CONS X NIL)))

          VALUE IS ...

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

```
(MAPLIST <LIST> <FCTN>)      (LAMBDA (L FN)    (COND
                                     ((NULL L) NIL)
NORMAL; SUBR                         (T (CONS (APPLY FN (LIST L))
                                        (MAPLIST (CDR L) FN)))    ))
```

MAPLIST is similar to MAP except that the result of MAPLIST is a list of all the results of applying <FCTN> to successive CDRs of <LIST>.

```
e.g.,     EVAL:

            (MAPLIST "(A B C D) "(LAMBDA (X) (CONS X NIL)))

          VALUE IS ...

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

```
(SASSOC <S> <LIST> <FCTN>)     (LAMBDA (S L FN)    (COND
                                       ((NULL L) (APPLY FN NIL))
NORMAL; SUBR                           ((EQUAL S (CAAR L)) (CAR L))
                                       (T (SASSOC S (CDR L) FN))   ))
```

SASSOC searches its second argument, which must be a list of dotted pairs, for an element whose CAR is EQUAL to <S>. If such an element is found it is returned as the value of SASSOC. If none is found then <FCTN>, which must be a function of no arguments, is applied and the result is the value of SASSOC.

```
e.g.,     EVAL:

             (SASSOC "B "((A . 1) (B X) (C . 2) (B . 5)) NIL)

          VALUE IS ...

             (B X)
```

(SASSOC uses internal calls to CAR and CDR to search <LIST>. If it is ill-formed an error will result.)

```
(SEARCH <LIST> <FCT1> <FCT2> <FCT3>)    (LAMBDA (L F1 F2 F3)    (COND
                                           ((NULL L) (APPLY F3 (LIST NIL)))
NORMAL; SUBR                               ((APPLY F1 (LIST L))
                                              (APPLY F2 (LIST L)))
                                           (T (SEARCH (CDR L) F1 F2 F3))    ))
```

SEARCH applies <FCT1>, which must be a function of one argument, to <LIST> and successive CDRS of <LIST> until a non-null result is obtained or the end of <LIST> is reached. In the latter case <FCT3>, which must be a function of one argument, is applied to NIL and the result is the value of SEARCH. In the former case the value of SEARCH is the result of applying <FCT2>, which must also be a function of one argument, to the remaining portion of the list. Thus SEARCH is used to apply some function to a portion of a list depending on a condition which must be met by some elements of the list.

3.3.6 ARITHMETIC FUNCTIONS AND PREDICATES

Unless otherwise noted arithmetic functions will take either fixed point or floating point numbers and convert them if necessary in order to perform their specific operation. If integer overflow occurs the result of the computation is automatically converted to floating point. If floating point overflow occurs an error is generated unless legal operations with infinity are to be allowed. (Illegal operations with infinity are noted in the function descriptions.) A non-numeric argument to an arithmetic function will generate an illegal argument error unless otherwise noted. Functions of multiple arguments will resolve the mixed mode problem by converting integers to floating point and performing floating point operations.

(ADD1 <NUMBER>)

NORMAL; SUBR

ADD1 returns a number of the same type as its argument, if possible, whose value is <NUMBER> + 1.

(DIFFERENCE <N1> <N2>)

NORMAL; SUBR

DIFFERENCE returns a number whose value is <N1> - <N2>.

(∞ - ∞ is always illegal.)

```
(DIVIDE <N1> <N2>)          (LAMBDA (N1 N2) (LIST (QUOTIENT N1 N2)
                                  (REMAINDER N1 N2)))
```

NORMAL; SUBR

DIVIDE returns a list of two numbers, the first is (QUOTIENT <N1> <N2>) and the second is (REMAINDER <N1> <N2>).

(Refer to notes under QUOTIENT and REMAINDER.)

(FIX <NUMBER>)

NORMAL; SUBR

FIX returns an integer type atom (a copy of the argument is returned if it was already an integer). The integer returned is obtained by truncating the magnitude of <NUMBER>. An error is generated if the number is too large for representation as an integer.

```
e.g.,                       EVAL:
                               (FIX 4.9)
                            VALUE IS ...
                               4
                            EVAL:
                               (FIX -4.9)
                            VALUE IS ...
                               -4
```

(FIXP <S>)

PREDICATE; SUBR

FIXP returns true if its argument is a fixed point numeric atom and false if it is any other S-expression.

(FLOAT <NUMBER>)

NORMAL; SUBR

FLOAT returns a floating point numeric atom (a copy of the argument is returned if it was already a floating point atom) whose value is equal to <NUMBER>.

(FLOATP <S>)

PREDICATE; SUBR

FLOATP returns true if its argument is a floating point number. It returns false if it is any other S-expression.

(GREATERP <N1> <N2>)

PREDICATE; SUBR

GREATERP returns true if <N1> is algebraically greater than <N2>. It returns false if <N1> is less than or equal to <N2>. If either argument is floating point then the comparison is done in floating point.

(INFP <S>)

PREDICATE; SUBR

INFP returns true if its argument is either plus or minus infinity. It returns false if it is any other S-expression.

(LESSP <N1> <N2>)   (LAMBDA (N1 N2) (GREATERP N2 N1))

PREDICATE; SUBR

LESSP returns true if <N1> is algebraically less than <N2> and false if it is greater than or equal to <N2>. If either argument is floating point then the comparison is done in floating point.

(LOGAND <N1> <N2> ...)

NORMAL; FSUBR

LOGAND returns an integer atom which is obtained by evaluating its arguments and forming the bit by bit logical AND. Any floating point atoms are converted to integer prior to ANDing.

(LOGOR <N1> <N2> ...)

NORMAL; FSUBR

LOGOR returns an integer atom which is obtained by evaluating its arguments and forming the bit by bit logical inclusive OR. Any floating point atoms are converted to integer prior to ORing.

(LOGXOR <N1> <N2> ...)

NORMAL; FSUBR

LOGXOR returns an integer atom which is obtained by evaluating its arguments and forming the bit by bit logical exclusive OR. Any floating point atoms are converted to integer prior to ORing.

(LSHIFT <N1> <N2>)

NORMAL; SUBR

LSHIFT returns an integer atom obtained by performing a shift on the integer value of <N1>. <N2> is the shift count: if <N2> is positive the shift is end-around to the left by <N2> bits; if <N2> is negative the shift is end-off (with sign replication) to the right by - <N2> bits.

(MAX <N1> <N2> ...)

NORMAL; FSUBR

MAX evaluates all its arguments and returns a copy of the argument which is algebraically largest.

(MIN <N1> <N2> ...)

NORMAL; FSUBR

MIN evaluates all its arguments and returns a copy of the argument which is algebraically smallest.

(MINUS <NUMBER>)

NORMAL; SUBR

MINUS returns a numeric atom whose value is the negative of <NUMBER>.

(MINUSP <S>)

PREDICATE; SUBR

MINUSP returns true if its argument is a numeric atom whose value is less than zero. It returns false if it is any other S-expression.

(ONEP <S>)

PREDICATE; SUBR

ONEP returns true if its argument is a numeric atom whose value is 1 (or 1.0). It returns false if it is any other S-expression.

(PLUS <N1> <N2> ...)

NORMAL; FSUBR

PLUS evaluates all its arguments and returns a numeric atom whose value is their sum. (∞ - ∞ is always illegal.)

(QUOTIENT <N1> <N2>)

NORMAL; SUBR

QUOTIENT returns a numeric atom whose value is <N1>/<N2>. QUOTIENT always returns a floating point result (unless <N1> and <N2> are both integers and <N2> divides <N1>).

(0/0 and ∞/∞ are always illegal.)

(RANDOM <BOOL>)

NORMAL; SUBR

RANDOM returns a pseudo-random number between 0.0 and 1.0. If <BOOL> is NIL the sequence is reinitiated. Calls with <BOOL> non-null return the next number in the sequence.

```
(RECIP <NUMBER>)            (LAMBDA (N) (QUOTIENT 1 N))
```

NORMAL; SUBR

RECIP returns a numeric atom whose value is equal to the reciprocal of <NUMBER>. (This is of floating point type unless <NUMBER> = 1.)

```
(REMAINDER <N1> <N2>)       (LAMBDA (N1 N2) (DIFFERENCE N1
                              (TIMES N2 (FIX (QUOTIENT N1 N2)))))
```

NORMAL; SUBR

REMAINDER returns a numeric atom whose value is calculated in the usual manner:

REMAINDER = <N1> - integer part (<N1>/<N2>) * <N2>

(Due to the fact that FIX truncates the magnitude of its argument the remainder will always be the same sign as <N1>.)

REMAINDER will return a fixed point result if <N1> and <N2> are both fixed point, otherwise floating point. An error will be generated if the quotient is too large for integer format (i.e., >32767 or <-32768). (The following are undefined and illegal operations: (REMAINDER 0 0) (REMAINDER ∞ X) where X ≠ 0).

(SUB1 <NUMBER>)

NORMAL; SUBR

SUB1 returns a number whose value is <NUMBER> - 1.

(TIMES <N1> <N2> ...)

NORMAL; FSUBR

TIMES evaluates all its arguments and returns a numeric atom whose value is their product. ($0 \times \infty$ is always illegal.)

(ZEROP <S>)

PREDICATE; SUBR

ZEROP returns true if its argument is a numeric atom whose value is zero. It returns false if it is any other S-expression.

3.3.7 DEBUGGING AND ERROR PROCESSING FUNCTIONS

(ERROR <S>)

PSEUDOFUNCTION; SUBR

ERROR causes a recoverable error to occur. The argument will appear as part of an error message printed in response to this function. Continued execution of the current top level evaluation will be aborted unless an ERRSET function is in control.

e.g.,
```
                 EVAL:
                    (ERROR "MESSAGE)
                 VALUE IS ...
                    *****ERROR MESSAGE
```

(ERRSET <EXP> <BOOL1> <BOOL2>)

If no errors occur ERRSET is equivalent to:
(LAMBDA (E B1 B2) (CONS (EVAL E) NIL))

PSEUDOFUNCTION; SUBR

ERRSET is a function which allows LISP to recover from a recoverable error without aborting the top level evaluation in progress. If no error occurs during the evaluation of <EXP> (note that since ERRSET is a SUBR the arguments are evaluated prior to calling ERRSET; during this evaluation the current call to ERRSET offers no protection) then the value of ERRSET is a list whose single element is the result of evaluating <EXP>. If a recoverable error occurs the result of the most recent ERRSET is NIL. At the time the error occurs <BOOL1> controls printing of the error message itself--if it is false the message is not printed. <BOOL2> controls printing of the back trace--if it is false the back trace is not printed.

e.g.,

```
EVAL:
   (ERRSET " "A T T)
VALUE IS ...
   (A)
EVAL:
   (ERRSET "A T T)
VALUE IS ...
   *****ERROR A8 A
   (EVAL)
   NIL
EVAL:
   (ERRSET A T T)
VALUE IS ...
   *****ERROR A8 A
   (EVAL EVLIS EVAL MAINLOOP)
EVAL:
```

(KILL <S>)

PSEUDOFUNCTION; SUBR

KILL is similar to ERROR except it causes a non-recoverable (fatal) error which terminates the LISP run.

```
e.g.,                EVAL:

                        (KILL 'MESSAGE)

                     VALUE IS ...

                        *****KILLED MESSAGE

                        (MAINLOOP)

                        .
```

(LOOK <NUMBER>)

NORMAL; SUBR

LOOK returns an integer atom whose value is the contents of the memory location specified by <NUMBER>. (A trap to 4 will result if non-existent memory is referenced.) If <NUMBER> is odd it is decremented prior to being used as an address.

```
(TRACE <LAT>)                 (LAMBDA (L)      (COND
                                      ((NULL L) NIL)
PSEUDOFUNCTION; SUBR                  ((PUT (CAR L) 'TRACE 0)
                                         (TRACE (CDR L)))    ))
```

TRACE puts a counter (initialized to 0) on the property list of each atom in <LAT> (the value of TRACE is always NIL). After that time (and until an UNTRACE call removes the counter) a message is printed each time a function whose name appeared in <LAT> is called.

The message is of the form:

(<number>) LEVEL ARGUMENTS OF <NAME>

where <NAME> is the name of the function and <number> is the nesting level of the current call. Following the message the arguments are printed one per line (evaluated in the case of expressions and SUBRS--one unevaluated list of arguments in the case of FSUBRS and FEXPRS). Every function that is entered is also exited. At the time the function is exited a message of the following form is printed:

(<number>) LEVEL RESULT OF <NAME>.

On the following line is the actual value returned by the function. <number> indicates the nesting level and should be paired with the most recent occurrence of the same number in the "arguments of" message to compare the arguments and result of the <number>th level call to the function. As noted SUBRS and FSUBRS can be traced; however, internal calls are not traced. Furthermore, COND, PROG, QUOTE, LABEL, FUNCTION cannot be traced because they are handled specially.

```
(TRACESET <LAT>)              (LAMBDA (L)      (COND
                                      ((NULL L) NIL)
PSEUDOFUNCTION; SUBR                  (T (CONS (PUT (CAR L) 'TRACESET T)
                                         (TRACESET (CDR L))))   ))
```

TRACESET puts a flag on the property list of each atom in <LAT>. The value of TRACESET is the same <LAT>. The TRACESET flag on an EXPR or FEXPR function atom causes all SET or SETQ calls at the top level of a top level PROG call to be traced. A message of the form <variable> = <assignment> is printed for each such SET call encountered.

(UNDEF <S>)

PREDICATE; FSUBR

UNDEF returns true if its argument is a <LITATM> that does not have an APVAL or a binding on the association list. It returns false for all other situations.

(UNTRACE <LAT>)

PSEUDOFUNCTION; SUBR

UNTRACE removes the TRACE counter and indicator from the property list of each atom in <LAT>. The value of UNTRACE is always NIL.

(UNTRACESET <LAT>)

PSEUDOFUNCTION; SUBR

UNTRACESET removes the TRACESET flag from the property lists of each atom in <LAT>. The value of UNTRACESET is its unchanged argument.

3.3.8 MISCELLANEOUS FUNCTIONS

(ADDR <S>)

NORMAL; SUBR

ADDR returns an integer atom whose value is the address of the first cell of the argument.

(ALIST)

PSEUDOFUNCTION; SUBR

ALIST returns the current association list at the time ALIST is called. This shows all the bindings but not the APVALS of the literal atoms.

(FREE)

PSEUDOFUNCTION; SUBR

FREE returns an integer atom whose value is the number of free cells immediately available.

(GENSYM <LITATM>)

PSEUDOFUNCTION; FSUBR

GENSYM returns an entirely new literal atom each time it is called. The literal atom has a print name of XNNNNNN where X is a single letter obtained from <LITATM> and NNNNNN is a unique octal number. The letter used is the first letter of the <LITATM> print name and the number is the next number in the series for that letter. If no (legal) argument is specified to GENSYM the letter from the last call is used. If no call precedes a call with no argument (or illegal argument) then the letter G is used. The atoms created by GENSYM are not placed on the OBLIST. If an atom with the same print name is subsequently read in it will not be related to the GENSYM atom. In order to keep GENSYM names from being discarded some portion of existing active S-expressions or property lists must contain references to them.

(INTERN <STR>)

PSEUDOFUNCTION; SUBR

INTERN creates a <LITATM> whose print name is the value of the string atom <STR> and inserts it on the OBLIST. If an atom of the same print name already exists INTERN returns the existing atom rather than create a new one. The value of INTERN is a literal atom with a print name equivalent to <STR>. (INTERN modifies <STR> to a literal atom. NIL is returned if a bad argument has been specified.)

(LABEL <LITATM> <FCTN>)

PSEUDOFUNCTION; FSUBR

LABEL accomplishes almost the same thing as DEFINE except that it binds the function definition to the name on the association list rather than

incorporating it on the atom's property list. This is similar to the case for variables, which may have bindings on the ALIST or APVALs. For function atoms, the EXPR or FEXPR definition takes precedence over the LABEL binding in the same way that the APVAL supersedes the ALIST binding for variables.
(There is no FSUBR implemented for LABEL. It is an indicator specifically recognized by APPLY.)

```
(MAINLOOP)                  (LAMBDA () (PROG ()
                                    A (SETQ %ANS (PRINT (EVAL (READ))))
PSEUDOFUNCTION; SUBR                   (GO A)   ))
```

MAINLOOP is the actual driver for LISP and is LISP callable. While in control it expects to read an S-expression, evaluate it, and print it. It remains in control until an end-of-file on SYSIN is detected, at which time it exits with NIL to the caller. If there was no caller, it looks for an unfinished SYSIN file and if not found, it exits LISP. If the %T switch is on, a timing message is printed for each top level EVAL.

(RECLAIM)

PSEUDOFUNCTION; SUBR

RECLAIM causes a garbage collection to occur whether or not it is needed. The value of RECLAIM is always NIL.

(REMOB <LITATM>)

PSEUDOFUNCTION; SUBR

REMOB removes <LITATM> from the OBLIST. If no active lists refer to <LITATM> then the cells used to store it will be collected in the next garbage collection. In any case if an atom with the same print name is subsequently read in it will have no relation to the (previously) existing atom. The value of REMOB is always NIL.

(TEMPUS)

PSEUDOFUNCTION; SUBR

TEMPUS returns a floating point atom which represents the RT-11 system time in seconds.

3.3.9 INPUT/OUTPUT FUNCTIONS

(CLOSE <N>)

PSEUDOFUNCTION; SUBR

CLOSE releases channel <N> by closing the associated ENTER'd or LOOKUP'd file, releasing the handler if no other channel requires it, and releasing the I/O buffers. The value of CLOSE is always NIL.

(ENTER <STR>)

PSEUDOFUNCTION; SUBR

<STR> must be a legal file specification for a new output file. The new file is opened and writing to it is done via the OUTPUT function. If <STR> is illformed or if an attempt to open too many files is made an error is generated. The value of ENTER is an integer, the channel number assigned.

(A maximum of 9 channels may be open at any one time: SYSIN, SYSOUT, and 7 others.)

(INPUT <N>)

PSEUDOFUNCTION; SUBR

INPUT works just like READ except that the LOOKUP'd file associated with channel <N> is used instead of the current SYSIN. INPUT returns the next S-expression from channel <N>.

(LOOKUP <STR>)

PSEUDOFUNCTION; SUBR

<STR> must be a legal file specification for an already existing file to be used for input. The new file is opened and input is done via the INPUT function. If <STR> is illformed or if an attempt is made to open too many files an error is generated. The value of LOOKUP is an integer, the channel number assigned.
(A maximum of 9 channels may be open at any one time: SYSIN, SYSOUT, and 7 others.)

(OUTPUT <N> <S>)

PSEUDOFUNCTION; SUBR

OUTPUT works just like PRINT except that the ENTER'd file associated with channel <N> is used instead of the current SYSOUT. The value of OUTPUT is <S>.

(PRINT <S>)

PSEUDOFUNCTION; SUBR

PRINT causes its argument to be printed to the current SYSOUT. The value of PRINT is its unchanged argument. If the %R switch is on, all non-standard atoms and strings are printed in LISP readable format.

If the %O switch is on all integers are printed in octal format. A carriage-return/line-feed is done after <S> is printed.

(PRIN1 <S>)

PSEUDOFUNCTION; SUBR

PRIN1 is similar to PRINT except that the final carriage-return/line-feed is not done. This allows multiple S-expressions to be printed on a line.

(READ)

PSEUDOFUNCTION; SUBR

READ causes an S-expression to be read from the current SYSIN. All <LITATM>s are either found on the OBLIST or put there. The value of READ is the S-expression read in.

READ automatically recognizes atoms of the form CAAR, CADR, CDDR, CAAAR, ..., and when installing them on the OBLIST puts a special SUBR indicator on their property lists. This feature is inhibited by turning the %I switch on. The echoing features of READ are controlled by the %L switch. (There is never any echo or printing while reading or evaluating LISP subsystems.)

(SYSIN <STR>)

PSEUDOFUNCTION; SUBR

<STR> must be a legal file name specification for an exiting file on an input device. SYSIN causes <STR> to be immediately opened as the next system input file. Input comes from <STR> until an end-of-file is detected at which time the previous SYSIN file is used. SYSIN may

be used to read in LISP subsystems in which case the results of all the evaluations are written to the output file. An error is generated if <STR> is an illformed argument or if an attempt was made to open too many channels. SYSIN returns the channel number of the previous SYSIN.

(SYSOUT <STR>)

PSEUDOFUNCTION; SUBR

SYSOUT changes the current output file to <STR>. <STR> must be a legal output specification; a new file is entered on the output device. If <STR> is NIL then SYSOUT closes the current output file and reverts to the previous one. If <STR> is illformed or if an attempt was made to open too many channels an error is generated. In addition, if an attempt is made to close the final SYSOUT file by calling SYSOUT with NIL then an error will be generated and the file will remain open. It can only be closed by exiting LISP.

SYSOUT returns NIL or the channel number of the previous SYSOUT.

(TERPRI)

PSEUDOFUNCTION; SUBR

The value of TERPRI is NIL. It prints a carriage-return/line-feed to the output file. A call on PRIN1 followed by a call to TERPRI is equivalent to a call on PRINT.

3.3.10 STRING FUNCTIONS

Most string functions will perform correct operations using the print name of a literal atom as an argument. However, GENSYM atoms should not be used since they will usually not be handled recognizably.

(PNAME <ATOM>)

NORMAL; SUBR

PNAME returns a string whose value is the print name of a literal atom or the ASCII print image of a number. It returns NIL if argument is not atomic.

(POSITION <STR1> <STR2> <N>)

NORMAL; SUBR

POSITION searches <STR1> for the first occurrence of <STR2> starting with the <N>th character of <STR1>. It returns a number <M> such that <STR2> matches <STR1> beginning at the <M>th character of <STR1>. If no match is found or if arguments are illegal POSITION returns 0. (If <STR2> is the NULL string 0 is returned.)

(Strings made from GENSYM atoms not handled recognizably.)

(SEGMENT <STR> <N1> <N2>)

NORMAL; SUBR

SEGMENT returns a new string atom made up of the characters from the <N1>th position through the <N2>th position of <STR>. If <STR> is not a valid argument (i.e., a number or list) NIL is returned; if there are no characters between <N1> and <N2> (e.g., N2<N1) then the NULL string is returned.

(STRING <S1> <S2> ...)

NORMAL; FSUBR

After evaluating each argument STRING creates a new string atom by concatenating all its arguments in the following manner: Non-atomic

S-expressions as arguments are ignored; literal atom print names are concatenated along with strings; numbers are converted to characters by converting each number to an integer and truncating to 7 bits. Strings are limited to a length of 255 characters. (GENSYM atoms will not concatenate recognizably.)

(STRLEN <STR>)

NORMAL; SUBR

STRLEN returns an integer whose value is the number of characters in <STR>. NIL is returned if the argument is a number or list. (Characters in GENSYM atoms are counted incorrectly.)

(VALUE <STR>)

NORMAL; SUBR

VALUE returns the numeric atom which is represented by the STRING <STR>. (If the argument to VALUE is a number it is returned unchanged; if it is a list then NIL is returned.) If <STR> is not proper numeric syntax then NIL is returned. (If the resulting number is too large and the %Z switch is off then an error is generated.)

3.3.11 CHARACTER FUNCTIONS

(ADVANCE <BOOL>)

PSEUDOFUNCTION; SUBR

ADVANCE will advance the input pointer and read the next character of the input file if <BOOL> is NIL. If <BOOL> is non-null ADVANCE will back up the input pointer and read the previous character again

(leaving the pointer pointing to the previous character). Attempts to back up more than 6 characters will result in %EOF being returned. The value of ADVANCE is either the %EOF atom or an integer atom whose value is the ASCII code of the character read.
(In interactive mode the user should be aware that characters are not available to ADVANCE until a carriage-return is typed.)

(CHLEX <N1> <N2>)

PSEUDOFUNCTION; SUBR

CHLEX changes the lexical class of the character whose ASCII code is <N1> to lexical class <N2> (<N2> must be between 0-19 or the call is ignored). The change takes effect immediately. The value of CHLEX is the old lexical class of character <N1> (or NIL if the call was ignored). (Refer to Section 2.3.1 for cautions.) If <N2> is NIL, CHLEX returns the current lexical class of character <N1> without changing it.

(EXPLODE <ATOM>)

NORMAL; SUBR

EXPLODE returns a list of integer atoms representing the ASCII codes of the characters in the print name of <ATOM> if it is a literal atom or string. It returns NIL if the argument was a number or non-atomic expression.
(GENSYM atoms will be recognizably exploded.)

3.3.12 ARRAY FUNCTIONS

SET recognizes the form (SET "(X 1 3 9 2) "B) and attempts to interpret X as an array. If the atom X has the array property then the subscript list is evaluated, and the appropriate element is indexed and reset to point to B. When presented with (X 1 3 7) EVAL first looks for SUBR, FSUBR, EXPR, or FEXPR on the property list of X. It then looks for ARRAY and if found it evaluates the subscripts, indexes the appropriate element, and returns the associated value. If the ARRAY indicator is not found then an association on the ALIST is sought for X.

(ARRAY <LITATM> <LIST>)

PSEUDOFUNCTION; SUBR

ARRAY defines and allocates a new array whose name is <LITATM>. <LIST> is a list of dotted pairs which give the range of each subscript.

```
e.g.,              EVAL:
                        (ARRAY "X "((0 . 4)(-2 . 11)))
```

will allocate a 5 by 14 array, the first index ranges from 0 to 4 and the second from -2 to 11. Each element of the array can be a pointer to an arbitrary S-expression. ARRAY puts the indicator ARRAY with the value <LIST> on the property list of <LITATM>. ARRAY initializes each element to NIL. The value of ARRAY is NIL. If <LIST> is NIL, the ARRAY property of LITATM is removed and the space is reclaimed. (The space is then available to be allocated for another array or for I/O buffer and handler space.)

(CLARRAY <LITATM>)

PSEUDOFUNCTION; SUBR

<LITATM> must have an ARRAY property or an error is generated. CLARRAY sets all elements of the array to NIL. The value of CLARRAY is NIL.

4. SYSTEM ATOMS

The following alphabetical list includes all the system atoms which are present on the OBLIST after initialization. Notice that the command string switches G, I, L, O, R, T, W, X, Z are represented as system apvals %G, %I, %L, %O, %R, %T, %W, %X, %Z. These apvals are initialized to the values assigned to the command string switches and may be accessed or altered under program control. Notice also the existence of the atom %ANS; it always contains the result of the last top level evaluation. (Refer to the MAINLOOP function description.)

Atom	Type	Description
ADDR	SUBR	3.3.8
ADD1	SUBR	3.3.6
ADVANCE	SUBR	3.3.11
ALIST	SUBR	3.3.8
ALPHAP	SUBR	3.3.1
AND	FSUBR	3.3.3
APPEND	SUBR	3.3.2
APPLY	SUBR	3.3.3
APVAL	INDICATOR	
ARRAY	SUBR	3.3.12
ATOM	SUBR	3.3.1
CAR	SUBR	3.3.1
CDR	SUBR	3.3.1
CHLEX	SUBR	3.3.11
CLARRAY	SUBR	3.3.12
CLOSE	SUBR	3.3.9

Atom	Type	Description
CONC	FSUBR	3.3.2
COND	SUBR	3.3.3
CONS	SUBR	3.3.1
COPY	SUBR	3.3.2
DEFINE	SUBR	3.3.4
DEFLIST	SUBR	3.3.4
DIFFERENCE	SUBR	3.3.6
DIVIDE	SUBR	3.3.6
EFFACE	SUBR	3.3.2
ENTER	SUBR	3.3.9
EQ	SUBR	3.3.1
EQN	SUBR	3.3.1
EQUAL	SUBR	3.3.1
ERROR	SUBR	3.3.7
ERRSET	SUBR	3.3.7
EVAL	SUBR	3.3.3
EVALQUOTE	SUBR	3.3.3
EVLIS	SUBR	3.3.3
EXPLODE	SUBR	3.3.11
EXPR	INDICATOR	
F	APVAL	VALUE:=NIL
FEXPR	INDICATOR	
FIX	SUBR	3.3.6
FIXP	SUBR	3.3.6
FLOAT	SUBR	3.3.6

Atom	Type	Description
FLOATP	SUBR	3.3.6
FREE	SUBR	3.3.8
FSUBR	INDICATOR	
FUNARG	INDICATOR	
FUNCTION	INDICATOR	3.3.3
GENSYM	FSUBR	3.3.8
GET	SUBR	3.3.4
GO	FSUBR	3.3.3
GRADP	SUBR	3.3.1
GREATERP	SUBR	3.3.6
INFP	SUBR	3.3.6
INPUT	SUBR	3.3.9
INTERN	SUBR	3.3.8
KILL	SUBR	3.3.7
LABEL	INDICATOR	3.3.8
LAMBDA	INDICATOR	
LENGTH	SUBR	3.3.2
LESSP	SUBR	3.3.6
LIST	FSUBR	3.3.1
LOGAND	FSUBR	3.3.6
LOGOR	FSUBR	3.3.6
LOGXOR	FSUBR	3.3.6
LOOK	SUBR	3.3.7
LOOKUP	SUBR	3.3.9
LSHIFT	SUBR	3.3.6

Atom	Type	Description
MAINLOOP	SUBR	3.3.8
MAP	SUBR	3.3.5
MAPC	SUBR	3.3.5
MAPCAR	SUBR	3.3.5
MAPCON	SUBR	3.3.5
MAPLIST	SUBR	3.3.5
MAX	FSUBR	3.3.6
MEMBER	SUBR	3.3.1
MEMQ	SUBR	3.3.1
MIN	FSUBR	3.3.6
MINUS	SUBR	3.3.6
MINUSP	SUBR	3.3.6
NCONC	SUBR	3.3.2
NIL	APVAL	VALUE:=NIL
NOT	SUBR	3.3.1
NULL	SUBR	3.3.1
NUMBERP	SUBR	3.3.1
OBLIST	APVAL	VALUE:=LIST of hash buckets, i.e., contains all existing literal atoms
ONEP	SUBR	3.3.6
OR	FSUBR	3.3.3
OUTPUT	SUBR	3.3.9
PAIR	SUBR	3.3.2
PLUS	FSUBR	3.3.6
PNAME	SUBR	3.3.10

Atom	Type	Description
POSITION	SUBR	3.3.10
PRINT	SUBR	3.3.9
PRIN1	SUBR	3.3.9
PROG	FSUBR	3.3.3
PROGN	FSUBR	3.3.3
PROP	SUBR	3.3.4
PUT	SUBR	3.3.4
QUOTE	INDICATOR	3.3.1
QUOTIENT	SUBR	3.3.6
RANDOM	SUBR	3.3.6
READ	SUBR	3.3.9
RECIP	SUBR	3.3.6
RECLAIM	SUBR	3.3.8
REMAINDER	SUBR	3.3.6
REMOB	SUBR	3.3.8
REMPROP	SUBR	3.3.4
RETURN	SUBR	3.3.3
REVERSE	SUBR	3.3.2
REVERSIP	SUBR	3.3.2
RPLACA	SUBR	3.3.1
RPLACD	SUBR	3.3.1
SASSOC	SUBR	3.3.5
SEARCH	SUBR	3.3.5
SEGMENT	SUBR	3.3.10
SELECT	FSUBR	3.3.3

Atom	Type	Description
SET	SUBR	3.3.1
SETQ	FSUBR	3.3.1
SIZE	SUBR	3.3.2
STRING	FSUBR	3.3.10
STRINGP	SUBR	3.3.1
STRLEN	SUBR	3.3.10
SUBLIS	SUBR	3.3.2
SUBR	INDICATOR	
SUBST	SUBR	3.3.2
SUB1	SUBR	3.3.6
SYSIN	SUBR	3.3.9
SYSOUT	SUBR	3.3.9
T	APVAL	VALUE:=*T*
TEMPUS	SUBR	3.3.8
TERPRI	SUBR	3.3.9
TIMES	FSUBR	3.3.6
TRACE	SUBR	3.3.7
TRACESET	SUBR	3.3.7
UNDEF	FSUBR	3.3.7
UNTRACE	SUBR	3.3.7
UNTRACESET	SUBR	3.3.7
VALUE	SUBR	3.3.10
ZEROP	SUBR	3.3.6
%ANS	APVAL	VALUE:=last top level EVAL result
%EOF	INDICATOR	

Atom	Type	Description
%G	APVAL	VALUE:=<BOOL>==/G
%I	APVAL	VALUE:=<BOOL>==/I
%L	APVAL	VALUE:=0,1,2,3,==/L
%O	APVAL	VALUE:=<BOOL>==/O
%R	APVAL	VALUE:=<BOOL>==/R
%T	APVAL	VALUE:=<BOOL>==/T
%W	APVAL	VALUE:=<BOOL>==/W
%X	APVAL	VALUE:=<BOOL>==/X
%Z	APVAL	VALUE:=<BOOL>==/Z
T	APVAL	VALUE:=*T*

5. ERROR MESSAGES

Errors detected by LISP functions will cause the evaluation in progress to be aborted, printing an identification message and optional argument and a backtrace list. Operation resumes at the top level MAINLOOP unless an ERRSET function is encountered in the backtrace, in which case operation resumes by returning to the ERRSET caller with a NIL result. Non-recoverable errors will cause the LISP run to terminate with the KILLED message.

Recoverable errors

Code	Argument	Comment
A1	---	<UNUSED>
A2	offending function object	function object has no definition--APPLY
A3	---	<UNUSED>
A4	---	RETURN occurred outside PROG context
A5	---	GO occurred outside PROG context
A6	offending label atom	GO refers to an unassigned label
A7	offending argument	illegal first argument to SET
A8	offending variable atom	unbound variable--EVAL
A9	offending function object	function object has no definition--EVAL
A10	offending atom	CAR or CDR of atom (suppressed with X switch)
GC1	---	<UNUSED>
GC2	---	free space is exhausted

Code	Argument	Comment
G1	---	<UNUSED>
G2	---	recursion limit exceeded--stack space exhausted
I1	new array requested	array space exhausted
I2	argument	improper argument for numeric function
I3	---	<UNUSED>
I4	---	hard math error--infinity generated and not allowed--illegal operation with infinity
I5	argument	improper argument for array function
I01	channel/file specification	illegal file specification or channel number
I02	---	attempt to open too many channels
I03	---	device handler and I/O buffer space is exhausted
I04	---	attempt to close the last SYSOUT
R1	---	unexpected ,) or]
R2	---	. out of context--improper dotted pair construction
R3	---	unmatched (
R4	---	too many [or (. (max [= 29; max (= 255 per [ )
R5	---	undecodable number
R6	---	print name/string too long (>255 chars)
F1	---	improper number of SUBR arguments

Code	Argument	Comment
F2	PAIR result	improper number of EXPR arguments

Fatal system errors (internal LISP errors which should never occur)

T1	---	illegal trap code
T2	---	stacks corrupted
T3	---	unmatched marked stack item
01	---	input error
02	---	output error

5.1 INTERPRETATION OF BACKTRACE ON ERROR CONDITION

The backtrace list contains the names of the functions which have been entered but not completed. The leftmost function is the one in which the error was detected. (If an error occurred in a utility routine the PC of the utility routine is placed in the backtrace list.) The user should not be concerned by occurrences of functions which he did not call since many functions make internal calls and transfers to other functions. Some functions which are implemented in a recursive fashion (e.g., READ) may actually appear consecutively many times in the backtrace list.

5.2 GARBAGE COLLECTION WARNING MESSAGE

The garbage collector uses as much space as is available in order to do its job. If the recursion depth is almost at a maximum the collection will be slowed down but it will always continue to completion. If the collector fails to recover at least 1/64 of the total free space area then a warning message is printed. No other action is taken and no indication is available to the program.

6. IMPLEMENTATION CONSIDERATIONS

6.1 GENERAL

In this implementation of LISP full word space is not distinct from free space, but has the same list structure. The basic unit of free space is the cell. Each cell is two words long with the first word being at an address which is a multiple of 4. Thus, all pointers or links to other cells are only 14 bits long. The first word of each cell is the CAR pointer. The second word is either a full word of data or it is the CDR pointer. In this way full word space is incorporated in free space, only one garbage collection technique is needed, there is no bit map overhead, and the user is not faced with a decision as to the relative allocation of free versus full space. However, it is slightly more extravagant with respect to memory usage than the conventional approach to full word space.

SUBR and FSUBR function calls, utility function calls, and error processor calls are all implemented via the TRAP instruction.

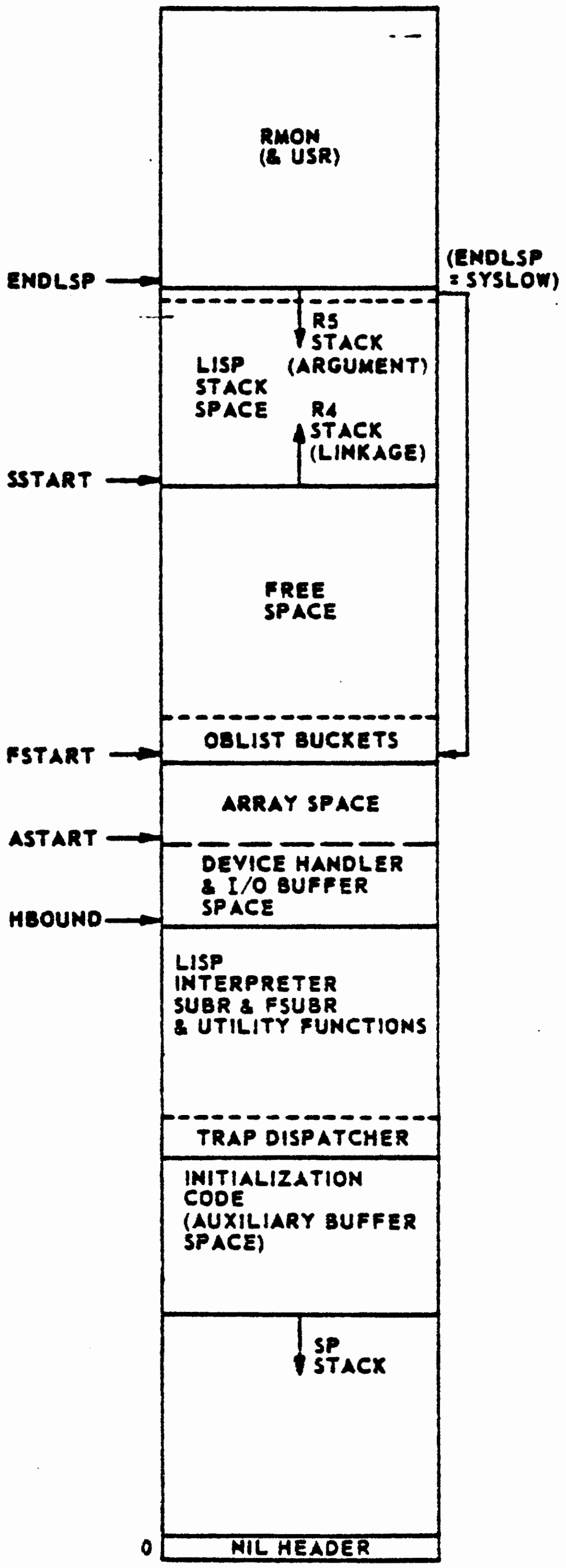
6.2 MEMORY LAYOUT

The following diagram shows the layout of the background partition after LISP has been loaded and started. There are three stacks used by LISP: SP, for general usage and arithmetic operations; R5, for LISP argument transmission; and R4, for LISP function linkage. With separate stacks for linkage and arguments backtracing after an error is facilitated as well as garbage collection, since only those S-expressions with pointers on the argument stack need be saved (the OBLIST is always on the R5 stack).

The initialization code, whose primary function is to build the OBLIST and install the system atoms, is converted to auxiliary buffer space (print name buffer*, parenthesis count buffer and I/O buffers) after execution.

The first item on the argument transmission stack (R5) is always the OBLIST, for convenience. The OBLIST consists of 32 consecutive cells at the beginning of the free space used as headers to hash buckets.

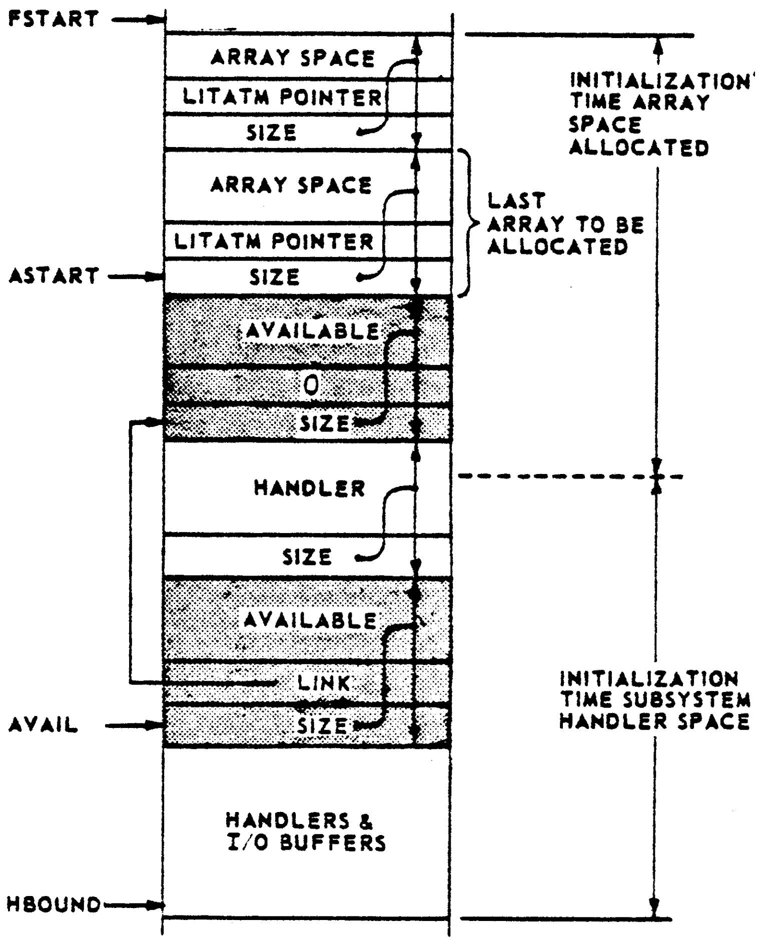
The first item on the function linkage (recursion) stack is always the address of the LISP exit routine. The exit routine closes any open files and restores the contents of locations 0 and 2 which were used by LISP as a special header for the atom NIL.



*The size of this buffer (256 characters) is the only limit on string size or atom print name size.

Device handlers, I/O buffers, and arrays share the same memory block.

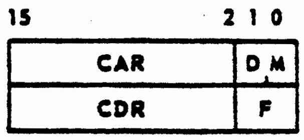
-At initialization time all handlers needed for reading subsystems are loaded contiguously starting at HBOUND. When a handler or I/O buffer is no longer needed it is released and the space is reclaimed on the AVAIL list. The block which abuts ASTART is available for an array. If no block abuts ASTART then array space is exhausted.



New arrays are allocated directly below ASTART. Whenever array space is released the arrays are slid up so that array space is always contiguous.

If a new handler is required but there is no block of available memory on AVAIL large enough to contain it then handler--I/O buffer space is exhausted.

6.3 CELL FORMAT



A cell consists of two consecutive words of memory such that the first word has an address which is a multiple of 4. Thus, all pointers to cells need only be 14 bits long.

The first word of a cell contains three fields as follows:

"MARK" M is a 1-bit field which is used by the garbage collector to mark a cell for preservation. M must not be set except in special circumstances relating to garbage collection (see the SUBR "SIZE"). M is set by the garbage collector during its marking phase, and cleared during its clean-up phase.

"DATA" D is a 1-bit field which governs the interpretation of the second word of the cell. If D is 0 then the second word consists of two fields as shown in the diagram with the interpretation given below. If D is one then the second word is interpreted as a single 16-bit field consisting of a full word of data.

CAR is a 14-bit field which is interpreted as a pointer to another cell--the CAR of the S-expression.

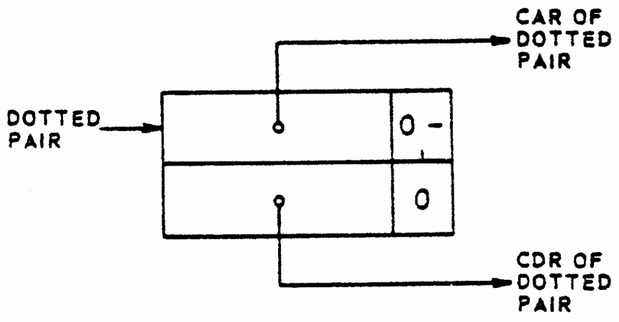
The second word of a cell contains two fields (if D is 0) as follows:

"FORMAT" F   is a 2-bit field which is used to distinguish literal atoms, string atoms, and lists. (Refer to Section 6.4.)

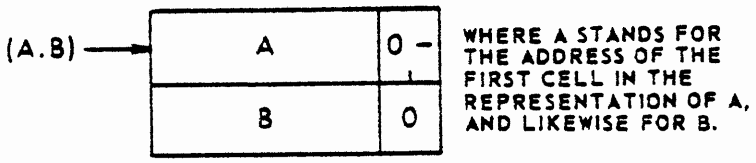
CDR   is a 14-bit field which is interpreted as a pointer to another cell--the CDR of the S-expression.

6.4 S-EXPRESSION REPRESENTATION

Dotted pairs are represented using a single cell. The first word contains the pointer to the CAR of the dotted pair. The second word contains the pointer to the CDR of the dotted pair. The format field is 0.

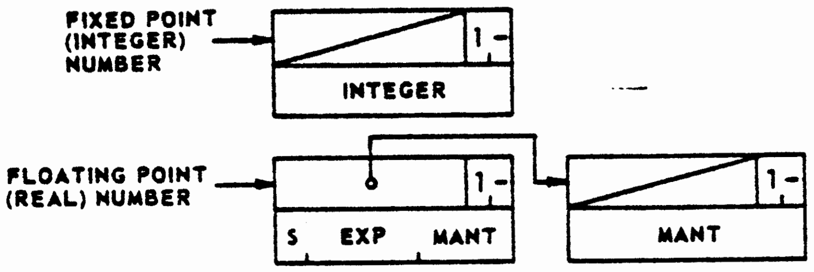


When a function returns a dotted pair (A.B) as its value it is returning a pointer to a cell as shown:


(A.B)
A
0 -
B
0
WHERE A STANDS FOR THE ADDRESS OF THE FIRST CELL IN THE REPRESENTATION OF A, AND LIKEWISE FOR B.

Atoms come in three classes as far as the representation of the header is concerned: literal atoms, numeric atoms, and string atoms.

The header or first cell of a numeric atom is structured as shown.



The diagonal slash represents a pointer to NIL (whose header is located at address 0; thus, it is a field which is 0).

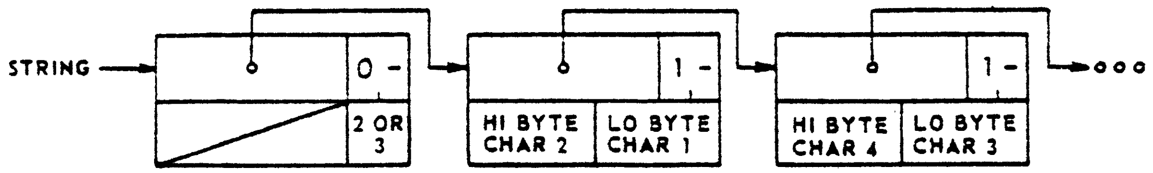
A fixed point number is represented in a single cell, with a null CAR pointer and data bit=1 in the first word, and a 16-bit integer in the second word.

A floating point number is represented using two cells. The first has a CAR pointer which points to the second. The data bits in both cells are on. The data in the first cell is interpreted as the most significant word in the standard PDP-11 floating point format. The data in the second cell is the least significant word (extension of the mantissa). Although multiple precision is not implemented the representation would merely involve extending the list of mantissa cells.

In addition to the usual PDP-11 floating point interpretation additional subclasses are recognized:

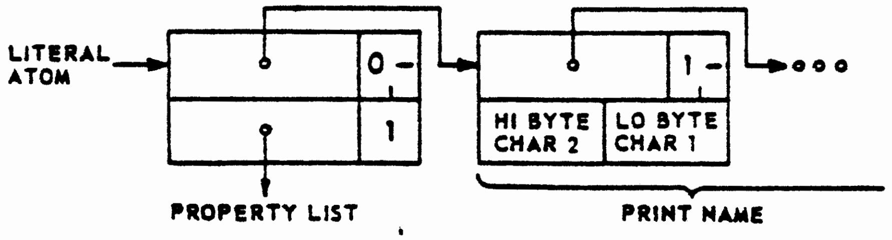
	Exponent word	Mantissa word	
floating point number	±X	X	
floating point zero	0	0	usual PDP-11 interpretation
(-0 interpreted as 0)	-0	0	
+ infinity	0	-1	
(interpreted as + infinity)	0	≠0	additions
- infinity	-0	-1	
(interpreted as - infinity)	-0	≠0	

String atoms are represented with a header cell followed by an arbitrary number (including none to represent the null string) of data cells. No distinction is made between the two format classes of strings in any of the LISP functions provided (with the exception of READ: it builds strings only with the format field=3); they merely propagate the existing format bits and process the data information identically.



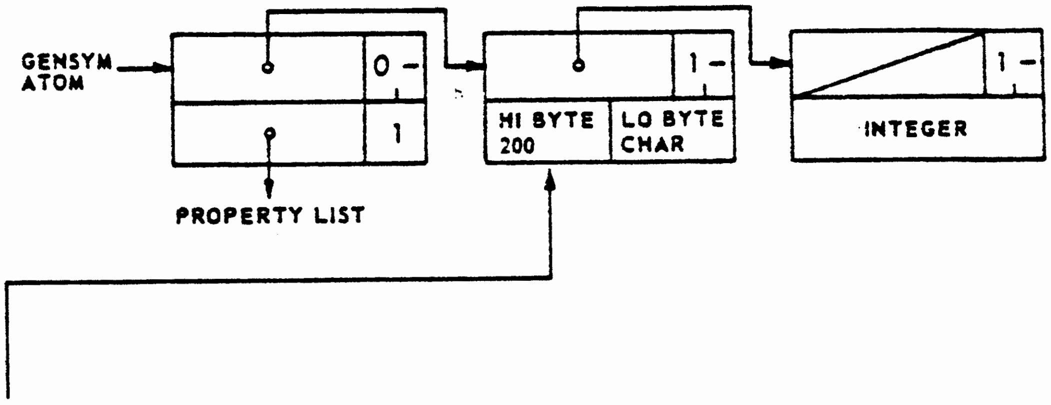
The CDR field of the header cell is presently unused and is set to zero by READ.

Literal atoms are similar to string atoms except that the CDR pointer of the header cell points to the atom's property list and the format field is 1. The string pointed to by the CAR of the header is the atom's print name.



Given a pointer to a cell in RO: (1) if D is non-zero at (RO) then RO points to a number, (2) else if F at 2(RO) is zero then RO points to a dotted pair, (3) else if F at 2(RO) is 2 or 3 then RO points to a string atom, (4) else RO points to a literal atom.

A special type of literal atom is the GENSYM atom represented as shown:

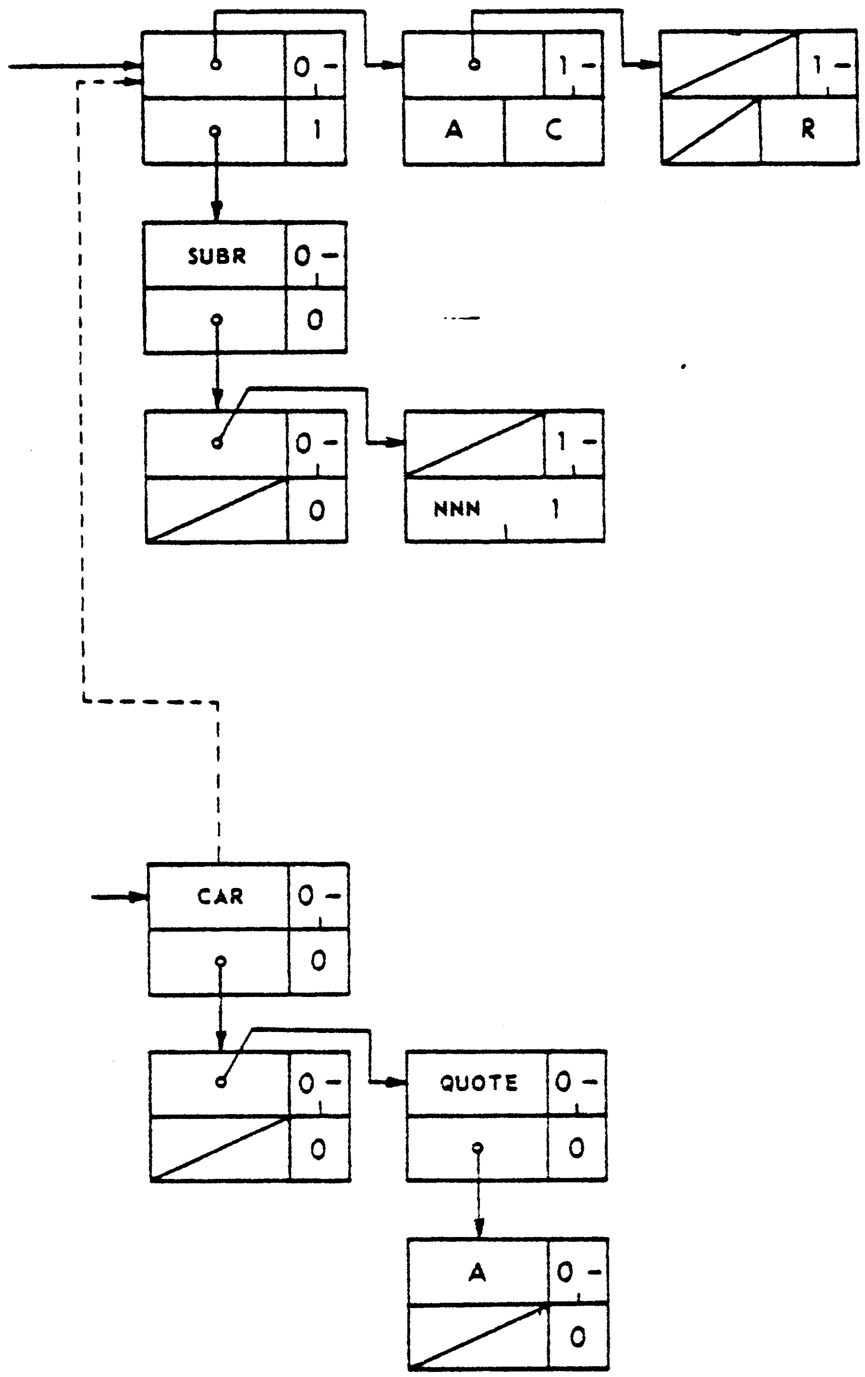


NOTE: Characters in strings and atom PNAMES are formed from 7-bit ASCII values; thus bit 15 can be used to signal a GENSYM atom. (This property is only needed on printout and is not checked or used by any other LISP functions besides GENSYM and PNAME.)

Examples:

THE ATOM CAR

THE LIST (CAR "A)



6.5 THE OBLIST

The OBLIST is a list of 32 (consecutive, to facilitate hash indexing) cells; the CAR of each points to a list (bucket) of objects (literal atoms) whose print names all hash to the same value. All objects on the OBLIST are automatically saved during a garbage collection. After initialization the OBLIST contains all the system atoms. Each literal atom encountered by READ is installed on the OBLIST if it does not already exist there. READ does this by hashing the print name and scanning the appropriate bucket for a match using EQN. If it is not found it is installed with a null property list. If it is found then READ identifies the existing atom with the one read in and any properties are inherited.

6.6 CALLING CONVENTION

Calls to SUBRs, FSUBRs, and utility routines are via the TRAP instruction which results in an effective JSR PC using the inverted R4(linkage) stack. Consequently, return from a function is via a JMP @-(R4). (Similar to the Fortran threaded code return except the stack is inverted.) R0 is used by the dispatcher but R1 through R3 are preserved.

All function atoms have the indicator SUBR or FSUBR on their property lists with an integer value associated. The low byte of the integer gives the number of arguments (not used for FSUBRs since they always take one list of arguments) and the high byte gives the TRAP code.

The number of arguments passed to a SUBR is checked within EVAL prior to calling the SUBR for an external call. There is no checking on internal calls from other functions.

7. PROGRAMMING CONVENTIONS

7.1 NAMING

Labeling of files, entry points, trap codes, etc., is based on atom names for LISP SUBRs or FSUBRs, e.g., the atom "CONS" represents a LISP SUBR. The actual assembly code is located in a file "CONS.MAC" whose global entry point is "CONS". The entry point must be the lowest core location of the function. The size of the function in bytes is given by the global "ZCONS". The SUBR is called by "QCONS" which is equal to TRAP+NNN. The global location "YCONS" contains the pointer to the atom "CONS". Utility routines follow this convention also. The following exceptions occur since the atom names are not unique within the first 5 characters:

FLOTP==FLOATP	MNUSP==MINUSP	REVIP==REVERSIP
RPLD==RPLACD	TRSET==TRACESET	UNTRSET==UNTRACESET
STRGP==STRINGP		

7.2 REGISTER USAGE

Registers R0 through R3 are available for use by any function. If a "closed" function uses one of them in a special manner that fact is noted in the module documentation. Only the explicit error TRAPs are noted in each module along with the explicit function calls. Unless otherwise specified the C bit is not significant on exit. If certain registers are always preserved within a function (and its support routines, etc.) that fact is also noted.

7.3 NOTATION

Notation for stack arguments is as follows: ARG1 represents the first argument to a routine and is located at (R5); ARG2 is located at 2(R5), etc. Non-LISP functions which pass arguments on the general stack (SP) refer to the corresponding arguments as ARG1(SP), ARG2(SP), etc. Register arguments are noted in additional comments.

7.4 MODULE FORMAT

Standard module header format:

```
; <FUNCTION TYPE:  UTILITY, SUBR, OR FSUBR> : <OPTIONAL NAME OR TITLE COMMENT>
; ENTRY:  <NUMBER OF ARGS> <ARG1 TYPE> ; <ARG2 TYPE> ; ...
(Unless otherwise specified arguments are assumed to be on R5 stack.)
; EXIT:  <NUMBER OF ARGS>   <ARG1 VALUE> ; <ARG2 VALUE> ; ...
; ERRORS: <ERROR CODE> , <ERROR EXPLANATION>
; CALLS:  <FUNCTION NAME> , <OPTIONAL EXPLANATION>
; <ADDITIONAL COMMENTS>
```

7.5 MISCELLANEOUS RULES

1) Recursive functions shall not save anything on SP stack when recursing, i.e., the SP stack is not to be used recursively.
2) Only list valued arguments or pointers outside the free space area may reside on the R5 stack.
3) All entries on the R5 stack must be even. Bit 0 is used as a special marker for ERRSET and PROG arguments.
4) S-expressions which are being modified or built should be saved on the R5 stack across CONS or GETC calls, i.e., do

not leave a pointer to an S-expression, which is not pointed to by any other S-expression, in a GPR (R1 to R3) across a possible GETC call since it will be lost if the garbage collector is called.

8. BUILDING LISP

8.1 BUILDING LISP FROM THE DISTRIBUTION PACKAGE

LISP sources are distributed as .MAC files. The initialization codes and trap handler are located in the file LISP.MAC. Parameters, definitions, and macros common to all modules are located in the file LSPMAC.MAC. The remaining modules are for the most part named as <ATOM>.MAC where ATOM is the corresponding function atom print name for the function implemented in the module. Utility routines are labeled as <UTILITY MNEMONIC>.MAC.

To build LISP, first edit LSPMAC.MAC to include the hardware option switches desired: FPU, FIS, EIS, EAE (the only modules affected by the switches are: LISP.MAC, ADR.MAC, CMR.MAC, DVI.MAC, DVR.MAC, IR.MAC, MLI.MAC, MLR.MAC, RI.MAC).

Then assemble each source module as follows:

```
<FILNAM>,<FILNAM>/N:TTM/C = LSPMAC,<FILNAM>
```

For convenience all the object modules (except LISP.OBJ) should be inserted in a library, e.g., LSPLIB.OBJ.

The functions included in a LISP load module are controlled by the option switches NOSTRG, NOARRY, NODBUG, NOMATH, NOAUXF. Only LISP.MAC is sensitive to these switches. Remnants of string, array, etc., code will still exist in other system SUBR's and FSUBR's; however, the memory savings shown in the table will result.

Finally link the object modules:

```
LISP, LISP = LISP,LSPLIB
```

Define:	To remove the functions:	and save approximately
NOSTRG	STRING,EXPLODE,VALUE,STRLEN,SEGMENT, POSITION	390 words
NOARRY	ARRAY, CLARRAY	280 words
NODBUG	ERRSET,TRACE,TRACESET,UNDEF,UNTRACE, UNTRACESET	220 words
NOMATH	ADD1,DIFFERENCE,DIVIDE,FLOAT,FIX, GREATERP,LSHIFT,MAX,MIN,MINUS,PLUS, RANDOM,SUB1,TIMES,QUOTIENT,REMAINDER, RECIP	1240-1600 words*
NOAUXF	GENSYM,LOGAND,LOGOR,LOGXOR,SEARCH, SELECT,SIZE,SUBLIS,SUBST,TEMPUS	490 words

*Depending on hardware options.

8.2 ADDING A SUBR OR FSUBR

When adding a SUBR or FSUBR the user should adhere to the programming conventions described in Section 7. Other SUBRs may be called by means of the appropriate global trap code. The proper number of arguments must be present on the stack prior to the call. Exit from a SUBR or FSUBR is via JMP @-(R4). When the result of a SUBR or FSUBR is reduced to a function of an existing quantity a shortcut can be used. The function may be called using its trap code QXXXX followed by a HALT (.WORD 0). The trap dispatcher recognizes this situation and does not save a return PC on the R4 stack so that exiting from the called function will return directly to the caller of the user's SUBR.

If an FSUBR is being written only one argument is passed--a list of the arguments to the function. The global ALIST contains a pointer to the association list. If multiple evaluations are done the ALIST should be saved on the stack and restored prior to each evaluation.

After editing the source, assemble it and insert it in the object module library. Then edit LISP.MAC to insert an ENTERS or ENTERF macro in an appropriate place and finish the LISP building operations. The arguments to the macros ENTERS and ENTERF give the unique universal name (unique within the first 5 characters--refer to Section 7), the number of arguments (not present for ENTERF), and optionally a bracketed print name if it differs from the first macro argument.

8.3 DELETING A SUBR OR FSUBR

A LISP function may be deleted simply by removing the appropriate ENTERS or ENTERF macro call within LISP.MAC and re-linking. The removed function cannot be called by any remaining function or an undefined global error will result. The conditional assembly parameters at the start of LISP.MAX define all the dependencies of the system functions. Thus a better method of removing a function is to simply leave the associated conditional variable undefined. If there is no associated conditional variable then the function is an integral part of the interpreter and cannot be removed.

8.4 ADDING AN APVAL OR INDICATOR

Adding indicators is trivial, only requiring an ENTERI macro call to be inserted within LISP.MAC. Adding an APVAL requires the addition of an ENTERA macro call as well as some code. Each APVAL must be initialized within LISP.MAC and furthermore they must be initialized in the same order as they occur in the macro calls.

APPENDIX A

THE LISP INTERPRETER: APPLY AND EVAL DESCRIPTION

The following descriptions are only suggestive of the actual workings of APPLY and EVAL. In particular, the tracing complications are left out along with certain error checking features and the multiple CAR/CDR function implementation.

These descriptions follow those given in the LISP 1.5 Programmer's Manual, which were used as a guide in developing this implementation.

```
(APPLY   (LAMBDA  (FN ARGS) (COND
         ((NULL FN)  NIL)
         ((ATOM FN)  (COND
                     ((GET FN "EXPR) (APPLY expr1 ARGS))
                     ((GET FN "SUBR) (subr2 (SPREAD3 ARGS)))
                     (T (APPLY (CDR (SASSOC FN (ALIST)
                          "(LAMBDA ()  (ERROR "A2)))) ARGS))   ))
         ((EQ (CAR FN)"LABEL) (PROGN (SETQ (ALIST) (CONS (CONS (CADR FN)
                          (CADDR FN)) (ALIST)))  (APPLY (CADDR FN) ARGS)))
         ((EQ (CAR FN)"FUNARG) (PROGN (SETQ (ALIST) (CADDR FN))
                          (APPLY (CADR FN) ARGS)   ))
         ((EQ (CAR FN)"LAMBDA) (PROGN (SETQ (ALIST) (NCONC
                          (PAIR (CADR FN) ARGS) (ALIST))) (EVAL (CADDR FN))   ))
         (T (APPLY (EVAL FN) ARGS))   )))
```

```
(EVAL  (LAMBDA (FORM) (COND
       ((NULL FORM) NIL)
       ((OR (NUMBERP FORM) (STRINGP FORM)) FORM)
       ((ATOM FORM) (COND
                     ((GET FORM "APVAL) apval[1])
                     (T (CDR (SASSOC FORM (ALIST) "(LAMBDA () (ERROR "A8))))) ))
       ((EQ (CAR FORM) "QUOTE) (CADR FORM))
       ((EQ (CAR FORM) "FUNCTION) (LIST "FUNARG (CADR FORM) (ALIST)))
       ((EQ (CAR FORM) "COND) ("COND (CDR FORM)))
       ((EQ (CAR FORM) "PROG) ("PROG (CDR FORM)))
       ((ATOM (CAR FORM)) (COND
            ((GET (CAR FORM) "EXPR) (APPLY expr[1] (EVLIS(CDR FORM))))
            ((GET (CAR FORM) "FEXPR) (APPLY fexpr[1] (LIST (CDR FORM) (ALIST))))
            ((GET (CAR FORM) "SUBR) (subr[2] (SPREAD[3] (EVLIS (CDR FORM)))))
            ((GET (CAR FORM) "FSUBR) ((fsubr[2] (CDR FORM)))
            ((GET (CAR FORM) "ARRAY) (INDEX[4] (CAR FORM) (EVLIS(CDR FORM))))
            (T (EVAL (CONS (CDR (SASSOC (CAR FORM) (ALIST)
                     "(LAMBDA () (ERROR "A9)))) (CDR FORM))))   ))
       (T (APPLY (CAR FORM) (EVLIS (CDR FORM))))   )))
```

[1] The value of GET is set aside so that it need not be calculated twice. This is the meaning of the lower case variable.

[2] The value of GET is set aside so that it need not be calculated twice. This value is an integer atom which contains the TRAP code as well as the number of arguments expected (for SUBR's only). Control is transferred to the SUBR or FSUBR by constructing the appropriate TRAP instruction and executing it. This is the meaning of the lower case variable in the function position.

[3] SPREAD is a pseudofunction describing a utility section within the EVAL code. It effectively takes a single list argument and returns multiple values by spreading the elements of the list on the R5 (argument transmission) stack.

[4] INDEX is a utility function which locates that portion of the array space referred to, calculates the offset to the desired element, and returns the value of that element.

APPENDIX B

EXAMPLE OF CONVERSATIONAL LISP

THE FOLLOWING EXAMPLE APPEARS IN THE LISP 1.5 PROGRAMMER'S MANUAL.

```
. R LISP
*/L:3/G/T

EVAL:
(SYSIN 'TSTJAB')
0              0

VALUE IS...

.. AFTER      0.133 SECONDS...
1

EVAL:
(DEFINE "(
0        1
(THEOREM (LAMBDA (S) (TH1 NIL NIL (CADR S) (CADDR S))))
2        3       4 4 4            5      5 5       5432


(TH1 (LAMBDA (A1 A2 A C) (COND ((NULL A)
2    3       4         4 4     56      6
        (TH2 A1 A2 NIL NIL C)) (T
        6                   65 5
        (OR (MEMBER (CAR A) C) (COND ((ATOM (CAR A))
        6   7       8     8  7 7     89     A     A9
        (TH1 (COND ((MEMBER (CAR A) A1) A1)
        9    A     BC       D     D   C   B
        (T (CONS (CAR A) A1))) A2 (CDR A) C))
        B  C     D     D   CBA    A     A  98
        (T (TH1 A1 (COND ((MEMBER (CAR A) A2) A2)
        8  9       A     BC       D     D   C   B
        (T (CONS (CAR A) A2))) (CDR A) C))))))))
        B  C     D     D   CBA A     A  98765432


(TH2 (LAMBDA (A1 A2 C1 C2 C) (COND
2    3       4             4 4
        ((NULL C) (TH A1 A2 C1 C2))
        56      6 6              65
        ((ATOM (CAR C)) (TH2 A1 A2 (COND
        56     7     75 6          7
        ((MEMBER (CAR C) C1) C1) (T
        89       A     A   9   8 8
        (CONS (CAR C) C1))) C2 (CDR C)))
        9     A     A   987    7     765
        (T (TH2 A1 A2 C1 (COND ((MEMBER
        5  6             7     89
        (CAR C) C2) C2) (T (CONS (CAR C) C2)))
        A     A   9   8 8  9     A     A   987
        (CDR C))))))
        7     765432
```

```
(TH (LAMBDA (A1 A2 C1 C2) (COND ((NULL A2) (AND (NOT (NULL C2))
2   3       4           4 4     56       6 6    7    8       87
        (THR (CAR C2) A1 A2 C1 (CDR C2)))) (T (THL (CAR A2) A1 (CDR A2)
        7    8      8          8      8765 5  6    7      7    7      7
        C1 C2)))))
             65432


(THL (LAMBDA (U A1 A2 C1 C2) (COND
2    3       4             4 4
        ((EQ (CAR U) (QUOTE NOT)) (TH1R (CADR U) A1 A2 C1 C2))
        56   7     7 7         76 6     7      7            65
        ((EQ (CAR U) (QUOTE AND)) (TH2L (CDR U) ~~A1~~ A2 C1 C2))
        56   7     7 7         76 6     7     7                65
        ((EQ (CAR U) (QUOTE OR)) (AND (TH1L (CADR U) A1 A2 C1 C2)
        56   7     7 7        76 6    7     8      8            7
        (TH1L (CADDR U) A1 A2 C1 C2) ))
        7     8       8            7 65
        ((EQ (CAR U) (QUOTE IMPLIES)) (AND (TH1L (CADDR U) A1 A2 C1
        56   7     7 7             76 6    7     8       8
        C2) (TH1R (CADR U) A1 A2 C1 C2) ))
          7 7     8      8            7 65
        ((EQ (CAR U) (QUOTE EQUIV)) (AND (TH2L (CDR U) A1 A2 C1 C2)
        56   7     7 7           76 6    7     8     8            7
        (TH2R (CDR U) A1 A2 C1 C2) ))
        7     8     8            7 65
        (T (ERROR (LIST (QUOTE THL) U A1 A2 C1 C2)))
        5  6      7     8         8              765
        )))
        432


(THR (LAMBDA (U A1 A2 C1 C2) (COND
2    3       4             4 4
        ((EQ (CAR U) (QUOTE NOT)) (TH1L (CADR U) A1 A2 C1 C2))
        56   7     7 7         76 6     7      7            65
        ((EQ (CAR U) (QUOTE AND)) (AND (TH1R (CADR U) A1 A2 C1 C2)
        56   7     7 7         76 6    7     8      8            7
        (TH1R (CADDR U) A1 A2 C1 C2) ))
        7     8       8            7 65
        ((EQ (CAR U) (QUOTE OR)) (TH2R (CDR U) A1 A2 C1 C2))
        56   7     7 7        76 6     7     7            65
        ((EQ (CAR U) (QUOTE IMPLIES)) (TH11 (CADR U) (CADDR U)
        56   7     7 7             76 6     7      7 7       7
        A1 A2 C1 C2))
                   65
        ((EQ (CAR U) (QUOTE EQUIV)) (AND (TH11 (CADR U) (CADDR U)
        56   7     7 7           76 6    7     8      8 8       8
        A1 A2 C1 C2) (TH11 (CADDR U) (CADR U) A1 A2 C1 C2) ))
                   7 7     8       8 8      8            7 65
        (T (ERROR (LIST (QUOTE THR) U A1 A2 C1 C2)))
        5  6      7     8         8              765
        )))
        432
```

```
(TH1L (LAMBDA (V A1 A2 C1 C2) (COND
2     3       4             4 4
        ((ATOM V) (OR (MEMBER V C1)
        56      6 6   7           7
        (TH (CONS V A1) A2 C1 C2) ))
        7   8         8         7 65
        (T (OR (MEMBER V C2) (TH A1 (CONS V A2) C1 C2) ))
        5  6   7           7 7      8         8      7 65
        )))
        432

(TH1R (LAMBDA (V A1 A2 C1 C2) (COND
2     3       4             4 4
        ((ATOM V) (OR (MEMBER V A1)
        56      6 6   7           7
        (TH A1 A2 (CONS V C1) C2) ))
        7         8         8   7 65
        (T (OR (MEMBER V A2) (TH A1 A2 C1 (CONS V C2))))
        5  6   7           7 7            8         8765
        )))
        432

(TH2L (LAMBDA (V A1 A2 C1 C2) (COND
2     3       4             4 4
        ((ATOM (CAR V)) (OR (MEMBER (CAR V) C1)
        56     7     76 6   7       8     8   7
        (TH1L (CADR V) (CONS (CAR V) A1) A2 C1 C2)))
        7     8      8 8     9     9   8         765
        (T (OR (MEMBER (CAR V) C2) (TH1L (CADR V) A1 (CONS (CAR V)
        5  6   7       8     8   7 7     8      8    8     9     9
        A2) C1 C2)))
          8      765
        )))
        432

(TH2R (LAMBDA (V A1 A2 C1 C2) (COND
2     3       4             4 4
        ((ATOM (CAR V)) (OR (MEMBER (CAR V) A1)
        56     7     76 6   7       8     8   7
        (TH1R (CADR V) A1 A2 (CONS (CAR V) C1) C2)))
        7     8      8       8     9     9   8   765
        (T (OR (MEMBER (CAR V) A2) (TH1R (CADR V) A1 A2 C1
        5  6   7       8     8   7 7     8      8
        (CONS (CAR V) C2))))
        8     9     9   8765
        )))
        432
```

```
(TH11 (LAMBDA (V1 V2 A1 A2 C1 C2) (COND
2     3         4                4 4
        ((ATOM V1) (OR (MEMBER V1 C1) (TH1R V2 (CONS V1 A1) A2 C1
        56         6 6   7             7 7         8           8
        C2)))
          765
        (T (OR (MEMBER V1 C2) (TH1R V2 A1 (CONS V1 A2) C1 C2)))
        5  6   7             7 7          8           8    765
        )))
        432
))
10

VALUE IS...

...AFTER       0.017 SECONDS...
(THEOREM TH1 TH2 TH THL THR TH1L TH1R TH2L TH2R TH11)

EVAL:
(TRACE "(TH]     ;EOF ON TSTJAB.LSP, SO INPUT IS NOW FROM TT:
0        1  0

VALUE IS...

...AFTER       0.017 SECONDS...
NIL

EVAL:
(THEOREM "(ARROW((OR A (NOT B)))((IMPLIES(AND P Q)(EQUIV P Q]
0          1     23     4     43223        4         44        0

VALUE IS...

[1]  LEVEL ARGUMENTS OF TH
NIL
((OR A (NOT B)))
NIL
((IMPLIES (AND P Q)(EQUIV P Q)))

[2]  LEVEL ARGUMENTS OF TH
(A)
NIL
NIL
((IMPLIES (AND P Q)(EQUIV P Q)))

[3]  LEVEL ARGUMENTS OF TH
(A)
((AND P Q))
NIL
((EQUIV P Q))

[4]  LEVEL ARGUMENTS OF TH
(Q P A)
NIL
NIL
((EQUIV P Q))

GARBAGE COLLECTION NUMBER      1  AT     439.933  FREED    1578  CELLS

[4]  RESULT OF TH
*T*
```

```
[3]  RESULT OF TH
*T*

[2]  RESULT OF TH
*T*

[2]  LEVEL ARGUMENTS OF TH
NIL
((NOT B))
NIL
((IMPLIES (AND P Q)(EQUIV P Q)))

[3]  LEVEL ARGUMENTS OF TH
NIL
NIL
(B)
((IMPLIES (AND P Q)(EQUIV P Q)))

[4]  LEVEL ARGUMENTS OF TH
NIL
((AND P Q))
(B)
((EQUIV P Q))

[5]  LEVEL ARGUMENTS OF TH
(Q P)
NIL
(B)
((EQUIV P Q))

[5]  RESULT OF TH
*T*

[4]  RESULT OF TH
*T*

[3]  RESULT OF TH
*T*

[2]  RESULT OF TH
*T*

[1]  RESULT OF TH
*T*

 ...AFTER      53.333 SECONDS...
*T*

EVAL:
(FREE)
0    0

VALUE IS...

 ...AFTER       0.033 SECONDS...
1083
```

```
EVAL:
(RECLAIM)
0       0

VALUE IS...

GARBAGE COLLECTION NUMBER      2  AT      526.650  FREED    1763  CELLS

...AFTER      3.667 SECONDS...
NIL

EVAL:
```