C-MU LISP

Crispin S. Perdue
and a host of others
2 September 1979

Carnegie-Mellon University
Department of Computer Science

The work resulting in the LISP system described in this manual and in the manual itself were funded in part by the Defense Advanced Research Projects Agency under contract No. F44620-73-C-0074.

Table of Contents

Preface

This manual was produced automatically from the on-line help database at Carnegie-Mellon University. It is the best available source of reference information on the descendant of Stanford LISP and UCI LISP that is in use at C-MU as of 2 September 1979. The formatting is sometimes deficient, but considerations of formatting and presentation had to defer to the goal of producing the best available reference information with minimum effort. Any pleasant aspects of the appearance are probably due to the use of the SCRIBE document formatting system, by Brian Reid.

In general, this document does not attempt to provide an explanation of the purposes of the facilities provided. On the other hand, the manual has been carefully organized and this may help the reader. Also, in certain cases some background information is included.

The maintainers of LISP will appreciate corrections and improvements especially if the helpful user will send text that can replace the current database entries. At C-MU the database is contained in the files INDEX.*[A311LISP], where the extension is same as the first alphabetic character in the name of the function, command, value, etc. described. Send comments and reports on problems to LISP@CMU-10A.

This system and its documentation is the result of many people's work. Maintenance at C-MU has been mainly done by Cris Perdue, who has made many miscellaneous improvements and wrote the system that produced this document. Don Cohen is responsible for the existence of the on-line help data and maintains it. The top level and numerous other improvements were made by Mark Stickel. The pretty printer, the most recent edition of the compiler, improved printing functions and other features came from the version of UCI LISP produced at Rutgers University by Rick LeFaivre. This is a descendant of UCI LISP, whose authors are Robert J. Bobrow, Richard R. Burton, Jeffrey M. Jacobs, and Daryle Lewis. UCI LISP is a descendant of Stanford LISP 1.6 by Lynn Quam, John Allen, and Whitfield Diffie. LISP 1.6 in turn was originally an adaptation of an early version of MacLISP at M.I.T..[1] The top level and editor in particular are directly derived from facilities provided by INTERLISP, and we owe a debt to Warren Teitelman and all the others responsible for that system. The text of the documentation here is taken in part from the documentation of some version of each of the systems mentioned with the possible exception of MacLISP.

---

[1] A MacLISP manual of appropriate vintage states that the LISP described is a direct descendant of the first LISP interpreter written for the PDP-6, which was the first program ever written for the PDP-6.

1. LISP-PROPER

1.1 ELEMENTARY

The basic data types of LISP are numbers, strings, identifiers and S-expressions. Numbers are typed as one would expect. Strings are surrounded by double-quotes ("). Identifiers are also strings of characters, but some characters (such as blank, comma, dot, parens) terminate identifiers unless specially marked. Two special identifiers are T, which is interpreted as the constant true, and NIL, which is treated as the constant false and is used as a list terminator. S-expressions are defined as either objects of a basic data type or a dotted pair of S-expressions, written (e1 . e2). The latter are created by the function CONS. A special case of an S-expression is a "list" which is either NIL or (<S-expression> . <list>). Lists are written without dots or their corresponding parentheses, e.g. (1 . (2 . (3 . NIL))) is written (1 2 3). The interpreter expects function calls to be in the form of lists, e.g. (<function-name> <first argument> <second argument> ...).

1.1.1 OVERVIEW

Most of the interaction between users and the LISP system is handled by three programs. The three can be distinguished by their "prompts":

<n> or > is the prompt for the TOP-LEVEL, which reads what you type, executes it and prints the result. N is just a counter.

: or n: is a prompt from the BREAK-PACKAGE. The BREAK-PACKAGE is LISP's debugger. It is very similar to the top-level except that it understands special commands to recover from errors. The break-package counts the number of times it has been entered recursively.

or n# is an EDITOR prompt. Like the break-package, the editor counts the number of times it has been entered recursively. Unlike the other two, the editor understands only editor commands, and will not evaluate whatever expression you type to it.

Fortunately, all three of these programs understand the HELP command. To get help related to almost any word that is meaningful to LISP, just type "HELP <word>". For example "HELP TOP-LEVEL" or "HELP BREAK-PACKAGE". An outline of the entire lisp system is built into the help messages. If you are not sure what word to ask about, type "HELP INDEX" and get the outline's top level.

1.1.2 NUMBER

1.1.2.1 INUM

INUMs are integers of absolute value less than 2^16. They are represented as pointers outside of the normal LISP address space.

1.1.2.2 FIXNUM

FIXNUMs are integers of absolute value between 2^16 and 2^36. The FIXNUM property is used to store values of FIXNUMS.

1.1.2.3 FLONUM

FLONUMs are floating point numbers. The FLONUM property is used to store values of FLONUMs.

1.1.3 (QUOTE "E") [FSUBR]

returns E without evaluating it.

1.1.4 NIL [VALUE]

is a primitive constant of LISP used to terminate lists and to represent falsehood (as the value of predicates). Woe be unto them that would change the value of NIL (from NIL)! In keeping with its character as both an atom and the representation of the empty list, the atom NIL has been modified so that its CAR and CDR are both NIL. One can now, for example, pick up the arguments to an FEXPR via a sequence of CAR/CDR combinations, with missing arguments automatically set to NIL. Note that NIL now has a usable property list, although it is not stored as the CDR of NIL as with other atoms (GET, PUTPROP, etc. are all aware of its actual location).

1.1.5 T [VALUE]

is the primitive constant that LISP uses for TRUE (as in the value of a predicate).

1.1.6 (HELP "word1" ... "wordn") [FSUBR]

The HELP function prints messages associated with the words given as its arguments. Any word ending with "@" will find all of the help for words that are the same up to the final @, e.g. (HELP X@) will explain all that starts with X. In addition there is a semantic index to the entire LISP system, in tree-like form where the pointers are "index" help messages. Do (HELP INDEX) for a list of top level topics in the help tree.

The HELP function now treats (HELP) as if it were (HELP OVERVIEW). In order to understand HELP you are strongly urged to see HELPFILTER.

1.1.6.1 (HELPFILTER word attributes) [FSUBR]

decides whether or not to print a comment (HELP message). The word is the one found by GETDEF and (with permission) to be explained. If HELPFILTER returns NIL then the message will not be printed. You can program your own help by writing a new HELPFILTER.

The attributes used by HELP are BASIC, GENERAL, DETAIL, EXAMPLES, XREF (backward pointers to SEE, UNDER and INDEX), SEE (for related topics), UNDER (where the explanation REALLY is) and INDEX (list of sub-categories with explanations), and the attributes STANDARD, EDIT-COMMAND, TOP-LEVEL-COMMAND and BREAK-COMMAND to indicate that this explanation refers to the interpretation given to the word by the editor, top-level or break package (only for words that are meaningful to more than one). BASIC is the vanilla-flavored attribute for most first or only entries of a name. The attribute LONG flags the messages that are longer than one screenful. OBSOLETE entries are for words that are no longer meaningful, and the entry points to the new features that replace the obsolete one. OLD flags are for things about to become obsolete and NEW flags are for features that are about to appear (or are on an experimental LISP).

The default helpfilter sets LASTHELP to contain the current word after it decides what to return. It uses LASTHELP to automatically print the first entry for any word and asks the user whether or not to print the following entries, with the following exceptions: - SEE entries are always printed (and preceded by "See") since they are always short. - UNDER entries are always printed (with a message) since they are always short. - XREF entries are always printed (and preceded by "Pointed to by") since they are (hopefully) always short. No distinction is made between being pointed to from an INDEX, UNDER or SEE entry. - OBSOLETE entries are always printed as a message saying that the word is obsolete and you should use <entry> instead. - First entries are not printed automatically if they are long. - First entries are not printed automatically if they have any of the flags STANDARD, TOP-LEVEL-COMMAND, BREAK-COMMAND, EDIT-COMMAND since the user probably wanted only one of these explanations. The new HELPFILTER asks for one letter responses to its questions. The response is stored as part of LASTHELP.

1.1.6.2 LASTHELP [VALUE]

contains the last word that has been HELPed and the (ascii code of) the character that the user typed in response to the question of what to do next. It is set and used by the default

HELPFILTER.

1.2 EVAL-S-EXP

1.2.1 EVAL

```
(*EVAL E) [SUBR]
 (EVAL E) [LSUBR]
```

*EVAL and EVAL evaluate the S-expression E.

```
Example:        (EVAL (LIST (QUOTE ADD1) 3)) = 4
```

The difference is that EVAL (but not *EVAL) allows a second argument which is interpreted as a Binding Context Pointer (BCP).

1.2.2 APPLY

```
(*APPLY FN ARGS)  [SUBR]
(APPLY FN ARGS) [LSUBR]
```

APPLY evaluates ARGS and binds each s-expression of that result to the corresponding argument of the function FN. The value of FN is then returned.

APPLY can also be given a third argument which will be interpreted as a BCP. *APPLY does not take a third argument, and is used for compiled calls on APPLY which do not have three arguments.

Example:

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

1.2.3 (APPLY* FN ARGS) [SUBR]

APPLY* is similar to APPLY except that FN may be a function of any type including MACRO.

Note that when either APPLY or APPLY* is given an EXPR as its first argument, the second argument is evaluated by APPLY* or APPLY, but the elements of the resulting list are directly bound to the lambda variables of the first argument, and are not evaluated again even though it is an EXPR.

1.2.4 FUNARG

(To be ridiculously brief about it,) There are times when you would like to evaluate

expressions in contexts other than the one from which the request for evaluation is made. In LISP the solution is to specify the context as a pointer into the SPDL. These pointers are called BCPs.

For related information see FEXPR and SPDL.

1.2.5 BCP

A "binding context pointer" (BCP) is a pointer into the SPECIAL PUSHDOWN LIST designating a level in recursive variable binding. BCPs are now simply displacements from the bottom of the SPDL. When EVAL and APPLY receive a BCP as their last argument, all SPECIAL (VALUE) CELLS are restored to the values they had at the time the BCP was generated. This then causes EVAL and APPLY to reference these variables in the binding context which existed at the time of BCP generation. This feature primarily is useful to prevent variable name conflicts when using EVAL, APPLY, and functional arguments. As with the A-LIST, when EVAL and APPLY exit, the previous bindings are restored. There are two ways to generate a BCP: If an FEXPR is defined with two arguments, then the second argument will be bound to the SPECIAL PUSHDOWN LIST level at the time the FEXPR is called. The second way to generate a BCP is with *FUNCTION.

Example using the BCP feature:

NOTE This example will not work, because at present the values of the variables are not restored into the stack. Therefore, when the current use of the BCP ends, the next reference to that variable will return the old value.

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

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

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

1.2.6 (*FUNCTION "FN") [FSUBR]

*FUNCTION returns a list of the following form:

```
(FUNARG FN . <BCP>)
```

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

1.3 LAMBDA-EXP

1.3.1 LAMBDA

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

(Note: LAMBDA is not considered to be a function.) An expression of this form denotes the function whose value (action) on the given list of arguments is the result of evaluating the body. Except for LEXPRs, the argument list is a list of identifiers. Lambda expressions with more than five arguments can't be compiled.

Examples: `(LAMBDA NIL 1)`

is the constant function (whose value is always one) of no arguments.

```
(LAMBDA (X) (TIMES X X))
```

is a function which returns the square of its argument if it is a number. Otherwise an error will result.

1.3.2 (FUNCTION "FN") [FSUBR]

FUNCTION is the same as QUOTE in the interpreter. In the compiler, FUNCTION causes the S-expression FN to be compiled as if it were another named function, whereas QUOTE generates an S-expression constant.

1.3.3 FEXPR

A FEXPR is an identifier which has a LAMBDA expression of one dummy variable on its property list with property name FEXPR. FEXPRs are evaluated by binding the actual argument list to the dummy variable without evaluating any arguments. DF is useful for defining FEXPRs. The compiled form of an FEXPR is an FSUBR. FEXPRs can be defined with two arguments, in which case the second is interpreted as a BCP.

```
(DF LISTQ (L) L)
        (LISTQ A (B) C) = (A (B) C)
        (LISTQ) = NIL
```

1.3.4 LABEL

```
(LABEL "ID" "LAMBDA-EXPR")
```

(Note: LABEL is not considered to be a function.) LABEL creates a temporary name ID for its LAMBDA expression by creating a local variable of that name whose value is the LAMBDA expression. This makes it possible to construct recursive functions with temporary names. Example:

```
(DE REVERSE (L)
   ((LABEL REVEVSE1
           (LAMBDA (L M)
                   (COND ((ATOM L) M)
                         (T (REVERSE1 (CDR L) (CONS (CAR L) M))))))
    L NIL))
```

1.3.5 LEXPR

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

```
(LAMBDA "ID" "FORM")
```

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

```
(DE MAX N
   (PROG (M)
         (SETQ M (ARG N))
       L (SETQ N (SUB1 N))
         (COND ((ZEROP N) (RETURN M))
               ((GREATERP (ARG N) M) (SETQ M (ARG N))))
         (GO L)))
(MAX 1 1.2 4 3 -50) = 4
```

1.3.5.1 (ARG N) [SUBR]

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

1.3.5.2 (SETARG N V) [SUBR]

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

1.3.6 EXPR

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

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

1.3.7 MACRO

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

HELP could be defined by:

```
(DM HELP (L) (CONS (QUOTE GETDEF) (CONS <NAME OF HELP FILE>  (CDR L))))
```

FOR-EACH is defined by:

```
(DEFPROP FOR-EACH
(LAMBDA (L)
 (CONS (COND ((MEMQ (CADR L)
                    (QUOTE (MAP MAPC MAPCAN MAPCAR MAPCON MAPCONC MAPLIST)))
                (SETQ L (CDR L)) (CAR L))
               (T (QUOTE MAPC)))
       (CONS (CONS (QUOTE FUNCTION)
                   (NCONS
                    (CONS (QUOTE LAMBDA)
                          (COND ((ATOM (CADR L)) (CONS (NCONS (CADR L))
                                                       (CDDDR L)))
                                (T (CONS (CADR L) (NTH (CDDDR L)
                                                       (LENGTH (CADR L))
                                   )))))))
             (COND ((ATOM (CADR L)) (NCONS (CADDR L)))
                   (T (LDIFF (CDDR L) (NTH (CDDDR L) (LENGTH (CADR L))))))
       ))))
MACRO)
```

1.3.7.1 (*EXPAND L FN) [SUBR]

```
(*EXPAND1 L FN) [SUBR]
```

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

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

With PLUS defined as

```
(DM PLUS (L) (*EXPAND L(QUOTE *PLUS)))
```

(PLUS A B C D) expands to:

```
(*PLUS (*PLUS (*PLUS A B) C) D)
```

1.4 DEFINITIONS

1.4.1 (DE "NAME" "ARGUMENT-LIST" "FORM1" ... "FORMn") [FSUBR]

DE, DF and DM are used to define EXPRs, FEXPRs and MACROs. They place the form (LAMBDA ARGUMENT-LIST FORM1 ... FORMn) on the property list of NAME under property EXPR, FEXPR or MACRO. DE, DF, and DM will generate an error if there are fewer than three arguments, the first argument is not a literal atom, or the second argument is not a list (or literal atom for DE).

For related information see MARK!CHANGED, CHANGES, and //PUTPROP.

If the function being defined was not previously defined, the function name will be returned. Otherwise, a list consisting of the function name and "EQUAL" or "REDEFINED" will be returned, depending on whether an EQUAL definition was already present. In the cases of a new definition or a redefinition, MARK!CHANGED will be called to record the fact that the function has been changed. The new definition will always be at the front of the property list, insuring that it will be used as the definition of the function. DE, DF, and DM now call //PUTPROP rather than PUTPROP so they will be undoable.

1.4.2 (DV "atom" "value") [FSUBR]

is equivalent to (SETQ atom (QUOTE value)). DV is undoable and calls MARK!CHANGED.

1.4.3 (DEFPROP "I" "V" "P") [FSUBR]

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

For related information see //PUTPROP, MARK!CHANGED, CHANGES, and GRINPROPS.

DEFPROP has been modified to generate an error if it is called with other than three arguments, its first argument is not a literal atom, its third argument is neither a literal atom nor an INUM, or its first argument is T or NIL and its third argument is VALUE. If P is in GRINPROPS (the list of property names "seen" by GRINDEF) and the new property value is not EQUAL to the old one, MARK!CHANGED will be called to record the fact that the definition of I has been changed. DEFPROP now calls //PUTPROP rather than PUTPROP so it will be undoable. If the property defines a function, it will always be placed at the front of the property list, insuring that it will be used as the definition.

1.4.4 (DEFLIST "L" {"defval"} "prop") [FSUBR]

DEFLIST is useful for placing a property on a number of atomic symbols. L should be a list of items, each of which is either an atomic symbol A or a two-element list (A val). Each A will have a prop property placed on its property list, with a value of val if present, or defval if only the atomic symbol was given. Defval is optional, with a default value of T assumed. DEFLIST is undoable and calls MARK!CHANGED. As an example of the use of DEFLIST, the following will give TOM and BOB ages of 15, and SAM an age of 20 (i.e., the 20 overrides the default value of 15):

```
(DEFLIST (TOM BOB (SAM 20)) 15 AGE)
```

1.4.5 (DEFSYNON "at1" "at2" "prop") [FSUBR]

Places the <prop> property of <at2> onto <at1>. DEFSYNON is undoable and calls MARK!CHANGED. It may be used to give two synonymous names to a variable. If the property defines a function, the property will always be placed at the front of the property list, insuring that it will be used as the definition.

1.5 CONTROL

1.5.1 CONDITIONALS

1.5.1.1 (COND Clause1 Clause2 ...) [FSUBR]

where Clausei is a list of expressions, (E<i,1> E<i,2> ... E<i,n>).

The COND is evaluated by evaluating the E<i,1>s starting from i=1 until one is found that evaluates to something other than NIL. Then the rest of the expressions in its list are evaluated, the value of the last being the value returned from the COND. If all of the E<i,1> evaluate to NIL the value of the COND is NIL.

Examples:

```
(DE ABS (X) (COND ((MINUSP X) (MINUS X)) (T X)))
(DE NOT (X) (COND (X NIL) (T)))
```

1.5.1.2 (SELECTQ X "Y1" "Y2" ... "Yn" Z) [FSUBR]

This function is used to select a sequence of instructions based on the value of its first argument X. Each of the Yi is a list of the form (Si E[1,i] E[2,i] ... E[k,i]) where Si is the "selection key".

If Si is an atom the value of X is tested to see if it is EQ to Si (which is not evaluated). If so, the expressions E[1,i] ... E[k,i] are evaluated in sequence, and the value of SELECTQ is the value of the last expression evaluated, i.e. E[k,i]. If Si is a list, and if any element of Si is EQ to the value of X, then E[1,i] ... E[k,i] are evaluated in turn as above. If Yi is not selected in one of the two ways described then Y[i+1] is tested, etc. until all the Y's have been tested. If none is selected, the value of SELECTQ is the value of Z. Z must be present.

An example of the form of a SELECTQ is:

```
(SELECTQ (CAR W)
         (Q (PRINT FOO) (FIE W))
         ((A E I O U) (VOWEL W))
         (COND (W (QUOTE STOP))))
```

which has two cases, Q and (A E I O U) and a default condition which is a COND.

SELECTQ compiles open, and is therefore very fast; however, it will not work if the value of X is a list, a large integer, or floating point number, since it uses EQ. Compiled SELECTQs bind the variable SELECTQ to the value computed as the selection key.

1.5.2 MAPPING

"Mapping" refers to a loop which is controlled by a list. Typically one wants to do something for each element of a list. The FOR-EACH function makes direct use of the mapping functions almost obsolete. However, the user must still understand what the mapping functions do in order to get FOR-EACH to do it instead. Similarly, FORALL and EXISTS are the reasonable ways to use the lower level EVERY and SOME functions.

All of the map functions have been extended to allow called functions which need more than one argument. The function FN to be called is still the first argument. Arguments 2 thru N (N < 6) are used as arguments 1 thru N-1 for FN. If the arguments to the map functions are of unequal length, the map function terminates when the shortest list becomes NIL. The functions behave the same as the previous definitions of the functions when used with two arguments.

Example: This will set the values of A, B and C to 1, 2 and 3, respectively.

```
(MAPC (FUNCTION SET) (QUOTE (A B C)) (QUOTE (1 2 3)))
NIL
```

1.5.2.1 (MAP FN L) [LSUBR]

MAP applies the function FN to list L and to successive CDRs (or "tails") of L until L is reduced to NIL. The value of MAP is NIL.

```
Example:        (MAP (FUNCTION PRINT) (QUOTE (X Y Z))) *
        PRINT:          (X Y Z)
        PRINT:          (Y Z)
        PRINT:          (Z)
        RETURN:         NIL
```

1.5.2.2 (MAPC FN L) [LSUBR]

MAPC is identical to MAP except that MAPC applies function FN to the CAR of the remaining list at each step. I.e. Fn is applied to each element of the list L. The value of MAPC is NIL.

```
Example:        (MAPC (FUNCTION PRINT) (QUOTE (X Y Z))) *
        PRINT:          X
        PRINT:          Y
        PRINT:          Z
        RETURN:         NIL
```

1.5.2.3 (MAPCON FN ARG) [LSUBR]

MAPCON applies the function FN to the list ARG. It then takes the CDR of ARG and applies FN to it. It continues this until ARG is NIL. The value of MAPCON consists of all of the values returned by FN NCONC'ed together. For a single list MAPCON is equivalent to:

```
(DE MAPCON (FN ARG)
           (COND ((NULL ARG) NIL)
                 (T (NCONC (FN ARG)
                           (MAPCON FN (CDR ARG)))))))
```

Example

```
* (MAPCON (FUNCTION COPY) (QUOTE (1 2 3 4)))
(1 2 3 4 2 3 4 3 4 4)
```

1.5.2.4 (MAPCAN FN ARG) [LSUBR]

MAPCAN is similar to MAPCON except it calls FN with the CAR of successive CDRs of ARG instead of the whole list. For example, a function to remove all of the vowels from a word can be easily written as:

```
(READLIST (MAPCAN (FUNCTION VOWELTEST) (EXPLODE WORD)))
```

where VOWELTEST is a procedure which takes one argument, LET, and returns NIL if LET is a vowel, and (LIST LET) otherwise.

For related information see SET-OF.

1.5.2.5 MAPCONC [LSUBR]

is the same as MAPCAN

1.5.2.6 (MAPLIST FN L) [LSUBR]

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

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

1.5.2.7 (MAPCAR FN L) [LSUBR]

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

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

1.5.2.8 (MAPATOMS fn) [SUBR]

applies fn (a function of one argument) to every atom in OBLIST and returns NIL. It compiles in line.

1.5.3 (FOR-EACH {MAPfn} "FORMAL" LIST "FORM1" ... "FORMn") [MACRO]

FOR-EACH is a MACRO that expands to a form which successively assigns to variable FORMAL an element of LIST and evaluates FORM1 ... FORMn with that variable value. The generated form is (MAPC (FUNCTION (LAMBDA (FORMAL) FORM1 ... FORMn)) LIST). If the optional argument, MAPfn, is included, then that mapping function is used instead of MAPC. Multiple formals may be supplied in a list in which case there must be a LIST argument for each. EXPAND-FE is the function that expands FOR-EACH.

```
(FOR-EACH X '(1 2 3) (PRINT X)) ;; prints 1, 2 and 3.  (Returns nil.)
(FOR-EACH MAPCAR X '(1 2 3) (+ X X))    ;; returns (2 4 6).
(FOR-EACH MAPCAN P PEOPLE
        (COND ((FEMALE P)(NCONS P))))   ;; equivalent to
                        ;; (SET-OF P PEOPLE (FEMALE P))
(FOR-EACH MAPCAR (X Y) '(1 2 3) '(3 2 1) (EQ X Y)) ;; returns (NIL T NIL)
```

1.5.4 (SET-OF <var> <list> <predicate>) [MACRO]

Although SET-OF is related to EXISTS and FORALL from the user's point of view, it actually expands into a call on MAPCAN. It returns a list of those elements of <list> satisfying <predicate>. <var> is bound to the argument (members of <list>) in <predicate>. SET-OF always creates new cells at the top level of the list it returns.

Like other mapping functions, SET-OF can map along several lists at once if <var> is specified as a list of variables and more than one list is given. However the value returned

will only include the members of the first list for which the predicate was satisfied. Any extra arguments will be ignored (only one form is used).

```
(SET-OF X '(1 3 5 2 6) (> X 4)) = (5 6)
(SET-OF (X Y Z) '(1 2 3) '(2 4 6) '(3 4 5) (NEQ Y Z)) = (1 3)
```

1.5.5 PROGRAMS

1.5.5.1 (PROG "VARLIST" "BODY") [FSUBR]

PROG is a function which takes as arguments VARLIST, a list of program variables which are initialized to NIL when the PROG is entered, and a BODY which is a list of labels (which are identifiers) and statements (which are non-atomic S-expressions). PROG evaluates its statements in sequence until either a RETURN or GO is evaluated, or the list of statements is exhausted. In the first case the prog exits with the value passed to the RETURN. In the second case the execution continues at the label passed to the GO. In the last case the prog exits with the value of NIL.

Note: Both RETURN and GO should only occur either at the top level of a PROG, or in compositions of COND, AND, OR, and NOT which are at the top level of a PROG. (Unfortunately,) Prog and Go work at lower levels, and even from functions called in the PROG, but this is usually not intended and can make for bugs that are very hard to find.

1.5.5.2 (GO "ID") [FSUBR]

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

1.5.5.3 (RETURN X) [SUBR]

RETURN causes the PROG containing it to be exited with the value X. RETURN should be used at the top level of a PROG or at the top level of a COND, AND, OR, and NOT which are themselves at the top level of a PROG.

1.5.5.4 (PROG2 X1 X2 ... Xn) [SUBR]

(For n<6) PROG2 evaluates all expressions X1 X2 ... Xn, and returns the value of X2.

1.5.5.5 (PROG1 X1 X2 ... Xn) [SUBR]

(For n<6) PROG1 evaluates all expressions X1 X2 ... Xn and returns X1 as its value.

1.5.5.6 (PROGN X1 X2 ... Xn) [FSUBR]

PROGN evaluates all expressions X1 X2 ... Xn and returns Xn as its value.

1.5.5.7 (SETQ "ID" V) [FSUBR]

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

1.5.5.8 (SET E V) [SUBR]

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

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

1.5.6 SIGNALS

1.5.6.1 (ERRSET E "F") [FSUBR]

ERRSET evaluates the S-expression E and if no error occurs during its evaluation, ERRSET returns a list whose only element is the value computed. If an error occurs, then if F = NIL the error message is suppressed, the break package is not entered and ERRSET returns NIL. If F = 0 (zero) then the error message is printed on the current output device. Otherwise (including the case in which F is not specified) the error message is printed on the teletype.

1.5.6.2 (ERR E) [SUBR]

ERR returns the value of E to the most recent ERRSET, or to the top level if there is none. There is now a special case of ERR. If the value of E is ERRORX, then ERR will return to the most recent ERRSET which has F=ERRORX. This allows two levels of user errors. If a Control-G (or whatever character that has been changed to by ERRCH) is typed in by the user it generates a (ERR (QUOTE ERRORX)). This means that the user can now protect himself against this type of input error.

1.5.6.3 (CATCH "<expr>" {"<label>"}) [FSUBR]

```
(CATCH "‹EXPR›" (("‹L1›" "‹E11›" . . .)) (("‹L2›" "‹E21›" . . .)) . . .)
(THROW ‹VALUE› ("‹LABEL›"))    [FSUBR]
```

CATCH and THROW provide a more convenient method of programming transfers to a higher level in the control hierarchy than ERRSET/ERR, which (as the names imply) were originally designed for error handling rather than planned (programmed) transfers. CATCH

simply evaluates <expr>, and if no THROWs are executed during that evaluation, returns the value of <expr>. If a THROW is evaluated and the CATCH has no <label> then the CATCH is immediately exited with <value> as its value (regardless of whether the THROW had a <label> or not). An unlabeled CATCH will thus catch a value thrown by any THROW. If the CATCH has a <label>, it will catch values thrown only by a THROW with the same label; other THROWs are passed on in search of a higher-level CATCH with a matching label.

Finally, a single CATCH can catch a variety of different THROWs via a SELECTQ-like mechanism as shown above. Each <L> is either a <label> or a list of <label>s; if a THROW <label> matches an <L> or a member of an <L>, the corresponding <e>s are evaluated and the value of the last one is returned as the value of the CATCH. If no labels match, the THROW is passed on in search of a higher level CATCH. Note that a missing THROW <label> is equivalent to a <label> of NIL, and may be caught as such. CATCH and THROW are compiled in-line. The variable THROW is given the value of the first argument to THROW (the value being thrown), and the variable CATCH is bound to the label (if any) specified by the throw.

1.5.6.4 (THROW value {"label"}) [FSUBR]

returns to the next higher CATCH which recognizes its label. The full explanation may be found under CATCH.

1.5.7 REPETITION

1.5.7.1 DO, FOR, UNTIL and WHILE [MACRO]

are all forms of the same iteration macro expanded by EXPAND-DO. The call is scanned for keywords related to a for-loop variable. If the word FOR is found, the next word is taken to be the name of a variable (which is bound in a prog so the other expressions can use it). If one of the symbols {Gets, =, _, :=} is found, the next expression is taken to be the initial value of the loop variable (if there is one). Otherwise it is initialized to the value 1. If one of the symbols {Step, By} is found the next expression is taken as the increment. The default is 1. If the word TO is found, the next expression is taken to be the limit. The default is no limit (loop forever). The call expands into a Prog. First the loop variable is initialized, if there is one. Then comes the body of the loop. Finally the variable is incremented, the new value is tested against the limit (exiting the loop if it is greater - negative increments are not understood), and the loop is restarted. The body of the loop consists of the elements of the list that were not specially interpreted, with their order in the call preserved. The only exceptions are Do, While and Until. Do's are ignored entirely. When a While is found, the next expression is treated as a test. If its value is NIL the loop is exited. Until works the

same way but the exit occurs if the test is non-NIL.

The limit expression is reevaluated on every test. Any exit from the loop from a While test, Until test or increment past the limit results in a value of NIL. It is possible to exit the loop at any time and with any value by using a RETURN. For nil, Step nil, To nil, Gets nil, and their equivalents (using the other words) have the effect of supplying default values for the various parameters. For nil causes the others to be ignored, since the resulting loop has no loop variable. To nil causes the test against the limit to be skipped. The other two parameters default to 1.

```
(For i to 10 (print i)) ;;  print numbers from 1 to 10.
(Do (print 'hello)) ;;  print hello forever.
(While nil (print 'hello)) ;;  do nothing.
(Do (print 'hello) until T) ;;  print hello once.
(For i by 5 until (= i 11) (print i)) ;;  print 1 and 6.
(For i by 5 (print i) until (= i 11)) ;;  print 1, 6 and 11.
(For i to (+i i) (print i)) ;; print 1,2,3, ... (forever).
```

1.5.7.2 (EXPAND-DO form) [SUBR]

is the program that expands the DO, FOR, UNTIL and WHILE macros.

1.6 PREDICATES

A predicate is a test, or a boolean function. Originally predicates were expected to return either T (for true) or NIL (for false), but since the functions that use the values of predicates consider anything other than NIL to mean true, many predicates have been generalized to return more useful values than T. Another useful tidbit is that predicates have traditionally been given names ending in P, such as ZEROP. This convention is not universal, but when you see a function whose name ends with P, chances are good that it's a predicate.

1.6.1 S-EXP-PRED

1.6.1.1 (EQ X Y) [SUBR]

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

For related information see OBLIST.

```
Examples:       (EQ T T)          = T
                (EQ T NIL)        = NIL
                (EQ 'A 'B)        = NIL
                (EQ 1 1.0)        = NIL
                (EQ 1 1)          = T
                (EQ 1.0 1.0)      = NIL
```

1.6.1.2 (NEQ X Y) [SUBR]

returns T if X is not EQ to Y, otherwise NIL.

1.6.1.3 (EQUAL X Y) [SUBR]

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

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

1.6.1.4 (NULL L) [SUBR]

= T if L is NIL, otherwise NIL.

1.6.1.5 (MEMQ X Y) [SUBR]

returns the first tail of Y whose CAR is EQ to X, NIL if there is none. I.e. it returns a non-nil value if X is EQ to an element of Y.

1.6.1.6 MEMB [SUBR]

is the same as MEMQ

1.6.1.7 (MEMBER X Y) [SUBR]

returns the first tail of Y whose CAR is EQUAL to X, NIL if there is none. I.e. it returns a non-nil value if X is equal to an element of Y.

1.6.1.8 (INP X Y) [SUBR]

INP returns T if X is EQ to some subexpression of Y, NIL otherwise. (The search stops at atoms.)

1.6.1.9 (CONSP X) [SUBR]

Returns X if X is a cons cell, otherwise NIL

1.6.1.10 (ATOM X) [SUBR]

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

1.6.1.11 (EQP X Y) [SUBR]

EQP returns T if X and Y are EQ or are EQUAL numbers, otherwise NIL.

1.6.1.12 (LITATOM X) [SUBR]

The value of LITATOM is T if X is a literal atom, i.e., an atom but not a number, otherwise NIL.

1.6.1.13 (PATOM X) [SUBR]

The value of PATOM is T if X is an atom or X is a pointer outside of free storage, otherwise NIL.

1.6.1.14 (STRINGP X) [SUBR]

The value of STRINGP is T if X is a string, otherwise NIL.

1.6.1.15 (TAILP X Y) [SUBR]

The value of TAILP is X if X is a list and a tail of Y, i.e., X is EQ to some number of CDRs (including 0) of Y. The search stops when a CDR returns an atom or NIL. If X is not a tail of Y, then TAILP returns NIL.

1.6.1.16 (BOUNDP X) [SUBR]

BOUNDP returns T if X is a literal atom with a value cell whose cdr is not UNBOUND, i.e., if X is a bound variable (other than a local compiled variable), NIL otherwise.

1.6.2 QUANTIFIERS

The EXISTS and FORALL functions are convenient user interfaces to the lower level SOME and EVERY functions. These are closely related in function to the SET-OF function and the FOR-EACH function. Look them up if the quantifiers don't quite serve your purposes.

1.6.2.1 (SOME SOMEX SOMEFN1 SOMEFN2) [SUBR]

SOME returns the first tail of SOMEX for which SOMEFN1 of its CAR returns a non-NIL value. Otherwise nil is returned. Successive tails of SOMEX, whose first elements are tested

by SOMEFN1, are computed by repeated applications of SOMEFN2. Thus, the function SOMEFN1 is first applied to (CAR SOMEX), then to (CAR (SOMEFN2 SOMEX)), then to (CAR (SOMEFN2 (SOMEFN2 SOMEX))), etc., until the remainder of SOMEX is atomic or NIL. If SOMEFN2 is NIL, then CDR is used.

1.6.2.2 (EVERY EVERYX EVERYFN1 EVERYFN2) [SUBR]

EVERY returns T if the result of applying function EVERYFN1 to each selected element of list EVERYX is non-NIL, NIL otherwise. EVERYFN2 is used to compute successive tails of EVERYX to whose first elements EVERYFN1 will be applied. Thus, the function EVERYFN1 is first applied to (CAR EVERYX), then to (CAR (EVERYFN2 EVERYX)), then to (CAR (EVERYFN2 (EVERYFN2 EVERYX))), etc., until the remainder of EVERYX is atomic or NIL. If EVERYFN2 is NIL, then CDR is used.

1.6.2.3 (EXISTS <var> <list> <predicate> {<next>}) [MACRO]

This expands into a call on SOME in much the way FOR-EACH expands into a mapping function. The effect is that <var> expands into the formal parameter of a function whose body is the predicate, which must be a single expression. <list> must return the list to be searched. The optional <next> actually must return the next TAIL. If <next> is ommited, the function CDR is assumed. <var> will be bound to its CAR. The value returned is the first tail of <list> whose CAR satisfies <predicate> (or NIL if there is none). Only one list and formal parameter may be given. Anything after the optional last argument will be ignored.

```
(EXISTS I '(1 2 3 4 5) (> I 3))
```

This expanses to:

```
(SOME (QUOTE (1 2 3 4 5)) (FUNCTION (LAMBDA (I) (> I 3))) NIL)
```

and returns (4 5)

1.6.2.4 (FORALL <var> <list> <predicate> {<tail-fn>}) [MACRO]

returns T if every element of <list> satisfies <predicate> and NIL otherwise. In the evaluation of <predicate> the element of <list> being tested is bound to <var>. Unlike other maping functions only one list can be given. The optional last argument is a tail computing function (as in EVERY). If <tail-fn> is ommited then the function CDR is assumed. Anything after that argument is ignored. FORALL expands into a call on EVERY in the same way as EXISTS expands into a call on SOME.

1.6.2.5 (NOTEVERY EVERYX EVERYFN1 EVERYFN2) [SUBR]

NOTEVERY is defined to be (NOT (EVERY EVERYX EVERYFN1 EVERYFN2)).

1.6.2.6 (NOTANY SOMEX SOMEFN1 SOMEFN2) [SUBR]

NOTANY is defined to be (NOT (SOME SOMEX SOMEFN1 SOMEFN2)).

1.6.3 NUMERICAL-PRED

1.6.3.1 (NUMBERP X) [SUBR]

= T if X is a number of any type, Nil otherwise

1.6.3.2 (INUMP X) [SUBR]

INUMP returns X if X is an INUM. It returns NIL otherwise.

1.6.3.3 (NUMTYPE X) [SUBR]

returns the type of the number X - FIXNUM (inc. INUM) or FLONUM.

1.6.3.4 (ZEROP X) [SUBR]

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

1.6.3.5 (=0 X) [SUBR]

(=0 X) is identical to (ZEROP X).

1.6.3.6 (ONEP X) [SUBR]

ONEP returns T if X is EQUAL to 1, NIL otherwise.

1.6.3.7 (MINUSP X) [SUBR]

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

1.6.3.8 (= X Y) [SUBR]

(= X Y) is identical to (EQP X Y).

1.6.3.9 (GREATERP X1 X2 ...Xn) [LSUBR]

True if (*GREAT X1 X2) and (*GREAT X2 X3) and ... (*GREAT Xn-1 Xn). Error if any Xi is non-numerical. NIL otherwise.

1.6.3.10 (> X1 ... Xn) [LSUBR]

> is identical to GREATERP.

1.6.3.11 (*GREAT X Y) [SUBR]

Returns Y if X > Y, and NIL otherwise. Error if either X or Y is not a number.

1.6.3.12 (LESSP X1 X2 ... Xn) [LSUBR]

True if X1 to Xn are in strictly ascending numerical order, otherwise NIL.

1.6.3.13 (< X1 ... Xn) [LSUBR]

< is identical to LESSP.

1.6.3.14 (*LESS X Y) [SUBR]

Returns Y if X < Y, and NIL otherwise. It generates an error if either X or Y is not a number.

1.6.4 BOOLEAN-PRED

1.6.4.1 (NOT X) [SUBR]

= T if X is NIL, NIL otherwise

1.6.4.2 (OR X1 X2 ... Xn) [FSUBR]

= The first non-NIL argument or NIL if all Xi are NIL. OR only evaluates it arguments until it finds one that is non-NIL.

1.6.4.3 (AND X1 X2 ... Xn) [FSUBR]

= Xn if all Xi are non-NIL, NIL otherwise. AND only evaluates its arguments up to the first one that is NIL.

1.6.4.4 (BOOLE N X1 X2 ... Xm) [LSUBR]

BOOLE causes a 36 bit Boolean operation to be performed on its arguments. The value of N specifies which of 16 Boolean operations to perform. For m = 2, the ith bit in (BOOLE N A B) is defined: (-X is used as an abbreviation for (not X).)

N	result	N	result
0	0	8	-Ai and -Bi
1	Ai and Bi	9	Ai equiv Bi
2	-Ai and Bi	10	-Ai
3	Bi	11	-Ai or Bi
4	Ai and -Bi	12	-Bi
5	Ai	13	Ai or -Bi
6	Ai neq Bi	14	-Ai or -Bi
7	Ai or Bi	15	1

For m > 2, BOOLE is defined: (BOOLE N ... (BOOLE N (BOOLE N X1 X2) X3) ... Xm)

The method in this madness (in case anyone cares): Let A be 5 (0101 binary) and B be 3 (0011). Then (BOOLE N A B) returns N in the last four bits (for N between 0 and 15).

1.7 FUN-ON-S-EXP

1.7.1 GETTING-COMPONENTS

1.7.1.1 (CAR L) [SUBR]

The CAR of a non-atomic S-expression is the first element of that dotted pair. CAR of NIL is NIL. CAR of any other atom is undefined and tends to lead to an illegal memory reference.

1.7.1.2 (CADR s-exp) [SUBR]

(also CADDR, CDDAAR etc.) All of the compositions of CAR and CDR functions are available up to four As and Ds. e.g.

```
(CADR L) = (CAR (CDR L))
(CDAADR L) = (CDR (CAR (CAR (CDR L))))
```

1.7.1.3 (CDR L) [SUBR]

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

1.7.1.4 (LAST x) [SUBR]

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

```
        (DE LAST (L)
            (COND ((ATOM (CDR L)) L)
                  (T (LAST (CDR L)))))
Examples:       (LAST (QUOTE (A B C))) = (C) = (C.NIL)
                (LAST (QUOTE (A B . C))) = (B.C)
```

1.7.1.5 (NTH X N) [SUBR]

The value of NTH is the tail of X beginning with the Nth element, e.g. if N=2, the value is (CDR X), if N=3, (CDDR X), etc. If N=1, the value is X, if N=0, for consistency, the value is (CONS NIL X).

1.7.2 BUILD

1.7.2.1 BUILD-NONDESTRUCTIVE

1.7.2.1.1 (CONS X Y) [SUBR]

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

For related information see FREE-STG, GC, GCGAG, SPEAK, GCGOT, and METER.

```
Examples:       (CONS (QUOTE A) (QUOTE B)) = (A . B)
                (CONS (QUOTE A) (QUOTE (C))) = (A C)
```

1.7.2.1.2 (XCONS X Y) [SUBR]

= (CONS Y X)

1.7.2.1.3 (NCONS X) [SUBR]

= (CONS X NIL)

1.7.2.1.4 (LIST X1 ... Xn) [FSUBR]

= (CONS X1 (CONS X2 ...(CONS Xn NIL)...)) List evaluates all of its arguments and returns a list of their values.

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

1.7.2.1.5 (QUOTE! "FORM1" ... "FORMn") [FSUBR]

QUOTE! is a complement of the LIST function. LIST forms a list by evaluating each form in the argument list; evaluation is suppressed if the form is QUOTEd. In QUOTE!, each form is implicitly QUOTEd. To be evaluated, a form must be preceded by of one of the evaluate operators ! and !!. ! FORM evaluates FORM and the value is inserted in the place of the call; !! FORM evaluates FORM and the value is spliced into the place of the call. Use of the evaluate operators can occur at any level in a form argument.

```
(QUOTE! CONS ! (CONS 1 2) 3) = (CONS (1 . 2) 3)
(QUOTE! 1 !! (LIST 2 3 4) 5) = (1 2 3 4 5)
(QUOTE! TRY ! '(THIS ! ONE)) = (TRY (THIS ! ONE))
```

1.7.2.1.6 (*APPEND X Y) [SUBR]

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

1.7.2.1.7 (APPEND X1 X2 ...Xn) [LSUBR]

=(*APPEND X1 (*APPEND X2 ...(*APPEND Xn NIL)...))

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

1.7.2.1.8 (COPY X) [SUBR]

returns a copy of X. All of the list cells at all levels are copied. (COPY X) is equivalent to (SUBST 0 0 X).

1.7.2.1.9 (KWOTE X) [SUBR]

KWOTE is defined as (LIST (QUOTE QUOTE) X).

1.7.2.2 BUILD-DESTRUCTIVE

1.7.2.2.1 (NCONC X1 X2 ... Xn) [LSUBR]

NCONC is similar in effect to APPEND, but NCONC does not copy list structures. NCONC modifies list structures by replacing the last element of X1 by a pointer to X2, the last element of X2 by a pointer to X3, etc. The value of NCONC is the modified list X1, which is the concatenation of X1, X2, ..., Xn.

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

1.7.2.2.2 (//NCONC L1 ... LN) [LSUBR]

//NCONC is the same as NCONC except it is undoable.

1.7.2.2.3 (TCONC PTR X) [SUBR]

TCONC is useful for building a list by adding elements one at a time at the end. This could be done with NCONC. However, unlike NCONC, TCONC does not have to search to the end of the list each time it is called. It does this by keeping a pointer to the end of the list being assembled, and updating this pointer after each call. The savings can be considerable for long lists. The cost is the extra word required for storing both the list being assembled, and the end of the list. PTR is that word: (CAR PTR) is the list being assembled, (CDR PTR) is (LAST (CAR PTR)). The value of TCONC is PTR, with the appropriate modifications to its CAR

and CDR. Note that TCONC is a destructive operation, using RPLACA and RPLACD.

```
*(MAPC (FUNCTION (LAMBDA (X) (SETQ FOO (TCONC FOO X))))
       (QUOTE (5 4 3 2 1)))
*FOO
((5 4 3 2 1) 1)
```

TCONC can be initialized in two ways. If PTR is NIL, TCONC will make up a ptr. In this case, the program must set some variable to the value of the first call to TCONC. After that it is unnecessary to reset since TCONC physically changes PTR thus:

```
*(SETQ FOO (TCONC NIL 1))
((1) 1)
*(MAPC (FUNCTION (LAMBDA (X) (TCONC FOO X)))
       (QUOTE (4 3 2 1)))
*FOO
((1 4 3 2 1) 1)
```

If PTR is initially (NIL), the value of TCONC is the same as for PTR=NIL, but TCONC changes PTR, e.g.

```
*(SETQ FOO (NCONS NIL))
(NIL)
*(MAPC (FUNCTION (LAMBDA (X) (TCONC FOO X)))
       (QUOTE (5 4 3 2 1)))
*FOO
((5 4 3 2 1) 1)
```

The latter method allows the program to initialize, and then call TCONC without having to perform SETQ on its value.

1.7.2.2.4 (//TCONC PTR X) [SUBR]

//TCONC is the same as TCONC except it is undoable.

1.7.2.2.5 (LCONC PTR X) [SUBR]

Where TCONC is used to add elements at the end of a list, LCONC is used for building a list by adding lists at the end. For example:

```
*(SETQ FOO (NCONS NIL))
(NIL)
*(LCONC FOO (LIST 1 2))
((1 2) 2)
*(LCONC FOO (LIST 3 4 5))
((1 2 3 4 5) 5)
*(LCONC FOO NIL)
((1 2 3 4 5) 5)
```

Note that LCONC uses the same pointer conventions as TCONC for eliminating searching to the end of the list, so that the same pointer can be given to TCONC and LCONC interchangeably.

```
*(TCONC FOO NIL)
((1 2 3 4 5 NIL) NIL)
*(LCONC FOO (LIST 3 4 5))
((1 2 3 4 5 NIL 3 4 5) 5)
```

1.7.2.2.6 (//LCONC PTR L) [SUBR]

//LCONC is the same as LCONC except it is undoable.

1.7.2.2.7 *NCONC [SUBR]

is the same as NCONC but for only 2 arguments

1.7.2.2.8 //*NCONC [SUBR]

is the same as //NCONC but for only 2 arguments

1.7.2.2.9 (NCONC1 L X) [SUBR]

NCONC1 destructively adds the element X to the end of the list L. It is equivalent to (NCONC L (LIST X)). It generates an error if L is atomic.

1.7.2.2.10 (//NCONC1 L X) [SUBR]

//NCONC1 is the same as NCONC1 except is is undoable.

1.7.2.2.11 (ATTACH X L) [SUBR]

ATTACH destructively attaches element X to the beginning of list L. It generates an error if L is atomic.

1.7.2.2.12 (//ATTACH X L) [SUBR]

//ATTACH is the same as ATTACH except it is undoable.

1.7.2.2.13 (MERGE DATA1 DATA2 COMPAREFN) [SUBR]

MERGE returns the merged list of the two input sorted lists DATA1 and DATA2 using binary comparison function COMPAREFN. (COMPAREFN X Y) should return something non- NIL if X can precede Y in sorted order, NIL if Y must precede X. If COMPAREFN is NIL, LEXORDER will be used. (COMPAREFN should be thought of as "less or equal".) MERGE changes both of its data arguments.

For related information see LEXORDER.

1.7.2.2.14 (INSERT X L COMPAREFN NODUPS) [SUBR]

INSERT destructively inserts element X into list L in sorted order using COMPAREFN as a binary comparison function. (COMPAREFN X Y) should return something non- NIL if X can

precede Y in sorted order, NIL if Y must precede X. If COMPAREFN is NIL, LEXORDER will be used. If NODUPS is non-NIL, an element will not be inserted if an equal element is already in the list. INSERT does binary search to determine where to insert the new element.

1.7.2.2.15 (//INSERT X L COMPAREFN NODUPS) [SUBR]

//INSERT is the same as INSERT except it is undoable.

1.7.3 TRANSFORM

1.7.3.1 TRANSFORM-NONDESTRUCTIVE

1.7.3.1.1 (LENGTH L) [SUBR]

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

```
(DE LENGTH (L)
    (COND ((ATOM L) 0)
          (T (ADD1 (LENGTH (CDR L))))))
```

1.7.3.1.2 (SUBST X Y S) [SUBR]

SUBST returns the result of substituting X for all EQUAL occurrences of Y in S-expression S. SUBST is equivalent to:

```
(DE SUBST (X Y S)
    (COND ((EQUAL Y S) X)
          ((PATOM S) S)
          (T (CONS (SUBST X Y (CAR S))
                   (SUBST X Y (CDR S))))))
Example:        (SUBST 5 (QUOTE FIVE) (QUOTE (FIVE PLUS FIVE IS TEN)))
                = (5 PLUS 5 IS TEN)
```

For related information see DSUBST.

1.7.3.1.3 (REVERSE L) [SUBR]

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

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

For related information see DREVERSE.

1.7.3.1.4 (LDIFF X Y) [SUBR]

Y must be a tail of X, i.e. EQ to the result of applying some number of CDRs to X. LDIFF

gives a list of all elements in X but not in Y, i.e., the list difference of X and Y. Thus (LDIFF X (MEMB FOO X)) gives all elements in X up to the first FOO. Note that the value of LDIFF is always new list structure unless Y=NIL, in which case (LDIFF X NIL) is X itself. If Y is not a tail of X, LDIFF generates an error. LDIFF terminates on a NULL check.

1.7.3.1.5 (LSUBST X Y Z) [SUBR]

Like SUBST except X is substituted as a segment. Note that if X is NIL, LSUBST returns a copy of Z with all Y's deleted. For example:

```
(LSUBST (QUOTE (A B)) (QUOTE Y) (QUOTE (X Y Z))) = (X A B Z)
```

1.7.3.1.6 (SUBLIS ALST EXPR) [SUBR]

ALST is a list of pairs ((U1 . V1) (U2 . V2) ... (Un . Vn)) with each Ui atomic. The value of SUBLIS is the result of (simultaneously) substituting each V for the corresponding U in EXPR. Example:

```
*(SUBLIS (QUOTE ((A . X) (C . Y))) (QUOTE (A B C D)))
(X B Y D)
```

New structure is created only if needed, e.g. if there are no substitutions, value is EQ to EXPR. Note: SUBLIS and SUBPAIR do not substitute copies of the appropriate expression, but substitute the identical structure.

1.7.3.1.7 (SUBPAIR OLD NEW EXPR) [SUBR]

Similar to SUBLIS except that elements of NEW are substituted for corresponding atoms of OLD in EXPR. Example:

```
*(SUBPAIR (QUOTE (A C)) (QUOTE (X Y)) (QUOTE (A B C D)))
(X B Y D)
```

Note: SUBLIS and SUBPAIR do not substitute copies of the appropriate expression, but substitute the identical structure.

1.7.3.1.8 (REMOVE X L) [SUBR]

Removes all top level occurrences of X from the list L, giving a COPY of L with all top level elements EQUAL to X removed.

For related information see DREMOVE.

1.7.3.2 TRANSFORM-DESTRUCTIVE

1.7.3.2.1 (RPLACA X Y) [SUBR]

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

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

Note that this actually changes X, as opposed to creating a new list.

1.7.3.2.2 (//RPLACA X Y) [SUBR]

//RPLACA is the same as RPLACA except it is undoable.

1.7.3.2.3 (RPLACD X Y) [SUBR]

RPLACD replaces the CDR of X by Y. The value of RPLACD is the modified S-expression X. Note: this actually changes X as opposed to creating a new list.

1.7.3.2.4 (//RPLACD X Y) [SUBR]

//RPLACD is the same as RPLACD except it is undoable.

1.7.3.2.5 (DREMOVE X L) [SUBR]

Similar to REMOVE, but uses EQ instead of EQUAL, and actually modifies the list L when removing X, and thus does not use any additional storage. More efficient than REMOVE.

For related information see REMOVE.

NOTE: If X = (L ... L) (i.e. a list of any length all of whose top level elements are EQ to L) then the value returned by (DREMOVE X L) is NIL, but even after the destructive changes to X there is still one CONS cell left in the modified list which cannot be deleted. Thus if X is a variable and it is possible that the result of (DREMOVE X L) might be NIL the user must set the value of the variable given to DREMOVE to the value returned by the function.

1.7.3.2.6 //DREMOVE [SUBR]

the same as DREMOVE but undoable

1.7.3.2.7 (DSUBST X Y Z) [SUBR]

Similar to SUBST, but uses EQ and does not copy Z, but changes the list structure Z itself. DSUBST substitutes with a copy of X. More efficient than SUBST.

For related information see SUBST.

1.7.3.2.8 (//DSUBST X Y Z) [SUBR]

//DSUBST is the same as DSUBST except it is undoable.

1.7.3.2.9 (DREVERSE L) [SUBR]

The value of (DREVERSE L) is EQUAL to (REVERSE L), but DREVERSE destroys the original list L and thus does not use any additional storage. More efficient than REVERSE.

For related information see REVERSE.

1.7.3.2.10 //DREVERSE [SUBR]

//DREVERSE is the same as DREVERSE except it is undoable.

1.7.3.2.11 (SORT DATA COMPAREFN) [SUBR]

SORT destructively sorts the list DATA using COMPAREFN as a binary comparison function. (COMPAREFN X Y) should return something non-NIL if X can precede Y in sorted order, NIL if Y must precede X. If COMPAREFN is NIL, LEXORDER will be used. Pointers to the head of DATA will not generally continue to do so after a SORT. The value returned is a pointer to the new head of the list.

For related information see LEXORDER.

1.7.4 UNDOABLE-FNS

Several destructive list modification functions have undoable variants which work by calling //RPLACA and //RPLACD. These have the same effect as their (permanent) counterparts but they remember the list that was to be discarded, and from where. Thus it becomes possible to undo the effects of those functions (see the top level command UNDO and UNDOERRSET). It should be recognized that this facility is not foolproof. For example you can get into trouble by undoing things in the wrong order.

1.7.4.1 #UNDOSAVES

#UNDOSAVES (initially -1) can be reset to indicate when the user should be prompted concerning saving additional inverses of undoable changes. If it is less than 1 no inverses will be saved.

1.7.4.2 (UNDOERRSET "form") [FSUBR]

UNDOERRSET is like ERRSET with second argument NIL since it evaluates form and does not

generate a break if an error occurs. UNDOERRSET locally records undoable changes for the // functions (see UNDOABLE-FNS) and undoes them if an error occurs. If no error occurs the undoable changes are recorded with the event (as usual so they can be undone by the UNDO command in the top level) and a list whose only element is the result of evaluating form is returned (just like ERRSET).

1.7.5 SEARCH

1.7.5.1 (ASSOC X L) [SUBR]

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

ASSOC is equivalent to:

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

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

1.7.5.2 (ASSOC* X Y) [SUBR]

Similar to ASSOC, but uses EQUAL instead of EQ.

1.7.5.3 (SASSOC X L FN) [SUBR]

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

```
(DE SASSOC (X L FN)
    (COND ((NULL L) (FN))
          ((EQ X (CAAR L)) (CAR L))
          (T (SASSOC X (CDR L) FN))))
Example:        (SASSOC 0 (QUOTE ((1.ONE) (2.TWO)))
                          (FUNCTION (LAMBDA NIL (QUOTE LOSE)))) = LOSE
```

1.8 PROPERTY-LIST

Every atom in the OBLIST has a "property list" which is what you see if you ask for the PLIST of the atom. (CDR happens to do the same thing except in the case of NIL which keeps its property list elsewhere. Use of CDR is more implementation-dependent.) The property list is a list which alternates between property names and the corresponding property values. See PROPERTIES for a list of properties used by the system.

1.8.1 (GET I P) [SUBR]

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

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

1.8.2 (GETL I L) [SUBR]

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

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

1.8.3 (PUTPROP I V P) [SUBR]

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

1.8.4 (//PUTPROP I V P) [SUBR]

//PUTPROP is the same as PUTPROP except it is undoable.

1.8.5 (REMPROP I P) [SUBR]

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

1.8.6 (//REMPROP I P) [SUBR]

//REMPROP is the same as REMPROP except it is undoable.

1.8.7 (PLIST x) [SUBR]

returns the property list of the atom x (which is the CDR of x if x is not NIL).

1.8.8 PNAME

PNAME is the name of the property under which print-names of atoms are stored.

For related information see OBLIST.

1.8.9 PROPERTIES

```
        ;; This is a list of properties used by the system.  They
        ;;should therefore be avoided for purposes other than those
        ;;for which the system uses them.  Many of these are described
        ;; elsewhere.
PNAME           VALUE           EXPR            FEXPR
MACRO           ERXACTION       BROKEN          BROKEN-IN
SIDE            UNDEF           SUBR            FSUBR
LSUBR           LEXPR           NAMED-EVENT     BRKARGS
ALIAS           TRACED-IN       TRACED          SYM
EDIT-SAVE       LASTVALUE       TRACE           CHANGES
PRINTMACRO      NAMESCHANGED    PPCOM           LOSTPROP
COMMENT         FIXNUM          FLONUM          EXTRAFNS
CALLS           CONSES          MSEC
FUNTYPE         READMACRO       %READIN         BEFORE-SIDE
```

Properties are used by LISP to interpret everything. For example when you do an assignment such as (SETQ X 1), the atom X is given a VALUE property containing 1. When you define a function called X the body of the function will be added as a different property of X. Properties are stored in PROPERTY-LISTs.

For related information see OBLIST.

1.9 IDENTIFIERS

1.9.1 OBLIST

In order for atoms with the same print-names to be recognized as the same (EQ) LISP keeps a symbol-table in the form of a special list called OBLIST. It is organized as a hash table (a list of buckets) , each element of which is a list of identifiers. Property lists are stored as part of the identifiers.

For related information see PROPERTIES and PROPERTY-LIST.

1.9.2 (INTERN I) [SUBR]

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

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

1.9.3 (REMOB "X1" "X2 ... "Xn") [FSUBR]

and REMOB [VALUE]

REMOB removes all of the identifiers X1, X2, ... , Xn from the OBLIST and returns NIL. None of the Xi's are evaluated. See NOCALL for explanation of the value.

1.9.4 (REMOB1 "id") [SUBR]

is the same as REMOB but for only one argument.

1.9.5 (GENSYM) [SUBR]

GENSYM increments the generated symbol counter and returns a new identifier specified by the counter. The GENSYM counter is initialized to the identifier G0000. Successive executions of (GENSYM) will return G0001, G0002, G0003, ... Note: GENSYM does not INTERN its result.

1.9.6 (CSYM "I") [FSUBR]

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

```
Example:        (CSYM ARY00)     = ARY00
                (GENSYM)         = ARY01
                (GENSYM)         = ARY02
```

etc.

1.10 IDENTIFIER-NAMES

1.10.1 (EXPLODE L) [SUBR]

EXPLODE transforms an S-expression into a list of single character identifiers identical to the sequence of characters which would be produced by PRIN1. These identifiers are always symbols (literal atoms), never numbers.

For related information see LOWER-CASE.

```
(EXPLODE '(DX //- DY)) = (/( D X / // // - / D Y /))
(EXPLODE 'F1) = (F /1)
```

1.10.2 (EXPLODEC L) [SUBR]

EXPLODEC transforms an S-expression into a list of single character identifiers identical to the sequence of characters which would be produced by PRINC. These identifiers are always symbols (literal atoms), never numbers.

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

1.10.3 (FLATSIZE L) [SUBR]

```
= (LENGTH (EXPLODE L))
```

For related information see LOWER-CASE.

1.10.4 (FLATSIZEC L) [SUBR]

```
= (LENGTH (EXPLODEC L))
```

1.10.5 (MAKNAM L) [SUBR]

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

For related information see LOWER-CASE and RDNAM.

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

1.10.6 (READLIST L) [SUBR]

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

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

For related information see LOWER-CASE.

1.10.7 (LEXORDER X Y) [SUBR]

The value of LEXORDER is T iff X is lexically less than or equal to Y. Note: Both arguments must be atoms. Numeric arguments are all lexically less than symbolic atoms.

```
Examples:    (LEXORDER (QUOTE ABC) (QUOTE CD))      = T
             (LEXORDER (QUOTE B) (QUOTE A))         = NIL
             (LEXORDER 123999 (QUOTE A))            = T
             (LEXORDER (QUOTE B) (QUOTE B))         = T
```

For related information see SORT and MERGE.

1.10.8 (SUBSTRING str m n) [SUBR]

Returns a new string consisting of characters m through n of str. m and n may be positive integers (count from left) or negative integers (count from right). If m is non-numeric a value of 1 (first character) is assumed, and if n is non-numeric -1 (last character) is assumed. Although str will typically be a string, SUBSTRING will actually work with any argument; the indicated characters are simply extracted from the PRINC character string of str and formed into a string.

1.10.9 (EQSTR at1 at2) [SUBR]

Compares the PNAMEs of at1 and at2, returning T if they are identical and NIL if they differ. Useful for use with any uninterned atomic symbols (including, of course, strings).

1.10.10 (EQNAM X Y) [SUBR]

EQNAM is slower than EQSTR, but does more checks and is more general. Two quantities are EQNAM if they are EQP or if they are both LITATOM and EQSTR. This means that quantities are EQNAM if they are EQ, or they are EQUAL numbers, or they are atoms that print out the same.

1.10.11 (NTHCHAR X N) [SUBR]

```
  = (CAR (NTH (EXPLODEC L) N))    if N>0
= (CAR (NTH (REVERSE (EXPLODEC L)) N))    if N<0
= NIL    if (ABS N) = 0 or > (FLATSIZEC L)
```

1.10.12 (CHRVAL X) [SUBR]

CHRVAL returns the ASCII representation of the first character of the print name of X.

1.10.13 (ASCII N) [SUBR]

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

Example: (ASCII 65) is an identifier with print name "A".

Note: ASCII does not INTERN its result.

1.10.14 (BIGRATOM n) [SUBR]

returns the atom made up of the input characters up to the first occurrence of the character whose ascii code is n. This is used for reading comments.

1.11 ARITHMETIC

1.11.1 (ABS X) [SUBR]

= absolute value of X

1.11.2 (ADD1 X) [SUBR]

= X+1

1.11.3 (+I X) [SUBR]

(+I X) is identical to (ADD1 X).

1.11.4 (*DIF X Y) [SUBR]

= X - Y

1.11.5 (DIFFERENCE X1 X2 ... Xn) [~~MACRO~~ LSUBR]

= X1 - X2 - ... - Xn

1.11.6 (- X1 ... Xn) [LSUBR]

- is identical to DIFFFERENCE.

1.11.7 (MINUS X) [SUBR]

= -X

1.11.8 (DIVIDE X Y) [SUBR]

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

1.11.9 (FIX X) [SUBR]

returns the largest integer not greater than X (floor function).

1.11.10 (GCD X Y) [SUBR]

returns the greatest common divisor of integers X and Y.

1.11.11 (LSH X N) [SUBR]

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

1.11.12 (*MAX X Y) [SUBR]

Maximum of X and Y.

1.11.13 (MAX X1 X2 ... Xn) [LSUBR]

Maximum of X1 ... Xn.

1.11.14 (*MIN X Y)[SUBR]

Minimum of X and Y.

1.11.15 (MIN X1 X2 ... Xn) [LSUBR]

Minimum of X1 ... Xn.

1.11.16 (*PLUS X Y) [SUBR]

= X + Y

1.11.17 (PLUS X1 X2 ... Xn) [LSUBR ~~MACRO~~]

= X1 + X2 + ... + Xn

1.11.18 (+ X1 ... Xn) [LSUBR]

+ is identical to PLUS.

1.11.19 (*QUO X Y) [SUBR]

= X / Y

For integer arguments this returns the integer part of the answer.

1.11.20 (QUOTIENT X1 X2 ... Xn) [LSUBR ~~MACRO~~]

= X1 / X2 / ... / Xn For integer arguments this returns the integer part of the answer.

1.11.21 (// X1 ... Xn) [LSUBR]

// is identical to QUOTIENT.

1.11.22 (REMAINDER X Y) [SUBR]

= X - (X / Y) * Y

It is NOT defined for non-integer arguments.

1.11.23 (SUB1 X) [SUBR]

= X - 1

1.11.24 (-I X) [SUBR]

(-I X) is identical to (SUB1 X).

1.11.25 (*TIMES X Y) [SUBR]

= X * Y

1.11.26 (TIMES X1 X2 ... Xn) [MACRO]

= X1 * X2 * ... * Xn

1.11.27 (* X1 ... Xn) [LSUBR]

* is identical to TIMES.

1.11.28 SCIENTIFIC-SUBR

Scientific functions have been set up by Cris Perdue for use with LISP based on code stolen from MIT by Dan Hinsley. This is NOT the UCI-FORTRAN package. The functions currently reside on ARITH.MAC, ARITH.REL on [A311L15P]. This version contains a couple of kludges and may be changed when the cause of the kludges is fixed. Beware of copying these files since the current versions may not work when things are fixed. These algorithms are available "as is". All we did was interface them to our LISP. Use the LOAD function to load them. Arguments out of range cause normal LISP errors with messages appropriate to each function. It is conceivable that overflow or underflow may occur with theoretically in-range arguments. This will cause a generic arithmetic overflow error from LISP.

1.11.28.1 (SIN X) [SUBR]

where x is in radians returns the sin of x. This function is NOT built in but must be loaded by the user.

1.11.28.2 (COS x) [SUBR]

returns the cosine of x. where x is in radians. It is NOT built in but must be loaded by the user.

1.11.28.3 (ATAN x y) [SUBR]

is not built in but must be loaded by the user. ATAN returns the angle in radians of which the tangent is y/x. The signs of y and x determine the quadrant of the result.

1.11.28.4 (SQRT x) [SUBR]

returns the square root of x. It is NOT built in but must be loaded by the user.

1.11.28.5 (LOG x) [SUBR]

returns the natural log of x. It is NOT built in but must be loaded by the user.

1.11.28.6 (EXP x) [SUBR]

returns e raised to the power x. It is NOT built in but must be loaded by the user.

1.11.29 OVERFLOW

An overflow message comes from arithmetic functions when they detect either fixed or floating point overflows (i.e. when they compute a result that does not fit into one machine word).

1.12 (ARRAY "ID" TYPE B1 B2 ... Bn) [FSUBR]

(For n<6) ARRAY is a function which declares an array with name ID, and places an array referencing function on the property list of ID.

For related information see BPS.

TYPE determines the type of an array as follows:

```
TYPE      INITIAL VALUE   ARRAY ELEMENT
T         NIL             LISP S-expressions stored as
                          pointers;; 2 per word.
NIL       0.0             REAL numbers stored one per word in
                          ;;PDP-6/10 floating point representation.
36.       0               36 bit 2/s complement integers stored
                          ;;1 per word.
0<n<36.   0               n bit positive integers packed
                          ;;[36./n] per word.
```

B1 B2 ... Bn are array subscript bounds which should evaluate to either positive integers Si, or to dotted pairs of integers (Li . Ui) where Li .le. Ui, which specify lower and upper subscript bounds as follows:

```
Bi        LOWER BOUND     UPPER BOUND     LENGTH
Si        0               Si - 1          Si
(Li . Ui) Li              Ui              Ui-Li + 1
```

The elements of an array are referenced by: (<array name> I1 I2 ... In) where Lj .le. Ij < Uj. Note: Both ARRAY and EXARRAY consume Binary Program Space .

The ARRAY subscripts Ij must be integers. References to memory locations outside of the area reserved for the array are prohibited and will cause an illegal memory reference message. Array elements are stored in BINARY PROGRAM SPACE. Redeclaring arrays will allocate new binary program space - the old space will not be reclaimed.

1) To declare a 1 dimensional array CHARS of 7 bit characters and with subscripts 1 to 50: (ARRAY CHARS 7 (QUOTE (1 . 50))) The first element of CHARS is referenced: (CHARS 1)

2) To declare a 2-dimensional array A of REAL numbers and with subscripts 0 .le. i < N, 0 .le. j < M: (ARRAY A NIL N M)

3) To declare a 1-dimensional array FOO of S-expressions and with subscripts -K .le. i .le. K: (ARRAY FOO T (CONS (MINUS K) K))

1.12.1 (EXARRAY "ID" TYPE B1 B2 ... Bn) [FSUBR]

(For n<6) EXARRAY is identical to ARRAY except that array elements are stored in the body of a subroutine loaded by the LOADER and exarray elements are not initialized. The array referencing subroutine is stored in BINARY PROGRAM SPACE as with ARRAY. EXARRAY searches symbol tables as does GETSYM . Note: Both ARRAY and EXARRAY consume BINARY PROGRAM SPACE. If there is insufficient room there the error message "BINARY PROGRAM SPACE EXCEEDED" will result.

1.12.2 (STORE ("ID" i1 i2 ... in) value) [FSUBR]

STORE changes the value of the specified array element to value, and returns value. Note: STORE evaluates its second argument first.

Examples: With the arrays declared previously:

```
(STORE (FOO 0) (QUOTE (A B)))
(STORE (FOO (BAZ L)) 1)
(STORE (A I J) (A J I))
(STORE (CHARS 1) 17)
```

1.13 MEASUREMENT

"Measurement" has to do with finding out how expensive your programs are to run. Naturally, the most interesting measures of this are time and space. Metering tells you how often functions are called, how much time they take and how much space they use (in terms of the number of CONSes they do). The timimg functions tell how much CPU time has been used (or how much of it has been used garbage-collecting). Counting tells you how often each branch of a program is executed. It is good for finding out which sections are used often (and thus ought to be optimized) and which are used seldom (and thus may not be debugged).

1.13.1 (METER "F1" ... "Fn") [FSUBR]

METER sets up functions F1, ..., Fn for metering. Each function may be specified as an

atomic function name, (function name IN function name), or (function name , meter condition), just as for BREAK.

1.13.1.1 (UNMETER "F1" ... "Fn") [FSUBR]

UNMETER stops metering of functions F1, ..., Fn and prints their statistics. (UNMETER) unmeters all the metered functions and (UNMETER T) unmeters the most recently metered function, just as for UNBREAK and UNTRACE.

1.13.1.2 ZZMC1

ZZMC1 (and ZZMC2, ZZMC3, ZZMC4) are constants which compensate for the overhead in the METER package.

1.13.1.3 BREAK1M [SUBR]

Performs the overhead when a function is metered

1.13.1.4 (METERS "F1" ... "Fn") [FSUBR]

METERS prints the statistics for F1, ..., Fn without unmetering them. (METERS) prints the statistics for all metered functions and (METERS T) prints the statistics for the most recently metered function.

1.13.1.5 METEREDFNS [VALUE]

is just a list of the functions currently being METERed.

1.13.2 (COUNT "fn1" "fn2" ...) [FSUBR]

modifies the definitions of the (interpreted) functions in the argument list to keep track of the number of times each piece of code in the definition is evaluated. Every sub-expression that is to be evaluated is replaced by (# 0 <sub-expression>). To see how many times it has been evaluated simply look at the function definition (via PP or DSKOUT). This facility is good for finding the sections of code that are executed much more often than expected (and thus should be optimized) or less often (and thus may contain bugs even though the program works). UNCOUNT removes the counters.

1.13.2.1 (UNCOUNT "fn1" "fn2" ...) [FSUBR]

undoes the counting for the given functions. Like UNTRACE and UNBREAK, a null argument causes COUNTEDFNS to be used, and an argument of T causes the last counted function to be used.

1.13.2.2 COUNTEDFNS

COUNTEDFNS is a list of functions that are COUNTed.

1.13.2.3 (COUNT1 fn) [SUBR]

Does the same thing as COUNT but is a SUBR and takes only one function

1.13.2.4 (UNCOUNT1 fn) [SUBR]

Does the same thing as UNCOUNT but is a SUBR and takes only one function for an argument.

1.13.2.5 (# <number> <expression>) [FSUBR]

is used for counting (see COUNT). It merely increments the number and returns the result of evaluating the expression.

1.13.2.6 #-ERROR

When functions are UNCOUNTed (see COUNT, UNCOUNT, #) a few checks are made. If the call on # (the counting function) is not of the right form then the user is warned and the questionable portions are not changed. When this error occurs the offending functions should be fixed (in the editor) and then UNCOUNTed again. To fix such a function look for calls to # (via "F #" in the editor). Either the first argument is non-numeric or there are more than two arguments. See # for the correct form.

1.13.3 (TIME) [SUBR]

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

For related information see DATESTR and MSTIME.

1.13.4 (GCTIME) [SUBR]

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

1.13.5 (TIME-GCTIME) [SUBR]

TIME-GCTIME returns the difference of run time and garbage collection time in milliseconds,

i.e., (*DIF (TIME) (GCTIME)). Note that the difference in values of (TIME-GCTIME) for two occasions is the run time excluding garbage collection time for the intervening computation. Run time excluding garbage collection time is often a more meaningful statistic than just run time since garbage collection occurring at "random" times can invalidate run times, and frequency of garbage collection is dependent on amount of free storage allocated as well as the algorithm whose time is being measured.

1.13.6 (SPEAK) [SUBR]

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

2. INPUT-OUTPUT

2.1 SAVE-STATE

Users typically define functions in lisp and then want to save them for the next session. If you do (CHANGES), a list of the functions that are newly defined or changed will be printed. When you type (DSKOUTS), the functions associated with files will be saved in the new versions of those files. In order to associate functions with files you can either add them to the filefns list of an existing file or create a new file to hold them. This is done with the FILE function. If you type (FILE NEW) the system will create a variable called NEWFNS. You may add the names of the functions to go into that file to NEWFNS. After you do (CHANGES) the functions which are in no other file are stored in the value of the atom CHANGES. To put these all in the new file, (SETQ NEWFNS (APPEND NEWFNS CHANGES)). Now if you do (CHANGES), all of the changed functions should be associated with files. In order to save the changes on the files, do (DSKOUTS). All of the changed files (such as NEW) will be written. To recover the new functions the next time you run LISP, do (DSKIN NEW).

(Only blank lines were deleted from this recording, comments are added)

```
<2>(de square(x) (* x x))                          ;; define a new function
SQUARE                                             ;;
<3>(changes)                                       ;;
FILE            FUNCTIONS                          ;;
<NO-FILE>       SQUARE                             ;;
T                                                  ;;
<4>(file new)                                      ;;
NEW                                                ;;
<5>(setq newfns (append newfns changes))           ;; add the functions which
                ;; are associated with no file to the new file, NEW
((QUOTE (VERSION 0 "19-JUL-78 15:30:59")) SQUARE) ;; the value of NEWFNS
<6>(changes)                                       ;;
FILE            FUNCTIONS                          ;;
NEW             SQUARE                    ;;square is now associated with NEW
T                                                  ;;
<7>(dskouts)                                       ;;
NEW                                                ;; the file is written
NIL                                                ;;
<8>(dskin new)                                     ;; we now read it in
NEWFNS (VERSION 1 "19-JUL-78 15:31:43") (SQUARE EQUAL)
FILES-LOADED                    ;; SQUARE is read in (same as before)
```

2.1.1 (DSKIN "LIST OF FILE-NAMES") [FSUBR]

READ-EVAL-PRINTs the contents of the given files. This is the function to use to read files created by DSKOUT. DSKIN also declares the files that it reads (if a FILE-FNS list is defined and the file is otherwise declarable by FILE), so that changes to it can be recorded. DSKIN [VALUE] determines how the expressions read are reported. If it is NIL then nothing will be printed. If it is PRINT then the values are printed one per line, and if the value is T (the default) the values are printed with new lines starting only when a value does not fit on the

current line.

Example:

```
(DSKIN (FUNCS.LSP) DTA0: (DATA.LSP))
```

Reads FUNCS.LSP from DSK: and DATA.LSP from DTA0:.

```
(DSKIN (C410 HB00) (DSKLOG.LSP))
```

Reads DSKLOG.LSP from the disk area of [C410HB00].

2.1.2 (DSKOUTS "FILE1" ... "FILEn") [FSUBR]

DSKOUTS applies DSKOUT to and prints the name of each FILEi (with no additional arguments, assuming filenameFNS to be a list to be GRINLed) for which FILEi is either not in FILELST (meaning it is a new file not previously declared by FILE or given as an argument to DSKIN, DSKOUT, or DSKOUTS) or is in FILELST and has some recorded changes to definitions of atoms in filenameFNS, as recorded by MARK!CHANGED and noted by CHANGES. If FILE1 ... FILEn is not specified, FILELST will be used. This is the most common way of using DSKOUTS. Typing (DSKOUTS) will save every file reported by (CHANGES) to have changed definitions.

For related information see GRINPROPS.

2.1.3 (%READIN channel print) [SUBR]

executes a loop which reads from the specified channel, evaluates what was read and if the second argument is non-NIL prints the result. It terminates when the read generates an error.

2.1.4 (FILE "FILE") [FSUBR]

FILE declares its argument to be a file to be used for reporting and saving changes to functions by adding the file name to a list of files FILELST. It adds the file name to FILELST only if the extension is not LAP, LBK, or Lnn (nn some integer). It also prevents you from declaring a file which would use the same FILEFNS list as a current LIBRARY file. FILE is called for each file argument of DSKIN, DSKOUT, and DSKOUTS. If the user executes the FILE function, the associated fileFNS list has a version number (initially 0) added to its beginning if none exists. DSKOUT automatically increments this number. FILE understands devices and ppns. If you declare a file which has the same name (excluding device, ppn and extension, i.e. the same filefns list) then the new declaration will supersede the old one. Thus, if you want (FILE.EXT) to be put on TEMP when you do a DSKOUT, just redeclare it: (FILE TEMP:

(FILE.EXT))

2.1.5 FILELST [VALUE]

is the list of files whose contents are (supposedly) contained in the current core image.

For related information see LIBRARIES.

2.1.6 (FILE-FNS FILE) [SUBR]

FILE-FNS returns the name of the fileFNS list for its file argument FILE.

2.1.7 (CHANGES flag) [FSUBR]

CHANGES [PROPERTY] and CHANGES [VALUE]

Changes computes a list containing an entry for each file which defines atoms that have been marked changed. The entry contains the file name and the changed atoms defined therein. There is also a special entry for changes to atoms which are not defined in any known file. (FILELST contains the "known" files.) If no flag is passed this result is printed in human readable form and the value returned is T if there were any changes and NIL if not. Otherwise nothing is printed and the computed list is returned. The global variable CHANGES contains the atoms which are marked changed but not (yet) associated with any file. The CHANGES function attempts to associate these names with files, and any that are not found are considered to belong to no file. The CHANGES property is the means by which changed functions are associated with files. When a file is read in or written out its CHANGES property is removed.

2.1.8 (MARK!CHANGED F) [SUBR]

MARK!CHANGED records the fact that the definition of F has been changed. It is automatically called by DE, DF, DEFPROP, DM, DV, and the editor when a definition is altered.

2.1.9 FILE-SEARCH

2.1.9.1 (GETDEF "FILE" "I1" ... "In") [FSUBR]

GETDEF selectively executes definitions for atoms I1, ..., In from the specified file. Any of the words to be defined which end with "@" will be treated as patterns in which the @ matches any suffix (just like the editor). GETDEF is driven by GETDEFTABLE (and thus may

be programmed). It looks for lines in the file that start with a word in the table. (The first character must be a "(" or "[" followed by the word followed by a space, return or something else that will not be considered as part of the identifier by RDNAM - "(" i unacceptable.) When one is found the next word is read. If it matches one of the identifiers in the call to GETDEF then the table entry is executed. The table entry is a function of the expression starting in this line. Output from DSKOUT is in acceptable format for GETDEF. GETDEF returns a list of the words (which match the ones it looked for) for which it found (but not necessarily executed) definitions in the file.

2.1.9.1.1 GETDEFPROPS [VALUE]

is used by the standard programs in GETDEFTABLE to decide whether an expression read by GETDEF is to be executed. For example, DE's are executed if GETDEFPROPS contains EXPR or is NIL. DC's are evaluated if GETDEFPROPS contains COMMENT or is NIL. DEFPROP's enter properties if they are in GETDEFPROPS or GETDEFPROPS is NIL. This is meant to provide a convenient way to program GETDEF. The initial value of GETDEFPROPS is (EXPR FEXPR MACRO VALUE SUBR FSUBR LSUBR).

2.1.9.1.2 GETDEFTABLE [VALUE]

is the table that drives GETDEF. It is in the form of an association list. Each element is a dotted pair consisting of the name of a function for which GETDEF searches and a function of one argument to be executed when it is found.

2.1.9.1.3 (GETDEFACT id prop exp) [SUBR]

is used by GETDEF to do what PUTPROP does with the following bells and whistles added: If an atom's property was not previously defined and is given a definition, the atom, property, and "DEFINED" will be printed. If the atom's property was previously defined, the atom, property, and "EQUAL" or "***REDEFINED***" will be printed, depending on whether the new definition is EQUAL to the old definition. (LAP function definitions are not checked for equality.) For each definition of one of the desired atoms defining a property not selected by GETDEFPROPS, the atom, property, and "BYPASSED" will be printed.

2.1.9.1.4 (GETDEFEVAL "ID" exp "PROP") [FSUBR]

simply evaluates exp. It is useful only because it is recognized by GETDEF, which only executes the GETDEFEVAL if PROP is acceptable .

2.1.9.2 (LIBRARY "file1" "file2" ...) [FSUBR]

LIBRARY declares files to be libraries. A library is a file that has not been read in (and

thus is not in FILELST), but its contents are known. After a file is declared as a library, you can get function definitions and comments from it (with USERHELP and GETDEFNS) without having to mention it by name. LIBRARY understands devices and ppns just like DSKIN. The compiler will put into every LAP file a declaration of its source as a library. Thus if your compiled function FOO does not work you can find out what it does or get the interpreted code with USERHELP or GETDEFNS. When a file is DSKIN'd its LIBRARY declaration automatically goes away.

2.1.9.3 LIBRARIES [VALUE]

is the counterpart to FILELST. However the files in LIBRARIES have not been (officially) read. To add a file to LIBRARIES use the LIBRARY function.

2.1.9.4 (GETDEFNS fn1 fn2 ...) [MACRO]

tries to find the names in the argument list (with FINDFILES using LIBRARIES and FILELST), and does a GETDEF from those files of the words given. Note that if a word is defined in two different files then they will both be read (in arbitrary order).

2.1.9.5 (USERHELP word1 word2 ...) [FSUBR]

USERHELP is supposed to be just like HELP, but for getting user comments. Like HELP, it accepts the "@" to mean "any suffix". It looks for comments in the files in FILELST and those in LIBRARIES. Also like HELP, it decides whether or not to show the comment by calling HELPFILTER.

2.1.9.6 (FINDFNS file-list name-list) [SUBR]

returns a (sorted) list of names. Any name in name-list that does not end with "@" will be in the list. Those that do end with "@" match any name that can be made by replacing the "@" with a suffix. FINDFNS looks for such names in the FILEFNS lists of the files in the file-list, and in the functions marked as having been changed but associated with no file. Typically the file-list will be either FILELST or LIBRARIES.

2.1.9.7 (FINDFILES file-list name-list) [SUBR]

returns those members of the file-list which contain names in the name-list. The name-list may have words that end with "@" to mean any suffix. The file-list is normally either FILELST or LIBRARIES. In order to determine whether a name is in a file, the file must at least have a FILEFNS list. This function is used by GETDEFNS and USERHELP to decide which files to search.

2.1.10 (DSKOUT "FILE" "FORM1" ... "FORMn") [FSUBR]

DSKOUT is an extension of the UCI LISP DSKOUT function. If FORM1 ... FORMn is specified, each FORMi (or (GRINL FORMi) if FORMi is atomic) is evaluated with all printing directed to FILE. Any previous version of FILE will be renamed to have extension LBK, deleting a previous LBK file if necessary.

For related information see GRINPROPS, DSKOUTS, GRINL, FILE, and FILE-FNS.

If FORM1 ... FORMn is not specified, i.e., only a file argument is specified, DSKOUT assumes the list named filenameFNS (i.e., the file name, excluding extension, concatenated with FNS) contains a list of function names, etc., to be GRINLed. DSKOUT adds (if not already present) as first element of filenameFNS a list containing a version number and creation time for the file. The version number and creation time are updated each time the file is DSKOUTed (but only when FORM1 ... FORMn are not specified). Any previous version of FILE will be renamed to have extension Lnn where nn is the previous version number. Successive uses of DSKOUT with only a file argument will create a sequence of files FILE.L01, FILE.L02, ..., FILE.Lnn, FILE, leaving a history of changes to the file.

DSKOUT recognizes line printer device names LPT:, LPT0:, and LPT1: and suppresses the attempts to delete old backup files and rename the current file to be a backup file (illegal operations for the line printer). A file name must still be specified when DSKOUTing to the line printer. The FILE function will not be called for files DSKOUTed to the line printer. Thus, a file printed by DSKOUT on the line printer will still (if there are changes) be written to the disk by (DSKOUTS).

2.1.10.1 COMMENT [PROPERTY]

is the name of the property in which comments are saved.

2.1.10.1.1 (DC word {id} {(descriptor1 descriptor2 ...)}) <text> <esc> [FSUBR]

DC defines comments. It is exceptional in that its behavior is very context dependent. When DC is executed from DSKIN it simply records the fact that the comment exists. It is expected that in interactive mode comments will be found via GETDEF (as in HELP) - this allows large comments which do not take up space in your LISP core image. When DC is executed from the terminal it expects you to type a comment. DSKOUT will write out the comments that you define and also copy the comments on the old version of the file, so that the new version will keep the old comments even though they were never actually brought into core. The optional id is a mechanism for distinguishing among several comments

associated with the same word. It defaults to NIL. However if you define two comments with the same id, the second is considered to be a replacement for the first. The optional list of attributes is meant to be passed (along with the word being commented) to HELPFILTER in order to decide whether or not to print the comment as HELP.

The behavior of DC is determined by the value of the global variable DEF-COMMENT. DEF-COMMENT contains the name of a function that is run. Its arguments are the word, id and attribute list. DEF-COMMENT is initially DC-DEFINE. Other functions rebind it to DC-HELP, DC-USERHELP, and the value of DSKIN-COMMENT.

The comment property of an atom is a list of entries, each representing one comment. Atomic entries are assumed to be identifiers of comments on a file but not in core. In-core comments are represented by a list of the id, the attribute list and the comment text. The comment text is an uninterned atom. Comments may be deleted or reordered by editing the comment property.

2.1.10.1.2 DEF-COMMENT [VALUE]

names a function that will be called when a DC is done. It is initially DC-DEFINE, but is rebound by various functions to DC-HELP, DC-USERHELP etc. depending on how comments are to be treated at various times. Of course, this provides a handy way to program your own treatment of comments too.

2.1.10.1.3 (DC-DEFINE name id attributes) [SUBR]

is the function that defines comments. It reads a comment and puts it in the right place.

2.1.10.1.4 (DC-DSKIN name id attributes) [SUBR]

is the function that DC normally calls during a DSKIN. It simply records the existence of the comment, without actually reading it in.

2.1.10.1.5 (DC-HELP name id attributes) [SUBR]

is the function that DC calls during a HELP. It simply decides whether to print the comment (by calling HELPFILTER) and then either prints it or skips it.

2.1.10.1.6 (DC-IGNORE) [SUBR]

is the function that DC-DSKIN calls to skip a comment.

2.1.10.1.7 (DC-USERHELP name id attributes) [SUBR]

is the function that DC uses during a USERHELP. It is like DC-HELP except that it does some extra bookkeeping to tell you what may be wrong

2.1.10.1.8 DSKIN-COMMENT [VALUE]

contains the value give to DEF-COMMENT when DSKIN is called. It is initially DC-DSKIN.

2.1.10.1.9 (*** comment) [FSUBR]

is the standard comment function. It returns NIL. It also has a special printmacro property which block-prints the comment. These comments are not normally printed unless COMMENTFLG is non-NIL. PP* and PPL* force COMMENTFLG to T.

2.1.10.1.10 (TRANSPRINT) [SUBR]

simply reads and prints characters (the same ones) until it reads an altmode, at which point it returns NIL.

2.1.10.2 (FILBAK FILE NEWEXT) [SUBR]

, FILBAK [VALUE]

FILBAK is a SUBR that attempts to rename FILE with the extension of NEWEXT. FILE can be either a FILNAM or a FILESPEC. FILBAK returns T if the renaming was successful and NIL if it fails. As a value FILBAK is the standard extension (initially LBK) for backup files.

2.1.10.3 *NOPOINTDSK [VALUE]

determines whether DSKOUT puts decimal and octal points after numbers. NIL means that they will be added, T means they won't. This feature is meant to solve the compatibility problems that arise from saving a file in one base and reading it in another.

For related information see *NOPOINT, BASE, and IBASE.

2.1.10.4 LISTDEVS [VALUE]

LISTDEVS is a list of output devices that are not also storage. It is used by DSKOUT in deciding whether changed definitions that have been written out are now "saved" and need not show up as "changes" any more. LISTDEVS is initially (LPT: LPT0: LPT1: TTY:).

2.2 FILES

In addition to arbitrary text files (which require assigning and selecting I/O channels etc.), there are three special types of files that LISP supports. The RECORDFILE is simply a transcript of your terminal session. Core images may be saved with the SAVE function, described under SAVE-JOB. Finally, there is an elaborate mechanism (described under SAVE-STATE) for assigning your functions, variables etc. to files which are pretty-printed and can be read by DSKIN or searched by GETDEF.

2.2.1 FILESPEC

FSUBRs that read filenames generally accept an optional device (e.g. DSKC:) followed by an optional PPN (e.g. (C410 HB00)) followed by a file name (e.g. FILE1 or (FILE1 . EXT)). (Whenever multiple files are specified the device and ppn apply to all following files until they are overridden.) However the explicit references to FILESPEC are for SUBRs, in which a file must always be one argument. In this case the argument is of the form (dev: filename), where dev: is optionally absent or a PPN list and file names are the same format as before.

2.2.1.1 (%DEVP X) [SUBR]

%DEVP is a previously undocumented UCI LISP function whose value is non-NIL if X is a device, NIL otherwise. X is a device if it is either atomic and ends in ":" or is a list whose cdr is not atomic.

2.2.1.2 (%GETDEV filespec) [SUBR]

%GETDEV is the standard filespec scanner used by many system functions. It returns (CONS (LIST DEV PPN) TAIL), where TAIL begins with the first element of filespec which is not a device or PPN. DEV is defaulted to DSK:. If no PPN is scanned over, the value returned is (CONS (LIST DEV) TAIL).

2.2.1.3 PPN

Project-Programmer Numbers for Disk I/O In all I/O functions (including INPUT and OUTPUT), the use of a two element list (not a dotted pair) in place of or in addition to a device will cause the function to use the list as the project-programmer number. CMU PPNs are specified in LISP as a (<project> <prog name>) list e.g. (A310 DN10) for A310DN10. e.g. (DSKIN (A310 DN10) (APE . LSP)) DEC PPNs are specified as a list of two numbers. e.g. (5551 601)

The function MYPPN will show your PPN in DEC format.

2.2.1.4 (MYPPN) [SUBR]

MYPPN returns the user's project programmer number in a form suitable for use by the directory and I/O functions.

2.2.2 SAVE-JOB

2.2.2.1 (SAVE "FILE-SPEC" "EXCISE") [FSUBR]

SAVE saves the present core image. If no device is specified, DSK: will be used. If EXCISE is absent or non-NIL, (EXCISE) will be executed. This is generally desirable since it releases I/O buffers and expanded core resulting in a smaller save file. Note that open channels cannot be preserved in a save file in any case, so loss of I/O buffers created by EXCISE is irrelevant. (All channels (other than to the teletype) should be closed prior to doing SAVE.) The SAVE function has been modified to work properly under the 6.02 monitor. It works exactly as before, but for the time being it is unfortunately necessary for it to exit to the monitor. The user must type SAVE or NSAVE (new type of save), then START to continue. The save function allows the high segment not to be saved.

The saved core image will continue execution when RUN or STARTed exactly where the saved program left off, i.e., the first thing the saved core image will do is return the file name for the successfully saved file. SAVE thus preserves the entire execution context, and can be invoked from inside the editor, the break package, or a user function, and control will be returned to that location in both the running job and the saved core image.

Since the allocation procedure is bypassed when RUNning or STARTing a core image saved by the SAVE function (so that execution can continue where it left off in the saving program), if the core image is run in more core than it was saved in, this excess core will be unavailable to LISP (except as I/O buffers or expanded core) until the core image is exited and STARTed again to enter the allocation procedure (or the CORE function is used to make the extra core available). (START only continues the program from where the saving program left off the first time; later STARTs will enter the allocation procedure as usual.) SAVE uses the name to rename to core image, and the PPN and device (as well as name) are used to do a SETSYS. Thus if you specify another PPN the hiseg will still go onto your area, but your .LOW file will tell the monitor to look on the specified PPN for the hiseg. In order to save a core image and have the saved core image allocate whatever core is available when it is RUN or STARTed and go to the LISPX top level, (PROGN (SAVE filename) (CORE (CORE NIL))) may be executed.

2.2.2.2 (SETSYS file-spec) [FSUBR]

SETSYS enables the user to create his own sharable system. In order to create the system, the user must Control-C out and do an SSA <file>, then run the system. After this procedure, the user has write priveleges and may load code into the sharable high segment. (Note that the user need not use this to save a low segment only). This procedure is not necessary for generating the system. (SETSYS OLD: LISP) will cause a .SAV file made with the old LISP to get its high segment from OLD instead of SYS. No high segment need be saved by the user. (The first time you can R OLD:LISP or ASSIGN OLD SYS before R LISP.) The SAVE function does a SETSYS for you.

2.2.2.3 HISEG [VALUE]

causes the SAVE function to save a high segment if it is not NIL. Normally HISEG is NIL, and SAVE only saves a low segment.

2.2.2.4 VERSION [SPECIAL VALUE]

VERSION is numeric and specifies a .SAV file version number to be made part of the name of the next .SAV file written by SAVE.

2.2.3 (RECORDFILE "FILE") [FSUBR]

and RECORDFILE [VALUE] RECORDFILE controls the writing of a file which records all the input from and output to the teletype (with a few exceptions, cited below). If a record file is currently open when RECORDFILE is executed, it will be closed. If a file name is specified, a new record file with that name will be opened. A message ("RECORD FILE file OPENED" or "RECORD FILE file CLOSED") will be printed when a file is opened or closed. An improper specification of a file name to RECORDFILE will result in no record file being opened and the absence of the OPENED message. Use (RECORDFILE) to stop recording. The value of the atom RECORDFILE is the name of the recording file.

Everything read or written by the standard LISP input/output functions (READ, PRINT, etc.) will be copied to the record file. Certain LISP messages are stored in SIXBIT and printed by UUOs and are not copied by the record file facility. These are the error messages ("UNDEFINED FUNCTION", "UNBOUND VARIABLE", etc.) and garbage collection messages. Note that, for example, all user interaction with the top level, editor, or break package is copied to the record file.

Some things can be done in or to LISP to cause the record file to be lost. If the user exits the core image and executes the monitor START command, a RESET UUO will be performed by

LISP, releasing the channel on which the record file was being written without closing the file. The CORE function, when used to acquire more core or to reallocate excised storage, has nearly the same effect as exiting the core image and doing a START. This condition is checked for, and if the allocation procedure is to be entered (whenever the argument to CORE is greater than or equal to the current size of the low segment), the currently open record file (if any) is closed. Executing EXCISE, running out of push down list, and possibly some other severe LISP errors of this type may also cause loss of the record file. If LISP expects to be writing a record file, and finds it is no longer open because of one of these events, it prints the message "RECORD FILE file NO LONGER OPEN" and ceases outputting to the record file. The no longer open record file may sometimes be recovered by executing the monitor CLOSE command.

2.2.4 UFDS

2.2.4.1 (UFDINP CHANNEL PPN) [SUBR]

UFDINP is analogous to the function INPUT in that it opens a file on a specified channel. The channel must be selected via INC in order to be read. The file is opened in binary image mode and should not be read by the normal LISP read functions. UFDINP opens the directory of PPN on CHANNEL. It returns the value of CHANNEL as its result. PPN is either of the form (PROJ PROG) or NIL. If PPN is NIL the user's directory is assumed.

2.2.4.2 (RDFILE) [SUBR]

RDFILE returns the next file in the directory that is open on the current input channel. It return a file which is either an atom or an atomic dotted pair. It does an (ERR EOF) when it reaches the end of file.

For related information see DIR.

2.2.4.3 (LOOKUP DEV FILNAM) [SUBR]

LOOKUP is a SUBR that determines whether the file DEV FILNAM exists or not. LOOKUP returns NIL if it can't find the file and (LIST DEV FILNAM) if the file does exist. If DEV is NIL, DSK: is assumed and (LIST FILNAM) is returned.

2.2.4.4 (LOOKUPFILE file) [SUBR]

LOOKUPFILE is a SUBR that takes as its argument a FILESPEC and returns a non-NIL value if there is a file of that description that can be looked up. (In fact LOOKUPFILE performs a LOOKUP monitor call.) In many cases LOOKUPFILE is more convenient than the LOOKUP

function. It is also more general.

2.2.4.5 (FILELENGTH) [SUBR]

returns a number whose right half contains the number of words in the last file for which a LOOKUP or LOOKUPFILE was done.

2.2.5 (TY "file1" "file2" ... "filen") [FSUBR]

TY imitates the monitor type command by reading (and typing) the specified files in a TYI - TYO loop. The loop may be terminated by typing any character while it is in progress (you don't have to type ^C . ^O is ineffective. Try a space.).

2.2.6 (DELETE "FILNAM1" "FILNAM2" ...) [FSUBR]

DELETE is an FSUBR that deletes the files in the list. The DEV's are optional, and a DEV is effective over the following FILNAM's until a new DEV is encountered. DELETE always returns NIL. The user's disk area is assumed if no DEV has been specified.

2.2.7 (DIRF {ppn} {filespec}) [FSUBR]

prints a list of files in the specified directory (defaults to (MYPPN)) which "match" filespec. Filespec is a file name except that either half may be replaced with an asterisk (*) to match anything. The default filespec is (*.*).

```
Example: (DIRF (C410 HB00) (* . LSP))
```

2.2.8 (DIR PPN) [SUBR]

DIR returns a list of files from the directory of PPN. If PPN is NIL, the user's directory is assumed.

For related information see PPN.

2.2.9 (RENAME "FILNAM1" "FILNAM2") [FSUBR]

RENAME is an FSUBR that renames FILNAM1 to FILNAM2. RENAME returns T if the renaming is successful and NIL if it fails.

2.2.10 (*RENAME FILESPEC1 FILESPEC2) [SUBR]

*RENAME is a SUBR that renames FILESPEC1 to FILESPEC2. It returns T if the rename is

successful and NIL if it fails. If a device is specified in FILESPEC1 and no device is specified in FILESPEC2 the device specified in FILESPEC1 is carried over to FILESPEC2

2.3 PRETTY-PRINTING

2.3.1 (PP <a1> {<a2>} . . .) [FSUBR]

For each <a> in the argument list: If <a> is atomic, each property of <a> which appears on the list PRETTYPROPS is printed in readable format. (If no such properties appear a message to that effect is printed - this message may be suppressed by setting NOPRETTYPROPS to NIL). Each non-atomic <a> is simply printed via SPRINT unless its CAR is defined as a "prettyprint command", in which case the expression is evaluated. <a> may also be a list consisting of a LAP expression followed by a sequence of LAP code; such a list will be printed in standard LAP format.

2.3.2 (GRINDEF "F1" "F2" "F3" ... "FN") [FSUBR]

GRINDEF is the same as PP.

2.3.3 (PP* I1 I2 ...) [FSUBR]

is like PP but it forces COMMENTFLG to T so all comments are shown.

2.3.4 (SPRINT EXPR IND) [SUBR]

SPRINT is the function which does the "pretty printing" of GRINDEF. EXPR is printed in a human readable form, with the levels of list structure shown by indentation along the line. This is useful for printing large complicated structures or function definitions. The initial indentation of the top level list is IND-1 spaces. In normal use, IND should be given as 1.

2.3.5 (PPL <var1> {<var2>} . . .) [FSUBR]

Each <var> should be an atomic symbol which is bound to a prettyprint list to be passed on to PP. This prettyprint list may contain atomic symbols whose properties are to be printed, prettyprint command expressions, and other expressions which are to be SPRINTed. Each <var> which is not already a member of its prettyprint list will be printed so that if dumping to a file its value will be restored when the file is subsequently loaded. The prettyprint list will disappear when the file is compiled, however (i.e., it will not appear in the LAP file).

2.3.6 (GRINL "F1" "F2" ... "FN") [FSUBR]

GRINL is the same as PPL.

2.3.7 (PPL* l1 l2 ...) [FSUBR]

is like PPL but it forces COMMENTFLG to T, causing all comments to be shown.

2.3.8 PRETTYPROPS [VALUE]

In its simplest form, PRETTYPROPS (or GRINPROPS) is a list of atomic symbols which gives the properties which PP is to print. Each atomic argument to PP which has a property on PRETTYPROPS will be printed as a DEFPROP expression. Occasionally, however, it is desirable to print certain properties in something other than DEFPROP format. This may be accomplished by putting a consed pair of the form (<prop> . <fn>) onto PRETTYPROPS; when an atom with a <prop> property is encountered, PP simply prints a carriage return and calls <fn>. <fn> will be passed three arguments: the atom currently being PPed, the value stored under the property <prop>, and the atom <prop> itself. The function can then print anything it wishes before returning to PP, at which time another carriage return will be printed. For example, the functions PP-VALUE and PP-RMACS are provided by the prettyprint package to print VALUE and READMACRO properties in special form.

2.3.8.1 (PP-VALUE atom value (Quote VALUE)) [SUBR]

is the default pretty-printing function for values

2.3.8.2 (PP-FUNCTION atom function-defn fn-prop) [SUBR]

is the function used to pretty-print EXPR, FEXPR and MACRO properties by PP.

2.3.8.3 (PP-RMACS atom readmacro-defn (Quote READMACRO)) [SUBR]

is the default pretty-printing function used for readmacros.

2.3.8.4 (PP-DCCOMMENT ID VAL PROP) [SUBR]

PP-DCCOMMENT is the standard function for printing DC-style comments. Comments are frequently stored only in a file, of course, so PP-DCCOMMENT is more complex than most prettyprint functions. When PP-DCCOMMENT is called from DSKOUT, the variables it uses for communications are appropriately set up by DSKOUT. The system is initialized so that when it is called directly from PP, no printing will be done.

PP-DCCOMMENT only attempts to print the DC-style comments (referred to henceforth as comments) if the variable PRINT-COMMENT is true. PRINT-COMMENT is initially NIL. If the variable COMMENT-CDF is bound and not NIL, the comments will be searched for on the file and channel specified. The format of COMMENT-CDF is exactly the format of an argument list to INPUT, and it must include a channel. An INPUT must already have been performed on the channel, but PP-DCCOMMENT will select the channel as needed. COMMENT-CDF is initially unbound. If the variable FILUPDATFLG is true, comments that are stored in primary memory will be deleted so the space can be reclaimed. The deletion is undoable -- if DSKOUT should abort for some reason, the operation can ordinarily be undone and the comments will be restored. FILUPDATFLG is initially NIL, and should stay that way for most user applications.

If COMMENT-CDF (CDF = ChanDevFile) is true and a comment is searched for without success, the user will be asked to help in finding the comment unless COMMENT-CDF is unbound. A file spec will be requested from the user and searched for the comment. If the first item on the line containing the file spec is the atom "DEFAULT:" and COMMENT-CDF is bound (though possibly NIL), COMMENT-CDF will be set up so that subsequent calls to PP-DCCOMMENT will search the new file automatically rather than the old one. The idea here is that if PP is called directly, the variable will ordinarily be unbound, and the user will not want to set up a GLOBAL default file to search for comments.

The variable #7LINECTR is used by the system routines that search the COMMENT-CDF file for comments. It keeps track of the "place" that LISP is in searching the file. When a COMMENT-CDF is set up by the user, #7LINECTR should be initialized to 1.

2.3.9 PRINTMACRO

SPRINT normally operates by formatting the expression being printed using indentation to produce "pretty" output. It is occasionally desirable to have certain subexpressions printed in some special format for increased readability. Such a capability is provided via the use of printmacros. Any function may be flagged as a printmacro by placing the macro definition on the property list of the atom under the indicator PRINTMACRO. Whenever an atom with such a property appears as the first element in a list being prettyprinted, SPRINT takes special action, such action depending on the value of the PRINTMACRO property:

(1) If the value is a string the string is simply PRINCed and the CADR of the original expression (if present) is SPRINTed. This serves as an inverse for READMACROs of the '<e> -> (QUOTE <e>) type. (If the expression has a non-NIL cddr then the printmacro will be ignored.) (2) If the value is the special atomic symbol BRACKETS then the expression is printed by SPRINT in the normal way, except that each top-level non-atomic argument will be

printed with brackets [...] instead of the usual parentheses (...). This gives the user one more method of producing more readable output. COND, SELECTQ, AND, OR, and CATCH are initialized as printmacros of this type. To disable the use of brackets for these functions simply REMPROP the PRINTMACRO property from their property lists. (3) If the value of the PRINTMACRO property is neither a string nor the atomic symbol BRACKETS it is assumed to be a true printmacro function (or, more typically, the name of a function). This function will be passed the expression being printed as its only argument, and may print it in any format it wishes.

The QUOTE printmacro (which is already in the system) could be defined either by: (DEFPROP QUOTE "'" PRINTMACRO) or

```
(DEFPROP QUOTE
(LAMBDA (E) (PRINC '"'") (SPRINT (CADR E) (CURPOS)))
  PRINTMACRO)
```

2.3.9.1 (PP-COMMENT exp) [SUBR]

is the printmacro that block-prints *** comments.

2.3.9.2 (PP-FORMAT <e> <n> <flag>) [SUBR]

Prints the expression <e> with the first <n>+1 elements (the function name and <n> arguments) printed on one line. <flag> specifies how the remaining arguments are to be printed: if <flag>=NIL (standard format), all remaining arguments are printed under the first argument; if <flag>=MISER, the remaining arguments are placed under the function name; if <flag>=LABELS, all non-atomic arguments are printed under the first argument, with atoms placed to the left as labels.

2.3.9.3 (PP-LABELS exp) [SUBR]

is used by the pretty-printer to print PROG expressions. It is equivalent to (PP-FORMAT exp 1 'LABELS).

2.3.9.4 (PP-MISER exp) [SUBR]

is equivalent to (PP-FORMAT exp 1 'MISER). It is the default printmacro for LAMBDA and DEFPROP.

2.3.10 PRETTY-PRINT-COMMANDS

Prettyprint commands may be used as arguments to PP or in PPL prettyprint lists to perform a variety of special formatting tasks. A prettyprint command is simply an expression

whose CAR is a function name with a non-NIL PPCOM property. Such expressions are evaluated when encountered by PP, thus providing a mechanism for "grabbing control" during the prettyprint process. The user may define his own prettyprint commands, or may use the following functions supplied by the system. Note that in addition to appearing as prettyprint commands in PPL lists, these expressions may be used in other contexts as well.

2.3.10.1 PPCOM [PROPERTY]

PPCOM is the property that identifies an atom as a pretty-print command. If the value of the property is not T then it should be the name of a function which when applied to a call on the pretty-print command will give the words whose definitions are to be printed by the call on the prettyprint command. For example, the PPCOM property of P: is CDDR. (P: (value expr) x y z) would print definitions of the words x, y and z. This is used to help keep track of what functions are defined in which files so that the CHANGES computation will work right.

2.3.10.2 (P: <props> <x1> {<x2>} . . .) [FSUBR]

PRETTYPROPS is set to <props>, the <x>s are passed on to PP, and PRETTYPROPS is restored.

2.3.10.3 (*PG*) [SUBR]

prints a page-eject (^L). It is useful as a PRETTY-PRINT-COMMAND.

2.3.10.4 (MBD: <fn> <x1> {<x2>} . . .) [FSUBR]

Passes the <x>s on to PP in such a way that they will be prettyprinted inside of an expression starting with <fn>. For example, to prettyprint F1 and F2 inside of a PROGN expression (perhaps so they will not be compiled) one could do:

```
(MBD: PROGN F1 F2)
```

2.3.10.5 (FORMS: <x1> {<x2>} . . .) [FSUBR]

Each <x> is passed directly to SPRINT - may be used to print atoms and prettyprint command expressions which would normally be handled specially by PP.

2.3.10.6 (E: <e1> {<e2>} . . .) [FSUBR]

The <e>s are simply evaluated. For example, the inclusion of the following in a prettyprint list could be used to change the base in the middle of a print:

```
(E: (SETQ BASE 10.))
```

2.3.11 PRETTYFLG [VALUE]

determines whether SPRINT prints the nice readable things we all know and love (when the value is T which is the default) or a fast, compact (and not very readable) symbolic dump (when the value is NIL).

2.3.12 PPMAXLEN [VALUE]

is a general limit on the number of characters (after initial spacing) that the prettyprinter will put on a line.

2.4 INPUT-FNS

2.4.1 (READ) [SUBR]

READ causes the next S-expression to be read from the selected input device, and returns the internal representation of the S-expression. READ uses INTERN to guarantee that references to the same identifier are EQ. READ has been altered so that (except when processing a read macro), reading from the teletype flushes all input on the same line after the thing is read. Thus, inputs to successive reads cannot be placed on the same line.

For related information see INTERNSTR and LOWER-CASE.

2.4.2 (RDNAM) [SUBR]

RDNAM functions in the same manner as READ except that it does not INTERN the atoms that it reads. Thus an atom read by RDNAM and an atom read by READ are **NOT** EQ.

For related information see MAKNAM.

2.4.3 (READCH) [SUBR]

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

For related information see LOWER-CASE.

2.4.4 (TYI) [SUBR]

TYI causes the next character to be read from the selected input device and returns the

ASCII code for that character.

2.4.5 (LINEREAD) [SUBR]

LINEREAD reads a line, returning it as a list. If some expression takes more than one line or a line terminates in a comma or tab, then LINEREAD continues reading until an expression ends at the end of a line. This is the function used by the EDITOR and BREAK Package supervisors to read in commands, and may be useful for other supervisor-type functions. Note that a blank no longer forces LINEREAD to continue reading. This allows LINEREAD to read things created by functions such as PRINT.

For related information see LOWER-CASE.

2.4.6 (LINEREADP) [SUBR]

LINEREADP is just like LINEREAD, but if the input buffer contains a CRLF before any interesting characters are read it will return NIL instead of waiting for a list to be read. This is convenient for defaulting input. Warning: Not all input commands read to the end of the line. For example if you type "ABC" followed by a crlf and then do 3 TYIs there will still be a crlf in the input stream. This would cause LINEREADP to return NIL even before the next line was typed. In general LINEREADP should only be used after LINEREAD or another LINEREADP.

2.4.7 (PEEKC) [SUBR]

PEEKC returns the ASCII code for the next character in the input buffer without actually reading it.

2.4.8 (UNTYI n) [SUBR]

UNTYI unreads the character whose ascii code is n (puts it in the front of the input stream) and returns n. This only works for one character (the size of the buffer). UNTYI is really only guaranteed not to lose characters when it is called immediately after a TYI.

2.4.9 (TYIO n) [SUBR]

copies from the input channel to the output channel until the character whose ascii code is n appears as input.

2.4.10 (YESNO X) [SUBR]

YESNO returns T if X is T, Y, or YES, returns NIL if X is NIL, N, or NO, and returns X otherwise. It is useful for interpreting yes/no answers typed by the user.

2.4.11 (TTYESNO) [SUBR]

reads from the tty. If the read generates an error the value is NIL, otherwise the input is passed to YESNO.

2.5 OUTPUT-FNS

2.5.1 (PRINT S) [SUBR]

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

For related information see LOWER-CASE.

2.5.2 (PRIN1 S) [SUBR]

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

For related information see LOWER-CASE.

2.5.3 (PRINC S) [SUBR]

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

2.5.4 (TYO N) [SUBR]

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

2.5.5 (MSG <i1> {<i2>} . . .) [FSUBR]

MSG provides a general message-printing facility for LISP. Each <i> is an instruction which

provides a specific formatting capability:

```
"<string>"          Print <string> using PRINAC
+<number>           Space <number> spaces
(T <n>)             Tab to position <n>
T                   Move to new line
-<number>           Print <number> blank lines
(E <expr>)          Evaluate <expr>
other               Eval and print using PRINA
```

Note that MSG prints the desired message on the currently selected output device. MSG is compiled in-line.

```
(MSG T "X = " 5 X T)
```

is equivalent to:

```
(PROGN (TERPRI)
       (PRINAC '"X = ")
       (SPACES 5)
       (PRINA X)
       (TERPRI))
```

2.5.6 (TTYMSG <i1> {<i2>} . . .) [FSUBR]

TTYMSG is identical to MSG, except printed output is directed to the terminal instead of the currently selected output device. To insure that the message will appear on the terminal even if ^O has been struck, a TALK is performed before printing. TTYMSG is compiled in-line.

2.5.7 (PRINA x {pos}) [LSUBR]

Like PRIN1, except if an atom won't fit on the line, a tab to position pos on the next line is performed before printing resumes. Pos is optional, with a value of 1 assumed if omitted.

2.5.8 (PRINAC x {pos}) [LSUBR]

is the same as PRINA but PRINC is used to print atoms instead of PRIN1.

2.5.9 (SPACES n {ident}) [LSUBR]

Spaces over <n> spaces, using tab characters when possible. If <n> spaces won't fit on the current line, SPACES performs a TERPRI instead. If ident is specified then SPACES indents this amount after the TERPRI.

2.5.10 (LINES n) [SUBR]

Prints <n> blank lines. Note that the next print position will always be at the start of a

line, so (LINES 0) may be used as a "conditional TERPRI" which outputs a carriage return if not already at the start of a line.

2.5.11 (PRINL <I>) [LSUBR]

Prints the list <I> without the outermost parentheses, i.e., prints the elements of <I> separated by spaces. Each element is printed using PRINA, with a <pos> of 1.

2.5.12 (PRINLC <I>) [LSUBR]

Identical to PRINL, except uses PRINAC instead of PRINA to print the list elements.

2.5.13 (TERPRI X) [SUBR]

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

Example: `(PRINC(TERPRI X))`

is the same as

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

2.5.14 (TAB N) [SUBR]

TAB tabs to position N on the output line doing a TERPRI if the current position is already past N. Note that TAB outputs spaces only when necessary and outputs tab characters otherwise.

2.5.15 (PRINTLEV EXPRESSION DEPTH) [SUBR]

PRINTLEV is a printing routine similar to PRINT. PRINTLEV, however, only prints to a depth of DEPTH. In addition, PRINTLEV recognizes lists which are circular down the CDR and closes these with '...]' instead of ')'. The combination of these two features allows PRINTLEV to print any circular list without an infinite loop. The value of PRINTLEV is the value of EXPRESSION. This means that PRINTLEV should not be used at the top level if EXPRESSION is a circular list structure, since the LISP executive would then attempt to print the circular structure which is returned as the value. (Instead say (NULL (PRINTLEV ...)) .)

2.5.16 (PRINLEV EXPRESSION DEPTH) [SUBR]

PRINLEV is the same as PRINTLEV but no preceeding (TERPRI) is done and no trailing blank

is added.

2.5.17 (PLEV exp) [SUBR]

does (PRINLEV exp %LOOKDPTH). PLEV is the default value of %PRINFN (used by the break package).

2.5.18 %LOOKDPTH [VALUE]

This is the variable that determine the depth to which PLEV goes. It is rebound in a few places by the break package.

2.6 I-O-CHANNELS

2.6.1 (INPUT "CHANNEL" . "FILENAME-LIST") [FSUBR]

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

2.6.2 (INC CHANNEL ACTION) [SUBR]

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

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

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

A crude approximation to the TY function (for typing a file) is (PROG NIL (INC (INPUT XYZ

FILE) NIL) L: (TYO (TYI)) (GO L:)).

2.6.3 (OUTPUT "CHANNEL" . "FILENAME-LIST") [FSUBR]

OUTPUT initializes for output on the specified channel the single file specified by the filename-list. OUTPUT does not evaluate its arguments, and returns the channel name if specified, T otherwise. Note that OUTPUT does NOT SELECT that channel for output. This is done by OUTC.

2.6.4 (OUTC CHANNEL ACTION) [SUBR]

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

Examples: At the top level:

```
(OUTC (OUTPUT LPT:) T)  ;; now all output goes to LPT:NIL
(OUTC NIL T)  ;; now output comes to TTY: and LPT:NIL is closed
(OUTPUT FOO DSK:  BAZ)  ;; output still on TTY: but FOO is declared
(OUTC (QUOTE FOO) NIL)  ;; now the channel FOO is selected.  Thus
       ;;output goes to DSK:BAZ.  The previous file (TTY:) is not
  ;;closed.  (Actually TTY: is always opened when nothing else is.)
```

2.6.5 (INCH) [SUBR]

returns the name of the currently selected input channel.

2.6.6 (OUTCH) [SUBR]

returns the name of the currently selected output channel.

2.6.7 (TTYIN FORM1 ... FORMn) [MACRO]

TTYIN is a MACRO that produces code that evaluates FORM1 through FORMn with the currently selected input device forced to be the teletype. The value of FORMn is returned as the value of TTYIN. For this to be useful, some of FORM1 ... FORMn should do some reading.

Note that no error protection is provided. If an error occurs, TTYIN will lose control and fail to reset the input channel to its previous value. For this reason, TTYIN should be used only to surround the expressions which actually do the input, and ERRSET should generally be

used to protect against the user typing control-G so that control is retained by TTYIN. Generally, the function should be (and is most useful when) used as in, for example, (TTYIN (ERRSET (READ) ERRORX)).

2.6.8 (TTYOUT FORM1 ... FORMn) [MACRO]

TTYOUT is a MACRO that produces code that evaluates FORM1 through FORMn with the currently selected output device forced to be the teletype. The value of FORMn is returned as the value of TTYOUT. For this to be useful, some of FORM1 through FORMn should do some writing.

Note that no error protection is provided. If an error occurs, TTYOUT will lose control and fail to reset the output channel to its previous value. For this reason, TTYOUT should be used only to surround the expressions which actually do the output. Generally, the function should be (and is most useful when) used as in, for example, (TTYOUT (PRINT X)).

2.6.9 (GETCHN) [SUBR]

allocates an I/O channel. It returns the channel it got (a number).

2.6.10 (GIVCHN chan) [SUBR]

deallocates an I/O channel. The argument should be the number of an allocated channel.

2.6.11 (EXCISE) [SUBR]

EXCISE contracts core to its length after ALLOCATION or the last START. This removes I/O buffers, and all RELOC programs. It also closes all files and releases all devices. The usual reasons for expanding core are 1) using I/O channels, 2) using the loader and 3) getting more Binary Program Space.

For related information see BPS, CORE, LOAD, and SAVE.

2.7 I-O-MODE

2.7.1 BASE [VALUE]

BASE specifies the output radix for integers (initially 10 (decimal)). Warning: the default used to be 8 (octal). If BASE is negative then negative numbers will be printed as unsigned 36 bit numbers (where the radix is -BASE), i.e. 777777777777Q instead of -1Q.

2.7.2 IBASE

IBASE determines the input radix for integers not followed by a decimal point or octal point. The default value is 10 (decimal). Warning - IBASE used to default to 8 (octal).

2.7.3 *NOPOINT [VALUE]

This variable determines whether decimal points will be printed after decimal numbers and octal points after octal numbers - T means they won't.

For related information see *NOPOINTDSK.

2.7.4 OCTAL-POINT

The octal point "Q" is printed after octal numbers (if *NOPOINT is NIL) to distinguish them from numbers in other bases, just as the decimal point is used in base 10.

For related information see *NOPOINTDSK.

"Why 'Q'?", you ask. "Why not?", we reply.

2.7.5 INTERNSTR [VALUE]

determines whether strings will be INTERNed by READ. The default value is NIL meaning they are not. This flag is turned on by LAP.

2.7.6 (PGLINE) [SUBR]

When reading an input file, it is sometimes desirable to know the page and line being read from. PGLINE returns the dotted pair (page number . line number) for the selected input file. The page number is applicable only to STOPGAP files. If the file has no line numbers, PGLINE will always return (1 . 0).

2.8 CHARACTERS

2.8.1 COMMENT-CHAR

Note: For commenting code you should see the explanations of *** and COMMENT. The features described here are not good for that purpose. Comments are useful for allowing descriptive text in files which will be COMPLETELY IGNORED WHEN READ. (Since lisp doesn't

see this text it can't save it in the revised file.) Comments also make it possible to extend atoms (identifiers, strings and numbers) across line boundaries without any of the characters in the comment becoming part of the atom. When the comment character is seen by LISP the rest of the input line is ignored. The comment character is ?4 in SOS or ^Z (control Z).

2.8.2 LETTER-QUOTE

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

2.8.3 (CHQUOTE n) [SUBR]

(CHQUOTE n) changes to quoting (slashifying) character to character number n. (CHQUOTE NIL) returns the current quoting character.

2.8.4 (MODCHR CH N) [SUBR]

The value of MODCHR is the old read-table entry for CH. If N is non-NIL it must be a number which represents a valid table entry. The entry for CH is changed to N. If N is NIL, no change is made, e.g. to make "." a letter (so it will behave like the letter "A") execute (MODCHR 46 (MODCHR 65 NIL)).

2.8.5 (SETCHR CH N) [SUBR]

SETCHR is similar to MODCHR except that it only modifies the portion of the entry associated with read macros.

For related information see READMACRO.

2.8.6 *DIGITS [VALUE]

is simply a list of the digits 0-9 (as characters, not numbers).

2.8.7 *LETTERS [VALUE]

is just a list of the letters A-Z.

2.8.8 LOWER-CASE

READ and LINEREAD now map lower case letters into upper case letters inside identifiers. Letters in strings are not mapped. Lower-case letters in identifiers other than strings are slashified on output by PRINT or PRIN1. READCH maps lower case to upper case, but the mapping cannot be turned off by changing the read tables. To disable this feature, redefine READCH to be (INTERN (ASCII (TYI))). TYI is unaffected by the mapping, distinguishing upper and lower case as before.

Programs using no lower-case identifiers and not using TYI will notice no difference except that input from a file or the keyboard may now be in lower case. If there are lower case identifiers (or strings), the change will affect the actions of EXPLODE, FLATSIZE, MAKNAM, and READLIST. EXPLODEC and FLATSIZEC are unaffected because, like PRINC, they ignore any possible slashification. NTHCHAR is also unaffected.

To read in existing files containing lower case identifiers, the lower case letters may be slashified with TECO before being input to LISP. Alternatively, LISP can be modified by a method like the one for removing the lower case mapping feature. Remove the lower case mapping feature, but record the results of the MODCHRs performed on the lower case characters. Read in the file. Do MODCHRs restoring the original table values, then write the file out again. This will cause the lower case identifiers to be slashified.

To remove the lower case mapping, do the following:

```
(PROG (IT)
      (SETQ IT  (CHRVAL 'A))
 LOOP (COND  ((NOT (> IT (CHRVAL'Z)))
             (MODCHR (+ IT 32)(MODCHR IT NIL)) (SETQ IT (+I IT))(GO LOOP))]
```

2.9 TTY-CONTROL

2.9.1 (CLRBFI) [SUBR]

CLRBFI clears the Teletype input buffer.

2.9.2 (DDTIN X) [SUBR]

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

S-expression being read and start reading again.

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

2.9.3 (INITPROMPT N) [SUBR]

Whenever LISP is forced back to the top level (e.g. by an error or Control-G), the prompt character is reset. INITPROMPT is similar to PROMPT except that it sets the top level prompt character. (INITPROMPT NIL) returns the ASCII value of the top level prompt character without changing it.

2.9.4 (PROMPT N) [SUBR]

The LISP READ routines type out a "prompt character" for the user when they expect to read from the teletype. For example the top level prints ">" and the editor prints "*". PROMPT resets this prompt character. N is the ASCII representation of the new prompt character. The ASCII representation of the old prompt character is returned as the value of PROMPT. (PROMPT NIL) returns the current prompt character without changing it.

2.9.5 (TTYECHO) [SUBR]

TTYECHO complements the Teletype echo switch. The value of TTYECHO is T if the echo is being turned on, and NIL if it is being turned off.

2.9.6 (READP) [SUBR]

READP returns T if a character can be input and NIL otherwise. READP does not input a character.

2.9.7 (ERRCH N) [SUBR]

ERRCH changes the bell character that causes an (ERR (QUOTE ERRORX)). N is the ASCII representation of the character. ERRCH returns the ASCII representation of the old character. Note that if the new character is not a break character to the monitor, it will not be processed until it is read in the normal course of reading.

2.9.8 (TALK) [SUBR]

Undoes the effect of a previous ^O. May be used to insure that a message will appear on the terminal (see TTYMSG). Note that a TALK is automatically performed whenever an error condition is encountered, including an end of file on any input device.

2.10 LINE-CONTROL

2.10.1 (CURPOS) [SUBR]

CURPOS returns the current position on the output line of the currently selected channel; this is computed by (ADD1 (*DIF (LINELENGTH NIL) (CHRCT))).

2.10.2 (CHRCT) [SUBR]

CHRCT returns the number of character positions remaining on the output line of the selected output channel. When characters are output, if CHRCT is made negative, an ASCII 176 followed by a carriage-return and a line-feed are output. These characters are completely ignored on input.

2.10.3 (SETCURPOS N) [SUBR]

Immediately following (SETCURPOS N), (CURPOS) will return N. This allows the user to send characters that do not print or that do cursor positioning, and inform PRINT and friends of where one really is on the line.

2.10.4 (LINELENGTH N) [SUBR]

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

2.10.5 LPTLENGTH [VALUE]

(or DSKLENGTH) is supposed to tell the system how many characters it can put in a line in a DSKOUT. There are some known pathological cases in which it is worth changing.

2.11 READMACRO

Read Macros allow the user to specify a function to be executed each time a selected character is read during input of his data or programs. This function is generally used to

produce one or more elements of the input list which are built up in some way from later characters of the input string. There are two types of Read Macros; Normal Read Macros whose result is used as an element of the input list in the position where the macro character occurred, and Splice Macros whose result (must be a list which) is spliced sequentially into the input list.

```
Examples: (DRM * (LAMBDA () (NCONS (READ))))
```

If the expression (A B *C D) is read the apparent input is (A B (C) D).

```
(DRM = (LAMBDA () (REVERSE (READ))))
```

If the expression (A B = (C D E) F G) is read the apparent input is (A B (E D C) F G).

```
(DSM : (LAMBDA () (CONS NIL (READ))))
```

If the expression (A B :C) is read the apparent input is (A B NIL . C).

Splice macros place the result of the function evaluation into the input stream minus the outermost set of parentheses.

WARNING: Read macro characters will not be recognized if they occur inside of an atom name unless the character is first defined to be equivalent to a break or separator character (e.g. space or comma) using MODCHR. Read macros are defined by DRM (normal) and DSM (splice).

2.11.1 (DRM "CHARACTER" "FUNCTION") [FSUBR]

CHARACTER is defined as a Normal Read Macro with "FUNCTION" being a function name or a LAMBDA expression of no arguments which will be evaluated each time CHARACTER is detected as a macro during input. FUNCTION is put on the property list of CHARACTER under the property READMACRO. The value of DRM is CHARACTER.

2.11.2 (DSM "CHARACTER" "FUNCTION") [FSUBR]

DSM is exactly like DRM except that CHARACTER is defined as a Splice Macro.

2.11.3 (%DEREAD number lambda-exp type) [SUBR]

defines the character with the given number as a readmacro (if type is 10) or a splicemacro (if type is 11).(Normal value is 21.)

2.11.4 QUOTE-CHAR

The character "'" is a readmacro that will translate into the QUOTE function. So, '(A B C) is the same as (QUOTE (A B C)).

2.11.5 EDRM [EXPR]

This is the edit read macro. I.e. if you type (DRM $ EDRM) then typing $ <fn name> will cause the editor to be called for the given function.

2.11.6 EVSM [EXPR]

This is the eval splice macro. I.e. if you type (DSM _ EVSM) (and (SETQ EVSM 95)) then whenever you type __ <exp> the expression will be evaluated (and the value printed) and spliced into the input. Whenever you type _ <exp> the expression will be evaluated and the value will be printed but with no change to the apparent input.

2.11.7 (PPRM) [EXPR]

is the GRINDEF read macro. I.e. if you do (DRM # PPRM) then # <fn name> will cause fn to be pretty-printed.

2.11.8 (P1RM) [EXPR]

is the PROG1 read macro. I.e. if you type (DRM ; P1RM) then whenever you type ;<exp> the expression will be evaluated but NIL will be returned.

3. ERROR-RECOVERY

3.1 INTERRUPTS

In case you want to interactively stop a computation for any of several purposes, LISP has an interrupt-handling facility. This provides two flavors of entry to the debugger and about four flavors of aborting of the computation, five perhaps depending on how you count.

The interrupt routine is entered by striking a single ^C (Control-C) if awaiting terminal input, or two consecutive ^C's if computing. The interrupt routine types:

```
;;          Interrupt (?=help):
```

and awaits an interrupt character from the user. Typing "?" will produce the following list of choices:

```
;;          CR = Continue (Ignore ^C)
;;          ^D = Return to Top Level
;;          ^X = Exit to Monitor via (EXIT T)
;;          ^H = Break Next Fn Call
;;          ^B = Back Up and Break Last Fn Call
;;          ^G = (ERR ERRORX)
;;          ^E = (ERR NIL)
;;          ^R = Restore System OBLIST
```

1. Control-H: This will cause the computation to continue, but a break will occur the next time a function is called (except for a compiled function called by a compiled function). A message of the form (-- BROKEN) is typed and the user is in BREAK. WARNING: It is possible to get into an infinite loop that does not include calls to functions other than compiled functions called by compiled functions. These will continue to run. (In such cases, try one of the other control characters).

2. Control-B: This will cause the system to back up to the last expression to be evaluated and cause a break (putting the user in BREAK1 with all the power of BREAK1 at the user's command. This does not include calls to compiled functions by other compiled functions.

3. Control-G: This causes an (ERR ERRORX) which returns to the last (ERRSET ERRORX). This enables the user to Control-C out of the Break package or the Editor, reenter and return to the appropriate command level. (i.e. if the user were several levels deep in the Editor for example, Control-G will return him to the correct command level of the Editor).

4. Control-E: This does an (ERR NIL), which return NIL to the last ERRSET. (See section on changes to ERR and ERRSET).

5. Control-R: This restores the normal system OBLIST (as the value of the atom OBLIST). Another of the above control characters must be typed after this character is typed. This

will often recover after a GARBAGED OBLIST message.

6. Carriage return causes LISP to continue doing whatever it was doing when the ^C was typed.

7. ^D causes an immediate return to the top level of LISP .

8. ^X causes an exit to the monitor (see EXIT). A subsequent CONTINUE or START at monitor level will cause LISP to continue where it left off.

Any other character causes a user interrupt (a feature that is not implemented and just causes an error). The system is fully protected against a ^C interrupt occuring at the wrong time; for example, if ^C is typed during a garbage collection the garbage collection is completed before the interrupt is recognized. Note that a REENTER at monitor level is equivalent to a START if LISP was exited normally (via ^C ^X or EXIT). If LISP was somehow exited without going through the normal exit procedure, REENTER will cause the ^C interrupt routine to be entered.

3.2 BREAK-PACKAGE

Whenever LISP types a message of the form (-- BROKEN) or (Error from --) followed by : or n: the user is then "talking to" the function BREAK1, and he is "in a break." BREAK1 allows the user to interrogate the state of the world and affect the course of the computation. It uses the prompt character : to indicate it is ready to accept input(s) for evaluation, in the same way as the top level of LISP uses >. The n before the : is the level number which indicates how many levels of BREAK1 are currently open. The user may type in an expression for evaluation and the value will be printed out, followed by another :. Or the user can type in one of the commands specifically recognized by BREAK1 (a break command).

Since the user can type in arbitrary expressions to be evaluated he has all of the power of LISP at his command. For example he can call the standard top level by typing (TOP-LEVEL), although the simple TL command is shorter. He can define new functions or edit existing ones, set breaks, or trace functions. The user may evaluate an expression, see that the value was incorrect, call the editor, change a function, and evaluate the expression again, all without leaving the break.

It is important to emphasize that once a break occurs, the user is in complete control of the flow of the computation, and the computation will not proceed without specific instruction from him. Only if the user gives one of the commands that exits from the break (GO, OK, RETURN, FROM?=, EX) will the computation continue. If the user wants to abort the

computation, this also can be done (using ^ or ^^).

3.2.1 (BREAK1 BRKEXP BRKWHEN BRKFN BRKCOMS BRKTYPE) [SUBR]

The heart of the debugging package is a function called BREAK1. BREAK and TRACE redefine your functions in terms of BREAK1. When an error occurs control is passed to BREAK1.

Note that BREAK1 is just another LISP function, not a special system feature like the interpreter or the garbage collector. It has arguments and returns a value, the same as any other function. The arguments to BREAK1 are: BRKWHEN is a LISP function which is evaluated to determine if a break will occur. If BRKWHEN returns NIL, BRKEXP is evaluated and returned as the value of the BREAK1. Otherwise a break occurs. BRKFN is the name of the function being broken and is used to print an identifying message. BRKCOMS is a list of command lines (as returned by READLINE) which are executed as if they had been typed in from the teletype. The command lines on BRKCOMS are executed before commands are accepted from the teletype, so that if one of the commands on BRKCOMS causes a return, a break occurs without the need for teletype interaction. BRKTYPE identifies the type of the break. It is used by the error package and in all cases the user can use NIL for this argument.

The value returned by BREAK1 is called 'the value of the break.' The user can specify this value explicitly by using the RETURN command described below. In most cases, however, the value of the break is given implicitly, via a GO or OK command, and is the result of evaluating 'the break expression,' BRKEXP.

BRKEXP is, in general, an expression equivalent to the computation that would have taken place had no break occurred. In other words, one can think of BREAK1 as a fancy EVAL, which permits interaction before and after evaluation. The break expression then corresponds to the argument to EVAL. For BREAK and TRACE, BRKEXP is a form equivalent to that of the function being traced or broken. For errors, BRKEXP is the form which caused the error. For DDT breaks, BRKEXP is the next form to be evaluated.

For related information see BREAK, BREAKIN, TRACE, BREAKO, and BREAKMACROS.

3.2.1.1 LASTPOS [VALUE]

All information pertaining to the evaluation of forms in LISP is kept on the special push down stack. Whenever a form is evaluated, that form is placed on the special push down stack. Whenever a variable is bound, the old binding is saved on the special push down

stack. The context (the bindings of free variables) of a function is determined by its position in the stack. When a break occurs, it is often useful to explore the contexts of other functions on the stack. BREAK1 allows this by means of a context pointer, LASTPOS, which is a pointer into the special push down stack. BREAK1 contains commands to move the context pointer and to evaluate atoms or expressions as of its position in the stack. For the purposes of this document, when moving through the stack, "backward" is considered to be toward the top level or, equivalently, towards the older function calls on the stack.

3.2.1.2 BRKEXP [VALUE]

The argument passed to BREAK1 which is evaluated (and returned as the value of the break) when you type GO or OK. It should be equivalent to the expression which caused the break.

3.2.1.3 BRKWHEN [VALUE]

The argument to BREAK1 which is evaluated to determine whether a break will occur. If the value is NIL then BRKEXP is evaluated and returned, otherwise a break occurs.

3.2.1.4 BRKFN [VALUE]

The argument passed to BREAK1 used to tell the user what function was broken.

3.2.1.5 BRKCOMS [VALUE]

The list of commands that is pased as a parameter to BREAK1 to execute when a break is entered.

3.2.1.6 BRKTYPE [VALUE]

The argument to BREAK1 identifying the type of the break. The user can use NIL for this argument when calling BREAK1.

3.2.1.7 (//BREAK1) [SUBR]

//BREAK1 is the function that BREAK1 calls to do all the work.

3.2.1.8 NAMESCHANGED [PROPERTY]

When you break (or trace) (fn1 in fn2) then fn1 is added to the NAMESCHANGED property of fn2.

3.2.1.9 BRKAPPLY [SUBR]

(same as apply but used by the breakpackage)

3.2.2 BREAK-COMMANDS

3.2.2.1 GO [BREAK COMMAND]

Releases the break and allows the computation to proceed. BREAK1 evaluates BRKEXP, its first argument, prints the value, and returns it as the value of the break. BRKEXP is the expression set up by the function that called BREAK1. For BREAK or TRACE, BRKEXP is equivalent to the body of the definition of the broken function. For the error package, BRKEXP is the expression in which the error occurred. For DDT breaks, it is the next form to be evaluated.

3.2.2.2 OK [BREAK COMMAND]

Same as GO except that the value of BRKEXP is not printed.

3.2.2.3 EVAL [BREAK COMMAND]

Causes BRKEXP to be evaluated. The break is maintained and the value of the evaluation is printed and bound on the variable !VALUE. (The evaluation is done in an errset whose value is bound to FULL!VALUE.) Typing GO or OK will not cause reevaluation of BRKEXP following EVAL but another EVAL will. EVAL is a useful command when the user is not sure whether or not the break will produce the correct value and wishes to be able to do something about it if it is wrong.

3.2.2.4 RETURN form [BREAK COMMAND]

The form is evaluated and its value is returned as the value of the break. For example, one might use the EVAL command and follow this with

```
RETURN (REVERSE !VALUE)
```

3.2.2.5 ^ [BREAK COMMAND]

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

3.2.2.6 ^^ [BREAK COMMAND]

This returns control directly to the top level of LISP.

3.2.2.7 > expr [BREAK COMMAND]

For use either with UNBOUND ATOM error or UNDEFINED FUNCTION error. Replaces the expression containing the error with expr (not the value of expr) e.g.,

```
FOO1
UNDEFINED FUNCTION
(FOO1 BROKEN)
1: > FOO
```

changes FOO1 to FOO and continues the computation. Expr need not be atomic, e.g.,

```
FOO
UNBOUND ATOM
(FOO BROKEN)
1: > (QUOTE FOO)
```

For UNDEFINED FUNCTION breaks, the user can specify a function and its first argument, e.g.,

```
MEMBERX
UNDEFINED FUNCTION
(MEMBERX BROKEN)
1: > MEMBER X
```

Note that in the some cases the form containing the offending atom will not be on the stack (notably, after calls to APPLY) and in these cases the function definition will not be changed. In most cases, however, > will correct the function definition.

> has been altered to interface better to the standard USERTOP function. The problem is that if you type "mumble foo" to top-level, the standard usertop will, if there is no such function as mumble, assume it to be an EXPR. It will accordingly change foo to (QUOTE foo), ending up with (mumble 'foo). Occasionally one mistypes the name of an FSUBR or FEXPR such as "hlp foo". If you get an error and type "> HELP", you do not want the arguments quoted. The new version of > unquotes the arguments that are quoted if brkexp is the form typed to top-level in the last event.

3.2.2.8 FROM?= {form} [BREAK COMMAND]

FROM?= exits from the break by undoing the special push down stack back to LASTPOS. If FORM is NIL or missing, re-evaluation continues with the form on the push down stack at LASTPOS. If FORM is not NIL, the function call on the push down stack at LASTPOS is replaced by FORM and evaluation continues with FORM. FORM is evaluated in the context of LASTPOS. There is no way of recovering the break because the push down stack has been undone. FROM?= allows the user to, among other things, return a particular value as the value of any function call on the stack. To return 1 as the value of the previous call to FOO:

```
: F FOO
: FROM?= 1
```

Since form is evaluated after it is placed on the stack, a value of NIL can be returned by using (QUOTE NIL).

For related information see LASTPOS and SPDL.

3.2.2.9 EX [BREAK COMMAND]

EX exits from the break and re-evaluates the form at LASTPOS. EX is equivalent to FROM?= NIL.

3.2.2.10 USE x FOR y [BREAK COMMAND]

Causes all occurrences of y in the form on the stack at LASTPOS (for Error breaks, unless a F command has been used, this form is the one in which the error occurred.) to be replaced (RPLACA'ed) by x. Note: This is a destructive change to the s-expression involved and will, for example, permanently change the definition of a function and make an edit step unnecessary.

3.2.2.11 F arg1 arg2 ... argN [BREAK COMMAND]

& arg1 arg2 ... argN [BREAK-COMMAND]

Resets the variable LASTPOS, which establishes a context for the commands ?=, USE, EX and FROM?=, and the backtrace commands described below. LASTPOS is the position of a function call on the special push down list. It is initialized to the function just before the call to BREAK1. F takes the rest of the teletype line as its list of arguments. F first resets LASTPOS to the function call just before the call to BREAK1, and then for each atomic argument, F searches backward for a call to that atom. The following atoms are treated specially: F, &, numbers, _

For related information see LASTPOS and SPDL.

When "F" or "&" is used as the first argument LASTPOS is not reset to above BREAK1 but continues searching from the previous position of LASTPOS.

When the arguments are numbers, they are added to lastpos. Thus positive numbers move toward the last error while negative numbers move toward the top-level.

_ causes the search to change direction.

If the special push-down stack looks like

```
BREAK1 (13)
FOO (12)
SETQ (11)
COND (10)
PROG (9)
FIE  (8)
COND (7)
FIE  (6)
COND (5)
FIE  (4)
COND (3)
PROG (2)
FUM  (1)
```

then F FIE COND will set LASTPOS to to (7) F & COND will then set LASTPOS to (5) F FUM _ FIE will stop at (4) F & 2 will then move LASTPOS to (6) F will reset LASTPOS to (12)

If F cannot successfully complete a search, for argN or if argN is a number and F cannot move the number of functions asked, "argN?" is typed. In either case, LASTPOS is restored to its value before the F command was entered. Note: It is possible to move past BRKEXP (i.e. into the break package functions) when searching or moving forwards. When F finishes, it types the name of the function at LASTPOS. F can be used on BRKCOMS. In which case, the remainder of the list is treated as the list of arguments. (i.e. (F FOO FIE FOO)

3.2.2.12 EDIT arg1 arg2 ... argN [BREAK COMMAND]

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

For related information see LASTPOS.

3.2.2.13 FIX arg1 arg2 ... [BREAK COMMAND]

is equivalent to EDIT followed by FROM?=. The edited expression will immediately be evaluated when the editor is exited.

3.2.2.14 ?= arg1 arg2 ... argN [BREAK COMMAND]

This is a multi-purpose command. Its most common use is to interrogate the value(s) of the arguments of the broken function, (ARGS is also useful for this purpose.) e.g. if FOO has three arguments (X Y Z), then typing ?= to a break of FOO, will produce:

```
;;        n:?=
;;        X = value of X
;;        Y = value of Y
;;        Z = value of Z
```

?= takes the rest of the teletype line as its arguments. If the argument list to ?= is NIL, as

in the above case, it prints all of the arguments of the function at LASTPOS. If the user types

```
?= X (CAR Y)
```

he will see the value of X, and the value of (CAR Y). The difference between using ?= and typing X and (CAR Y) directly into BREAK1 is that ?= evaluates its inputs as of LASTPOS. This provides a way of examining variables or forms as of a particular point on the stack.

```
F (FOO FOO)
?= X
```

will allow the user to examine the value of X in an earlier call to FOO. ?= also recognizes numbers as referring to the correspondingly numbered argument. Thus

```
:F FIE
:?= 2
```

will print the name and value of the second argument of FIE (providing FIE is not compiled). ?= can also be used on BRKCOMS, in which case the remainder of the list on BRKCOMS is treated as the list of arguments. For example, if BRKCOMS is ((EVAL) (?= X (CAR Y)) GO)), BRKEXP will be evaluated, the values of X and (CAR Y) printed, and then the function exited with its value being printed.

For related information see LASTPOS.

3.2.2.15 ARGS [BREAK COMMAND]

Prints the names and the current values of the arguments of BRKFN. In most cases, these are the arguments of the broken function.

3.2.2.16 HELP [BREAK-COMMAND]

is a break command that calls the HELP function and uses the remainder of the command line as the argument list for HELP.

3.2.2.17 TL [BREAK-COMMAND]

TL calls (TOP-LEVEL). To return to the break package just use the RETURN top-level command.

3.2.2.18 DO form [BREAK-COMMAND]

resets the I/O channels and prompt to what they were before the error, evaluates the expression given as an argument, returns to the break I/O status and shows the value computed. This is worth while if you want to find out where in the file you were reading. Just "do (read)".

3.2.2.19 BKE [BREAK-COMMAND]

prints a backtrace of function calls that are pending. BKE may be followed by a number which will limit the number of entries that are printed. BKEV (or BKEV number) will also print the values of the variables.

For related information see ERXACTION and BKTRC.

3.2.2.20 BK [BREAK COMMAND]

prints a backtrace of expressions being executed. BK may be followed by a number which will limit the number of entries that are printed. BKV (or BKV number) will also print the values of the variables.

For related information see ERXACTION and BKTRC.

3.2.2.21 BKF [BREAK COMMAND]

prints a backtrace of the names of the functions that are pending. BKF may be followed by a number which will limit the number of entries printed. BKFV (or BKFV number) will also print the values of the variables.

For related information see ERXACTION and BKTRC.

3.3 BREAKING

"Breaking" is what you do to functions to get them to stop in the middle so that you can see what's going on. It is also what happens to you when an error occurs and you get a message that something is broken. To see how to continue, recover from errors etc. see ERROR-RECOVERY.

The function BREAK is usually used to set a break on all calls on some function. BREAK (and TRACE) use a function BREAK0 to do the actual modification of function definitions. When BREAK0 breaks a SUBR or an FSUBR, it prints a message of the form (<NAME> ARGUMENT LIST?). The user should respond with a list of arguments for the function being broken. (FSUBR's take only one argument and BREAK0 checks for this.) The arguments on this list are actually bound during the calls to the broken function and care should be taken to insure that they do not conflict with free variables. These arguments are remembered as the value of the BRKARGS property of the broken function. If the function being broken or traced already has a BRKARGS property, its value is used as the argument list. The case of a null argument list must be treated separately: the value T indicates no arguments. For

LSUBR's, the atom N? is used as the argument. It is possible to GRINDEF and edit functions that are traced or broken. BROKENFNS is a list of the functions currently broken. TRACEDFNS is a list of the functions currently traced.

3.3.1 (BREAK fn1 fn2 ...) [FEXPR]

BREAK is an FEXPR. For each atomic argument, it breaks the function named each time it is called. For each list in the form (fn1 IN fn2), it breaks only those occurrences of FN1 which appear in FN2.

This feature is very useful for breaking a function that is called from many places, but where one is only interested in the call from a specific function, e.g. (RPLACA IN FOO), (PRINT IN FIE), etc. For each list not in this form, it assumes that the CAR is a function to be broken; the CADR is the break condition; (When the function is called, the break condition is evaluated. If it returns a non-NIL value, the break occurs. Otherwise, the computation continues without a break.) and the CDDR is a list of command lines to be performed before an interactive break is made.

```
(BREAK FOO1 (FOO2 (GREATERP N 5) (ARGS)))
```

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

```
(BREAK ((FOO4 IN FOO5) (MINUSP X)))
```

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

For related information see BRKWHEN, BRKCOMS, and BREAK1.

3.3.1.1 BROKENFNS [VALUE]

a list of the functions currently broken.

For related information see BREAKING.

3.3.1.2 (UNBREAK x1 x2 ...) [FSUBR]

UNBREAK is an FSUBR. It takes a list of functions modified by BREAK or BREAKIN and restores them to their original state. It's value is the list of functions that were "unbroken". (UNBREAK T) will unbreak the function most recently broken. (UNBREAK) will unbreak all of the functions currently broken (i.e. all those on BROKENFNS). If one of the functions is not broken, UNBREAK has a value of (fn NOT BROKEN) for that function and no changes are made to fn. If UNBREAK refuses to work try BREAKing and UNBREAKing it again.

For related information see BREAKIN.

3.3.2 (BREAKIN function {where} {BRKWHEN} {BRKCOMS}) [FSUBR]

inserts a BREAK in the function at the place specified as (BEFORE loc), (AFTER loc) or (AROUND loc) where loc is an editor location specification (defaults to (AROUND TTY:). The optional arguments BRKWHEN (default is T) and BRKCOMS (default is NIL) are the same as for BREAK1.

For related information see BREAK1.

BREAKIN enables the user to insert a break, i.e., a call to BREAK1, at a specified location in an interpreted function. For example, if FOO calls FIE, inserting a break in FOO before the call to FIE is similar to breaking FIE. However, BREAKIN can be used to insert breaks before or after prog labels, particular SETQ expressions, or even the evaluation of a variable. This is because BREAKIN operates by calling the editor and actually inserting a call to BREAK1 at a specified point inside of the function.

The user specifies where the break is to be inserted by a sequence of editor commands. These commands are preceded by BEFORE, AFTER, or AROUND, which BREAKIN uses to determine what to do once the editor has found the specified point, i.e., put the call to BREAK1 BEFORE that point, AFTER that point, or AROUND that point. For example, (BEFORE COND) will insert a break before the first occurrence of COND, (AFTER COND 2 1) will insert a break after the predicate in the first COND clause, (AFTER BF (SETQ X F)) after the last place X is set. Note that (BEFORE TTY:), (AROUND TTY:) or (AFTER TTY:) permit the user to type in commands to the editor, locate the correct point, and verify it for himself using the P command, if he desires. Upon exit from the editor with OK, the break is inserted. (A STOP command typed to TTY: produces the same effect as an unsuccessful edit command in the original specification, e.g., (BEFORE CONDD). In both cases, the editor aborts, and BREAKIN types (NOT FOUND).)

For BREAKIN BEFORE or AFTER, the break expression is NIL, since the value of the break is usually not of interest. For BREAKIN AROUND, the break expression will be the indicated form. When in the break, the user can use the EVAL command to evaluate that form, and see its value, before allowing the computation to proceed. For example, if the user inserted a break after a COND predicate, e.g., (AFTER (EQUAL X Y)), he would be powerless to alter the flow of computation if the predicate were not true, since the break would not be reached. However, by breaking (AROUND (EQUAL X Y)), he can evaluate the break expression, i.e., (EQUAL X Y), see its value and evaluate something else if he wished.

The message typed for a BREAKIN break identifies the location of the break as well as the function, e.g., ((FOO (AFTER COND 2 1)) BROKEN).

BREAKIN is an FEXPR which has a maximum of four arguments. The first argument is the function to be broken in. The second argument is a list of editor commands, preceded by BEFORE, AFTER, or AROUND, which specifies the location inside the function at which to break. If there is no second argument, a value of (AROUND TTY:) is assumed. (See earlier discussion.) The third and fourth arguments are the break condition and the list of commands to be performed before the interactive break occurs, (BRKWHEN and BRKCOMS for BREAK1) respectively. If there is no third argument, a value of T is assumed for BRKWHEN which causes a break each time the BREAKIN break is executed. If the fourth argument is missing, a value of NIL is assumed. For example,

```
(BREAKIN FOO (AROUND COND))
```

inserts a break around the first call to COND in FOO.

It is possible to insert multiple break points, with a single call to BREAKIN by using a list of the form ((BEFORE ...) ... (AROUND ...)) as the second argument. It is also possible to BREAK or TRACE a function which has been modified by BREAKIN, and conversely to BREAKIN a function which is broken or traced. UNBREAK restores functions which have been broken in. GRINDEF makes no attempt to correct the modification of BREAKIN so functions should be unbroken before they are stored on disk.

```
(BREAKIN FOO (AROUND TTY:) T (?= M N) ((*PLUS X Y)))
(BREAKIN FOO2 (BEFORE SETQ) (EQ X Y))
```

3.3.3 (TRACE x1 x2 ...) [FSUBR]

TRACE is an FEXPR. For each atomic argument, it traces the function named each time it is called. For each list in the form (fn1 IN fn2), it traces only those calls to FN1 that occur within FN2. For each list argument not in this form, the CAR is the function to be traced, and the CDR is a list of variables (or forms) the user wishes to see in the trace.

For example, (TRACE (FOO1 Y) (SETQ IN FOO3)) will cause both FOO1 and SETQ IN FOO3 to be traced. SETQ's argument will be printed and the value of Y will be printed for FOO1. Further examples:

```
(TRACE FOO)
(TRACE *TIMES (SELECTQ IN DOIT))
(TRACE (EVAL IN FOO))
(TRACE (TRY M N X (*PLUS N M)))
```

Trace gives the traced function a TRACE property indicating where the actual code for the

function is to be found. The original function is replaced by a call to that new function (whose name is generated by gensym) embedded in a call to BREAK. TRACE uses the global variable #%INDENT to keep its position on the line. The printing of output by TRACE is printed using %PRINFN . TRACE can therefore be pretty printed by:

```
(SETQ %PRINFN (QUOTE PRETPRIN))
(DE PRETPRIN (FORM)
      (SPRINT FORM (*PLUS 10 #%INDENT)))
```

TRACE [BREAK-COMMAND]

Does the work of tracing. Tracing is equivalent to BREAKing a function with BRKCOMS containing the single command TRACE. Thus you can make a normal BREAK act like a TRACE by simply typing this command (No guarantees if the thing broken is not a function). Similarly you can use the TRACE command to cause conditional tracing. (The default that you get with the TRACE function is always to trace.)

3.3.3.1 #%INDENT [VALUE]

is used by the break (trace) package. It is the number of columns to indent before printing. (Notice that in a trace the indenting shows the level of function nesting.)

3.3.3.2 (UNTRACE x1 x2 ...) [FSUBR]

UNTRACE is an FSUBR. It takes a list of functions modified by TRACE and restores them to their original state. It's value is the list of functions that were "untraced". It also undoes TRACEINs. (UNTRACE T) will unbreak the function most recently traced. (UNTRACE) will untrace all of the functions currently traced (i.e. all those on TRACEDFNS). If one of the functions is not traced, UNTRACE has a value of (fn NOT BROKEN) for that function and no changes are made to fn. If UNTRACE refuses to work try TRACEing and UNTRACEing it again.

For related information see TRACEIN.

3.3.3.3 TRACEDFNS [VALUE]

is a list of the functions currently traced

3.3.4 (TRACEIN fn {(AROUND $1) (AROUND $2) ...}) [FSUBR]

where fn is a function name and $1, $2 etc. are editor location specifications. (If no (AROUND ...) arguments are given the user is put in the editor to find the desired expression. When it is the current expression type OK (to exit) and that expression will be traced.) TRACEIN is undone by UNTRACE. TRACEIN is the ultimate tracing facility in that it shows

everything that happens in the execution of the specified code. This is done by using the editor to alter the function definition to trace the evaluation of the located expressions and all of their subexpressions.

Examples: (TRACEIN F1 (AROUND TYO) (AROUND COND 3))

(TRACEIN F2) (then find the desired subexpression in the editor)

WARNING: TRACEIN assumes that the expressions it is given are "well-formed" in the sense that they are to be executed as a unit. Thus it is all right to TRACEIN (Cond ((Null x) T)), but you will get into trouble if you try to TRACEIN just the conditional clause ((Null x) T). This is because it will be interpreted as a function call where the function is the result of (Null x) and the argument is T. The correct way to do this is to TRACEIN the two expressions separately (the way they will be evaluated).

3.3.4.1 (EVL-FIX exp type-of-fix) [SUBR]

and EVL-FIX [PROPERTY] EVL-FIX accepts an expression as its argument and modifies it for expression tracing or counting (depending on the second argument). The second argument is a list of items to be placed nondestructively in front of the form in the embedding process. EVL-FIX is a utility for COUNT and TRACEIN that may find other uses.

For related information see COUNT.

The EVL-FIX property enables the user to explain his FEXPRs and MACROs to EVL-FIX and thus to TRACEIN and COUNT. The EVL-FIX property is a pattern to be applied to the tail of the function call to determine which arguments will EVENTUALLY be evaluated and so should be embedded in EVL-TRACE (or # 0). If an element of the pattern is T, the expression in that position will be embedded by EVL-FIX. If it is NIL, it will not. If an element is a list, that expression will not be embedded, but its subexpressions will have that list applied to them as a pattern.

If the pattern element is TAIL, the pattern element following it is applied to the current expression, if any, and any expressions following it.

If TEST or EVAL is the CAR of a pattern, the pattern is treated specially:

TEST uses the pattern element following it as a condition to determine whether that expression should or should not be embedded.

EVAL is similar to TEST, but the value returned is treated not as a boolean, but as a pattern to be used in place of the pattern whose CAR is EVAL.

The value of the free variable EXP is the function call the pattern will apply to. Also, if before returning the pattern, the free variable NOEMBED is set to T, the function call ITSELF will not be embedded. This feature is not generally needed.

The pattern for COND would be (TAIL (TAIL T)). The clauses of the COND would not be embedded, but their elements would be.

The pattern for SETQ would be (NIL T).

To create a pattern by arbitrary processing, use the pattern (EVAL <whatever>).

3.3.4.2 (EVL-TRACE exp) [FSUBR]

EVL-TRACE evaluates and traces its (expression) argument. EVL-TRACEs are automatically inserted by EVL-FIX, and may also be inserted by the user.

3.3.5 BREAKMACROS [VALUE]

is a list of elements of the form: (atom args ttyline1 ... ttylinen)

Whenever an atomic command is encountered by BREAK1 that it does not recognize, either via BRKCOMS or the teletype, it searches (using ASSOC) the list BREAKMACROS to see if the atom has been defined as a break macro. The form of BREAKMACROS definitions is (... (atom args ttyline1 ... ttylineN) ...). ATOM is the command name. ARGS is the argument(s) for the macro. The arguments of a breakmacro are assigned values from the remainder of the command line in which the macro is called. If ARGS is atomic, it is assigned the remainder of the command line as its value. If ARGS is a list, the elements of the rest of the command line are assigned to the variables, in order. If there are more variables in ARGS then items in the rest of the command line, a value of NIL is filled in. Extra items on the command line are ignored. The TTYLINEs are the body of the breakmacro definition and are lists of break commands or forms to be evaluated. If the atom is defined as a macro, (i.e. is found on BREAKMACROS) BREAK1 assigns values to the variables in ARGS, substitutes these values for all occurrences of the variables in TTYLINEs and appends the TTYLINEs to the front of BRKCOMS. When BREAK1 is ready to accept another command, if BRKCOMS is non-NIL it takes the first element of BRKCOMS and processes it exactly as if it had been a line input from the teletype. This means that a macro name can be defined to expand to any arbitrary collection of expressions that the user could type in. If the command is not contained in BREAKMACROS, it is treated as a function or variable as before.

Example: a command PARGS to print the arguments of the function at LASTPOS could be

defined by evaluating:

```
(NCONC BREAKMACROS (QUOTE ((PARGS NIL (?=)))))
```

A command FP which finds a place on the SPD stack and prints the form there can be defined by:

```
(NCONC BREAKMACROS  (QUOTE (FP X (F .  X)  ((PRINT (SPDLRT
LASTPOS))))))
```

3.3.6 (BREAK0 FN WHEN COMS) [SUBR]

For related information see BREAK1 and BROKENFNS.

BREAK0 is a SUBR. It sets up a break on the function FN by redefining FN as a call to BREAK1 with BRKEXP a form equivalent to the definition of FN, and WHEN, FN and COMS as BRKWHEN, BRKFN, and BRKCOMS, respectively .

BREAK0 also adds FN to the front of the list BROKENFNS. It's value is FN. If FN is non-atomic and of the form (fn1 IN fn2), BREAK0 first calls a function which changes the name of fn1 wherever it appears inside of fn2 to that of a new function, fn1-IN-fn2, which is initially defined as fn1. Then BREAK0 proceeds to break on fn1-IN-fn2 exactly as described above. This procedure is useful for breaking on a function that is called from many places, but where one is only interested in the call from a specific function, e.g. (RPLACA IN FOO), (PRINT IN FIE), etc. This only works in interpreted functions. If fn1 is not found in fn2, BREAK0 returns the value (fn1 NOT FOUND IN fn2). If FN is non-atomic and not of the above form, BREAK0 is called for each member of FN using the same values for WHEN and COMS specified in this call to BREAK0. This distributivity permits the user to specify complicated break conditions without excessive retyping. If FN is non-atomic, the value of BREAK0 is a list of the individual values.

```
(BREAK0 (QUOTE (FOO1 ((PRINT PRIN1) IN (FOO2 FOO3))))
          (QUOTE (EQ X T))
          (QUOTE ((EVAL) (?= Y Z) OK)))
```

will break on FOO1, PRINT-IN-FOO2, PRINT-IN-FOO3, PRIN1-IN-FOO2, and PRIN1-IN-FOO3.

BREAK0 can be used to trace the changing of particular values by SETQ in the following manner:

```
      ›(SETQ VARLIST (QUOTE (X Y FOO)))
      ›(BREAK0 (QUOTE SETQ) (QUOTE (MEMQ (CAR XXXX) VARLIST))
      ›        (QUOTE ((TRACE) (?=) (UNTRACE))))
      (SETQ ARGMENTS?) › (XXXX)
```

(Note: the last line is a question followed by an answer.)

SETQ will be traced whenever CAR of its argument (SETQ is an FSUBR) is a member of

VARLIST.

3.4 SPDL

The Special PushDown List is used for saving forms to be evaluated in the form of an "eval-blip" which is used for backtraces. An eval-blip contains NIL in the left half (spdlft) and the form in the right half (spdlrt). The SPDL is also used for saving variable bindings. The left half of such an entry points to the special cell and the right side to the value. Finally, the interpreter uses the SPDL to hold things which always contain something other than NIL in the left half. LASTPOS and (SPDLPT) indicate a distance from the bottom of the SPDL. In the user's programs, stack pointers are always represented as INUMs. This allows the program to easily modify them with the standard arithmetic functions so that a program can step either up (toward the most recent Eval-Blip) or down (toward the top level of the interpreter) of the stack at will. All of the functions in this group take INUM's for the pointer arguments. The actual pointer to the stack element requires an offset from the beginning of the stack. For the user to obtain a true LISP pointer he must call the function STKPTR (with an INUM argument also). (i.e. if the user wishes to do an RPLACA or RPLACD on an element of the stack, he must get a pointer via STKPTR.)

The SPDL is implemented via PDP10 stack instructions. The stack pointer is kept in register 15 (17 octal) - the right half points to the top of the stack and the left half contains the negative of the number of words available.

3.4.1 (SPDLPT) [SUBR]

The value of SPDLPT is a stack pointer to the current top of the stack. (Returns an INUM).

3.4.2 (SPDLFT P) [SUBR]

The value of SPDLFT is the left side of the stack item pointed to by the stack pointer P.

3.4.3 (SPDLRT P) [SUBR]

The value of SPDLRT is the right side of the stack item pointed to by the stack pointer P.

3.4.4 (STKPTR P) [SUBR]

The value of STKPTR is a true LISP pointer to a stack item.

3.4.5 (NEXTEV P) [SUBR]

If the stack pointer P is a pointer to an Eval-Blip, the value of NEXTEV is P. Otherwise, NEXTEV searches down the stack, starting from P, and returns a stack pointer to the first Eval-Blip it finds. If NEXTEV can not find an Eval-Blip it returns NIL.

3.4.6 (PREVEV P) [SUBR]

PREVEV is similar to NEXTEV except that it moves up the stack instead of down it.

3.4.7 (STKCOUNT NAME P PEND) [SUBR]

The value of STKCOUNT is the number of Eval-Blips with a STKNAME of NAME occurring between stack positions P-1 and PEND, where PEND < P.

3.4.8 (STKNAME P) [SUBR]

If position P is not an Eval-Blip, the value of STKNAME is NIL. If position P is an Eval-Blip and the form is atomic, then the value of STKNAME is that atom. If the form is non-atomic, STKNAME returns the CAR for the form, i.e. the name of the function.

3.4.9 (STKNTH N P) [SUBR]

The value of STKNTH is a stack pointer to the Nth Eval-Blip starting at position P. If N is positive, STKNTH moves up the stack, and if N is negative, STKNTH moves down the stack.

3.4.10 (STKSRCH NAME P FLAG) [SUBR]

The value of STKSRCH is a stack pointer to the first Eval-Blip with a STKNAME of NAME. The direction of the search is controlled by FLAG. If FLAG=NIL, STKSRCH moves down the stack. Otherwise STKSRCH moves up the stack. STKSRCH never returns P for its value, i.e. it steps once before checking for NAME.

3.4.11 (FNDBRKPT P) [SUBR]

The value of FNDBRKPT is a stack pointer to the beginning of the Eval-Block that P is in. The beginning of a Eval-Block is defined as an Eval-Blip which does not contain the next higher Eval-Blip within it. This function is used by the backtrace functions.

3.4.12 (OUTVAL P V) [SUBR]

OUTVAL adjusts P to an Eval-Blip and returns from that position with V.

3.4.13 (SPREDO P V) [SUBR]

SPREDO adjusts P to an Eval-Blip and re-evaluates from that point.

3.4.14 (SPREVAL P V) [SUBR]

SPREVAL evaluates its argument v in its local context to get a form, and then it returns to the context specified by P and evaluates the form in that context, returning from that context with the value. This is very similar to SPREDO except that the EVAL-blip on the stack is changed.

3.4.15 (EVALV A P) [SUBR]

The value of EVALV is the value of the atom A evaluated as of position P. If A is not an atom then it must be the special cell of an atom. By using the special cell instead of the atom, special variables can be handled properly. EVALV is similar to EVAL with two arguments, but is more efficient.

3.4.16 (RETFROM FN VAL) [SUBR]

RETFROM returns VAL from the most recent call to the function FN with the value VAL. For RETFROM to work, there must be an Eval-Blip for FN. The only way to be sure to get an Eval-Blip in compiled code is to call the function with no arguments inside of an ERRSET, e.g. (ERRSET (FUNC)).

3.5 ERROR-OTHER

3.5.1 (ERROR E) [SUBR]

ERROR generates a real LISP error. E is evaluated and printed (unless error messages are suppressed) and then a break occurs just as for any other LISP error.

3.5.2 (ERRORX x) [SUBR]

ERRORX is called when an error occurs. Its argument signifies whether the error is

considered serious (T) or recoverable (NIL). ERRORX first does (USERERRORX x) (if there is a usererrorx) and if that returns a non-nil value it is used. Otherwise it calls //BREAK1 (the break package) and continues the computation with whatever is returned from there (if anything).

Unfortunately the argument to ERRORX is not much help in recovering from errors. Almost all errors are considered serious (the messages can not be suppressed). The exceptions are:

```
UNDEFINED FUNCTION                UNBOUND VARIABLE - EVAL
NON-NUMERIC ARGUMENT              TOO MANY ARGUMENTS SUPPLIED - APPLY
UNDEFINED FUNCTION - APPLY        TOO FEW ARGUMENTS SUPPLIED - APPLY
UNDEFINED PROG TAG - GO           CAN'T ENTER FILE
NO INPUT - INC                    NO OUTPUT - OUTC
CAN'T FIND FILE - INPUT           CAN'T EXPAND CORE
```

For related information see ERRSET.

3.5.3 %PRINFN [VALUE]

Nearly all printing from the error package is done by calling (%PRINFN expr). %PRINFN is an atom (not a function) which should evaluate to the name of a printing function of one argument. %PRINFN is initialized to use PLEV because it can print circular lists, which sometimes result from errors. There has been some small effort to protect against errors that occur in %prinfn, but for the most part, if your %prinfn isn't debugged you are asking for trouble. It is suggested that even if you use another %prinfn, it should use PLEV, because (a) you might get circular lists you didn't want, and (b) some of the break package functions rebind %lookdpth so as to act more appropriately when PLEV is used.

3.5.4 (BKTRC) [SUBR]

BKTRC prints a backtrace of compiled functions. This information is not available from the break-package backtrace commands which only show the interpreted forms on the stack. The format is a list of pairs of functions, one of which called the other.

For related information see BK, BKV, BKE, BKEV, BKF, and BKFV.

3.5.5 (*RSET flag) [SUBR]

sets the flag that determines whether errors will cause the break package to be entered (the default, T) or whether they will just cause a return to the top level (NIL). *RSET returns the old value of the flag. The value of ERRORX is also allowed. It will supress the printing of error messages, but otherwise acts like T.

3.5.6 ERXACTION [PROPERTY]

is a property given to functions in the break package so that they will not appear in backtraces, which would confuse the user (since the break package is supposed to be transparent to the user).

The ERXACTION property is a list of length four. The default (if the atom of interest has no such property) is (T T T T). If the first element is NIL then BKV, BKEV and BKFV act like BK, BKE and BKF respectively for the atom with this property. If the second is NIL then BK and BKV act like BKE and BKEV. If the third is NIL then BKE and BKEV act like BKF and BKFV. If the last one is NIL then BKF and BKFV do not mention this atom at all. Any combination of NILs and Ts is meaningful.

3.5.7 USERERRORX [VALUE]

may be set to the name of a function of one argument which will be called before the error package is entered. If it returns NIL then the error package will be called as usual. Otherwise its value will be used as if it were the value of the break. The argument signifies whether the error is considered to be serious (T) or recoverable (NIL).

```
(DE USERERRORX (FLAG) (PROG (BAD-FORM)
 (SETQ BAD-FORM (SPDLRT (NEXTEV (SUB1 (STKSRCH 'ERRORX (SPDLPT) NIL)))))
 (*** - IF I CAN FIGURE OUT WHAT THE ANSWER SHOULD HAVE BEEN RETURN IT)
 (*** - OTHERWISE (IF I RETURN NIL)
                  THE BREAK PACKAGE WILL BE ENTERED)))
(DV USERERRORX USERERROX)
```

4. THE-TOP-LEVEL

The "top level" is the function which reads what you type at your terminal and decides what to do with it (usually evaluate it and print the result). The default top level function is called TOP-LEVEL. It prompts you for input with numbers in angle brackets (like "<1>").

4.1 (TOP-LEVEL) [SUBR]

TOP-LEVEL is the LISP top level function. As well as being the top level function with which the user interacts, it can be called recursively by the user or any function. Thus, the top level can be invoked from inside the editor, break package, or a user function to make its commands available to the user.

The LISP top-level uses LINEREAD rather than READ. The difference will not usually be noticeable. The principal thing to be careful about is that input to the function or system being called cannot appear on the same line as the top-level call. For example, typing (EDITF FOO) P on one line will edit FOO and evaluate P, not edit FOO and execute the P command in the editor. In order to understand how input lines are interpreted, reading the explanation of USERTOP is strongly recommended.

4.1.1 TOP-LEVEL-COMMANDS

4.1.1.1 RETURN <form> [TOP-LEVEL COMMAND]

returns the result of evaluating form as the value of TOP-LEVEL.

4.1.1.2 FIX <event-spec> [TOP-LEVEL COMMAND]

calls the editor for each of the specified events and then executes them. The event-spec may be optionally followed by editor commands in which case the editor commands will be applied to the events and the user will not be asked to edit them.

4.1.1.3 EDIT <event-spec> [TOP-LEVEL-COMMAND]

is the same as the FIX top-level command but it does not execute the fixed events.

4.1.1.4 REDO <event-spec> [TOP-LEVEL COMMAND]

(re-)executes the events specified.

4.1.1.5 EVENT-SPEC

All of the top-level commands that use event-specifications (??, FIX, EDIT, USE, SUBST,

REDO, UNDO, NAME, FORGET) use the same syntax and conventions to specify events on the history list. An event address identifies one event on the history list. It consists of a sequence of commands for moving an imaginary cursor up or down the list. The cursor position at the end of the list of commands points to the event specified. If any command fails the history command is aborted.

```
<NUMBER>        ;;moves forward (backward if neg) that many events
>       ;;> <atom> searches backward for an event whose function is atom.
<PAT>   ;;searches backward for an event matching the (editor) pattern pat.
_       ;;changes the direction of motion for the next command
=       ;;= <pat> same as <pat> but matches the values of the events
TO      ;;<event1> TO <event2> specifies the sequence of events starting
        ;;with event1 and going up to (not including) event2.
THRU    ;;<event1> THRU <event2> specifies the sequence of events starting
        ;;with event1 and going up to and including event2.
<empty> ;;leaving out an event-spec lets it default to -1.
AND     ;;<event-spec1> AND <event-spec2> joins two lists of events.
@      ;;@ <name> specifies the events named by the name.
```

4.1.1.6 ^^^ [TOP-LEVEL COMMAND]

changes the default top level (INITFN) to the old LISP 1.6 top level and exits from TOP-LEVEL.

4.1.1.7 ?? <event-spec> [TOP-LEVEL COMMAND]

prints the specified events.

4.1.1.8 USE args FOR vars IN event-spec [TOP-LEVEL COMMAND]

substitutes arguments for variables in the specified events. The events are then executed. The number of arguments must be a multiple of the number of variables. For example,

```
<3>(FAC 0)
1
<4>USE 1 2 3 FOR 0 IN FAC
1
2
6
```

4.1.1.9 SUBST args FOR vars IN event-spec [TOP-LEVEL COMMAND]

is like USE but it does not execute the results.

4.1.1.10 UNDO <event-spec> [TOP-LEVEL COMMAND]

undoes the recorded (undoable) side effects of the events specified.

4.1.1.11 NAME <name> <event-spec> [TOP-LEVEL COMMAND]

saves the specified events on the NAMED-EVENTS property of name. This allows those

events (and their values and side effects) to be referenced (by that name) even after they are deleted from the history list (when they are no longer recent).

4.1.1.12 RETRIEVE <name> [TOP-LEVEL COMMAND]

adds the events specified by the name (via the NAME command) to the history list (at the end).

4.1.1.13 AFTER <name> [TOP-LEVEL-COMMAND]

adjusts the (undoable) side effects to reflect the situation after the events named by name (via the NAME command) are executed.

4.1.1.14 BEFORE <name> [TOP-LEVEL-COMMAND]

Adjusts the (undoable) side effects to reflect the situation before the events named by name (via the NAME command) were executed.

4.1.1.15 FORGET <event-spec> [TOP-LEVEL COMMAND]

deletes the information which allows the undoable side effects of the specified events to be undone. (Thus they are no longer undoable.) This is useful for conserving space (if you never want to undo those events).

4.1.2 (VALUEOF "EVENT-SPECIFICATION") [FSUBR]

VALUEOF returns the value(s) of the event(s) specified by EVENT-SPECIFICATION. If a single event is specified, its value will be returned. If more than one event is specified, or an event has more than one subevent (as for REDO, etc.), a list of values will be returned.

4.1.3 TOP-LEVELMACROS [VALUE]

provides a crude macro facility for the top level. The value of TOP-LEVELMACROS is a list of elements of the form (MACRONAME FORMALS TTYLINE1 ... TTYLINEN). This list is used just like BREAKMACROS. TTYLINEi must be formated as if it is a list returned by LINEREAD. That is, TTYLINEi must be a list whose elements are one line of input typed at TOP-LEVEL.

4.1.4 (CHANGESLICE N) [SUBR]

CHANGESLICE sets to N the maximum number of events that will be retained on the history list. The maximum number of events in the history list is initially set to 30.

4.1.5 LISPXHIST [VALUE]

contains the name of the atom (LISPXHISTORY) containing history, current event number, max event number and history slice size information.

4.1.6 LISPXHISTORY [VALUE]

Unless LISPXHIST is changed, LISPXHISTORY contains history and state information for the LISP top level. It is a list containing four elements. The first element is the actual history. The second is the current number for the numeric prompts. The third is the number of events being remembered. The fourth is the maximum number the prompts may reach (which must be more than the third element).

4.1.7 USERTOP [VALUE and SUBR]

USERTOP gives the user a chance to pre-process the input to TOP-LEVEL. If the value of USERTOP is non-NIL then the function USERTOP is called with the line just read by LINEREAD as a parameter. The result is used as if it had been the result of LINEREAD. TOP-LEVEL expects a list of lines, each consisting of a list of expressions to be evaluated. In this respect USERTOP is like TOP-LEVELMACROS. The default USERTOP attempts to allow the user to leave out the outermost set of parentheses.

The default USERTOP does nothing if the input starts with a left parenthesis or a top-level-command. If the input is a single atom, it will normally be left alone. (The exceptions are atoms which are functions but have no values, and some special cases such as CHANGES where one normally wants to call the function rather than see the value. To see the value of CHANGES type "eval changes".) Otherwise the atom is treated as a function and any other expressions on the line are treated as its arguments. If the function is an fexpr, fsubr or macro then the line is simply enclosed in parentheses. Otherwise its arguments are QUOTE'd before enclosing the line in parentheses. Exceptions to this rule are T, NIL and numbers which are not quoted. The quoting may be prevented by preceding an argument with the atom ! (as in QUOTE!).

```
help help       is interpreted as (help help) - help is an fsubr
plist plist     is interpreted as (plist 'plist) - plist is a subr
+ a b           is interpreted as (+ 'a 'b) - not what you mean
+ ! a ! b       is interpreted as (+ a b)
+ !a !b         is interpreted as (+ '!a '!b) - the ! must be separated
                ;;by spaces so as not to look like part of another atom
+ 2 3           is interpreted as (+ 2 3) - numbers are not quoted
setq x y        is interpreted as (setq x y) - probably what you meant
changes         is interpreted as (changes) - an exception
fix 3           is interpreted as itself - fix is a top-level-command
oblist          is interpreted as itself - even if there is a function
                ;;with that name
```

4.1.8 (**TOP**) [SUBR]

simply returns to the top-level. It is called when you type ^^.

4.2 (INITFN FN) [SUBR]

INITFN selects the function of no arguments FN as an initialization function which is evaluated after a LISP error return to the top level has occurred or whenever a BELL (^G) is typed. INITFN returns the previously selected initialization function. Initialization functions are useful when it is desirable to change the top level of LISP. For instance,

```
(INITFN (FUNCTION EVALQUOTE))
```

causes the top level to become EVALQUOTE instead of EVAL.

5. EDITOR

The most frequent use of the editor is to change function definitions (see EDITF), values (EDITV), properties (EDITP), and expressions (EDITE). The beginner is advised to start with the following (very basic) commands: OK, UNDO, P, # (under which are explained two different basic commands which start with numbers) and F.

5.1 EDIT-ATTN

Attention-changing commands (in the editor) do not actually change the thing being edited, but rather allow you to look at a different part of it. The sub-structure upon which the editor's attention is centered is called "the current expression". Thus "changing" the current expression means shifting attention and not actually modifying any structure.

5.1.1 CURRENT-EXPRESSION

At any given moment, the editor's attention is centered on some substructure of the expression being edited. This substructure is called the current expression, and it is what the user sees when he gives the editor the command P, for print. Initially, the current expression is the top level one, i.e., the entire expression being edited.

5.1.2

n (n>0) [EDIT-COMMAND] Adds the nth element of the current expression to the front of the edit chain, thereby making it be the new current expression. Sets LASTAIL for use by UP. Generates an error if the current expression is not a list that contains at least n elements.

-n (n>0) Adds the nth element from the end of the current expression to the front of the edit chain, thereby making it be the new current expression. Sets LASTAIL for use by UP. Generates an error if the current expression is not a list that contains at least n elements.

0

Sets edit chain to CDR of edit chain, thereby making the next higher expression be the new correct expression. Generates an error if there is no higher expression, i.e., CDR of edit chain is NIL. Note that 0 usually corresponds to going back to the next higher left parenthesis, but not always. For example, if the current expression is (A B C D E F G), and the user does

```
# UP P
... C D E F G)
#3 UP P
... E F G)
#0 P
... C D E F G)
```

If the intention is to go back to the next higher left parenthesis, regardless of any intervening tails, the command !0 can be used. (!0 is pronounced bang-zero.)

5.1.3 UP [EDIT-COMMAND]

(1) If a P command would cause the editor to type ... before typing the current expression, i.e., the current expression is a tail of the next higher expression, UP has no effect; otherwise (2) UP modifies the edit chain so that the old current expression (i.e., the one at the time UP was called) is the first element in the new current expression. (If the current expression is the first element in the next higher expression UP simply does a 0. Otherwise UP adds the corresponding tail to the edit chain.

The current expression in each case is (COND ((NULL X) (RETURN Y))).

```
1. #1 P
COND
#UP P
(COND (& &))
2. #-1 P
((NULL X) (RETURN Y))
#UP P
... ((NULL X) (RETURN Y)))
#UP P
... ((NULL X) (RETURN Y)))
3. #P NULL P
(NULL X)
#UP P
((NULL X) (RETURN Y))
#UP P
... ((NULL X) (RETURN Y)))
```

The execution of UP is straightforward, except in those cases where the current expression appears more than once in the next higher expression. For example, if the current expression is (A NIL B NIL C NIL) and the user performs 4 followed by UP, the current expression should then be ... NIL C NIL). UP can determine which tail is the correct one because the commands that descend save the last tail on an internal editor variable, LASTAIL. Thus after the 4 command is executed, LASTAIL is (NIL C NIL). When UP is called, it first determines if the current expression is a tail of the next higher expression. If it is, UP is finished. Otherwise, UP computes (MEMB current-expression next-higher-expression) to obtain a tail beginning with the current expression. (The current expression should always be either a tail or an element of the next higher expression. If it is neither, for example the user has directly (and incorrectly) manipulated the edit chain, UP generates an error.) If there are no other instances of the current-expression in the next higher expression, this tail is the correct one. Otherwise UP uses LASTAIL to select the correct tail. (Occasionally the user can get the edit chain into a state where LASTAIL cannot resolve the ambiguity, for example if there were two non-atomic structures in the same expression that were EQ, and the user

descended more than one level into one of them and then tried to come back out using UP. In this case, UP selects the first tail and prints LOCATION UNCERTAIN to warn the user. Of course, we could have solved this problem completely in our implementation by saving at each descent both elements and tails. However, this would be a costly solution to a situation that arises infrequently, and when it does, has no detrimental effects. The LASTAIL solution is cheap and resolves 99% of the ambiguities.

5.1.4 !0 [EDIT-COMMAND]

Does repeated 0's until it reaches a point where the current expression is not a tail of the next higher expression, i.e., always goes back to the next higher left parenthesis.

5.1.5 ^ [EDIT-COMMAND]

Sets edit chain to LAST of edit chain, thereby making the top level expression be the current expression. Never generates an error.

5.1.6 NX [EDIT-COMMAND]

Effectively does an UP followed by a 2, thereby making the current expression be the next expression. Both NX and BK operate by performing a !0 followed by an appropriate number, i.e. there won't be an extra tail above the new current expression, as there would be if NX operated by performing an UP followed by a 2. An error is generated if the current expression is the last one in a list. (However, !NX will handle this case.)

(NX n) Equivalent to n NX commands, except if an error occurs, the edit chain is not changed. (NX -n) is the same as (BK n).

5.1.7 !NX [EDIT-COMMAND]

Makes current expression be the next expression at a higher level, i.e., goes through any number of right parentheses to get to the next expression.

```
#PP
(PROG (UP)
(SETQ UP L)
LP (COND ((NULL (SETQ L (CDR L))) (ERR NIL))
((NULL (CDR (MEMQ# (CAR L) (CADR L))))
(GO LP)))
(EDITCOM (QUOTE NX))
(SETQ UNFIND UP)
(RETURN L))
#F CDR P
(CDR L)
#NX
NX ?
#!NX P
(ERR NIL)
#!NX P
((NULL &) (GO LP))
#!NX P
(EDITCOM (QUOTE NX))
```

!NX operates by doing 0's until it reaches a stage where the current expression is not the last expression in the next higher expression, and then does a NX. Thus !NX always goes through at least one unmatched right parenthesis, and the new current expression is always on a different level, i.e., !NX and NX always produce different results.

5.1.8 BK [EDIT-COMMAND]

Makes the current expression be the previous expression in the next higher expression. Generates an error if the current expression is the first expression in a list. (BK n) Equivalent to n BK commands, except if an error occurs, the edit chain is not changed. Note: (NX -n) is equivalent to (BK n), and vice versa.

5.1.9 (NTH n) n>0 [EDIT-COMMAND]

Equivalent to n followed by UP, i.e., causes the list starting with the nth element of the current expression. ((NTH 1) is a NOP.) Causes an error if current expression does not have at least n elements

(NTH $) - Generalized NTH command. Effectively performs (LCL . $), Followed by (BELOW \), followed by UP. In other words, NTH locates $, using a search restricted to the current expression, and then backs up to the current level, where the new current expression is the tail whose first element contains, however deeply, the expression that was the terminus of the location operation. For example:

```
#P
(PROG (& &) LP (COND & &) (EDITCOM &) (SETQ UNFIND UP) (RETURN L))
#(NTH UP)
#P
... (SETQ UNFIND UP) (RETURN L))
#
```

If the search is unsuccessful, NTH generates an error and the edit chain is not changed. Note that (NTH n) is just a special case of (NTH $), and in fact, no special check is made for $ a number; both commands are executed identically.

5.1.10 ::

(pattern :: . $) [EDIT-COMMAND] E.g., (COND :: RETURN). Finds a COND that contains a RETURN, at any depth. Equivalent to (F pattern N), (LCL . $) followed by (_ pattern). For example, if the current expression is (PROG NIL (COND ((NULL L) (COND (FLG (RETURN L))))) --), then (COND :: RETURN) will make (COND (FLG (RETURN L))) be the current expression. Note that it is the innermost COND that is found, because this is the first COND encountered when ascending from the RETURN. In other words, (pattern :: $) is not equivalent to (F pattern N), followed by (LCL . $) followed by \. Note that $ is a location specification, not just a pattern. Thus (RETURN :: COND 2 3) can be used to find the RETURN which contains a COND whose first clause contains (at least) three elements. Note also that since $ permits any edit command, the user can write commands of the form (COND :: (RETURN :: COND)), which will locate the first COND that contains a RETURN that contains a COND.

5.1.11 (BELOW com x) [EDIT-COMMAND]

Ascends the edit chain looking for a link specified by COM, and stops x links below that, i.e. BELOW keeps doing 0's until it gets to a specified point, and then backs off N 0's. (X is evaluated, e.g., (BELOW com (*PLUS X Y))) (BELOW com) Same as (BELOW com 1) For example, (BELOW COND) will cause the COND clause containing the current expression to become the new current expression. The BELOW command is useful for locating a substructure by specifying something it contains. For example, suppose the user is editing a list of lists, and wants to find a sublist that contains a FOO (at any depth). He simply executes F FOO (BELOW \).

BELOW operates by evaluating X and then executing COM, or (_ com) if COM is not a recognized edit command, and measuring the length of the edit chain at that point. If that length is M and the length of the current edit chain is N, then BELOW ascends n-m-y links where Y is the value of X. Generates an error if COM causes an error, i.e., it can't find the higher link, or if n-m-y is negative.

5.1.12 (NEX x) [EDIT-COMMAND]

Same as (BELOW x) followed by NX. For example, if the user is deep inside of a SELECTQ clause, he can advance to the next clause with (NEX SELECTQ).

Same as (NEX _). The atomic form of NEX is useful if the user will be performing repeated executions of (NEX x). By simply MARKing the chain corresponding to X, he can use NEX to step through the sublists.

5.1.13 EDIT-MATCH

All of the editor commands that search use the same pattern matching routine. (This routine is available to the user as EDIT4E).

A pattern PAT matches with X if

```
;;    1. PAT is EQ to X.
;;    2. PAT is &.
;;    3. PAT is a number and EQUAL to X.
;;    4. If (CAR PAT) is the atom *ANY*, (CDR PAT) is a
;;    list of patterns, and PAT matches X if and only
;;    if one of the patterns on (CDR PAT) matches X.
;;    5. If PAT is a literal atom or string, and (NTHCHAR
;;    pat -1) is &, then PAT matches with any literal
;;    atom or string which has the same initial
;;    characters as PAT, e.g.  VER& matches with
;;    VERYLONGATOM, as well as "VERYLONGSTRING".
;;    6. If (CAR PAT) is the atom --, PAT matches X if
;;     A. (CDR pat)=NIL, i.e. PAT=(--),
;;      e.g., (A --) matches (A) (A B C) and   (A . B)
;;     In other words, -- can match any tail of
;;     a list.
;;     B. (CDR PAT) matches with some tail of X,
;;      e.g.   (A -- (&)) will match with (A B
;;      C (D)), but not (A B C D), or (A B C
;;      (D) E). However, note that (A -- (&)
;;      --) will match with (A B C (D) E).
;;     In  other words, -- will match any
;;     interior segment of a list.
;;    7. If (CAR PAT) is the atom ==, PAT matches X if
;;    and only if (CDR PAT) is EQ to X. (This pattern
;;    is for use by programs that call the editor as a
;;    subroutine, since any non-atomic expression in a
;;    command typed in by the user obviously cannot be
;;    EQ to existing structure.)
;;    8. Otherwise if X is a list, PAT matches X if (CAR
;;    PAT) matches (CAR X), and (CDR PAT) matches (CDR
;;    X).
```

When searching, the pattern matching routine is called only to match with elements in the structure, unless the pattern begins with :::, in which case CDR of the pattern is matched against tails in the structure. (In this case, the tail does not have to be a proper tail, e.g. (::: A --) will match with the element (A B C) as well as with CDR of (X A B C), since (A B C) is a tail of (A B C).) Thus if the current expressiion is (A B C (B C)),

```
#F (B --)
#P
(B C)
#0 F (::: B --)
#P
... B C (B C))
#F (::: B --)
#P
(B C)
```

5.1.14 EDIT-SEARCH

Search Commands - All of the commands below set LASTAIL for use by UP, set UNFIND for use by \ , and do not change the edit chain or perform any CONSes if they are unsuccessful or aborted.

Editor Searching begins with the current expression and proceeds in print order. Searching usually means find the next instance of this pattern, and consequently a match is not attempted that would leave the edit chain unchanged. (Note, there is a version of the find command which can succeed and leave the current expression unchanged.) At each step, the pattern is matched against the next element in the expression currently being searched, unless the pattern begins with ::: in which case it is matched against the corresponding tail of the expression. (EQ pattern tail-of-expression)=T also indicates a successful match, so that a search for FOO will find the FOO in (FIE . FOO). The only exception to this occurs when PATTERN=NIL, e.g., F NIL. In this case, the pattern will not match with a null tail (since most lists end in NIL) but will match with a NIL element.

If the match is not successful, the search operation is recursive first in the CAR direction and then in the CDR direction, i.e., if the element under examination is a list, the search descends into that list before attempting to match with other elements (or tails) at the same level. (There is also a version of the find command which only attempts matches at the top level of the current expression, i.e., does not descend into elements, or ascend to higher expressions.)

However, at no point is the total recursive depth of the search (sum of number of CARs and CDRs descended into) allowed to exceed the value of the variable MAXLEVEL. At that point, the search of that element or tail is abandoned, exactly as though the element or tail had been completely searched without finding a match, and the search continues with the next element or tail for which the recursive depth is below MAXLEVEL. This feature is designed to enable the user to search circular list structures (by setting MAXLEVEL small), as well as protecting him from accidentally encountering a circular list structure in the course of normal editing. MAXLEVEL is initially set to 192. If a successful match is not found in the current expression, the search automatically ascends to the next higher expression, and

continues searching there on the next expression after the expression it just finished searching. If there is none, it ascends again, etc. This process continues until the entire edit chain has been searched, at which point the search fails, and an error is generated. If the search fails the edit chain is not changed (nor are any CONSes performed.)

If the search is successful, i.e., an expression is found that the pattern matches, the edit chain is set to the value it would have had had the user reached that expression via a sequence of integer commands.

If the expression that matched was a list, it will be the final link in the edit chain, i.e., the new current expression. If the expression that matched is not a list, e.g., is an atom, the current expression will be the tail beginning with that atom, (Except for situations where match is with Y in (X . Y), Y atomic and not NIL. In this case, the current expression will be (X . Y).) i.e., that atom will be the first element in the new current expression. In other words, the search effectively does an UP. (Unless UPFINDFLG=NIL (initially set to T).

5.1.14.1 F pattern [EDIT-COMMAND]

i.e., two commands: the F informs the editor that the next command is to be interpreted as a pattern. If no pattern is given on the same line as the F then the last pattern is used. If an F or BF has been done in this call to the editor, the variable FINDARG is bound in the editor to the pattern. This is the most common and useful form of the find command. If successful, the edit chain always changes, i.e., F pattern means find the next instance of PATTERN.

If (MEMB pattern current-expression) is true, F does not proceed with a full recursive search. If the value of the MEMB is NIL, F invokes the search algorithm described in EDIT-SEARCH. Thus if the current expression were (PROG NIL LP (COND (--(GO LP1))) ... LP1 ...), F LP1 would find the prog label, not the LP1 inside of the GO expression, even though the latter appears first (in print order) in the current expression. Note that 1 (making the atom PROG be the current expression), followed by F LP1 would find the first LP1.

(F pattern N)

Same as F pattern, i.e., finds the next instance of pattern, except the MEMB check of F pattern is not performed.

(F pattern T)

Similar to F pattern, except may succeed without changing edit chain, and does not perform the MEMB check. Thus if the current expression is (COND ..), F COND will look for the next COND, but (F COND T) will 'stay here'.

(F pattern n) n>0

Finds the nth place that pattern matches. Equivalent to (F pattern T) followed by (F pattern N) repeated n-1 times. Each time PATTERN successfully matches, n is decremented by 1, and the search continues, until n reaches 0. Note that the pattern does not have to match with n identical expressions; it just has to match N times. Thus if the current expression is (FOO1 FOO2 FOO3), (F FOO@ 3) will find FOO3. If the pattern does not match successfully N times, an error is generated and the edit chain is unchanged (even if the PATTERN matched n-1 times).

(F pattern) or (F pattern NIL)

Only matches with elements at the top level of the current expression, i.e., the search will not descend into the current expression, nor will it go outside of the current expression. May succeed without changing edit chain. For example, if the current expression is

```
(PROG NIL (SETQ X (COND & &)) (COND &) ...)
```

F (COND --) will find the COND inside the SETQ, whereas (F (COND --)) will find the top level COND, i.e., the second one.

5.1.14.2 (SECOND . $) [EDIT-COMMAND]

Same as (LC . $) Followed by another (LC . $) Except that if the first succeeds and second fails, no change is made to the edit chain.

5.1.14.3 (THIRD . $) [EDIT-COMMAND]

Similar to SECOND.

5.1.14.4 (FS pattern1 ... patternn) [EDIT-COMMAND]

Equivalent to F pattern1 followed by F pattern2 ... followed by F pattern n, so that if F pattern m fails, edit chain is left at place pattern m-1 matched.

5.1.14.5 (F= expression x) [EDIT-COMMAND]

is equivalent to (F (== . expression) x), i.e., searches for a structure eq to expression.

5.1.14.6 (ORF pattern1 ... patternn) [EDIT-COMMAND]

Equivalent to (F (*ANY* pattern1 ... patternn) N), i.e., searches for an expression that is matched by either pattern1 or ... patternn.

5.1.14.7 BF pattern [EDIT-COMMAND]

Backwards Find. Searches in reverse print order, beginning with expression immediately before the current expression (unless the current expression is the top level expression, in which case BF searches the entire expression, in reverse order.) BF uses the same pattern match routine as F, and MAXLEVEL and UPFINDFLG have the same effect, but the searching begins at the end of each list, and descends into each element before attempting to match that element. If unsuccessful, the search continues with the next previous element, etc., until the front of the list is reached, at which point BF ascends and backs up, etc. For example, if the current expression is

```
(PROG NIL (SETQ X (SETQ Y (LIST Z))) (COND ((SETQ W --) --)) --)
```

F LIST followed by BF SETQ will leave the current expression as (SETQ Y (LIST Z)), as will F COND followed by BF SETQ

(BF pattern T)

Search always includes current expression, i.e., starts at end of current expression and works backward, then ascends and backs up, etc. Thus in the previous example, where F COND followed by BF SETQ found (SETQ Y (LIST Z)), F COND followed by (BF SETQ T) would find the (SETQ W --) expression.

(BF pattern) is the same as BF pattern. (BF pattern NIL) is the same as BF pattern.

5.1.14.8 MAXLEVEL [VALUE]

is the maximum depth for editor searches

5.1.14.9 LOCATION-SPEC

Many of the more sophisticated editor commands use a more general method of specifying position called a LOCATION SPECIFICATION. A LOCATION SPECIFICATION is a list of edit commands that are executed in the normal fashion with two exceptions. First, all commands not recognized by the editor are interpreted as though they had been preceded by F. (Normally such commands would cause errors.) For example, the location specification (COND 2 3) specifies the 3rd element in the first clause of the next COND. (Note that the user could always write (F COND 2 3) for (COND 2 3) if he were not sure whether or not COND was the name of an atomic command.) Secondly, if an error occurs while evaluating one of the commands in the location specification, and the edit chain had been changed, i.e., was not the same as it was at the beginning of that execution of the location specification, the location operation will continue. In other words, the location operation keeps going unless it reaches a

state where it detects that it is 'looping', at which point it gives up. Thus, if (COND 2 3) is being located, and the first clause of the next COND contained only two elements, the execution of the command 3 would cause an error. The search would then continue by looking for the next COND. However, if a point were reached where there were no further CONDs, then the first command, COND, would cause the error; the edit chain would not have been changed, and so the entire location operation would fail, and cause an error.

The IF command and the ## function provide a way of using in location specifications arbitrary predicates applied to elements in the current expression.

The meta-symbol $ is used to denote a location specification. Thus $ is a list of commands interpreted as described above. $ Can also be atomic, in which case it is interpreted as (LIST $).

In INSERT, DELETE, REPLACE and CHANGE if $ is NIL (empty), the corresponding operation is performed here (on the current edit chain), e.g., (REPLACE WITH (CAR X)) is equivalent to (:(CAR X)). For added readability, HERE is also permitted, e.g., (INSERT (PRINT X) BEFORE HERE) will insert (PRINT X) before the current expression (but not change the edit chain). Note also that $ does not have to specify a location WITHIN the current expression, i.e., it is perfectly legal to ascend to INSERT, REPLACE, or DELETE. For example (INSERT (RETURN) AFTER ^ PROG -1) will go to the top, find the first PROG, and insert a (RETURN) at its end, and not change the current edit chain.

Finally, the A, B, and : commands, (and consequently INSERT, REPLACE, and CHANGE), all make special checks in E1 thru En for expressions of the form (## . coms). In this case, the expression used for inserting or replacing is a copy of the current expression after executing coms, a list of edit commands. (The execution of coms does not change the current edit chain.) For example, (INSERT (## F COND -1 -1) AFTER3) [not (INSERT F COND -1 (## -1) AFTER 3), which inserts four elements after the third element, namely F, COND, -1, and a copy of the last element in the current expression] will make a copy of the last form in the last clause of the next COND, and insert it after the third element of the current expression.

5.1.14.9.1 $

In descriptions of the editor, the meta-symbol $ is used to denote a location specification. Thus $ is a list of commands interpreted as described under LOCATION-SPEC. $ Can also be atomic, in which case it is interpreted as (LIST $).

5.1.14.9.2 (LC . $) [EDIT-COMMAND]

Provides a way of explicitly invoking the location operation, e.g. (LC COND 2 3) will perform

the search described under edit-search.

5.1.14.9.3 (LCL . $) [EDIT-COMMAND]

Same as LC except search is confined to current expression, i.e., the edit chain is rebound during the search so it looks as if the editor were called on just the current expression. For example, to find a COND containing a RETURN, one might use the location specification (COND (LCL RETURN) \) where the \ would reverse the effects of the LCL command, and make the final current expression be the COND.

5.1.15 EDIT-CHAIN

The edit-chain (the value of which is kept in the variable L) is a list of which the first element (CAR) is the current-expression (the one you are now editing), the next element is what would become the current-expression if you were to do a 0 (the edit command) (which is the next higher level expression of which this is an element) etc. until the last element which is the expression that was passed to the editor.

5.1.15.1 MARKLST [VALUE]

is an internal variable used by the editor to save and later retrieve intermediate copies of the edit chain.

5.1.15.2 MARK [EDIT-COMMAND]

Adds the current edit chain to the front of the list MARKLST.

5.1.15.3 _ [EDIT-COMMAND]

Makes the new edit chain be (CAR MARKLST). Generates an error if MARKLST is NIL, i.e., no MARKS have been performed, or all have been erased.

(_ pattern) [EDIT-COMMAND] Ascends the edit chain looking for a link which matches PATTERN. in other words, it keeps doing 0's until it gets to a specified point. If PATTERN is atomic, it is matched with the first element of each link, otherwise with the entire link. (If pattern is of the form (IF expression), EXPRESSION is evaluated at each link, and if its value is NIL, or the evaluation causes an error, the ascent continues.) For example:

```
#PP
(PROG NIL
      (COND ((NULL (SETQ L (CDR L)))
             (COND (FLG (RETURN L))))
            ((NULL (CDR (MEMB (CAR L) (CADR L)))))
             (GO LP))))
#F CADR
#(_ COND)
#P
(COND (& &) (& &))
#
```

Note that this command differs from BF in that it does not search inside of each link, it simply ascends. Thus in the above example, F CADR followed by BF COND would find (COND (FLG (RETURN L))), not the higher COND. If no match is found, an error is generated and the edit chain is unchanged.

5.1.15.4 __ [EDIT-COMMAND]

Similar to _ but also erases the MARK, i.e., performs (SETQ MARKLST (CDR MARKLST)).

5.1.15.5 \ [EDIT-COMMAND]

Makes the edit chain be the value of UNFIND. Generates an error if UNFIND=NIL. UNFIND is set to the current edit chain by each command that makes a "big jump", i.e., a command that usually performs more than a single ascent or descent, namely ^, _, __, !NX, all commands that involve a search, e.g., F, LC, ::, BELOW, et al and \ and \P themselves. (Except that UNFIND is not reset when the current edit chain is the top level expression, since this could always be returned to via the ^ command.) For example, if the user types F COND, and then F CAR, \ would take him back to the COND. Another \ would take him back to the CAR, etc.

5.1.15.6 \P [EDIT-COMMAND]

Restores the edit chain to its state as of the last print operation, i.e., P, ?, or PP. If the edit chain has not changed since the last printing, \P restores it to its state as of the printing before that one, i.e., two chains are always saved. For example, if the user types P followed by 3 2 1 P, \P will return to the first P, i.e., would be equivalent to 0 0 0. (Note that if the user had typed P followed by F COND, he could use either \ or \P to return to the P, i.e., the action of \ and \P are independent.) Another \P would then take him back to the second P, i.e., the user could use \P to flip back and forth between the two edit chains.

5.2 EDIT-PRINT

5.2.1 P [EDIT-COMMAND]

Prints current expression as though PRINTLEV were given a depth of 2. (P m) Prints mth element of current expression as though PRINTLEV were given a depth of 2. (P 0) : Same as P (P m n) Prints mth element of current expression as though PRINTLEV were given a depth of N. (P 0 n) Prints current expression as though PRINTLEV were given a depth of N. ? is the same as (P 0 100). Both (P m) and (P m n) use the general NTH command to obtain the corresponding element, so that m does not have to be a number, e.g. (P COND 3) will work. All printing functions print to the teletype, regardless of the primary output file. No printing function ever changes the edit chain. All record the current edit chain for use by \P.

5.2.2 ? [EDIT-COMMAND]

same as (P 0 100), i.e. prints the current expression as though PRINTLEV were given a depth of 100.

5.2.3 PP [EDIT-COMMAND]

pretty-prints the current expression.

5.2.4 PP*

is like PP, but forces comments to be shown.

5.2.5 AUTOP [VALUE]

After each line of editor commands is successfully executed, the current expression is automatically printed. Control of this facility is by the global variable AUTOP. Setting AUTOP to NIL suppresses automatic printing, setting AUTOP to an integer causes that integer to be used as the printing depth limit. AUTOP is initially 2. The printing will be suppressed if the last command executed was itself a printing command (P, ?, PP), one of several commands which do not affect the edited expression (such as E, M), or NIL. Appending NIL to a sequence of commands is the standard way of suppressing automatic printout without turning the facility off. AUTOP is a nop which forces automatic type-out if executed last.

5.3 EDIT-MOD

Implementation of Structure Modification Commands

Note: Since all commands that insert, replace, delete or attach structure use the same low level editor functions, the remarks made here are valid for all structure changing commands.

For all replacement, insertion, and attaching at the end of a list, unless the command was typed in directly to the editor, copies of the corresponding structure are used, because of the possibility that the exact same command, (i.e. same list structure) might be used again. (Some editor commands take as arguments a list of edit commands, e.g. (LP F FOO (1 (CAR FOO))). In this case, the command (1 (CAR FOO)) is not considered to have been "typed in" even though the LP command itself may have been typed in. Similarly, commands originating from macros, or commands given to the editor as arguments to EDITF, EDITV, et al, e.g. (EDITF FOO F COND (N --)) are not considered typed in.) Thus if the program constructs the command (1 (A B C)) via (LIST 1 FOO), and gives this command to the editor, the (A B C) used for the replacement will NOT be EQ to FOO. (The user can circumvent this by using the I command, which computes the structure to be used. In the above example, the form of the command would be (I 1 FOO), which would replace the first element with the value of FOO itself. See The rest of this section is included for applications wherein the editor is used to modify a data structure, and pointers into that data structure are stored elsewhere. In these cases, the actual mechanics of structure modification must be known in order to predict the effect that various commands may have on these outside pointers. For example, if the value of FOO is CDR of the current expression, what will the commands (2), (3), (2 X Y Z), (-2 X Y Z), etc., do to FOO?

Deletion of the first element in the current expression is performed by replacing it with the second element and deleting the second element by patching around it. Deletion of any other element is done by patching around it, i.e., the previous tail is altered. Thus if FOO is EQ to the current expression which is (A B C D), and FIE is CDR of FOO, after executing the command (1), FOO will be (B C D) (which is EQUAL but not EQ to FIE). However, under the same initial conditions, after executing (2) FIE will be unchanged, i.e., FIE will still be (B C D) even though the current expression and FOO are now (A C D). (A general solution of the problem just isn't possible, as it would require being able to make two lists EQ to each other that were originally different. Thus if FIE is CDR of the current expression, and FUM is CDDR of the current expression, performing(2) would have to make FIE be EQ to FUM if all subsequent operations were to update both FIE and FUM correctly. Think about it.) Both replacement and insertion are accomplished by smashing both CAR and CDR of the corresponding tail. Thus, if FOO were EQ to the current expression, (A B C D), after (1 X Y Z), FOO would be (X Y Z B C D). Similarly, if FOO were EQ to the current expression, (A B C D), then after (-1 X Y Z), FOO would be (X Y Z A B C D). The N command is accomplished by smashing the last CDR of the current expression a la NCONC. Thus, if FOO were EQ to any tail

of the current expression, after executing an N command, the corresponding expressions would also appear at the end of FOO.

In summary, the only situation in which an edit operation will not change an external pointer occurs when the external pointer is to a proper tail of the data structure, i.e., to CDR of some node in the structure, and the operation is deletion. If all external pointers are to elements of the structure, i.e., to CAR of some node, or if only insertions, replacements, or attachments are performed, the edit operation will always have the same effect on an external pointer as it does on the current expression.

5.3.1 *

(n) [EDIT-COMMAND] n>1 deletes the corresponding element from the current expression.

(n e1 ... em) n,m>1 replaces the nth element in the current expression with e1 ... em.

(-n e1 ... em) n,m>1 inserts e1 ... em before the n element in the current expression.

(N e1 ... em) (the letter "N" for "next" or "nconc", not a number) m>1 attaches e1 ... em at the end of the current expression.

All structure modificaton done by the editor is destructive, i.e., the editor uses RPLACA and RPLACD to physically change the structure it was given. However, all structure modification is undoable, see UNDO.

All of the above commands generate errors if the current expression is not a list, or in the case of the first three commands, if the list contains fewer than n elements. In addition, the command (1), i.e., delete the first element, will cause an error if there is only one element, since deleting the first element must be done by replacing it with the second element, and then deleting the second element. Or, to look at it another way, deleting the first element when there is only one element would require changing a list to an atom (i.e. to NIL) which cannot be done. (However, the command DELETE will work even if there is only one element in the current expression, since it will ascend to a point where it can do the deletion.)

5.3.2 INSERT-DELETE

5.3.2.1 (N e1 ... em) [EDIT-COMMAND]

m>1 attaches e1 ... em at the end of the current expression. This is needed because commands like (-2 ...) can't add to the end of the list.

5.3.2.2 (A e1 ... em) [EDIT-COMMAND]

Inserts e1 ... em after the current expression (or after its first element if it is a tail). Equivalent to UP followed by (-2 e1 ... em) or (N e1 ... em) or (N e1 ... em) whichever is appropriate.

5.3.2.3 (B e1 ... em) [EDIT-COMMAND]

Inserts e1 ... em before the current expression. Equivalent to UP followed by (-1 e1 ... em). (If the current expression is a tail then insert before the first element.) For example, to insert FOO before the last element in the current expression, perform -1 and then (B FOO).

5.3.2.4 (: e1 ... em) [EDIT-COMMAND]

Replaces the current expression by e1 ... em. Equivalent to UP followed by (1 e1 ... em). If the current expression is a tail then replace its first element. (:) is equivalent to DELETE.

5.3.2.5 DELETE or (:) [EDIT-COMMAND]

Deletes the current expression, or if the current expression is a tail, deletes its first element.

```
(DELETE . $)
```

Does a (LC . $) followed by DELETE. Current edit chain is not changed (Unless the current expression is no longer a part of the expression being edited, e.g., if the current expression is ... C) and the user performs (DELETE 1), the tail, (C), will have been cut off. Similarly, if the current expression is (CDR Y) and the user performs (REPLACE WITH (CAR X)).), but UNFIND is set to the edit chain after the DELETE was performed.

DELETE first tries to delete the current expression by performing an UP and then a (1). This works in most cases. However, if after performing UP, the new current expression contains only one element, the command (1) will not work. Therefore DELETE starts over and performs a BK, followed by UP, followed by (2). For example, if the current expression is (COND ((MEMB X Y)) (T Y)), and the user performs -1, and then DELETE, the BK-UP-(2) method is used, and the new current expression will be ... ((MEMB X Y))) However, if the next higher expression contains only one element, BK will not work. So in this case, DELETE performs UP, followed by (: NIL), i.e., it REPLACES the higher expression by NIL. For example, if the current expression is (COND ((MEMB X Y)) (T Y)) and the user performs F MEMB and then DELETE, the new current expression will be ... NIL (T Y)) and the original expression would now be (COND NIL (T Y)). The rationale behind this is that deleting (MEMB X Y) from ((MEMB X Y)) changes a list of one element to a list of no elements, i.e., () or NIL. Note that 2 followed by DELETE

would DELETE ((MEMB X Y)) NOT replace it by NIL.

For related information see FILESPEC.

5.3.2.6 (INSERT e1 ... em BEFORE . $) [EDIT-COMMAND]

Similar to (LC. $) followed by (B e1 ... em).

```
#P
(PROG (W Y X) (SELECTQ ATM & NIL) (OR & &) (PRIN1 &))
#(INSERT LABEL BEFORE PRIN1)
#P
(PROG (W Y X) (SELECTQ ATM & NIL) (OR & &) LABEL (PRIN1 &))
```

Current edit chain is not changed, but UNFIND is set to the edit chain after the B was performed, i.e., \ will make the edit chain be that chain where the insertion was performed. (INSERT e1 ... em AFTER . $) Similar to INSERT BEFORE except uses A instead of B. (INSERT e1 ... em FOR . $) Similar to INSERT BEFORE except uses : for B.

For related information see //INSERT.

5.3.2.7 (REPLACE $ WITH e1 ... em) [EDIT-COMMAND]

Here $ is the segment of the command between REPLACE and WITH. Same as (INSERT e1 ... em FOR . $). (BY can be used for WITH.)

```
Example: (REPLACE COND -1 WITH (T (RETURN L)))
```

5.3.2.8 (CHANGE $ TO e1'... em) [EDIT-COMMAND]

Same as REPLACE WITH

For related information see UP and EDIT-SEARCH.

5.3.2.9 UPFINDFLG

Form Oriented Editing and the Role of UP The UP that is performed before A, B, and : commands (and therefore in INSERT, CHANGE, REPLACE, and DELETE commands after the location portion of the operation has been performed.), makes these operations form-oriented. For example, if the user types F SETQ, and then DELETE, or simply (DELETE SETQ), he will delete the entire SETQ expression, whereas (DELETE X) if X is a variable, deletes just the variable X. In both cases, the operation is performed on the corresponding FORM and in both cases is probably what the user intended. Similarly, if the user types (INSERT (RETURN Y) BEFORE SETQ), he means before the SETQ expression, not before the atom SETQ. There is some ambiguity in (INSERT expr AFTER functionname), as the user might mean make expr be the function's first argument. Similarly, the user cannot write (REPLACE SETQQ WITH SETQ) to mean change the name of the function. The user must in these cases

write (INSERT expr AFTER functionname 1), and (REPLACE SETQQ 1 WITH SETQ). A consequent of this procedure is that a pattern of the form (SETQ Y --) can be viewed as simply an elaboration and further refinement of the pattern SETQ. Thus (INSERT (RETURN Y) BEFORE SETQ) and (INSERT (RETURN Y) BEFORE (SETQ Y --)) perform the same operation (Assuming the next SETQ is of the form (SETQ Y-)).) and, in fact, this is one of the motivations behind making the current expression after F SETQ, and F (SETQ Y --) be the same. Occasionally, however, a user may have a data structure in which no special significance or meaning is attached to the position of an atom in a list, as LISP attaches to atoms that appear as CAR of a list, versus those appearing elsewhere in a list. In general, the user may not even know whether a particular atom is at the head of a list or not. Thus, when he writes (INSERT expression AFTER FOO), he means after the atom FOO, whether or not it is CAR of a list. By setting the variable UPFINDFLG to NIL (Initially, and usually, set to T.) the user can suppress the implicit UP that follows searches for atoms, and thus achieve the desired effect. With UPFINDFLG = NIL, after F FOO, for example, the current expression will be the atom FOO. In this case, the A, B, and : operations will operate with respect to the atom FOO. If the user intends the operation to refer to the list which FOO heads, he simply uses instead the pattern (FOO --).

5.3.3 EMBED-EXTRACT

EXTRACT is an editor command which replaces the current expression with one of its subexpressions (from any depth). EMBED replaces the current expression with a new expression which contains it as a subexpression.

5.3.3.1 (XTR . $) [EDIT-COMMAND]

Replaces the original current expression with the expression that is current after performing (LCL . $). For example, if the current expression is (COND ((NULL X) (PRINT Y))), (XTR PRINT), or (XTR 2 2) will replace the COND by the PRINT. If the current expression after (LCL . $) is a tail of a higher expression, its first element is used. For example, if the current expression is (COND ((NULL X) Y) (T Z)), then (XTR Y) will replace the COND with Y. If the extracted expression is a list, then after XTR has finished, the current expression will be that list. Thus, in the first example, the current expression after the XTR would be (PRINT Y). If the extracted expression is not a list, the new current expression will be a tail whose first element is that non-list. Thus, in the second example, the current expression after the XTR would be ... Y followed by whatever followed COND. If the current expression initially is a tail, extraction works exactly the same as though the current expression were the first element in that tail. Thus is the current expression is (XTR PRINT) will replace the COND by the PRINT, leaving (PRINT Y) as the current expression.

5.3.3.2 (MBD x) [EDIT-COMMAND]

X is a list, substitutes (a la SUBST, i.e., a fresh copy is used for each substitution) the current expression for all instances of the atom * in x, and replaces the current expression with the result of that substitution. (MBD e1 ... em) : Equivalent to (MBD (e1 ... em *)). (MBD x) : X atomic, same as (MBD (x *)). All three forms of MBD leave the edit chain so that the larger expression is the new current expression. If the current expression initially is a tail, embedding works exactly the same as though the current expression were the first element in that tail.

Example: If the current expression is (PRINT Y), (MBD (COND ((NULL X) *) ((NULL (CAR Y)) * (GO LP))) would replace (PRINT Y) with (COND((NULL X) (PRINT Y)) ((NULL (CAR Y)) (PRINT Y) (GO LP))).

5.3.3.3 (EXTRACT $1 FROM $2) [EDIT-COMMAND]

($1 is the segment between EXTRACT and FROM.) Performs (LC . $2) And then (XTR . $1). Current edit chain is not changed, but UNFIND is set to the edit chain after the XTR was performed. Example: If the current expression is (PRINT (COND ((NULL X) Y) (T Z))) then following (EXTRACT Y FROM COND), the current expression will be (PRINT Y). (EXTRACT 2 -1 FROM COND), (EXTRACT Y FROM 2), (EXTRACT 2 -1 FROM 2) will all produce the same result.

5.3.3.4 (EMBED $ IN . x) [EDIT-COMMAND]

($ is the segment between EMBED and IN.) Does (LC . $) and then (MBD . x). Edit chain is not changed, but UNFIND is set to the edit chain after the MBD was performed. Example: (EMBED PRINT IN SETQ X), (EMBED 3 2 IN RETURN), (EMBED COND 3 1 IN (OR * (NULL X))). WITH can be used for IN, and SURROUND can be used for EMBED, e.g., (SURROUND NUMBERP WITH (AND * (MINUSP X))).

5.3.4 MOVE-COPY

5.3.4.1 (MOVE $1 TO com . $2) [EDIT-COMMAND]

($1 is the segment between MOVE and TO.) Where COM is BEFORE, AFTER, or the name of a list command, e.g., :, N, etc. Performs (LC . $1), Obtains the current expression there (or its first element, if it is a tail), let us call this expr; MOVE then goes back to original edit chain, performs (LC . $2), Peforms (com expr), then goes back to $1 and deletes expr. Edit chain is not changed. UNFIND is set to edit chain after (com expr) was performed.

If $2 is NIL, or (HERE), the current position specifies where the operation is to take place.

In this case, UNFIND is set to where the expression that was moved was originally located, i.e., $1. Finally, if $1 is NIL, the MOVE command allows the user to specify some place the current expression is to be moved to. In this case, the edit chain is changed, and is the chain where the current expression was moved to; UNFIND is set to where it was.

For example, if the current expression is (A B D C), (MOVE 2 TO AFTER 4) will make the new current expression be (A C D B). Note that 4 was executed as of the original edit chain, and that the second element had not yet been removed.

```
#?
(PROG (L) (EDLOC (CDDR C)) (RETURN (CAR L)))
#(MOVE 3 TO : CAR)
(PROG (L) (RETURN (EDLOC (CDDR C))))
#P
... (SELECTQ OBJPR & &) (RETURN &) LP2 (COND & & ))
#(MOVE 2 TO N 1)
... (SELECTQ OBJPR & & &) LP2 (COND & &))


#P
(OR (EQ X LASTAIL) (NOT &) (AND & & &))
#(MOVE 4 TO AFTER (BELOW COND))
(OR (EQ X LASTAIL) (NOT &))
#\ P
... (& &) (AND & & &) (T & &))
#P
(TENEX)
#(MOVE ^ F APPLY TO N HERE)
(TENEX (APPLY & & ))
#P
(SELECTQ OBJPR (&) (PROGN & &))
#(MOVE TO BEFORE LOOP)
... (SELECTQ OBJPR & &) LOOP (RPLACA DFPRP &) (RPLACD DFPRP &))
```

5.3.4.2 (MV com . $) [EDIT-COMMAND]

is the same as (MOVE HERE TO com . $)

5.3.4.3 (COPY $1 TO com . $2) [EDIT-COMMAND]

is like MOVE except that the source expression is not deleted.

For related information see SUBST.

5.3.4.4 (CP com . $) [EDIT-COMMAND]

is like MV except that the source expression is not deleted.

5.3.5 MOVE-PARENS

Commands That "Move Parentheses" The commands presented in this section permit modification of the list structure itself, as opposed to modifying components thereof. Their

effect can be described as inserting or removing a single left or right parenthesis, or pair of left and right parentheses. Of course, there will always be the same number of left parentheses as right parentheses in any list structure, since the parentheses are just a notational guide to the structure provided by PRINT. Thus, no command can insert or remove just one parenthesis, but this is suggestive of what actually happens. In all six commands, n and m are used to specify an element of a list, usually of the current expression. In practice, n and m are usually positive or negative integers with the obvious interpretation. However, all six commands use the generalized NTH command to find their element(s), so that nth element means the first element of the tail found by performing (NTH n). In other words, if the current expression is (LIST (CAR X) (SETQ Y (CONS W Z))), then (BI 2 CONS), (BI X -1), and (BI X Z) all specify the exact same operation. All six commands generate an error if the element is not found, i.e., the NTH fails. All are undoable.

5.3.5.1 (BI n m) [EDIT-COMMAND]

Both in. Inserts parentheses before the nth element and after the mth element in the current expression. Generates an error if the mth element is not contained in the nth tail, i.e., the mth element must be "to the right" of the nth element. Example: If the current expression is (A B (C D E) F G), then (BI 2 4) will modify it to be (A (B (C D E) F) G). (BI n) : Same as (BI n n). Example: If the current expression is (A B (C D E) F G), then (BI -2) will modify it to be (A B (C D E) (F) G).

5.3.5.2 (BO n) [EDIT-COMMAND]

Both out. Removes both parentheses from the nth element. Generates an error if nth element is not a list. Example: If the current expression is (A B (C D E) F G), then (BO D) will modify it to be (A B C D E F G).

5.3.5.3 (LI n) [EDIT-COMMAND]

Left in. Inserts a left parenthesis before the nth element (and a matching right parenthesis at the end of the current expression), i.e., equivalent to (BI n -1). Example: If the current expression is (A B (C D E) F G), then (LI 2) will modify it to be (A (B (C D E) F G)).

5.3.5.4 (LO n) [EDIT-COMMAND]

Left out. Removes a left parenthesis from the nth element. All elements following the nth element are deleted. Generates an error if nth element is not a list. Example: If the current expression is (A B (C D E) F G), then (LO 3) will modify it to be (A B C D E).

5.3.5.5 (RI n m) [EDIT-COMMAND]

Right in. Inserts a right parenthesis after the mth element of the nth element. The rest of the nth element is brought up to the level of the current expression. Example: If the current expression is (A (B C D E) F G), (RI 2 2) will modify it to be (A (B C) D E F G). Another way of thinking about RI is to read it as "move the right parenthesis at the end of the nth element IN to after the mth element."

5.3.5.6 (RO n) [EDIT-COMMAND]

Right out. Removes the right parenthesis from the nth element, moving it to the end of the current expression. All elements following the nth element are moved inside of the nth element. Generates an error if nth element is not a list. Example: If the current expression is (A B (C D E) F G), (RO 3) will modify it to be (A B (C D E F G)). Another way of thinking about RO is to read it as "move the right parenthesis at the end of the nth element OUT to the end of the current expression."

5.3.6 (R x y) [EDIT-COMMAND]

Replaces all instances of x by y in the current expression, e.g., (R CAADR CADAR). Generates an error if there is not at least one instance.

R operates by performing a DSUBST. The current expression is the third argument to DSUBST, i.e., the expression being substituted into, and y is the first argument to DSUBST, i.e., the expression being substituted. R computes the second argument to DSUBST, the expression to be substituted for, by performing (F x T). The second argument is then the current expression at that point, or if that current expression is a list and x is atomic, then the first element of that current expression. Thus x can be the S-expression (or atom) to be substituted for, or can be a pattern which specifies that S-expression (or atom). For example, if the current expression is (LIST FUNNYATOM1 FUNNYATOM2 (CAR FUNNYATOM1)), then (R FUN@ FUNNYATOM3) will substitute FUNNYATOM3 for FUNNYATOM1 throughout the current expression. Note that FUNNYATOM2, even though it would have matched with the pattern FUN@, is NOT replaced. Similarly, if (LIST(CAR X) (CAR Y)) is the first expression matched by (LIST --), then (R (LIST --) (LIST (CAR Y) (CAR Z))) is equivalent to (R (LIST (CARX) (CARY)) (LIST (CAR Y) (CAR Z))), i.e., both will replace all instances of (LIST (CAR X) (CAR Y)) by (LIST (CAR Y) (CAR Z)). Note that other forms beginning with LIST will not be replaced, even though they would have matched with (LIST --). To change all expressions of the form (LIST --) to (LIST (CAR Y) (CAR Z)), the user should perform (LP (REPLACE (LIST --) WITH (LIST (CAR Y) (CAR]. UNFIND is set to the edit chain following the find command so that

\ will make the current expression be the place where the first substitution occurred.

5.3.7 (SW n m) [EDIT-COMMAND]

Switches the nth and mth elements of the current expression. For example, if the current expression is (LIST (CONS (CAR X) (CAR Y)) (CONS (CDR Y))), (SW 2 3) will modify it to be (LIST (CONS (CDR X) (CDR Y)) (CONS (CAR X) (CAR Y))). The relative order of n and m is not important, ie, (SW 3 2) and (SW 2 3) are equivalent. SW uses the generalized NTH command to find the nth and mth elements, a la the BI-BO commands. Thus in the previous example, (SW CAR CDR) would produce the same result.

5.3.8 TO-THRU

TO and THRU EXTRACT, EMBED, DELETE, REPLACE, and MOVE can be made to operate on several contiguous elements, i.e., a segment of a list, by using the TO or THRU command in their respective location specifications. THRU and TO are not very useful commands by themselves, and are not intended to be used "solo", but in conjunction with EXTRACT, EMBED, DELETE, REPLACE, and MOVE. After THRU and TO have operated, they set an internal editor flag informing the above commands that the element they are operating on is actually a segment, and that the extra pair of parentheses should be removed when the operation is complete. TO and THRU can also be used directly with XTR (which takes a location specification), as in (XTR (2 THRU 4)) (from the current expression).

5.3.8.1 TO

($1 TO $2) [EDIT-COMMAND]

Same as THRU except last element not included.

($1 TO)

Same as ($1 THRU -1)

5.3.8.2 THRU

($1 THRU $2) [EDIT-COMMAND]

Does a (LC . $1), Followed by an UP, and then a (BI 1 $2), thereby grouping the segment into a single element, and finally does a 1, making the final current expression be that element. For example, if the current expression is (A (B (C D) (E) (F G H) I) J K), following (C THRU G), the current expression will be ((C D) (E) (F G H)). If both $1 and $2 are numbers, and $2 is greater than $1, then $2 counts from the beginning of the current expression, the same as $1. In other words, if the current expression is (A B C D E F G), (3 THRU 4) means (C

THRU D), not (C THRU F). In this case, the corresponding BI command is (BI 1 $2-$1+1).

```
($1 THRU)
```

same as ($1 THRU -1)

```
#P
(PROG NIL (SETQ A &) (RPLACA & &) (PRINT &) (RPLACD & &))
#(MOVE (3 THRU 4) TO BEFORE 5) P
(PROG NIL (PRINT &) (SETQ A &) (RPLACA & &) (RPLACD & &))
```

Note that when specifing $2 in the MOVE, 5 was used instead of 6. This is because the $2 is located after $1 is. The THRU location groups items together and thus changes the numeric location of the following items.

```
#P
(PROG NIL (PRIN1 &) (PRIN1 &) (SETQ IND &) (SETQ VAL &) (PRINT &))
#(MOVE (5 THRU 7) TO BEFORE 3)
(PROG NIL (SETQ IND &) (SETQ VAL &) (PRINT &) (PRIN1 &) (PRIN1 &))
#(DELETE (SETQ THRU PRI@))
= PRINT
(PROG NIL (PRIN1 &) (PRIN1 &))
#P
... LP (SELECTQ & & &) (SETQ Y &) OUT (SETQ FLG &) (RETURN Y))
#(MOVE (1 TO OUT) TO N HERE)
... OUT (SETQ FLG &) (RETURN Y) LP (SELECTQ & & &) (SETQ Y &))
```

5.4 EDIT-UNDO

Each command that causes structure modification automatically adds an entry to the front of UNDOLST containing the information required to restore all pointers that were changed by the command. The UNDO command undoes the last, i.e., most recent such command. Whenever the user continues an editing session as described under SAVE, the undo information of the previous session(s) is protected by inserting a special blip, called an undo-block on the front of UNDOLST. This undo-block will terminate the operation of a !UNDO, thereby confining its effect to the current session, and will similarly prevent an UNDO command from operating on commands executed in the previous session. Thus, if the user enters the editor continuing a session, and immediately executes an UNDO or !UNDO, UNDO and !UNDO will type BLOCKED, instead of NOTHING SAVED. Similarly, if the user executes several commands and then undoes them all, either via several UNDO commands or a !UNDO command, another UNDO or !UNDO will also type BLOCKED.

5.4.1 UNDO [EDIT-COMMAND]

Each command that causes structure modification automatically adds an entry to the front of UNDOLST containing the information required to restore all pointers that were changed by the command. The UNDO command undoes the last, i.e., most recent, structure modification command that has not yet been undone, and prints the name of that command, e.g., MBD

UNDONE. (Since UNDO and !UNDO causes structure modification, they also add an entry to UNDOLST. However, UNDO and !UNDO entries are skipped by UNDO, e.g., if the user performs an INSERT, and then an MBD, the first UNDO will undo the MBD, and the second will undo the INSERT. However, the user can also specify precisely which command he wants undone. In this case, he can undo an UNDO command, e.g., by typing UNDO UNDO, or undo a command other than that most recently performed.) The edit chain is then exactly what it was before the 'undone' command had been performed. If there are no commands to undo, UNDO types NOTHING SAVED.

5.4.2 !UNDO [EDIT-COMMAND]

Undoes all modifications performed during this editing session, i.e., this call to the editor. As each command is undone, its name is printed a la UNDO. If there is nothing to be undone, !UNDO prints NOTHING SAVED.

5.4.3 UNDOLST [VALUE]

Each editor command that causes structure modification automatically adds an entry to the front of UNDOLST containing the information required to restore all pointers that were changed by the command.

5.4.4 UNBLOCK [EDIT-COMMAND]

Removes an undo-block. If executed at a non-blocked state, i.e., if UNDO or !UNDO could operate, types NOT BLOCKED.

5.4.5 TEST [EDIT-COMMAND]

Adds an undo-block at the front of UNDOLST. Note that TEST together with !UNDO provide a 'tentative' mode for editing, i.e., the user can perform a number of changes, and then undo all of them with a single !UNDO command.

5.4.6 ?? [EDIT-COMMAND]

Prints the entries on UNDOLST. The entries are listed in the reverse order of their execution, i.e., the most recent entry first. For example:

```
#P
(CONS (T &) (& &))
#(1 COND) (SW 2 3) P
(COND (& &) (T &))
#??
SW (1 --)
```

5.5 EDIT-EVAL

5.5.1 E [EDIT-COMMAND]

Only when typed in, (i.e., (INSERT D BEFORE E) will treat E as a pattern) causes the editor to call the LISP interpreter giving it the next input as argument. For example,

```
#E (LENGTH (CAR L))
```

will print the length of the current expression (recall that L is the edit-chain and its CAR is the current expression).

(E x) Evaluates X, i.e., performs (EVAL x), and prints the result on the teletype. (E x T) Same as (E x) but does not print. The (E x) and (E x T) commands are mainly intended for use by MACROS and subroutine calls to the editor; the user would probably type in a form for evaluation using the more convenient format of the (atomic) E command.

5.5.2 (I c x1 ... xn) [EDIT-COMMAND]

Same as (c y1 ... yn) where yi=(EVAL xi). Example: (I 3 (CDR FOO)) will replace the 3rd element of the current expression with the CDR of the value of FOO. (The I command sets an internal flag to indicate to the structure modification commands not to copy expression(s) when inserting, replacing, or attaching.) (I N FOO (CAR FIE)) will attach the value of FOO and CAR of the value of FIE to the end of the current expression. (I F= FOO T) will search for an expression EQ to the value of FOO. If c is not an atom, it is evaluated as well. Example: (I (COND ((NULL FLG) (QUOTE -1)) (T 1)) FOO), if FLG is NIL, inserts the value of FOO before the first element of the current expression, otherwise replaces the first element by the value of FOO.

5.5.3 (## com1 com2 ... comn) [FSUBR]

is an FSUBR (not a command). Its value is what the current expression would be after executing the edit commands com1 ... comn starting from the present edit chain. Generates an error if any of com1 thru comn cause errors. The current edit chain is never changed. (Recall that A,B,:,INSERT, REPLACE, and CHANGE make special checks for ## forms in the expressions used for inserting or replacing, and use a copy of ## form instead. Thus, (INSERT (## 3 2) AFTER 1) is equivalent to (I INSERT (COPY (## 3 2)) (QUOTE AFTER) 1).) Example: (I R (QUOTE X) (## (CONS ..Z))) replaces all X's in the current expression by the first CONS containing a Z.

5.5.4 (COMS x1 ... xn) [EDIT-COMMAND]

Each xi is evaluated and its value executed as a command. The I command is not very convenient for computing an entire edit command for execution, since it computes the command name and its arguments separately. Also, the I command cannot be used to compute an atomic command. The COMS and COMSQ commands provide more general ways of computing commands. For example, (COMS (COND (X (LIST 1 X)))) will replace the first element of the current expression with the value of X if non-NIL, otherwise do nothing. (NIL as a command is a NOP.)

For related information see EDITL.

5.5.5 (COMSQ com1 ... comn) [EDIT-COMMAND]

Executes com1 ... comn. COMSQ is mainly useful in conjunction with the COMS command. For example, suppose the user wishes to compute an entire list of commands for evaluation, as opposed to computing each command one at a time as does the COMS command. He would then write (COMS (CONS (QUOTE COMSQ) x)) where x computed the list of commands, e.g., (COMS (CONS (QUOTE COMSQ) (GET FOO (QUOTE COMMANDS)))).

5.6 EDIT-TEST

5.6.1 (IF x) [EDIT-COMMAND]

Generates an error unless the value of (EVAL x) is non-NIL, i.e., if (EVAL x) causes an error or (EVAL x)=NIL, IF will cause an error. (IF x coms1 coms2) If (EVAL x) is non-NIL, execute coms1; if (EVAL x) causes an error or is equal to NIL, execute coms2. (IF x coms1) If (EVAL x) is non-NIL, execute coms1; otherwise generate an error. For some editor commands, the occurrence of an error has a well defined meaning, i.e., they use errors to branch on as COND uses NIL and non-NIL. For example, an error condition in a location specification may simply mean "not this one, try the next." Thus the location specification

```
(*PLUS (E (OR (NUMBERP (## 3)) (ERR NIL)) T))
```

specifies the first *PLUS whose second argument is a number. The IF command, by equating NIL to error, provides a more natural way of accomplishing the same result. Thus, an equivalent location specification is (*PLUS (IF (NUMBERP (## 3)))). For example, the command (IF (NULL A) NIL (P)) will print the current expression provided A=NIL.

5.6.2 (LP . coms) [EDIT-COMMAND]

Repeatedly executes coms, a list of commands, until an error occurs. For example, (LP F PRINT (N T)) will attach a T at the end of every PRINT expression. (LP F PRINT (IF (## 3) NIL ((N T)))) will attach a T at the end of each print expression which does not already have a second argument. (i.e. The form (## 3) will cause an error if the edit command 3 causes an error, thereby selecting ((N T)) as the list of commands to be executed. The IF could also be written as (IF (CDDR (##)) NIL ((N T))).) When an error occurs, LP prints n OCCURRENCES, where n is the number of times COMS was successfully executed. The edit chain is left as of the last complete successful execution of COMS. In order to prevent non-terminating loops, both LP and LPQ terminate when the number of iterations reaches MAXLOOP, initially set to 30.

5.6.3 (LPQ . Coms) [EDIT-COMMAND]

Same as LP but does not print n OCCURRENCES.

5.6.4 (ORR coms1 ... Comsn) [EDIT-COMMAND]

ORR begins by executing coms1, a list of commands. If no error occurs, ORR is finished. Otherwise, ORR restores the edit chain to its original value, and continues by executing coms2, etc. If none of the command lists execute without errors, i.e., the ORR "drops off the end", ORR generates an error. Otherwise, the edit chain is left as of the completion of the first command list which executes without error. (NIL as a command list is perfectly legal, and will always execute successfully. Thus, making the last 'argument' to ORR be NIL will insure that the ORR never causes an error. Any other atom is treated as (atom), i.e., the example given below could be written as

```
(ORR NX !NX NIL).)
```

For example, (ORR (NX) (!NX) NIL) will perform a NX, if possible, otherwise a !NX, if possible, otherwise do nothing. Similarly, DELETE could be written as (ORR (UP (1)) (BK UP (2)) (UP (: NIL))).

5.6.5 MAXLOOP [VALUE]

is the maximum number of iterations for an editor LP or LPQ command.

5.7 EDIT-MACROS

Many of the more sophisticated branching commands in the editor, such as ORR, IF, etc., are most often used in conjunction with edit macros. The macro feature permits the user to define new commands and thereby expand the editor's repertoire. (However, built in commands always take precedence over macros, i.e., the editor's repertoire can be expanded, but not modified.) Macros are defined by using the M command.

5.7.1 (M c . coms) [EDIT-COMMAND]

For c an atom, M defines c as an atomic command. (If a macro is redefined, its new definition replaces its old.) Executing c is then the same as executing the list of commands COMS. Macros can also define list commands, i.e., commands that take arguments. (M (c) (arg[1] ... arg[n]) . coms) C an atom. M defines c as a list command. Executing (c e1 ... en) is then performed by substituting e1 for arg[1], ... en for arg[n] throughout COMS, and then executing COMS. A list command can be defined via a macro so as to take a fixed or indefinite number of 'arguments'. The form given above specified a macro with a fixed number of arguments, as indicated by its argument list. If the 'argument list' is atomic, the command takes an indefinite number of arguments. (M (c) args . coms) C, args both atoms, defines c as a list command. Executing (c e1 ... en) is performed by substituting (e1 ... en), i.e., CDR of the command, for args throughout coms, and then executing coms.

For example, (M BP BK UP P) will define BP as an atomic command which does three things, a BK, an UP, and a P. Note that macros can use commands defined by macros as well as built in commands in their definitions. For example, suppose Z is defined by (M Z -1 (IF (NULL (##)) NIL (P))), i.e. Z does a -1, and then if the current expression is not NIL, a P. Now we can define ZZ by (M ZZ -1 Z), and ZZZ by (M ZZZ -1 -1 Z) or (M ZZZ -1 ZZ). We could define a more general BP by (M (BP) (N) (BK N) UP P). Thus, (BP 3) would perform (BK 3), followed by an UP, followed by a P. The command SECOND can be defined as a macro by

```
(M (2ND) X (ORR ((LC . X) (LC . X)))).
```

Note that for all editor commands, 'built in' commands as well as commands defined by macros, atomic definitions and list definitions are completely independent. In other words, the existence of an atomic definition for c in no way affects the treatment of c when it appears as CAR of a list command, and the existence of a list definition for c in no way affects the treatment of c when it appears as an atom. in particular, c can be used as the name of either an atomic command, or a list command, or both. In the latter case, two entirely different definitions can be used. Note also that once c is defined as an atomic command via a macro definition, it will not be searched for when used in a location specification, unless c is preceded by an F. Thus (INSERT -- BEFORE BP) would not search for BP, but instead perform a BK, an UP, and a P, and then do the insertion. The corresponding also holds true for list

commands.

5.7.2 (BIND . coms) [EDIT-COMMAND]

BIND is an edit command which is useful mainly in macros. It binds three dummy variables #1, #2, #3, (initialized to NIL), and then executes the edit commands COMS. Note that these bindings are only in effect while the commands are being executed, and that BIND can be used recursively; it will rebind #1, #2, and #3 each time it is invoked. (BIND is implemented by (PROG (#1 #2 #3) (EDITCOMS (CDR COM))) where COM corresponds to the BIND command, and EDITCOMS is an internal editor function which executes a list of commands.)

SW could be defined as

```
(M (SW) (N M) (NTH N) (S FOO 1) MARK 0 (NTH M) (S FIE 1) (I
1 FOO) __ (I 1 FIE))
```

(A more elegant definition would be

```
(M (SW) (N M) (NTH N)  MARK 0 (NTH M) (S FIE 1) (I 1 (## _ 1)) __ (I 1 FIE))
```

but this would still use one free variable.) Since SW sets FOO and FIE, using SW may have undesirable side effects, especially when the editor was called from deep in a computation. Thus we must always be careful to make up unique names for dummy variables used in edit macros, which is bothersome. Furthermore, it would be impossible to define a command that called itself recursively while setting free variables. The BIND command solves both problems. Thus we could now write SW safely as

```
(M (SW) (N M) (BIND (NTH N) (S #1 1) MARK 0 (NTH M) (S #2 1)
(I 1 #1) __ (I 1 #2))).
```

5.7.3 USERMACROS [VALUE]

This variable contains the users editing macros . Thus if you want to save your macros then you should save USERMACROS. You should probably also save EDITCOMSL.

5.7.4 EDITCOMSL [VALUE]

EDITCOMSL is the list of "list commands" recognized by the editor. (These are the ones of the form (command arg1 arg2 ...).)

5.8 EDIT-MISC

5.8.1 OK [EDIT-COMMAND]

exits from the editor.

5.8.2 SAVE [EDIT-COMMAND]

Exits from the editor and saves the 'state of the edit' on the property list of the function/variable being edited under the property EDIT-SAVE. If the editor is called again on the same structure, the editing is effectively "continued," i.e., the edit chain, mark list, value of UNFIND and UNDOLST are restored.

```
#P
(NULL X)
#F COND P
(COND (& &) (T &))
#SAVE
FOO


 . . .

*(EDITF FOO)
EDIT
#P
(COND (& &) (T &))
#\ P
(NULL X)
```

SAVE is necessary only if the user is editing many different expressions; an exit from the editor via OK always saves the state of the edit of that call to the editor. (On the property list of the atom EDIT, under the property name LASTVALUE. OK also remprops EDIT-SAVE from the property list of the function/variable being edited.) Whenever the editor is entered, it checks to see if it is editing the same expression as the last one edited. In this case, it restores the mark list, the undolst, and sets UNFIND to be the edit chain as of the previous exit from the editor. The user can always continue editing, including undoing changes from a previous editing session, if (1) No other expressions have been edited since that session; (since saving takes place at exit time, intervening calls that were exited via STOP will not affect the editor's memory of this last session.) or (2) It was ended with a SAVE command.

5.8.3 NIL [EDIT-COMMAND]

Unless preceded by F or BF, is always a NOP.

5.8.4 TTY: [EDIT-COMMAND]

Calls the editor recursively. The user can then type in commands, and have them executed.

The TTY: command is completed when the user exits from the lower editor (with OK or STOP). The TTY: command is extremely useful. It enables the user to set up a complex operation, and perform interactive attention-changing commands part way through it. For example the command (MOVE 3 TO AFTER COND 3 P TTY:) allows the user to interact, in effect, within the MOVE command. Thus he can verify for himself that the correct location has been found, or complete the specification "by hand". In effect, TTY: says "I'll tell you what you should do when you get there."

The TTY: command operates by printing TTY: and then calling the editor. The initial edit chain in the lower editor is the one that existed in the higher editor at the time the TTY: command was entered. Until the user exits from the lower editor, any attention changing commands he executes only affect the lower editor's edit chain. (Of course, if the user performs any structure modification commands while under a TTY: command, these will modify the structure in both editors, since it is the same structure.) When the TTY: command finishes, the lower editor's edit chain becomes the edit chain of the higher editor.

5.8.5 STOP [EDIT-COMMAND]

Exits from the editor with an error. Mainly for use in conjunction with TTY: commands that the user wants to abort. Since all of the commands in the editor are ERRSET protected, the user must exit from the editor via a command. STOP provides a way of distinguishing between a successful and unsuccessful (from the user's standpoint) editing session. For example, if the user is executing (MOVE 3 TO AFTER COND TTY:), and he exits from the lower editor with an OK, the MOVE command will then complete its operation. If the user wants to abort the MOVE command, he must make the TTY: command generate an error. He does this by exiting from the lower editor with a STOP command. In this case, the higher editor's edit chain will not be changed by the TTY: command.

5.8.6 HELP [EDIT-COMMAND]

HELP is both an atomic and list-type edit command. Both forms call the help function. The atomic form takes the entire remainder of the command line as the arguments to help, the list form takes only the tail of its list. Thus to obtain help on DATE and move back one s-expression with a single line of edit-commands, do: (help date) bk

5.8.7 TL [EDIT-COMMAND]

TL calls (TOP-LEVEL). To return to the editor just use the RETURN top-level command.

5.8.8 REPACK [EDIT-COMMAND]

Permits the 'editing' of an atom or string. For example:

```
#P
... "THIS IS A LOGN STRING")
#REPACK
EDIT
1#P
(/" T H I S / I S / A / L O G N / S T R I N G /")
1#(SW G N)
1#OK
"THIS IS A LONG STRING"
```

REPACK operates by calling the editor recursively on UNPACK of the current expression, or if it is a list, on UNPACK of its first element. If the lower editor is exited successfully, i.e. via OK as opposed to STOP, the list of atoms is made into a single atom or string, which replaces the atom or string being 'repacked.' The new atom or string is always printed.

(REPACK $)

Does (LC . $) followed by REPACK, e.g. (REPACK THIS@).

5.8.9 (MAKEFN form args n m) [EDIT-COMMAND]

Makes (CAR form) an EXPR with the nth through mth elements of the current expression with each occurance of an element of (CDR form) replaced by the corresponding element of args. The nth through mth elements are replaced by form. For example:

```
#P
... (SETQ A NIL) (SETQ B T) (CONS C D))
#(MAKEFN (SETUP C D) (W X) 1 3) P
... (SETUP C D))
#E (GRINDEF SETUP)
(DEFPROP SETUP
(LAMBDA(W X) (SETQ A NIL) (SETQ B T) (CONS W X))
EXPR)
#
```

(MAKEFN form args n)

Same as (MAKEFN form args n n).

5.8.10 EDITDEFAULT

Whenever a command is not recognized, i.e., is not 'built in' or defined as a macro, the editor calls an internal function, EDITDEFAULT to determine what action to take. If a location specification is being executed, an internal flag informs EDITDEFAULT to treat the command as though it had been preceded by an F. If the command is atomic and typed in directly, the procedure followed is as given below. 1) If the command is one of the list commands, i.e., a

member of EDITCOMSL, and there is additional input on the same teletype line, treat the entire line as a single list command. (Uses LINEREAD. Thus the line can be terminated by carriage return, right parenthesis or square bracket, or a list.) Thus, the user may omit parentheses for any list command typed in at the top level (which is not also an atomic command, e.g., NX, BK). If the command is on the list EDITCOMSL but no additional input is on the teletype line, an error is generated. 2) If the last character in the command is P, and the first n-1 characters comprise the command ___ _ UP, NX, BK, !NX, UNDO, or REDO, assume that the user intended two commands. 3) Otherwise, generate an error.

5.8.11 (EDITCOMS coms) [SUBR]

is an internal editor function which executes a list of edit commands.

5.8.12 (EDITRACEFN com) [VALUE and EXPR]

Is available to help the user debug complex edit macros, or subroutine calls to the editor. EDITRACEFN is to be defined by the user. Whenever the value of EDITRACEFN is non-NIL, the editor calls the function EDITRACEFN before executing each command (at any level), giving it that command as its argument. EDITRACEFN is initially equal to NIL, and undefined.

For example, defining EDITRACEFN as

```
(LAMBDA (C) (PRINT C) (PRINT (CAR L)))
```

will print each command and the corresponding current expression.

```
(LAMBDA (C) (BREAK1 T T NIL NIL NIL))
```

will cause a break before executing each command.

5.8.13 (S var . $) [EDIT-COMMAND]

Sets var (using SETQ) to the current expression after performing (LC . $). Edit chain is not changed. Thus (S FOO) will set FOO to the current expression, (S FOO -1 1) will set FOO to the first element in the last element of the current expression.

5.9 EDIT-FNS

5.9.1 (EDITF x) [FSUBR]

FSUBR function for editing a function. (CAR x) is the name of the function, (CDR x) an optional list of commands.

For the rest of the discussion, fn is (CAR x), and coms is (CDR x). If x is NIL, fn is set to the value of LASTWORD, coms is set to NIL, and the value of LASTWORD is printed. The value of EDITF is fn. (1) In the most common case, fn is an non-compiled function, and EDITF performs (EDITE (CADR (GETL fn (QUOTE (FEXPR EXPR MACRO)))) coms fn) and sets LASTWORD to fn. If the editor detects that the function has been changed by the edit, EDITF moves the definition to the front of the property list, insuring that the interpreted definition will be used in preference to a compiled definition. (2) If fn is not an editable function, but has a value, EDITF assumes the user meant to call EDITV, prints =EDITV, calls EDITV and returns. Otherwise, EDITF generates an fn NOT EDITABLE error.

5.9.2 (EDITE expr coms atm) [SUBR]

Edits an expression. Its value is the last element of (EDITL (LIST expr) coms atm NIL NIL). Generates an error if expr is not a list.

5.9.3 (EDITV editvx) [FSUBR]

FSUBR function, similar to EDITF, for editing values. (CAR editvx) specifies the value, (CDR editvx) is an optional list of commands. If editvx is NIL, it is set to the value of (NCONS LASTWORD) and the value of LASTWORD is printed. If (CAR editvx) is a list, it is evaluated and its value given to EDITE, e.g. (EDITV (CDR (ASSOC (QUOTE FOO) DICTIONARY))). In this case, the value of EDITV is T. However, in most cases, (CAR editvx) is a variable, e.g. (EDITV FOO); and EDITV calls EDITE on the value of the variable. If the value of (CAR editvx) is atomic then EDITV prints a NOT EDITABLE error message. When (if) EDITE returns, EDITV sets the variable to the value returned, and sets LASTWORD to the name of the variable. The value of EDITV is the name of the variable whose value was edited.

5.9.4 (EDITP x) [FSUBR]

Similar to EDITF for editing property lists. Like EDITF, LASTWORD is used if x is NIL. EDITP calls EDITE on the property list of (CAR x). When (if) EDITE returns, EDITP RPLACD's (CAR x) with the value returned, and sets LASTWORD to (CAR x). The value of EDITP is the atom whose property list was edited.

5.9.5 (EDITL L coms atm marklst mess) [SUBR]

EDITL is the editor.

Its first argument is the edit chain, and its value is an edit chain, namely the value of L at

the time EDITL is exited. (L is a special variable, and so can be examined or set by edit commands. For example, ^ is equivalent to (E (SETQ L(LAST L)) T).) Coms is an optional list of commands. For interactive editing, coms is NIL. In this case, EDITL types EDIT and then waits for input from the teletype. (If mess is not NIL EDITL types it instead of EDIT. For example, the TTY: command is essentially (SETQ L (EDITL L NIL NIL NIL (QUOTE TTY:))).) Exit occurs only via an OK, STOP, or SAVE command. If coms is NOT NIL, no message is typed, and each member of coms is treated as a command and executed. If an error occurs in the execution of one of the commands, no error message is printed , the rest of the commands are ignored, and EDITL exits with an error, i.e., the effect is the same as though a STOP command had been executed. If all commands execute successfully, EDITL returns the current value of L. Marklst is the list of marks. On calls from EDITF, Atm is the name of the function being edited; on calls from EDITV, the name of the variable, and calls from EDITP, the atom of which some property of its property list is being edited. The property list of atm is used by the SAVE command for saving the state of the edit. Thus SAVE will not save anything if atm=NIL i.e., when editing arbitrary expressions via EDITE or EDITL directly.

5.9.6 (EDITFNS x) [FSUBR]

FSUBR function, used to perform the same editing operations on several functions. (CAR x) is evaluated to obtain a list of functions. (CDR x) is a list of edit commands. EDITFNS maps down the list of functions, prints the name of each function, and calls the editor (via EDITF) on that function.

For example, (EDITFNS FOOFNS (R FIE FUM)) will change every FIE to FUM in each of the functions on FOOFNS.

The call to the editor is ERRSET protected, so that if the editing of one function causes an error, EDITFNS will proceed to the next function. Thus in the above example, if one of the functions did not contain a FIE, the R command would cause an error, but editing would continue with the next function. The value of EDITFNS is NIL.

5.9.7 (EDIT4E pat y) [SUBR]

Is the pattern match routine. Its value is T if pat matches y. See EDIT-MATCH For definition of 'match'.

Note: before each search operation in the editor begins, the entire pattern is scanned for atoms or strings that end in at-signs. These are replaced by patterns of the form (CONS (QUOTE /@) (EXPLODEC atom)). Thus from the standpoint of EDIT4E, pattern type 5, atoms or strings ending in at-signs, is really "If car[pat] is the atom @ (at-sign), PAT will match with

any literal atom or string whose initial character codes (up to the @) are the same as those in cdr[pat]."

If the user wishes to call EDIT4E directly, he must therefore convert any patterns which contain atoms or strings ending in at-signs to the form recgnized by EDIT4E. This can be done via the function EDITFPAT.

5.9.8 (EDITFPAT pat flg) [SUBR]

Makes a copy of pat with all patterns of type 5 (see EDIT-MATCH) converted to the form expected by EDIT4E. Flg should be passed as NIL (flg=T is for internal use by the editor).

5.9.9 (EDITFINDP x pat flg) [SUBR]

Allows a program to use the edit find command as a pure predicate from outside the editor. X is an expression, pat a pattern. The value of EDITFINDP is T if the command F pat would succeed, NIL otherwise. EDITFINDP calls EDITFPAT to convert pat to the form expected by EDIT4E, unless flg=T. Thus, if the program is applying EDITFINDP to several different expressions using the same pattern, it will be more efficient to call EDITFPAT once, and then call EDITFINDP with the converted pattern and flg=T.

6. SYSTEM-STUFF

SYSTEM-STUFF includes technical details, special hacks, and other things the typical user will have no use for or may not understand without prior arcane or non-LISP knowledge.

6.1 SYMBOL-TABLE

A large number of symbols defined in the MACRO-10 source for the basic LISP system are retained on disk to be loaded into core by the loader interface (see LOAD). When loaded into core, the symbol table serves three agencies: DDT, the loader, and the LISP symbol-table communication functions. DDT and the loader were designed to use this type of symbol table. They have not been modified at all in this respect. DDT uses the table to communicate with DDT users through symbolic names rather than addresses. Many .REL files must refer to the MACRO-10 symbols defined for LISP. These files cannot be properly loaded without the symbol table, which is the loader's only way to load .REL files that refer to things not defined within the file itself.

The LISP symbol table communication functions allow the LISP user to define arbitrary symbols for the loader to use. They also allow him to reference symbols created by the loader while loading .REL files. They provide the only way for a LISP user to make use of subroutines loaded by the loader.

6.1.1 (*GETSYM S) [SUBR]

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

6.1.2 (GETSYM "P" "S1" "S2" ... "Sn") [FSUBR]

GETSYM searches the DDT symbol table for each of the symbols Si and places the value on the property list of Si under property P. For example, (GETSYM SUBR DDT) causes DDT to be defined as a SUBR located at the value of the symbol DDT.

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

For related information see LOAD.

6.1.3 (*PUTSYM S V) [SUBR]

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

6.1.4 (PUTSYM "X1" "X2" ..."Xn") [FSUBR]

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

For related information see LOAD.

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

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

```
        MOVE X, VBPORG
(PUTSYM (MAPLST (QUOTE MAPLST)) (NUMBRP (QUOTE NUMBERP)))
(PUTSYM (MEMQ (GET(QUOTE MEMQ) (QUOTE SUBR))))
```

A RELOC program would call these functions as follows:

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

6.1.5 (*RGETSYM X) [SUBR]

gets the value of the symbol X, adds on the relocation and returns the cell pointed to as the value.

6.1.6 (RGETSYM P S1 S2 ...) [FSUBR]

searches the symbol table for Si and places the relocated value on the property list of Si with property P.

6.1.7 (*RPUTSYM SYM VAL) [SUBR]

puts VAL - relocation in the symbol table under SYM.

6.1.8 (RPUTSYM X1 X2 ...) [FSUBR]

(similar to GETSYM) IF Xi is an atom the Xi is placed in the symbol table with Xi less the relocation as its value. Otherwise (EVAL (CADR Xi)) is placed in the symbol table as the value of (CAR Xi).

6.2 LOAD

THE LOADER

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

(LOAD X) [SUBR]

When a * is typed, you are in the (LOAD X) loader, and the loader command strings are expected. As soon as an altmode is typed, the loader finishes and exits back to LISP. The loader is placed in expanded core. If X = NIL then loaded programs are placed in expanded core, otherwise (if X non-NIL) they are placed in BINARY PROGRAM SPACE. The loader removes itself and contracts core when it is finished. In other explanations a "RELOC" program will refer to any program which is suitable for loading with the loader. The output of FORTRAN, MACRO or FAIL is a RELOC program. The loader is very primitive. For example, it only understands UPPER CASE, and it doesn't understand C-MU PPNs.

Suppose you have copied CNTLSP.REL from A311LI5P (the PTY controller).

```
>(LOAD T)
*CNTLSP.REL/S            (The S means include the symbol table)

*<esc>

LOADER 1 K CORE          (or some such msg)
>(GETSYM SUBR PTYGO REPTY)
 ...                     (tells you where the subrs are)
>(PTYGO)
 ...
```

The LISP loader interface appears to have the following problems:

>In case of insufficient binary program space, symbols are defined although the code is not actually loaded.

>There is no notification as to whether BPS was sufficient or not, so BPORG must be checked before and after loading when using BPS.

>Allocating more core through LISP destroys the existing symbol table.

-CP01 2/4/77

6.3 DDT

DDT is a DEC supplied debugging package. It may be used in LISP by:

```
(LOAD)          ;; or (LOAD T)
/D<esc>         ;; D must be upper case, <esc> is the escape char.
(GETSYM SUBR DDT)
```

Then DDT will be defined as a SUBR that calls DDT. To return to LISP just type "POPJ 14,<esc>X".

6.4 STORAGE-ALLOCATION

Lisp partitions memory into seven areas which can independently vary in size. These areas and their uses are:

```
BINARY PROGRAM SPACE(BPS)  ;;for compiled functions and arrays
FREE STORAGE       ;;for LISP nodes (cons cells)
FULL WORD SPACE    ;;for print names and numbers
BIT TABLES ;;for the garbage collector
REGULAR PUSH-DOWN LIST(RPDL)       ;;for all function calls and
     ;;non-special variables in compiled functions
SPECIAL PUSH-DOWN LIST(SPDL)       ;;for interpreted variables and
     ;;special variables
EXPANDED CORE      ;;for I/O buffers, LOADER, and any loaded programs
```

6.4.1 BPS

Binary Program Space BPS is used for compiled code and arrays. Note: re-declaring arrays does not reclaim the old BPS. BPS is not garbage collected. Also the garbage collector does not collect structures pointed to by pointers in BPS (other than arrays of lists).

6.4.1.1 BPEND [VALUE]

BPEND contains the address of the end of Binary Program Space.

6.4.1.2 BPORG [VALUE]

BPORG contains the adress of the beginning of unused Binary Program Space.

6.4.2 FREE-STG

Free Storage is the area of memory in which cons-cells (the result of doing a CONS) are stored. Each cons-cell contains two pointers corresponding to the CAR and CDR. The cells that are known not to be used (pointed to by anything the user can name) are stored as a list called the FREE-LIST. The CONS function takes the first cell off of the free-list, fills it with the required pointers and returns it as the value of the CONS. When the free-list runs out,

the garbage collector is run to add to it any cells that are no longer reachable.

The free list is stored in register 13 (15 octal). Thus one can get the length of the free list by (length (numval 13)).

6.4.3 FULL-WORD-SPACE

Full word space is the area of memory used for the storage of character strings and numbers (except for INUMS). It is used and recreated (by the garbage collector) in the same way as free storage (see FREE-STG). The list of full words is kept in register 14 (16 octal).

6.4.4 RPDL

The Regular PushDown List is a stack that lisp uses for saving temporary values. It is also used in the garbage collector. The stack is implemented by PDP10 stack instructions. The stack pointer is kept in register 11 (13 octal). The right half contains the pointer to the top of the stack and the left half contains minus the number of words still available.

6.4.5 GARBAGE-COLLECTION

The garbage collector analyzes the state of list structures pointed to by the OBLIST, the REG. PDL, the SPEC. PDL, list arrays, and a few other special cells. By recursively marking all words on free and full word spaces which are pointed to in this manner, it is possible to determine which words are not pointed to and are therefore garbage. Such words are collected together on their respective free storage lists.

6.4.5.1 (GC) [SUBR]

GC causes a garbage collection to occur and returns NIL. Normally, a garbage collection occurs only when either free or full word space has been exhausted. It is possible to determine the length of the free storage list by:

```
(LENGTH (NUMVAL 13))     = length of free storage list
```

this feature may disappear or reappear often in the near future The system requested garbage collections (as well as user calls on GC) will be affected if the user redefines GC. Thus you can cause your own program to be called after each garbage collection by redefining GC to first call the system supplied GC and then your own program.

6.4.5.2 (GCGAG X) [SUBR]

flag = T initially. GCGAG sets a special flag in the interpreter to the value of X, and

returns the previous setting of the flag. When any garbage collection occurs, if the flag .ne. NIL, then the following is printed:

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

where x and y are numbers.

6.4.5.3 (GCGOT) [SUBR]

returns a dotted pair, the CAR of which is the number of free words and the CDR of which is the number of full words made available by the last garbage collection.

6.4.5.4 FREE

******WARNING******: The following two functions can catastrophically destroy the garbage collector by creating a circle in the free list if they are used to return to the free list any words which are still in use. Do not use these functions unless you are certain what you are doing. (They are only useful in rare cases where a small amount of working storage is needed by a routine which is called quite often.)

(FREE X) [SUBR]

FREE returns the word X to the free storage list and returns NIL.

(FREELIST X) [SUBR]

FREELIST returns all of the words on the top level of the list X to the free storage list and returns NIL. FREELIST terminates on a NULL check.

6.4.5.5 (GCMIN n1 n2) [SUBR]

where n1 and n2 are numbers resets the lower bounds for storage to be reclaimed by the garbage collector for free and full-word space respectively, and returns (as a dotted pair) the old values. If a garbage collection fails to find the minimum space then garbage-collection messages are turned on.

6.4.6 (REALLOC fws bps rpdl spdl fs) [SUBR]

REALLOC's arguments specify increments (in words) to be added to each of the five major allocation areas: fullword space, binary program space, the regular pushdown list, the special pushdown list, and free storage (i.e., list space). After expanding core as necessary and reallocating storage, REALLOC returns control directly to the top level of LISP. As with CORE there is no way to save the state of the computation through a reallocation of space

6.4.7 (EXPFWS n) [SUBR]

is the same as (REALLOC n 0 0 0 0)

6.4.8 (EXPBPS n) [SUBR]

is the same as (REALLOC 0 n 0 0 0)

6.4.9 (EXPFS n) [SUBR]

is the same as (REALLOC 0 0 0 0 n)

6.4.10 (EXPRPDL n) [SUBR]

is the same as (REALLOC 0 0 n 0 0)

6.4.11 (EXPSPDL n) [SUBR]

is the same as (REALLOC 0 0 0 n 0)

6.4.12 (CORE N) [SUBR]

Note - The CORE function is still reasonable for finding out how much core you are using, but the allocation functior. is better handled by the REALLOC function and its relatives.

If N is in the range [current size of low segment in words , 192K], CORE expands the low segment to that size and goes into the initialization procedure which will ask the user how he wants any newly available core allocated. If N is the current size of the low segment, the initialization procedure may still ask the user how he wants additional core allocated even though no additional core was obtained from the operating system. In this case, it is allocating space from what was expanded core, probably formerly used for I/O buffers for files now closed. (To allocate whatever space is available, evaluate (CORE (CORE NIL)).) When the initialization procedure is invoked, control is returned to the LISP top level. There is at present no way of allocating additional space and continuing where you left off. CORE closes all I/O channels when new core is allocated. If N is not numeric, or is not in the range indicated above, CORE returns the current size of the low segment.

The allocation procedure begins when LISP types "ALLOC?". You then type either Y or N (not followed by a carriage return). If you type Y then LISP will ask how many additional machine words should be allocated to each area by typing things like:

FULL WORDS=

There are three responses to each of these questions: 1) A space will cause the default value to be used. 2) An *OCTAL* number ended by a space causes the number to be used. 3) A carriage-return will cause the default value to be used for this and all following questions (which will not be asked). Note: if you use the reallocation procedure after having expanded core for any purpose, it will reallocate this additional core for its own purposes, thus destroying the contents of expanded core.

For related information see EXCISE.

6.5 COMPILED-CODE

To use the LISP compiler type "R LISPCO" to the monitor. In much the same way as you might normally type "(DSKIN file1 file2 ...)" you can type (COMPL file1 file2 ...). The compiler will read these files and produce .LAP files containing LAP code for the functions defined in those files. These may be read into LISP with DSKIN, but the compiled functions will be put in BPS.

The compiler is just another lisp program which has been compiled and loaded into the high segment to create another core image, called LISPCO. Therefore you can run LISPCO instead of LISP if you occasionally want to compile things. The LISPCO high segment is larger than the normal LISP high segment (since it contains the compiler), but LISPCO has been hacked to use the standard LISP high segment whenever it can. When you compile something the LISPCO high segment will be temporarily retrieved. Actually, the compiler requires some data that is not used by the normal LISP system, so the low segment is also larger than the standard one.

6.5.1 (DECLARE decl1 decl2 ...) [FSUBR]

allows declarations to be made to the compiler. Declarations are ignored by the interpreter. In the compiler each argument of DECLARE is evaluated.

Typical uses are

```
(DECLARE (SPECIAL X Y Z))
```

(followed by code which uses x, y and z as special variables followed by) (DECLARE (UNSPECIAL X Y Z)),

and `(DECLARE (*FSUBR MUMBLE))` before the first call to mumble

in a file before the FEXPR mumble (to be compiled) is defined.

6.5.1.1 (SPECIAL <var1> {<var2>} . . .) [DECLARATION]

Declares each <var> as a special variable, i.e., a variable which appears free in a function. Note that free variables in in-line LAMBDA expressions and LAMBDA expressions used as arguments to most system functions (e.g., the MAP functions) need not be declared SPECIAL, as such functions are compiled in-line. In addition ERRSETs are now compiled in-line, so variables in ERRSET expressions no longer have to be declared SPECIAL. All undeclared free variables in a file may be found by compiling the file and examining the error messages; for convenience, the compiler places all newly-discovered special variables on the list SPECIALS.

For related information see LAPLST and SPECBIND.

Special Variables In compiled functions, any variable which is bound in a LAMBDA or PROG and has a free occurrence elsewhere must be declared SPECIAL. Example:

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

The variable A which has a free occurrence must be declared SPECIAL if the outer LAMBDA expression is to be compiled. All variables in interpreted functions, and SPECIAL variables in compiled functions store their values in SPECIAL (or VALUE) cells. These variables are bound at the entry to a LAMBDA or PROG by saving their previous values on the SPECIAL pushdown list and storing their new values in the SPECIAL cells. All references to these variables are directly to their SPECIAL cells. When the LAMBDA or PROG is exited, the old values are restored from the SPECIAL pushdown list. In compiled functions, all variables not declared SPECIAL are stored on the REGULAR pushdown list, and the SPECIAL cells (if they exist) are not referenced.

6.5.1.2 (UNSPECIAL <var1> {<var2>} . . .) [DECLARATION]

This declaration may be used to inform the compiler that certain variables are no longer considered special, and should be compiled as normal local variables in subsequent functions.

6.5.1.3 (NOCALL <a1> {<a2>} . . .) [DECLARATION]

Each <a> should be either the name of a function to be compiled or a special variable. These functions and variables are assumed to be local to the file being compiled and will thus never be traced, called or referenced from functions not in this file, or used as entry points or top-level values. The compiler can compile references TO such functions as direct jumps, and the atoms may be REMOBed when the file is loaded (see DUMPATOMS).

NOCALL [VALUE]

If NOCALL is T when a function is being LOADED (read in from a LAP file) then all of the

function calls CONTAINED IN the code (except the calls to functions which were declared CALL during compilation) will be converted to direct jumps.

Removing Excess Entry Points - NOCALL Feature If, during compilation, a function has a non-NIL NOCALL property, all calls to that function are compiled as direct PUSHJ's to the entry point of that function with no reference to the atom itself. After loading, all functions used in this manner will be left as a list on the variable REMOB. This means that many functions which are not major entry points can often times be REMOBed to save storage. The user may use (NOCALL FOO1 FOO2 ... FOOn) [FSUBR] to make several NOCALL declarations. Like SPECIAL and DECLARE, when NOCALL is used outside of the compiler, it acts the same as NILL.

Warning: If a function, F, is compiled without the NOCALL property and a function, G, which calls F is compiled while F does have the NOCALL property, then the code for G will not be able to resolve its reference to F because of the lack of a SYM property on F (in spite of the fact that the SUBR property could be translated into the answer).

The NOCALL property does not affect the ability to call the function in the usual ways, but it does allow the atom (F) to be REMOBed after which it would still be accessible to the functions that were compiled (G) when its (F's) NOCALL property was non-NIL (whereas other functions (and the user) will no longer be able to call it). The safe and reasonable thing to do is to keep the NOCALL properties of all of the compiled functions the same throughout the compilation of all of the functions.

6.5.1.4 (CALL <fn1> {<fn2>} . . .) [DECLARATION]

Specifies that each <fn> should always be called via the function-calling mechanism and not changed to direct jumps. Necessary in rare cases when the NOCALL=T feature is being used. For example, any function which is undefined at compile time must be declared NOCALL unless it is to be loaded from another LAP file.

FUNCTION CALLING UUOs To allow ease in linking, debugging, and modifying of compiled functions, all compiled functions call other functions with special opcodes called UUOs. Several categories of function calls are distinguished: 1) Calls of the form (RETURN (FOO X)) are called terminal calls and essentially "jump" to FOO. 2) Calls of the form (F X) where F is a computed function name or functional argument is called a functional call. The function calling UUOS are:

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

where f is either the address of a compiled function or a pointer to the identifier for the

function, and n specifies the type of function being called as follows:

```
n = 0 to 5          specifies a SUBR call with n arguments
n = 16              specifies a LSUBR call
n = 17              specifies a FSUBR call.
```

The function calling UUOs are defined in MACRO by:

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

6.5.1.5 (NOCOMPILE exp) [DECLARATION]

Causes the compiler not to compile exp but to just transfer it to the output file. In interpretive mode exp is evaluated.

6.5.1.6 (GLOBALMACRO <mac1> {<mac2>} . . .) [DECLARATION]

Macro definitions are normally assumed to be used only by functions in the file in which they appear, and hence are not necessary after the file is compiled. Occassionally, however, it is desirable to keep the macro definitions after compilation by having them copied into the LAP file (PLUS is such a macro for example). The GLOBALMACRO declaration specifies that each <mac> is such a global macro and should be saved.

6.5.1.7 (*SUBR <fn1> {<fn2>} . . .) [DECLARATION]

(*FSUBR <fn1> {<fn2>} . . .) [DECLARATION] (*LSUBR <fn1> {<fn2>} . . .) [DECLARATION] FSUBRs and LSUBRs which are referenced before they are compiled must be declared (via *FSUBR and *LSUBR) so that the compiler can compile function references correctly. *SUBR declarations may also be made, although they are not necessary since all undefined functions are assumed to be SUBRs. *EXPR, *FEXPR, and *LEXPR may be used in place of *SUBR, *FSUBR, and *LSUBR if desired.

6.5.2 (COMPL file1 file2 ...) [FSUBR]

is only in the compiler core image which is run by typing "R LISPCO" to the monitor. The compiler now prints the name of each function before its compilation has begun. If an error occurs, the last name printed is the function in error. Note also that the value returned by LAP (and thus printed by DSKIN) is now a list consisting of the name of the function loaded followed by the number of words of binary program space required for the compiled code. Recall that strings are normally not interned by the READ routine so that they will be garbage collected when no longer referenced. Strings appearing in compiled code will always be referenced, however, so LAP has been modified to intern them (by setting INTERNSTR to

T). This has the advantage that functions which are compiled may reference the same string a number of times without penalty - only one copy will be stored.

6.5.3 (COMPLFNS LIST) [SUBR]

Available only in the LISPCO core image. LIST is to be a list of atoms, each of which is compiled and loaded into core by COMPLFNS. A scratch file, $TEMP$.LAP, is produced and deleted during this process. Should the compilation abort, this file can be deleted through the LISP DELETE function. It is not possible to give declarations in the function list, but they may be made in advance.

6.5.4 SYM

SYM : Symbol definitions for LAP are stored on the SYM property. These include opcodes, registers and NOCALL function locations.

6.5.5 VALUE

VALUE is the name of the property under which values of atoms (variables) are stored. IMPORTANT: It is a bad idea to change the value of an atom by simply replacing its VALUE property. In order to make compiled code more efficient, the value property of an atom is assumed to always point to the same list cell (so the address can be compiled into the code). The cdr of this cell points to the value. Thus if you must change the value property it should be done by RPLACD of the VALUE property.

6.5.6 SUBR

A SUBR is the compiled form of an EXPR

SUBR LINKAGE

SUBRs are compiled EXPRs which are the most common type of function. Consequently, considerable effort has been made to make linkage to SUBRs efficient. Arguments to SUBRs are supplied in accumulators 1 through n, the first argument in 1. There is a maximum of 5 arguments to SUBRs. To call a SUBR from compiled code, use call n,FUNC, where n is the number of arguments, and call is the appropriate UUO. (See CALL) The result from a SUBR is returned in A (=1).

6.5.7 FSUBR

An FSUBR is the compiled form of an FEXPR

FSUBR LINKAGE

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

```
PUSHJ P, *AMAKE
```

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

For related information see CALL and FEXPR.

6.5.8 LSUBR

LSUBR - the compiled form of an LEXPR

LSUBR LINKAGE

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

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

When a LSUBR is entered, it executes:

```
        JSP 3,*LCALL
```

which initializes the LSUBR. A will contain n. The ith argument can be referenced by: MOVE A, -i-1(P) Exit from an LSUBR with POPJ P, which returns to *LCALL to restore the stack.

6.5.9 COMPILE-HINTS

--- If you use the compiler you should be aware of the following --- - Subrs may have no more than 5 arguments. - Macros are expanded at compile time (the expansion is compiled). - Certain declarations are needed - see the help for DECLARE. - See help for NOUUO, NOCALL, DUMPATOMS

Note that when loading LAP files with NOCALL=T all functions are assumed to be either already defined when the files are loaded (e.g., system functions), or defined in the file. If any existing compiled functions (such as system functions) are to be redefined, they must either be defined before they are referenced or must have their SUBR, FSUBR, or LSUBR properties removed before loading. A warning will be printed if this is not done.

6.5.10 COMPILE-ERRORS

Explanations of LISPCO error messages (from Diffie at Stanford) User Errors These are errors in source code which cause the compiler to halt.

```
ARGNO-P1CONS              CONS has the wrong number of arguments.
ARGNOERR-BOOLEQ1          Wrong number of agruments to EQ.
ARGNOERR-COMPDEF          "DEFPROP" has the wrong number of arguments.
ARGNOERR-INTERNALLAMBDA   Differing numbers of variables and arguments.
ARGNOERR-P2CARCDR         Wrong number of arguments to CAR, CDR, CADR etc.
ATOMICVARLIST-P1BIND      An atom where a variable list was expected.
CONSTFUN-P1               Attempt to call a constant(number, T or NIL)
                          ;; as a function.
EXTRAARGS-P1SUBRARGS      EXPR or SUBR called with too many arguments.
                          ;; (More than the maximum for a SUBR)
EXTRAARGS-PASS1           Attempt to define a SUBR or EXPR with too many args.
NOTINPROG-P1GO            GO occurring outside of PROG.
NOTINPROG-P1RETURN        RETURN occurring outside of PROG.
NOTVARIABLE-P1BIND        A constant or non-atom in variable context.
NOTVARIABLE-P1SETQ        Attempt to SETQ a constant or non-atom.
PROGTOOSHORT-P1PROG       PROG must at least have a variable list.
READERR-FLUSHLAP          Read error while reading LAP in source file.
TOFEWARGS-P2PROG2
```

User Warnings These messages indicate that the compiler thinks there might have been an error. They do not interrupt compilation, but indicate conditions which can be expected to produce errors in object code.

```
REPEATED VARIABLE              Variable name repeated in a variable list.
MULTIPLY DEFINED TAG           Two PROG tags with same name.
UNUSED PROG VARIABLE           Some PROG variable not referenced.
UNDECLARED FREE VARIABLES      Variables found free in source code.
UNDEFINED TAG                  Undefined PROG tag.
```

Other Compiler Messages

```
FSUBR USED AS SUBR        A function previously called and presumed to be a SUBR
                          ;; has been defined to be an FSUBR.
LSUBR USED AS SUBR        A function previously called and presumed to
                          ;; be a SUBR has been defined to be an LSUBR.
MACRO USED AS SUBR        A function previously called and presumed to be
                          ;; a SUBR has been defined to be an MACRO.
(name . flag) USED AS SUBR      Same as above, except that the function has
                                ;; been defined by LAP in source file.
var LOCAL AND SPECIAL     A variable compiled as local in an earlier function
                          ;; is found free or declared special.  The compiler is
                          ;; worried that they might be the same variable.
```

Compiler Errors These are errors in the compiler itself. It halts and goes into a read eval print loop without unbinding the variables to facilitate debugging.

```
ATOM-NTHCDR  COUNTSDISAGREE-COMPFUNC  FUNNYVAR-BINDVARS
LDLSTLEFT-PASS2  LOSTVAR-ILOC1  NEGNUM-NTHCDR  NIL-RST
NOAC-RESTORE  NOTAC-GETSLOT  NOTLAMBDA-GENFUN  NOTONPDL-GETSLOT
NOTSLOT-GETSLOT  NULLLOC-MARKVAL  PDLSHORT-RESTORE  PDLTOOLONG-LSUBRCALL
SOMETHINGELSE-P2ELSE
```

6.5.11 COMPILE-IN-LINE

A number of system functions are compiled in-line by the compiler, either because they generate only a few words of code or because they are FEXPRs which evaluate one or more arguments (if calls to such functions were not compiled in-line, the uncompiled arguments would be passed to the interpreter, slowing down execution considerably). Functions currently compiled in-line include: ERRSET, CATCH, THROW, RPTQ, COND, AND, OR, SELECTQ, PROG1, PROG2, PROGN PROG, RETURN, GO, SETQ, MSG, TTYMSG, EVERY, SOME, NOTANY, NOTEVERY, All Map Functions, APPEND (as *APPENDs), NCONC (as *NCONCs), LIST (as CONSes), CAR, CDR, RPLACA, RPLACD, EQ, NEQ, NULL (and NOT), ZEROP, ARG, SETARG, STORE, EVAL (as *EVAL or a direct call if possible), APPLY (as *APPLY or a direct call if possible), and APPLY* (as a direct call if possible).

6.5.12 TAG

LAP code contains labels which consist of the letters T A G followed by digits. LAP remembers these labels and when it finishes loading a function it REMOBs them. (They are used to record information needed in the loading.) This means that if you load compiled code you should avoid naming your variables TAG16 (or any other tag), since they are likely to vanish out from under you.

6.5.13 LAP

THE LISP ASSEMBLER - LAP

LAP is a primitive assembler designed to load the output of the compiler. Normally, it is not necessary to use LAP for any other purpose. LAP is self loading. The format of a compiled function in LAP is: (LAP name type) [LAP is an FSUBR] <sequence of LAP instructions>

NIL

where name is the name of the function, and type is either SUBR, LSUBR, or FSUBR. A LAP instruction is either:

```
;;              1.  A label which is a non-NIL identifier.
;;              2.  A list of the form
;;                  (OPCODE AC ADDR INDEX)
;;                  a.  The index field is optional.
;;                  b.  The opcode is either a PDP-6/10 instruction
;;              which is defined to LAP and optionally suffixed
;;              by @ which designates indirect addressing, or
;;              a number which specifies a numerical opcode.
;;                  c.  The AC and INDEX fields should contain a number
;;              from 0 to 17, or P which designates register 14.
;;                  d.  The ADDR field may be a number, a label, or a
;;              list of one of the following forms:
;;                      (QUOTE S-expression)  to reference list structure.
;;              (SPECIAL x)             to reference the value of
;;                                        identifier x.
;;                      (E f)           to reference the function f.
;;                      (C OPCODE AC ADDR INDEX)
```

to reference a literal constant.

6.5.14 ACCUMULATORS

ACCUMULATOR USAGE TABLE

s means "sacred" to the interpreter p means "protected" during garbage collection

```
NIL   = 0       s,p     Header for the atom NIL.
A     = 1       p       Results  from  functions,  1st  arg  to
                                ;;functions
B     = 2       p       2nd arg
C     = 3       p       3rd arg
AR1   = 4       p       4th arg
AR2A  = 5       p       5th arg
T     = 6       p       used for LSUBR linkage
TT    = 7       p
T10   = 10      p       rarely used in the interpreter
S     = 11              rarely used in the interpreter
D     = 12
R     = 13
P     = 14      s,p     regular pushdown list pointer
F     = 15      s,p     free storage list pointer
FF    = 16      s,p     full word list pointer
SP    = 17      s,p     special pushdown list pointer.
```

6.5.15 (DEF-EV-PROP "I" V "P") [FSUBR]

DEF-EV-PROP is used by GETDEF to retrieve the names and properties (SUBR, etc.) of functions internal to the LISP system from the file SYS:REMOB.LSP. DEF-EV-PROP evaluates only its second argument.

For related information see NOCALL.

6.5.16 (GETSEGLISP) [SUBR]

gets the standard lisp high segment. This is used by the compiling functions when they

finish, so that the high segment of the lisp compiler need not be used when it is not needed.

6.5.17 (GETSEGLISPCO) [SUBR]

gets the high segment of the lisp compiler (LISPCO). This is used by the compiling functions (which need the LISPCO high segment).

6.6 (DEPOSIT N V) [SUBR]

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

6.7 (EXAMINE N) [SUBR]

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

6.8 SYSTEM-BUILD

Building your own (or, how this one was built): Contents of LISP source files are described in LISP.DIR[a311i5p]. Aside from compilation of .COM files as necessary, and creation of LISPS.REL by means of LISPS.CTL, the LISP system was created last by LSPSYS.CTL and BATCH. I hope that formula will still work, ignoring the use of the MOVE cusp.

6.8.1 (HGHCOR X) [SUBR]

(for creating your own system) If X=NIL the "read-only" flag is turned on (it is initially on) and HGHCOR returns T. Otherwise X is the amount of space needed for compiled code. The space is then allocated (expanding core if necessary), the "read-only" flag is turned off and HGHCOR returns T.

6.8.2 (HGHORG X) [SUBR]

(for creating your own system) If X=NIL the address of the first unused location is returned as the value of HGHORG. Otherwise the address of the first unused location is set to X and the old value of the high seg. origin is returned.

6.8.3 (HGHEND) [SUBR]

(for creating your own system) The value of HGHEND is the address of the last unused location in the high seg.

6.8.4 (UNBOUND) [SUBR]

UNBOUND returns the un-interned atom UNBOUND which the system places in the CDR of an atom's SPECIAL (VALUE) cell to indicate that the atom currently has no assigned value even though it has a SPECIAL (VALUE) cell on its property list.

6.8.5 (SYSCLR) [SUBR]

Re-initializes LISP to read the user's LISP.INI file when it returns to the top level, e.g. by a Control-G or a START, or a REENTER. SYSCLR also resets the garbage collection time indicator to 0 and the CONSes performed indicator to 0. It also performs an EXCISE.

6.8.6 (INITFL "FILELST") [FSUBR]

INITFL is an FSUBR that sets up the file list for the user's INIT file. FILELST may consist of more than one file. However, if there is more than one file in the list, the files following the first one must be found or an error will be generated. The first file in the list is optional. The INIT file is initially LISP.INI. INITFL returns the old file list as its result.

```
*(INITFL (INIT1 . LSP) (MYFILE . LSP) FOO)
((LISP . INI))
```

6.8.7 (GTBLK LENGTH GC) [SUBR]

returns a zeroed block of LENGTH words. If GC is NIL the contents of the BLOCK are ignored by the garbage collector. Otherwise the contents are treated as pointers and the cells pointed to will not be collected.

6.8.8 (BLKLST LIST LENGTH) [SUBR]

returns a pointer type BLOCK of length words. It chains the words in the block so that the CDR of each word is the succeeding word. The top level of LIST is then mapped into the CARs of the block. If length is NIL then the length of the list is used. If (LENGTH LIST) is less than LENGTH, then the CARs of the remainder of the block are set to NIL. If (LENGTH LIST) is greater than LENGTH the list is truncated.

6.8.9 LISPPN [VALUE]

is the PPN (as returned by MYPPN) where the system expects to find the greeting files, the system init files and the help files.

6.8.10 (SETNAM name) [SUBR]

changes the name of the running core image to name. This is displayed by ^T and systat. It is also used by SAVE.

6.9 (NOUUO X) [SUBR]

```
flag = T initially
```

NOUUO sets a special flag in the compiled function calling mechanism to the value of X and returns the previous setting of the flag. (Actually any non-NIL value is treated as T, returning T when reset.) Compiled functions initially call other functions with function calling UUOs which "trap" into the UUO mechanism of the interpreter. Ordinarily, such function calls involve searching the property list of the function being called for the functional property, and then (depending on whether the function is compiled or an S-expression) the function is called. If the NOUUO flag is set to NIL, then the overhead in calling a compiled function from a compiled function can be eliminated by replacing the CALL by PUSHJ and JCALL by JRST. CALLF and JCALLF are never changed. However, there are several dangers and restrictions when using (NOUUO NIL). Once the UUO's have been replaced by PUSHJ's then it is not possible to redefine or TRACE the function being called. It is therefore recommended that compiled functions be debugged with (NOUUO T).

6.10 SYSTEM-STUFF-MISC

6.10.1 (DEFSYM name number) [SUBR]

(the number is converted to an address) DEFSYM puts a SYM property on an atom, and if LAP has left some information about its usage before it was defined then some cleaning up is done. The atom is also CONSed onto the value of REMOB.

For related information see LAP, SYM, and REMOB.

6.10.2 (DUMPATOMS file) [FSUBR]

Note - this function is in RUTLIB.LSP[A311LI5P]. After loading a set of files which contain NOCALL declarations, DUMPATOMS may be called to REMOB all NOCALL atoms after first creating a file <file> which, when subsequently loaded, will restore the SUBR, FSUBR, LSUBR, VALUE, and SYM properties of each NOCALL atom. One can thus use DUMPATOMS to REMOB all NOCALL atoms (to save space), and if it is later discovered that one of the functions or

special variables is needed after all, DSKIN the DUMPATOMS file to restore things to their previous state. If <file> is missing, (REMOB.LSP) is assumed.

6.10.3 FIX1A

To convert the number in machine representation in A (in compiled functions) to its LISP integer representation use

```
PUSHJ P,FIX1A
```

6.10.4 GVAL [SUBR]

is an internal LAP function.

6.10.5 GWD

GWD (SUBR) is an internal LAP function.

6.10.6 INUM0

The SYM property of INUM0 is the magic constant for translating between INUMs and addresses. The address of the code of a SUBR in the high segment is (+ (GET <id> 'SUBR) (GET 'INUM0 'SYM)).

6.10.7 KLIST [VALUE]

contains discriptive information about LAP constants and is used by LAP to keep full word constants in unique locations in BPS.

6.10.8 LAPEVAL

LAPEVAL (SUBR) is an internal LAP function.

6.10.9 LAPKLST [VALUE]

is a list of constants that have been used by LAP code. This enables LAP code to share constants (thus saving space). However it does take up list space, so it is normally set to NIL after system generation. When you read in LAP code you will start growing it again.

6.10.10 LAPLST [VALUE]

contains the name and special cell of special variables to allow the printing of variable bindings in backtraces. Special variables are added to LAPLST iff the variable SPECIAL is non-NIL (initially T).

6.10.11 LAPQLST [VALUE]

is a list of list-cells which are accessed by compiled code. Its purpose is to protect these cells from the garbage collector. If you delete its members you are asking for trouble!

6.10.12 LAPSLST [VALUE]

is a list of special cells of NOCALLed special variables that are accessed by LAP code. Its purpose is to protect these cells from the garbage collector. If you delete its members you are asking for trouble!

6.10.13 (MAKNUM X TYPE) [SUBR]

considers X to be a number of the type specified (TYPE should be either FIXNUM or FLONUM) in machine representation and returns the LISP representation of the number. MAKNUM is a SUBR.

6.10.14 (NUMVAL n) [SUBR]

NUMVAL accepts a LISP number and returns the machine representation of that number. It is a SUBR.

6.10.15 (SIXBIT ATOM) [SUBR]

SIXBIT returns an integer whose bit pattern is the SIXBIT representation of its (atomic) argument. Lower case is converted to upper case as required. No error is given if characters not in the SIXBIT character set are present. Up to the first six characters of the PNAME of the atom are used.

6.10.16 (SIXATM N) [SUBR]

SIXATM returns the atom whose PNAME contains the characters from the ASCII set that correspond to the sixbit characters presumably contained in the bit pattern of the integer N.

6.10.17 QLIST [VALUE]

is a list of all S-expression constants referenced by compiled code. Their presence on QLIST protects them from garbage collection.

6.10.18 SPECBIND

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

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

SPECBIND saves the previous values of vari on the special pushdown list and binds the contents of accumulator ni to each vari. The vari must be pointers to special cells of identifiers. Any ni=0 causes the vari to be bound to NIL. Special variables are restored to their previous values by:

```
PUSHJ P, SPECSTR
```

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

6.10.19 (UUO UUO-TYPE) [SUBR]

(UUO UUO-TYPE) performs (UUOPARM 0 UUO-TYPE).

6.10.20 (UUOPARM N UUO-TYPE) [SUBR]

UUOPARM executes the CALLI 1, UUO-TYPE monitor call with the value of N loaded into register 1. The value returned by the UUO in register 1 is made into an integer and returned as the value of UUOPARM. The global variable !SKIP! is set to T if the UUO skipped, NIL otherwise.

7. MISC

7.1 DATES

7.1.1 (DATE) [SUBR]

DATE does (UUO 12) to return the DEC-style date, a number resembling the number of days since the start of 1964.

7.1.2 (DATESTR) [SUBR]

is equivalent to (datestrx (mstime) (date)), which returns a string with the current date and time.

7.1.3 (DATESTRX MSTIME DATE) [SUBR]

DATESTRX returns a string containing the date and time computed from MSTIME and DATE (typically computed by the MSTIME and DATE functions respectively). Note that PRINC will print the string without the quote marks.

7.1.4 (MSTIME) [SUBR]

MSTIME does (UUO 19) to return the time of day in milliseconds.

7.2 (EXIT flag) [SUBR]

LISP may be exited via the EXIT function. Flag specifies whether LISP's sharable high segment should be deleted (flag=NIL) or retained (flag=T) before exiting. There is normally no reason to retain the high segment, as it is automatically loaded when LISP is STARTed or CONTinued. By deleting the high segment, EXIT allows the user to exit from LISP and save the low segment as a runnable SAV file - when the file is later RUN, LISP's sharable high segment will be loaded automatically. (Note that an EXCISE or SYSCLR should be performed before exiting if the low segment is to be saved.) (EXIT T) is necessary only after SETSYS has been used to create a new sharable system, when both the low and high segments must be saved. At the top level, (EXIT) equivalent to (EXIT NIL).

7.3 FN-PROPS [VALUE]

is just a list of the function properties that LISP knows about. These are EXPR, FEXPR, MACRO, SUBR, FSUBR and LSUBR.

7.4 LASTWORD [VALUE]

is meant to contain the last word you defined to LISP. It is set by DE, DF, DM, the editor, DEFPROP (if the property was in GRINPROPS) and perhaps a few others that I've missed. This is used when, for example, you type (EDITF) as the name of the function to edit.

7.5 (NILL "X1" "X2" ... "Xn") [FSUBR]

NILL always returns NIL. This function allows the user to stick S-Expressions in the middle of a function definition (e.g. as a PROG element) without having them evaluated or otherwise noticed. NILL is also useful for giving a dummy definition to a function which has not yet been defined.

7.6 PROBLEMS

This section describes the undesirable features of our lisp system. Anyone who is interested in fixing any of them is invited to send mail to LISP@CMUA. Users are also invited to send in new entries.

Surely the worst feature of our system is that storage allocation can not be done without losing the state of the computation. The problems include relocating the stacks and their contents and moving IO buffers. It is conjectured that it would be safe to treat everything in the stacks as a pointer. Moving IO buffers involves communicating with the monitor. It is suggested that the buffer headers not be moved since this can not be done in earlier versions of the monitor. It would probably be more difficult to fix this problem than to convert all of the nice features of our lisp into MACLISP, which does not have this problem.

The SAVE function does not actually save your job. It just sets things up so that the right thing will be done when you type SAVE to the monitor. The reason is that the CMU special UUO for saving a job was not brought over to the new monitor.

The loader is ancient and crufty. It does not understand lower case. It does not understand PPNs. Anyone who wants to fix it is welcome to try.

Binary Program Space is not garbage collected. It is therefore a bad idea to write programs that will frequently declare local arrays. It is possible to reclaim BPS by hand (by resetting BPORG and BPEND), but if you are using BPS for anything other than arrays this will require a good understanding of LAP.

People who get used to the nice features of the top level become frustrated by their

absence in the break package. It would be nice if the break package could remember events and preprocess inputs through usertop. Any volunteers?

There is no known method for correctly dealing with un-planned-for unbinding of the pdls past functions that are trying to keep track of multiple I/O channels. The usual problem is that an error causes a break at some random time. If the user exits the break by partially unbinding the stack, the I/O functions cannot restore I/O channels. ^ and ^^ are handled correctly because these commands use information saved by the break package to restore I/O channels as of the invocation of the break package at the appropriate level. The proposed fix for this is a general mechanism for causing things to happen during unbinding. There would be some particular identifier which, when unbound would be EVAL'd. This could reset the IO channels and anything else which a program wanted to be sure about when control returned to it. This mechanism would hopefully be used mostly through higher level facilities. What we REALLY need is to be able to simply rebind IO channels.

There are some proposals for checkpointing in the LISP mail file. They are blocked by inadequate IO facilities, such as not being able to open a file for appending. In addition to this, it would be nice to be able to do random access file IO. For example, the HELP program might be made to run much faster if it did not have to search for words but could go right to them.

At the moment there is no reasonable way to try to automatically recover from errors. For example, the error message is printed before usererrorx is called and then usererrorx has no access to it. All of the errors in the lisp system ought to set a global variable to contain as much info (in a standard form) as the error message, and usererrorx should be called before the message is printed. Errorx should evaluate (usererrorx inside an errset, and if a non-atom is returned it should do an outval of the car. (This is instead of simply returning as it does now.) It would be even better if usererrorx did a dispatch on the type of the error to a function named by a variable (sounding like the type of error) so that error-handling could be done by rebinding variables - this would allow programs that knew how to handle certain kinds of errors to do so easily.

Index