CS 206 CLASS NOTES

UCI LISP MANUAL
(An Extended Stanford LISP 1.6 System)

by

Robert J. Bobrow
Richard R. Burton
Daryle Lewis

A reprint of UC Irvine Information and Computer Science
Technical Report No. 21, October 1972

Table of Contents

INTRODUCTION

UCI LISP is a compatible extension of the Stanford LISP 1.6 programming system for the DEC PDP-10. The extensions make UCI LISP a powerful and convenient interactive programming environment for research and teaching in artificial intelligence and advanced list processing applications. All Stanford LISP programs, (except those using the BIGNUM package) can be run directly in UCI LISP. In addition, the extended features of UCI LISP make it much easier to transfer interpreted LISP programs from BBN LISP and MIT AI LISP (we have already converted several large programs, including a version of the Woods' Augmented Transition Network Parser from BBN LISP, and a version of Micro-Planner from MIT AI LISP.)

This manual describes the extensions to the Stanford LISP 1.6 system, and should thus be read in conjunction with the latest Stanford LISP 1.6 manual, currently SAILON 28.6 (Stanford Artificial Intelligence Laboratory Operating Note 28.6). As can be seen from the relative sizes of the two documents UCI LISP represents a substantial extension to Stanford LISP, and from our own experience presents a major improvement in the habitability of the system for both naive and experienced users. (A majority of the extensions were suggested by the features of BBN LISP, probably the best interactive LISP system in existence, but unfortunately available only on TENEX, a paged virtual memory system for the PDP-10, produced by Bolt, Beranek and Newman Inc.)

The major extensions to Stanford LISP can be briefly described as follows:

1) Improvements in storage utilization:
   a) UCI LISP is reentrant and compiled code may be placed in the sharable high segment
   b) the allocator allows reallocation of all spaces (including Binary Program Space) at any time

2) Powerful interactive debugging facilities, including:
   a) Sophisticated conditional breakpoint and function tracing facilities

b) A powerful list structure editor for editing function definitions and data
c) Facilities for examining, correcting and continuing to run in the context of a program which has been interrupted by an error or by a user initiated temporary interrupt

3) Extensions to the I/O facilities available in the basic system, including:
   a) Convenient I/O to disk files, including use of project/programmer designations and ways to save and restore functions and data
   b) Read Macros (patterned after MIT AI LISP) for extending the LISP READ routine
   c) A routine for printing circular or deeply nested expressions
   d) Routines to modify the control table of the LISP READ routine
   e) Several useful functions for carriage positioning, teletype echo and prompt character control, reading input a line at a time, etc.

4) Functions for examining and modifying the special pushdown stack which holds the context of ongoing computations

5) Error protection facilities:
   a) NIL, T and other atoms cannot be easily damaged by RPLACA, RPLACD, SETQ and SET
   b) The system will no longer go into an infinite loop when searching for the function definition of the CAR of a form

6) Extended basic functions including:
   a) New predicates for data types, and predicates which return useful values
   b) New list construction and modification functions
   c) Multiple sequential form evaluation in LAMBDA expressions
   d) An efficient n-way switch
   e) Availability of the FORTRAN mathematical functions
   f) Mapping functions with several arguments, and ones which build new lists using NCONC to join segments

As mentioned, we have made UCI LISP a reentrant system which may be used by several users simultaneously. Thus, while the new features of UCI LISP require a larger system than the original Stanford LISP, this impact is minimized in any environment with more than one LISP user. In addition, since the basic LISP system contains many features previously available only in the various extension files (such as SMILE, ALVINE, TRACE, etc.) or which had to be written by the user, it is possible to write and debug meaningful jobs in the basic system, without getting extra core. The UCI LISP system has a sharable high segment of 14K and a user specific low segment of 8K. Thus, if there are two users the virtual core load is 30K, while getting the same capabilities in Stanford LISP would require a load of 32K for the two users, and of course the improvement is even more noticeable with more users sharing UCI LISP (about 8K is saved for each additional user).

The ability to put compiled code in the sharable segment and to reallocate Binary Program Space makes it possible to build systems in which much of the systems code is compiled LISP expressions. All of the advantages of higher level coding are obtained, and the LISP compiler (borrowed from Stanford with some small modifications) produces better results than most assembly language coders. Such partially compiled systems can now be used without closing off the possibility of the user extending Binary Program Space to store his own compiled code. In general, it is now possible to compile a system incrementally. The user can save the low segment which contains the partially compiled system, then test out new material in interpreted form before extending the Binary Program Space in the segment to load the new compiled material.

The debugging facilities form the bulk of the extensions to Stanford LISP, and are identical with the equivalent facilities available in BBN LISP in the summer of 1971. (BBN LISP has been extended in the intervening period.) They make it possible for the user to track down bugs in complicated recursive programs by making it easier for him to investigate the context in which the bug occurred (e.g. to see at what point erroneous data was passed as an argument, or at what point the flow of control went awry, etc.) The user does not have to plan in advance or set breakpoints to get access to the context of the error. The system holds the context of any error automatically,

allowing the user to perform whatever investigations he wishes, and make any corrections which may be useful. This also makes it possible to patch up a small error, like an unbound atom or simple undefined function, n the middle of a large computation and to continue the computation without having to start from scratch. Similarly, the user can try out ideas for correcting the error, without leaving the context of the error, and go on only when he has pinned down the error and its possible solution. If the information available at the time the LISP system discovers the error is insufficient to pin down the cause of the error, the user can have the system repeat the computation, with a special trace feature that prints out whatever the user wishes to know at various points in the computation. (The user can specify both what data is to be printed and under what conditions he wishes it printed.) The user can also force the system to establish a breakpoint anywhere in his computation, so he can investigate the context before the error has covered its tracks.

The UCI LISP editor (borrowed with some modifications from the BBN LISP system) is actually a language for incremental modification of list structures. It can be used by a user at a terminal to modify function definitions (even during the middle of a break while the function is still on the context stack) or to change complicated data structures. It can also be used as a subroutine by other functions, making it convenient for one function to modify another function. This is actually done by the BREAK package, to implement the function BREAKIN which inserts a breakpoint at any arbitrary point in a user function.

The editor can move around in a structure by small local motions, or by searching for a portion of the structure which matches some given pattern. It can insert new items, delete old ones, interchange items, change structure, embed old items in new structure or extract them from old structure, etc. In order to be able to edit a function which is still on the context stack and to have all of the portions on the context stack be changed at once, the modifications performed by the editor are physical changes of the existing structure. Although all the modifications are "destructive", using RPLACA and RPLACD to make changes in the given structure, all of the modifications can be selectively reversed by means of the UNDO feature. Thus the user can make modifications without worrying about completely destroying his function definitions by accident.

The editor is a very large, complicated function, and its documentation indicates that fact. However, the first part of the editor documentation gives a convenient rundown on how to use the editor as a novice, and with that the beginning user can get quite a bit done. By skimming the remainder of the editor chapter the user can get some idea of the many extra useful features available, and can slowly extend his capabilities with the editor. It has been a common observation that in the process of writing and debugging a large system, or even a small program, the average user spends most of his time in editing his functions. By becoming familiar with all the features of the list structure editor the user can cut his editing time considerably, and make large or subtle changes easily. The user should also bear in mind that the editor is available as a function which can be used by other functions. This can make many jobs substantially easier.

NOTE: ALVINE is no longer available in the standard version of UCI LISP because we believe that the new editor and I/O facilities are substantially better than those provided by ALVINE. (There is an assembly switch which makes it possible to run ALVINE in UCI LISP if necessary.)

Some of the extended I/O facilities of UCI LISP were available in SMILE, etc., but putting them in the shared system saves core. The Read Macro facility is a great convenience and makes using Micro-Planner much simpler. The user-modified READ control table is more general than that available in the Stanford SCAN package (which is still useful and available), and the new SPRINT is faster than the original. The other functions are quite convenient, and will make many tasks simpler.

The special pushdown list has been extended to provide the equivalent of the BBN LISP context stack. This is the backbone of the ERROR and BREAK packages, since it enables running programs to examine their context, and to change it if necessary. The stack functions, particularly RETFROM and OUTVAL make it possible to experiment with various control regimes, where subordinate functions can abort and return from higher level functions on the basis of local information. Indiscriminate fooling around with the stack is likely to produce peculiar and unwanted results, but the stack functions can be extremely helpful at times.

The error protection facilities are an attempt to catch some of the common errors of novices (and experienced users too) which can clobber the system. There are few things more confusing than what happens to the system when the value of NIL is no longer NIL, or if the value of T becomes NIL. In Stanford LISP this could easily happen if SETQ or SET received a list as a first argument. This can no longer happen in UCI LISP. Similarly, Stanford LISP occasionally went into infinite loops because a form had a CAR which was NIL or had no function definition and evaluated to NIL. This has been corrected.

The extended basic functions are ones which were of great use in writing the editor, BREAK package, etc., and in bringing up translated versions of BBN LISP and MIT AI LISP programs. The multiple form LAMBDA expression and the n-way switch SELECTQ should make many programming jobs much more convenient, as should the availability of mapping functions with several arguments. The user will almost certainly profit from skimming through the chapters on these extended features, just to know what is available.

Credits and Acknowledgements

The design and overall direction of the implementation of this system are the responsibility of Robert Bobrow, who also made the first modifications to Stanford LISP, including the original error package, accessible context stack and storage reallocator. In large part the existence of the final system and its extensive documentation is due to the Herculean efforts of Daryle Lewis, who did the bulk of the modifications to the assembly language code (including making Stanford LISP reentrant) and corrected the compiler and LAP systems. He singlehandedly transferred the entire BBN LISP editor and its documentation to our system, and in general performed vital and arduous design, programming and documentation tasks too numerous to mention. Richard Burton did yeoman's labor by transferring (and extending) the BBN LISP ERROR and BREAK packages, and providing their documentation. Whitfield Diffie of Stanford has helped us out of several sticky problems with the LISP system and its compiler. The original implementation of the editor and several I/O functions is due to Rodger Knaus, as well as many helpful suggestions. Finally, but of vital importance, is Alan Bell, whose great knowledge of the PDP-10 operating system helped us through many rough times, and who has done much of the transferring of BBN LISP and MIT AI LISP programs.

We are triply indebted to the designers, implementers and documenters of BBN LISP, particularly Daniel Bobrow and Warren Teitelman. Most of the debugging and interactive facilities as well as the general design philosophy of UCI LISP were inspired by the BBN LISP system. Secondly, we were able to use much of their code directly, since it was written in LISP, making it possible to obtain a large, well-written and debugged system in a fraction of the time and effort it would have taken to write it from scratch. Finally, we have made extensive use of the BBN LISP TENEX REFERENCE MANUAL as a source of raw material for our documentation. In particular, much of the material in the chapters on the BREAK and ERROR packages and the editor is a revised version of the material in the BBN LISP MANUAL. We take full responsibility for the errors and deficiencies produced by such an arrangement, while greatfully acknowledging BBN's aid in providing much of the basic documentation. We are also in debt to several people at BBN

for their aid in obtaining and explaining this material, particularly Jim Goodwin, Alice Hartley and the director of the Artificial Intelligence Group, Jaime Carbonell.

This manual is the work of many people as well as the listed authors - in particular Warren Teitelman, formerly of BBN and now at Xerox Palo Alto Research Center, who produced the original BBN LISP documentation and the lions share of the original code. We are also in debt to Marion Kaufman and Phyllis Siegel who did daily battle with the PDP-10 to produce the RUNOFF files from which this documentation is produced.

Last, but most assuredly not least in the roster of those who have made this system possible are Lynette Bobrow, Kathy Burton and Connie Lewis who lived through the many discussions, all night programming sessions and battle-fatigue of the year during which this system was implemented.

ENJOY, ENJOY!

DEBUGGING FACILITIES

Introduction

Debugging a collection of LISP functions involves isolating problems within particular functions and/or determining when and where incorrect data are being generated and transmitted. In the UCI LISP system, there are five facilities which aid the user in monitoring his program. One of these is the Error Package which takes control whenever an error occurs in a program and which allows the user to examine the state of the world (see section on 'ERROR PACKAGE'). Another facility allows the user to temporarily interrupt his computation and examine its progress. The other three facilities (BREAK, TRACE and BREAKIN) allow the user to (temporarily) modify selected function definitions so that he can follow the flow of control in his programs. All of these facilities use the same system function, BREAK1, as the user interface.

BREAK, BREAKIN and TRACE together are called the Break Package. BREAK and TRACE can be used on compiled and system functions as well as EXPR's, FEXPR's and MACRO's. BREAKIN can be used only with interpreted functions.

BREAK modifies the definition of a function FN, so that if a break condition (defined by the user) is satisified, the process is halted temporarily on a call to FN. The user can then interrogate the state of the machine, perform any computations, and continue or return from the call.

TRACE modifies a definition of a function FN so that whenever FN is called, its arguments (or some other values specified by the user) are printed. When the value of FN is computed it is printed also.

BREAKIN allows the user to insert a breakpoint inside an expression defining a function. When the breakpoint is reached and if a break condition (defined by the user) is satisfied, a temporary halt occurs and the user can again investigate the state of the computation.

The two examples on pages 1.3 and 1.4 illustrate these facilities. In the first example, the user traces the function FACTORIAL. TRACE redefines FACTORIAL so that it calls BREAK1 in such a way that it prints some information, in this case the arguments and value of FACTORIAL, and then

goes on with the computation. When an error occurs on the fifth recursion, BREAK1 reverts to interactive mode, and a full break occurs. The situation is then the same as though the user had originally performed (BREAK FACTORIAL) instead of (TRACE FACTORIAL), and the user can evaluate various LISP forms and direct the course of the computation. In this case, the user examines the variable N, instructs BREAK1 to change L to 1 and continue. The > command, following an UNBOUND ATOM or UNDEFINED FUNCTION error, tells BREAK1 to use the next expression instead of the atom which caused the error. The > command does a destructive replacement of, in this case, 1 for L, and saves an edit step by correcting the typo in the function definition. The rest of the tracing proceeds without incident. The function UNTRACE restores FACTORIAL to its original definition.

In the second example, the user has written Ackermann's function. He then uses BREAK to place a call to BREAK1 around the body of the function. He indicates that ACK is to be broken when M equals N and that before the break occurs, the arguments to ACK are to be printed. While calculating (ACK 2 1), ACK is called twice when M = N. During the first of these breaks, the user prints out a backtrace of the function names and variable bindings. He continues the computation with a GO which causes the value of (ACK 1 1), 3, to be printed before the break is released. The second break is released with an OK which does not print the result of (ACK 1 1). The function UNBREAK with an argument T restores the latest broken or traced function to its original definition.

For further information on how to use BREAK, TRACE and BREAKIN, see the section on The Break Package.

```
*(DE FACTORIAL (N)
     (COND ((ZEROP N) L)
           (T (TIMES N (FACTORIAL (SUB1 N))))))

FACTORIAL
*(TRACE FACTORIAL)

(FACTORIAL)
*(FACTORIAL 4)
ENTER FACTORIAL:
!     N = 4
!  ENTER FACTORIAL:
!  !     N = 3
!  !  ENTER FACTORIAL:
!  !  !     N = 2
!  !  !  ENTER FACTORIAL:
!  !  !  !     N = 1
!  !  !  !  ENTER FACTORIAL:
!  !  !  !  !     N = 0
L
UNBOUND VARIABLE - EVAL

(L BROKEN)
1:N

0

1:> 1

!  !  !  !  FACTORIAL = 1
!  !  !  FACTORIAL = 1
!  !  FACTORIAL = 2
!  FACTORIAL = 6
FACTORIAL = 30
30


(UNTRACE FACTORIAL)

(FACTORIAL)
*(FACTORIAL 4)

30
```

```
*(DE ACK (M N)
     (COND ((ZEROP M) (ADD1 N))
           ((ZEROP N) (ACK (SUB1 M) 1))
           (T (ACK (SUB1 M) (ACK M (SUB1 N))))))

ACK
*(BREAK (ACK (EQ N M) (ARGS)))

(ACK)
*(ACK 2 1)
   M = 1
   N = 1

(ACK BROKEN)
1:BKFV

    M = 1
    N = 1
ACK
    M = 2
    N = 0
ACK
    M = 2
    N = 1
ACK

1:GO

3



   M = 1
   N = 1

(ACK BROKEN)
1:OK

5
*(UNBREAK T)

(ACK)
```

Interrupting a computation-REE and DDT

A useful feature for debugging is a way to temporarily suspend computation. If the user wishes to know how his computation is proceeding (i.e. is he in an infinite loop or is system response poor). Then type Control-C twice (which will cause a return to the monitor) followed by either REE or DDT. After typing REE the user must respond with one of the following control characters; Control-H, Control-B, Control-G, Control-E or Control-Z. Typing DDT is equivalent to typing REE followed by Control-H.

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 BREAK1 (see the next section). He can examine the state of the world and continue or stop his computation using any of the BREAK1 commands. 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, type Control-C twice, followed by REE, followed by 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-Z: This returns the user to the top-level of LISP, (i.e. either the READ-EVAL-PRINT loop or the current INITFN).

BREAK1

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. The DDT break feature is also implemented using BREAK1.

Whenever LISP types a message of the form (-- BROKEN) followed by 'n:' the user is then 'talking to' 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 described below which are specifically recognized by BREAK1 (for summary of commands see Table I, page 1.25).

Since BREAK1 puts all of the power of LISP at the user's command, he can do anything he can do at the top level of LISP. For example, 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 ↑↑).

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. A call to BREAK1 has the form

(BREAK1 BRKEXP BRKWHEN BRKFN BRKCOMS BRKTYPE)

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 primarily 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.

WHAT YOU CAN DO IN A BREAK

Break Commands

Once in a break, in addition to evaluating expressions, the user can ask BREAK1 to perform certain useful actions by giving it atomic items as "break commands". The following commands can be typed in by the user or may be put on the list BRKCOMS. TABLE I (page 1.25) is a summary of these commands.

All printing in BREAK1 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 PRINTLEV because it can print circular lists, which quite often result from errors. PRINTLEV only prints lists to a depth of 6. This depth parameter may be changed by setting the value of %LOOKDPTH. PRINTLEV is necessarily slow and if you are not printing circular structures, traces can be speeded up greatly by changing the value of %PRINFN to PRIN1.

GO

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.

OK

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

EVAL

Causes BRKEXP to be evaluated. The break is maintained and the value of the evaluation is printed and bound on the variable !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.

RETURN form

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).

FROM?= form

This permits the user to release the break and return to a previous context with form to be evaluated. For details see context commands.

> [or ->] expr

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.

USE x FOR y

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 a edit step unnecessary.

↑

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.

↑↑

This returns control directly to the top level of LISP.

ARGS

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

Context Commands

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.

F [or &] arg1 arg2 ... argN

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

When used as the first argument caused LASTPOS not to be reset to above BREAK1 but continues searching from the previous position of LASTPOS.

Numbers

If negative, move LASTPOS back (i.e. towards the top level) that number of calls, if positive, forward.

← Search forward instead of backward for the next atom

Example:

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)

EDIT arg1 arg2 ... argN

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.

?= arg1 arg2 ... argN

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. For example,

```
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.

FROM?= [form]

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).

EX

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

Backtrace Commands

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

BKF

BKF gives a backtrace of the names of functions that are still pending.

BKE

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

BK

BK gives a full backtrace of all expressions still pending.

All of the backtrace commands may be suffixed by a 'V' and/or followed by an integer. If the integer is included, it specifies how many blocks are to be printed. The limiting point of a block is a function call. This form is useful when working on a Data Point. Using the integer feature in conjunction with the F command, which moves LASTPOS, the user can display any contiguous part of the backtrace. If a 'V' is included, variable bindings are printed along with the expressions in the backtrace.

Example:

BKFV would print the names and variable bindings of the functions called before LASTPOS.

BKV 5 would print everything (expressions and variables) for 5 blocks before LASTPOS.

The output of the backtrace commands deserves some explanation. Right circular lists are only printed up to the point where they start repeating and are closed with '...]' instead of a right parenthesis. Lists are only printed to a depth of 2. /#/ Is a notation which represents "the previous expression". For example, (SETQ FIE (FOO)) would appear in a BK backtrace as

```
(FOO)
(SETQ FIE /#/)
```

Breakmacros

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 ttyline1 ttyline2 ... 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))))))
```

BREAK PACKAGE

How To Set A Break

The following functions are useful for setting and unsetting breaks and traces.

Both 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 (--- . 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. 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.

BREAK

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 fuction 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 (see BRWHEN and BRKCOMS of BREAK1). For example,

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

will break all calls to FOO1 and all calls on FOO2 when N is greater than 2 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.

Examples:

```
(BREAK FOO)
(BREAK ((GET IN FOO) T (GO)))
(BREAK (SETQ (EQ N 1) ((PRINT (QUOTE N=1)))(?= M)))
```

TRACE

TRACE is an FEXPR. For each atomic argument, it traces the function named (see form on page 1.3) 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.

TRACE uses the global variable #%INDENT to keep its position on the line. The printing of output by TRACE is printed using %PRINFN (see page 1.9). TRACE can therefore be pretty printed by:

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

Examples:

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

Note: The user can always call BREAK0 himself to obtain combinations of options of BREAK1 not directly available with BREAK and TRACE (see section on BREAK0 below). These functions merely provide convenient ways of calling BREAK0, and will serve for most uses.

BREAKIN

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 fo 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 (BEFORE 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.

Examples:

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

UNBREAK

UNBREAK is an FEXPR. 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.

Note: If a function is both traced and broken in, either UNTRACE or UNBREAK will restore the original function definition.

UNTRACE

UNTRACE is an FEXPR. 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".

(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.

BREAK0 [FN WHEN COMS]

BREAK0 is an EXPR. 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 (see BREAK1). 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, e.g.,

```
(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.

If FN is non-atomic, the value of BREAK0 is a list of the individual values.

For example, 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)
```

SETQ will be traced whenever CAR of its argument (SETQ is an FSUBR) is a member of VARLIST.

ERROR PACKAGE

Introduction

When an error occurs during the evaluation of a LISP expression, control is turned over to the Error Package. The I/O is forced to the TTY (channel NIL) but will be restored to its previous channels if the user continues the evaluation. The idea behind the error package is that it may be possible to 'patch up' the form in which the error occurred and continue. Or, at least, that you can find the cause of the error more easily if you can examine the state of the world at the time of the error. Basically, what the Error Package does is call BREAK1 with BRKEXP set to the form in which the error occurred. This puts the user 'in a break' around the form in which the error occurred. BREAK1 acts just like the top level of the interpreter with some added commands (see section on BREAK1). The main difference when you are in the Error Package is that the variable bindings that were in effect when the error occurred are still in effect. Furthermore, the expressions that were in the process of evaluation are still pending. While in the Error Package, variables may be examined or changed, and functions may be defined or edited just as if you were at the top level. In addition, there are several ways in which you can abort or continue from the point of error. In particular, if you can patch up the error, you can continue by typing OK. If you can't patch the error, ↑ will get you out of the break. When you are in the error package, the prompt character is ':' and is preceded by a level number. Note: if you don't want the error package invoked for some reason, it can be turned off by evaluating (*RSET NIL). Similarly, (*RSET T) will turn the error package back on.

Commands

There are several atoms which will cause special actions when typed into BREAK1 (the error package). These actions are useful for examining the push down stack (e.g. backtraces), changing forms and exiting from the break in various ways. Table I (on the next page) gives a summary of the actions. For a complete description, see the section on 'What You Can Do In A Break'.

Table 1
Break Package Command Summary
(for complete description see pp. 1.8-1.16)

Command	Action
GO	Evaluates BRKEXP, prints its value, and continues with this value
OK	Same as GO but no print of value
EVAL	Reevaluate BRKEXP and print its value. Its value is bound to !VALUE
RETURN xx	Evaluate xx and continue with its value
↑	Escape one level of BREAK1
↑↑	Escape to the top level
> [->] expr	After an error, use expr for the erring atom
FROM?= form	Continues by re-evaluating form at LASTPOS
EX	Same as FROM?= NIL
USE x FOR y	Substitutes x for y in form at LASTPOS (destructively)
F [&] a1..aN	Resets LASTPOS (stack context)
EDIT A1..An	Resets LASTPOS and gives the form at LASTPOS to the LISP Editor
?= f1 ... fN	Evaluates forms fI as of LASTPOS
ARGS	Prints arguments of the broken function
BKF	Backtrace Function Names
BKE	Backtrace Function Calls
BK	Backtrace Expressions

Note: All of the backtrace commands can be combined with a 'V' or followed by an integer. The 'V' will cause the values of variables to be printed. The integer will limit

the trace to that number of blocks. For example, BK 3, BKEV, BKFV 5 and BKEV are all legitimate commands.

The LISP Editor

Contents

The LISP editor allows rapid, convenient modification of list structures. Most often it is used to edit function definitions, (often while the function itself is running) via the function EDITF, e.g., (EDITF FOO). However, the editor can also be used to edit the value of a variable, via EDITV, to edit special properties of an atom, via EDITP, or to edit an arbitrary expression, via EDITE. It is an important feature which allows good on-line interaction in the UCI LISP system.

This chapter begins with a lengthy introduction intended for the new user. The reference portion begins on page 15.

Introduction

Let us introduce some of the basic editor commands, and give a flavor for the editor's language structure by guiding the reader through a hypothetical editing session. Suppose we are editing the following incorrect definition of APPEND

```
(LAMBDA(X)
 Y
 (COND ((NUL X) Z)
       (T (CONS (CAR) (APPEND (CDR X Y))))))
```

We call the editor via the function EDITF:

```
#(EDITF APPEND)
EDIT
#
```

The editor responds by typing EDIT followed by #, which is the editor's ready character, i.e., it signifies that the editor is ready to accept commands. (In other words, all lines beginning with # were typed by the user, the rest by the editor.)

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. Thus:

```
#P
(LAMBDA (X) Y (COND & &))
#
```

Note that the editor prints the current expression, using PRINTLEV, to a depth of 2, i.e., sublists of sublists are printed as &. The command ? Will print the current expression as though PRINTLEV was given a depth of 100.

```
#?
(LAMBDA (X) Y (COND ((NUL X) Z) (T (CONS (CAR) (APPEND (CDR
X Y))))))
#
```

and the command PP (for PrettyPrint) will GRINDEF the current expression.

A positive integer is interpreted by the editor as a command to descend into the correspondingly numbered element of the current expression. Thus:

```
#2
#P
(X)
#
```

A negative integer has a similar effect, but counting begins from the end of the current expression and proceeds backward, i.e., -1 refers to the last element in the expression, -2 the next to the last, etc. For either positive integer or negative integer, if there is no such element, an error occurs. 'Editor errors' are not the same as 'LISP errors', i.e., they never cause breaks or even go through the error machinery but are direct calls to ERR indicating that a command is in some way faulty. What happens next depends on the context in which the command was being executed. For example, there are conditional commands which branch on errors. In most situations, though, an error will cause the editor to type the faulty command followed by a ? And wait for more input. In this case, the editor types the faulty command followed by a ?, and then another #. The current expression is never changed when a command causes an error. thus:

```
#P
(x)
#2
2 ?
#1
#P
X
#
```

A phrase of the form 'the current expression is changed' or 'the current expression becomes' refers to a shift in the editor's ATTENTION, not to a modification of the structure being edited.

When the user changes the current expression by descending into it, the old current expression is not lost. Instead, the editor actually operates by maintaining a chain of expressions leading to the current one. The current expression is simply the last link in the chain. Descending adds the indicated subexpression onto the end of the chain, thereby making it be the current expression. The command 0

is used to ascend the chain; it removes the last link of the chain, thereby making the previous link be the current expression. Thus:

```
#P
X
#0 P
(X)
#0-1 P
(COND (& Z) (T &))
#
```

Note the use of several commands on a single line in the previous output. The editor operates in a line buffered mode. Thus no command is actually seen by the editor, or executed until the line is terminated, either by a carriage return, or an escape (alt-mode).

In our editing session, we will make the following corrections to APPEND: delete Y from where it appears, add Y to the end of the argument list, (These two operations could be thought of as one operation, i.e., move Y from its current position to a new position, and in fact there is a MOVE command in the editor. However, for the purposes of this introduction, we will confine ourselves to the simpler edit commands.) change NUL to NULL, change Z to Y, add X after CAR, and insert a right parenthesis following CDR X.

First we will delete Y. By now we have forgotten where we are in the function definition, but we want to be at the "top," so we use the command ↑, which ascends through the entire chain of expressions to the top level expression, which then becomes the current expression, i.e., ↑ removes all links except the first one.

```
#↑ P
(LAMBDA (X) Y (COND & &))
#
```

Note that if we are already at the top, ↑ has no effect, i.e., it is a NOP. However, 0 would generate an error. In other words, ↑ means "go to the top," while 0 means "ascend one link."

The basic structure modification commands in the editor are

(n)

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 nth element in the current expression.

Thus:

```
#P
(LAMBDA (X) Y (COND & &))
#(3)
#(2 (X Y))
#P
(LAMBDA (X Y) (COND & &))
#
```

All structure modification done by the editor is destructive, i.e., the editor uses RPLACA and RPLACD to physically change the structure it was given. Note that all three of the above commands perform their operation with respect to the nth element from the front of the current expression; the sign of n is used to specify whether the operation is replacement or insertion. Thus, there is no way to specify deletion or replacement of the nth element from the end of the current expression, or insertion before the nth element from the end without counting out that element's position from the front of the list. Similarly, because we cannot specify insertion after a particular element, we cannot attach something at the end of the current expression using the above commands. Instead, we use the command N (for NCONC). Thus we could have performed the above changes instead by:

```
#P
(LAMBDA (X) Y (COND & &))
#(3)
#2 (N Y)
#P
(X Y)
#↑ P
#(LAMBDA (X Y) (COND & &))
#
```

Now we are ready to change NUL to NULL. Rather than specify the sequence of descent commands necessary to reach NULL, and then replace it with NULL, i.e., 3 2 1 (1 NULL), we will use F, the find command, to find NULL:

```
#P
(LAMBDA (X Y) (COND & &))
#F NUL
#P
(NUL X)
#(1 NULL)
#0 P
((NULL X) Z)
#
```

Note that F is special in that it corresponds to TWO inputs. In other words, F says to the editor, "treat your next command as an expression to be searched for." The search is carried out in printout order in the current expression. If the target expression is not found there, F automatically ascends and searches those portions of the higher expressions that would appear after (in a printout) the current expression. If the search is successful, the new current expression will be the structure where the expression was found, (If the search is for an atom, e.g., F NUL, the current expression will be the structure containing the atom. If the search is for a list, e.g., F (NUL X), the current expression will be the list itself.) and the chain will be the same as one resulting from the appropriate sequence of ascent and descent commands. If the search is not successful, an error occurs, and neither the current expression nor the chain is changed: (F is never a NOP, i.e., if successful, the current expression after the search will never be the same as the current expression before the search. Thus F EXPR repeated without intervening commands that change the edit chain can be used to find successive instances of EXPR.)

```
#P
((NULL X) Z)
#F COND P

COND  ?
#P
#((NULL X) Z)
#
```

Here the search failed to find a COND following the current expression, although of course a COND does appear earlier in the structure. This last example illustrates another facet of the error recovery mechanism: to avoid further confusion when an error occurs, all commands on the line beyond the one which caused the error (and all commands that may have been typed ahead while the editor was computing) are forgotten.

We could also have used the R command (for Replace) to change NUL to NULL. A command of the form (R e1 e2) will replace all occurrances of e1 in the current expression by e2. There must be at least one such occurrence or the R command will generate an error. Let us use the R command to change all Z's (even though there is only one) in APPEND to Y:

```
#↑ (R Z Y)
#F Z

Z  ?
#PP
(LAMBDA(X Y)
 (COND ((NULL X) Y)
       (T (CONS (CAR) (APPEND (CDR X Y))))))
#
```

The next task is to change (CAR) to (CAR X). We could do this by (R (CAR) (CAR X)), or by:

```
#F CAR
#(N X)
#P
(CAR X)
#
```

The expression we now want to change is the next expression after the current expression, i.e., we are

currently looking at (CAR X) in (CONS (CAR X) (APPEND (CDR X Y))). We could get to the APPEND expression by typing 0 and then 3 or -1, or we can use the command NX, which does both operations:

```
#P
(CAR X)
#NX P
(APPEND (CDR X Y))
#
```

Finally, to change (APPEND (CDR X Y)) to (APPEND (CDR X) Y), we could perform (2 (CDR X) Y), or (2 (CDR X)) and (N Y), or 2 and (3), deleting the Y, and then 0 (N Y). However, if Y were a complex expression we would not want to have to retype it. Instead, we could use a command which effectively inserts and/or removes left and right parentheses. There are six of these BI, BO, LI, LO, RI, and RO, for Both In, Both Out, Left In, Left Out, Right In, and Right Out. Of course, we will always have the same number of left parentheses as right parentheses, because the parentheses are just a notational guide to structure that is provided by our print program. (Herein lies one of the principal advantages of a LISP oriented editor over a text editor: unbalanced parentheses errors are not possible.) Thus, left in, left out, right in, and right out actually do not insert or remove just one parenthesis, but this is very suggestive of what actually happens.

In this case, we would like a right parenthesis to appear following X in (CDR X Y). Therefore, we use the command (RI 2 2), which means insert a right parentheses after the second element in the second element (of the current expression):

```
#P
(APPEND (CDR X Y))
#(RI 2 2)
#P
(APPEND (CDR X) Y)
#
```

We have now finished our editing, and can exit from the editor, to test APPEND, or we could test it while still inside of the editor, by using the E command:

```
#E (APPEND (QUOTE (A B)) (QUOTE (C D E)))
(A B C D E)
```

The E command causes the next input to be given to EVAL.

We GRINDEF APPEND, and leave the editor.

```
#PP
(LAMBDA(X Y)
 (COND ((NULL X) Y)
       (T (CONS (CAR X) (APPEND (CDR X) Y)))))
#OK
APPEND
*
```

Commands for the New User

This manual is intended primarily as a reference manual, and the remainder of this chapter is organized and presented accordingly. While the commands introduced in the previous scenario constitute a complete set, i.e., the user could perform any and all editing operations using just those commands, there are many situations in which knowing the right command(s) can save the user considerable effort. We include here as part of the introduction a list of those commands which are not only frequently applicable but also easy to use. They are not presented in any particular order, and are all discussed in detail in the reference portion of the chapter.

UNDO

Undoes the last modification to the structure being edited, e.g., if the user deletes the wrong element, UNDO will restore it. The availability of UNDO should give the user confidence to experiment with any and all editing commands, no matter how complex, because he can always reverse the effect of the command.

BK

Like NX, except makes the expression immediately before the current expression become current.

BF

Backwards Find. Like F, except searches backwards, i.e., in inverse print order.

<

Restores the current expression to the expression before the last "big jump", e.g., a find command, an ↑, or another <. 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.

<P

Like < except it restores the edit chain to its state as of the last print, either by P, ?, or PP. If the edit chain has not been changed since the last print, <P restores it to its state as of the printing before that one, i.e., two chains are always saved.

Thus if the user types P followed by 3 2 1 P, <P will take him back to the first P, i.e., would be equivalent to 0 0 0. Another <P would then take him back to the second P, i.e., he can use <P to flip back and forth between two current expressions.

&,--

The search expression given to the F or BF command need not be a literal S-expression. Instead, it can be a pattern. The symbol & can be used anywhere within this pattern to match with any single element of a list, and -- can be used to match with any segment of a list. Thus, in the incorrect definition of APPEND used earlier, F (NUL &) could have been used to find (NUL X), and F (CDR --) or F (CDR & &), but not F (CDR &), to find (CDR X Y).

Note that & and -- can be nested arbitrarily deeply in the pattern. For example, if there are many places where the varaible X is set, F SETQ may not find the desired expression, nor may F (SETQ X &). It may be necessary to use F (SETQ X (LIST --)). However, the usual technique in such a case is to pick out a unique atom which occurs prior to the desired expression and perform two F commands. This "homing in" process seems to be more convenient than ultra-precise specification of the pattern.

@ (at-sign)

Any atom ending in @ (at-sign) in a pattern will match with the first atom or string that contains the same initial characters. For example, F VER@ will find VERYLONGATOM. @ can be nested inside of the pattern, e.g., F (SETQ VER@ (CONS --)).
If the search is successful, the editor will print = followed by the atom which matched with the @-atom, e.g.,

```
#F (SETQ VER@ &)
=VERYLONGATOM
#
```

Frequently the user will want to replace the entire current expression or insert something before it. In order to do this using a command of the form (n e1,...,em) or (-n e1,...,em), the user must be above the current expression. In other words, he would have to perform a 0 followed by a command with the appropriate number. However, if he has reached the current expression via an F command, he may not know what that number is. In this case, the user would like a command whose effect would be to modify the edit chain so that the current expression became the first element in a new, higher current expression. Then he could perform the desired operation via (1 e1,...,em) or (-1 e1,...,em). UP is provided for this purpose.

UP

After UP operates, the old current expression is the first element of the new current expression. Note that if the current expression happens to be the first element in the next higher expression, then UP is exactly the same as 0. Otherwise, UP modifies the edit chain so that the new current expression is a tail (Throughout this chapter 'tail' means 'proper tail') of the next higher expression:

```
#F APPEND
(APPEND (CDR X) Y)
#UP P
... (APPEND & Y))
#0 P
(CONS (CAR X) (APPEND & Y))
#
```

The ... is used by the editor to indicate that the current expression is a tail of the next higher expression as opposed to being an element (i.e., a member) of the next higher expression. Note: if the current expression is already a tail, UP has no effect.

(B e1,...,em)

Inserts e1,...,em before the current expression, i.e., does an UP and then a -1.

(A e1,...,em)

Inserts e1,...,em after the current expression, i.e., does an UP and then either a (-2 e1,...,em) or an (N e1,...,em), if the current expression is the last one in the next higher expression.

(: e1,...,em)

Replaces current expression by e1,...,em, i.e., does an UP and then a (1 e1,...,em).

DELETE

Deletes current expression, i.e., equivalent to (:).

Earlier, we introduced the RI command in the APPEND example. The rest of the commands in this family:, BI, RO, LI, LO, and RO, perform similar functions and are useful in certain situations. In addition, the commands MBD and XTR can be used to combine the effects of several commands of the BI-BO family. MBD is used to embed the current expression in a larger expression. For example, if the current expression is (PRINT bigexpression), and the user wants to replace it by (COND (FLG (PRINT bigexpression))), he can acomplish this by (LI 1), (-1 FLG), (LI 1), and (-1 COND), or by a single MBD command.

XTR is used to extract an expression from the current expression. For example, extracting the PRINT expression from the above COND could be accomplished by (1), (LO 1), and (LO 1) or by a single XTR command. The new user is encouraged to include XTR and MBD in his repertoire as soon as he is familiar with the more basic commands.

Attention Changing Commands

Commands to the editor fall into three classes: commands that change the current expression (i.e., change the edit chain) thereby "shifting the editor's attention," commands that modify the structure being edited, and miscellaneous commands, e.g., exiting from the editor, printing, evaluating expressions.

Within the context of commands that shift the editor's attention, we can distinguish among (1) those commands whose operation depends only on the structure of the edit chain, e.g., Ø, UP, NX; (2) those which depend on the contents of the structure, i.e., commands that search; and (3) those commands which simply restore the edit chain to some previous state, e.g., <, <P. (1) and (2) can also be thought of as local, small steps versus open ended, big jumps. Commands of type (1) are discussed on pp. 2.15-2.21; type (2) on pp. 2.22-2.35; and type (3) on pp. 2.36-2.37.

Local Attention-Changing Commands

UP

(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.

Examples: 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.
```
#F 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.

n (n>0)

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 performs

```
# 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.)

!0

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.

↑

Sets edit chain to LAST of edit chain, thereby making the top level expression be the current expression. Never generates an error.

NX

Effectively does an UP followed by a 2. (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.) thereby making the current expression be the next expression. Generates an error if the current expression is the last one in a list. (However, !NX described below will handle this case.)

BK

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.

For example, if the current expression is (COND ((NULL X) (RETURN Y)))

```
#F RETURN P
(RETURN Y)
#BK P
(NULL X)
```

(NX n) n>0

Equivalent to n NX commands, except if an error occurs, the edit chain is not changed.

(BK n) n>0

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.

!NX

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.

For example:

```
#PP
(PROG (UF)
      (SETQ UF L)
 LP   (COND ((NULL (SETQ L (CDR L))) (ERR NIL))
            ((NULL (CDR (MEMQ# (CAR L) (CADR L))))
             (GO LP)))
      (EDITCOM (QUOTE NX))
      (SETQ UNFIND UF)
      (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. For example using the previous current expression:

```
#F CAR P
(CAR L)
#!NX P
(GO LP)
#<P P
(CAR L)
#NX P
(CADR L)
#
```

(NTH n) n>0

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.

A generalized form of NTH using location specifications is described on page 2.34.

Commands That Search

All of the editor commands that search use the same pattern matching routine. (This routine is available to the user directly, and is described later in this chapter in the section on "Editor Functions.") We will therefore begin our discussion of searching by describing the pattern match mechanism. 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 type 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)
#
```

Search Algorithm

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. (However, 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 300. 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). For discussion, see page 2.45).

Search Commands

All of the commands below set LASTAIL for use by UP, set UNFIND for use by < (p. 2.36), And do not change the edit chain or perform any CONSes if they are unsuccessful or aborted.

F pattern

i.e., two commands: the F informs the editor that the next command is to be interpreted as a 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 earlier.

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 F00@ 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.

(FS pattern[1] ... pattern[n])

Equivalent to F pattern[1] followed by F pattern[2] ... followed by F pattern n, so that if F pattern m fails, edit chain is left at place pattern m-1 matched.

(F= expression x)

Equivalent to (F (== . Expression) x), i.e., searches for a structure EQ to expression, see p. 2.22.

(ORF pattern[1] ... pattern[n])

Equivalent to (F (*ANY* pattern[1] ... pattern[n]) N), i.e., searches for an expression that is matched by either pattern[1] or ... pattern[n]. See p. 2.22.

BF pattern

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)
(BF pattern NIL)

Same as BF pattern.

Location Specification

Many of the more sophisticated commands described later in this chapter 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. IF and ## will be described in detail later in the chapter, along with examples ilustrating their use in location specifications.

Throughout this chapter, 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 $).

(LC . $)

Provides a way of explicitly invoking the location operation, e.g. (LC COND 2 3) will perform the search described above.

(LCL . $)

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.

(SECOND . $)

Same as (LC . $) Followed by another (LC . $) Except that if the first succeeds and second fails, no change is made to the edit chain.

(THIRD . $)

Similar to second.

(← pattern)

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.

(BELOW com x)

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. Thus if the current expression is as shown above, F CADR followed by (BELOW COND) will make the new expression be ((NULL (CDR (FMEMB (CAR L) CADR L)) (GO LP)), and is therefore equivalent to 0 0 0 0.

> 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.

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 <).

(NEX x)

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).

NEX

Same as (NEX ←).

The atomic form of NEX is useful if the user will be performing repeated executions of (NEX x). By simply MARKing (see p. 2.36) The chain corresponding to X, he can use NEX to step through the sublists.

(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 UF) (RETURN L))
#(NTH UF)
#P
... (SETQ UNFIND UF) (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.

(pattern :: . $)

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.

Commands That Save and Restore the Edit Chain

Three facilities are available for saving the current edit chain and later retrieving it. The commands are MARK, which marks the current chain for future reference, ←, (An atomic command; do not confuse with the list command (← pattern).) which returns to the last mark without destroying it, and ←←, which returns to the last mark and also erases it.

MARK

Adds the current edit chain to the front of the list MARKLIST.

←

Makes the new edit chain be (CAR MARKLIST). Generates an error if MARKLIST is NIL, i.e., no MARKS have been performed, or all have been erased.

←←

Similar to ← but also erases the MARK, i.e., performs (SETQ MARKLST (CDR MARKLST)).

If the user did not prepare in advance for returning to a particular edit chain, he may still be able to return to that chain with a single command by using < or <P.

<

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.

<P

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.

(S var . $)

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.

Commands That Modify Structure

The basic structure modifications commands in the editor are:

(n)

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)

m>1 attaches e1 ... em at the end of the current expression.

As mentioned earlier:

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 p. 2.76.

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.)

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 p. 2.63)

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.

The A,B,: Commands

In the (n), (n e1 ... em), and (-n e1 ... em) commands, the sign of the integer is used to indicate the operation. As a result, there is no direct way to express insertion after a particular element, (hence the necessity for a separate N command). Similarly, the user cannot specify deletion or replacement of the NTH element from the end of a list without first converting n to the corresponding positive integer. Accordingly, we have:

(B e1 ... em)

Inserts e1 ... em before the current expression. Equivalent to UP followed by (-1 e1 ... em).

For example, to insert FOO before the last element in the current expression, perform -1 and then (B FOO).

(A e1 ... em)

Inserts e1 ... em after the current expression. Equivalent to UP followed by (-2 e1 ... em) or (N e1 ... em) or (N e1 ... em) whichever is appropriate.

(: e1 ... em)

Replaces the current expression by e1 ... em. Equivalent to UP followed by (1 e1 ... em).

DELETE or (:)

Deletes the current expression, or if the current expression is a tail, deletes its first element.

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.

If the current expression is a tail, then B, A, and : will work exactly the same as though the current expression were the first element in that tail. Thus if the current expression were ... (PRINT Y) (PRINT Z)), (B (PRINT X)) would insert (PRINT X) before (PRINT Y), leaving the current expression ...(PRINT X) (PRINT Y) (PRINT Z)).

The following forms of the A, B, and : commands incorporate a location specification:

(INSERT e1 ... em BEFORE . $)
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.

(REPLACE $ WITH e1 ... em)
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)))

(CHANGE $ TO e1 ... em)
Same as REPLACE WITH

(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 (COR Y) and the user performs (REPLACE WITH (CAR X)).), but UNFIND is set to the edit chain after the DELETE was performed.

Example: (DELETE -1), (DELETE COND 3)

Note that 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 Em 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.

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) meaning 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 then following 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 --).

Extract and Embed

Extraction involves replacing the current expression with one of its subexpressions (from any depth).

(XTR . $)

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 by 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.

The extract command can also incorporate a location specification.

(EXTRACT $1 FROM $2)

($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.

While extracting replaces the current expression by a subexpression, embedding replaces the current expression with one containing it as a subexpression.

(MBD x)

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.

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))).

(MBD e1 ... em)

Equivalent to (MBD (e1 ... em *)).

Example: If the current expression is (PRINT Y), then (MBD SETQ X) will replace it with (SETQ X (PRINT Y)).

(MBD x)

X atomic, same as (MBD (x *)).

Example: If the current expression is (PRINT Y), (MBD RETURN) will replace it with (RETURN (PRINT Y)).

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. Thus if the current expression were (PRINT Y) with (SETQ X (PRINT Y)).

The embed command can also incorporate a location specification.

(EMBED $ IN . x)

($ 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))).

The MOVE Command

The MOVE command allows the user to specify (1) the expression to be moved, (2) the place it is to be moved to, and (3) the operation to be performed there, e.g., insert it before, insert it after, replace, etc.

(MOVE $1 TO com . $2)

($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.

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.

As the following examples taken from actual editing will show, the MOVE command is an extremely versatile and powerful feature of the editor.

```
#?
(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)
#P
... (SELECTQ OBJPR & & &) LP2 (COND & &))
#
```

```
#P
(OR (EQ X LASTAIL) (NOT &) (AND & & &))
#(MOVE 4 TO AFTER (BELOW COND))
#P
(OR (EQ X LASTAIL) (NOT &))
#< P
... (& &) (AND & & &) (T & &))
#
```

```
#P
((NULL X) (COND & &))
#(-3 (GO DELETE))
#(MOVE 4 TO N (← PROG))
#P
((NULL X) (GO DELETE))
#< P
(PROG (&) (COND & & &) (COND & & &) (COND & &))
#(INSERT DELETE BEFORE -1)
#P
(PROG (&) (COND & & &) (COND & & &) DELETE (COND & &))
#
```

Note that in the last example, the user could have added the prog label DELETE and moved the COND in one operation by performing (MOVE 4 TO N (← PROG) (N DELETE)).

Similarly, in the next example, in the course of specifying $2, the location where the expression was to be moved to, the user also performs a structure modification, via (N (T)), thus creating the structure that will receive the expression being moved.

```
#P
((CDR &) (SETQ CL &) (EDITSMASH CL & &))
#(MOVE 4 TO N 0 (N (T)) - 1]
#P
((CDR &) (SETQ CL &))
#< P
(T (EDITSMASH CL & &))
#
```

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. For example:

```
#P
(TENEX)
#(MOVE ↑ F APPLY TO N HERE)
#P
(TENEX (APPLY & & ))
#


#P
(T (PRIN1 C-EXP))
#(MOVE BF PRIN1 TO N HERE)
#P
(T (PRIN1 C-EXP) (PRIN1 &))
#
```

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.

```
#P
(SELECTQ OBJPR (&) (PROGN & &))
```

```
#(MOVE TO BEFORE LOOP)
#P
...(SELECTQ OBJPR & &) LOOP (RPLACA DFPRP &) (RPLACD DFPRP &))
#
```

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, p. 2.34, 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.

(BI n m)

Both in, inserts a left 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).

(BO n)

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).

(LI n)

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)).

(LO n)

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).

(RI n m)

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."

(RO n)

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."

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.

($1 THRU $2)

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)).

($1 TO $2)

Same as THRU except last element not included, i.e., after the BI, an (RI 1 -2) is performed.

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).

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. Thus:

```
#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)
#P
(PROG NIL (SETQ IND &) (SETQ VAL &) (PRINT &) (PRIN1 &) (PRIN1 &))
#(DELETE (SETQ THRU PRI@))
= PRINT
#P
(PROG NIL (PRIN1 &) (PRIN1 &))
#


#P
... LP (SELECTQ & & &) (SETQ Y &) OUT (SETQ FLG &) (RETURN Y))
#(MOVE (1 TO OUT) TO N HERE)
#P
... OUT (SETQ FLG &) (RETURN Y) LP (SELECTQ & & &) (SETQ Y &))
#
#PP
(PROG (TEMP1 TEMP2)
      (COND ((NOT (MEMQ REMARG LISTING))
             (SETQ TEMP1 (ASSOC REMARG NAMEDREMARKS))
             (SETQ TEMP2 (CADR TEMP1)))
            (T (SETQ TEMP1 REMARG)))
      (NCONC LISTING REMARG)
      (RETURN (CONS TEMP1 TEMP2)))
#(EXTRACT (SETQ THRU CADR) FROM COND) PP
(PROG (TEMP1 TEMP2)
      (SETQ TEMP1 (ASSOC REMARG NAMEDREMARKS))
      (SETQ TEMP2 (CADR TEMP1))
      (NCONS LISTING REMARG)
      (RETURN (CONS TEMP1 TEMP2)))
#
```

TO and THRU can also be used directly with XTR. (Because XTR involves a location specification while A,B,:, and MBD do not.) Thus in the previous example, if the current expression had been the COND, e.g., the user had first performed F COND , he could have used (XTR (SETQ THRU CADR)) to perform the extraction.

```
(#1  TO), (#1 THRU)
                        Both same as (#1 THRU -1), i.e.,
                        from #1 thru the end of the list.


#P
(VAL (RPLACA DFPRP &) (RPLACD & &) (RPLACA VARS &) (RETURN &))
#(MOVE (2 TO) TO N (← PROG))
#(N (GO VAR))
#P
(VAL (GO VAR))
#


#P
(T (COND &) (EDITSMASH CL & &) (COND &))
#(-2 (GO REPLACE))
#(MOVE (COND TO) TO N PROG (N REPLACE))
#P
(T (GO REPLACE))
#< P
(PROG (&) (COND & & &) (COND & & &) DELETE (COND & &) REPLACE
(COND &) (EDITSMASH CL & &) (COND &))
#


#PP
(LAMBDA(CLAUSALA X)
 (PROG (A D)
       (SETQ A CLAUSALA)
  LP   (COND ((NULL A) (RETURN NIL)))
       (SERCH X A)
       (RUMARK (CAR A))
       (NOTICECL (CAR A))
       (SETQ A (CDR A))
       (GO LP)))
#(EXTRACT (SERCH THRU NOT@) FROM PROG) P
= NOTICECL
(LAMBDA (CLAUSALA X) (SERCH X A) (RUMARK &) (NOTICECL &))
#(EMBED (SERCH TO) IN (MAP [FUNCTION (LAMBDA (A) *] CLAUSALA]
#PP
(LAMBDA(CLAUSALA X)
 (MAP (FUNCTION
       (LAMBDA(A)
        (SERCH X A)
        (RUMARK (CAR A))
        (NOTICECL (CAR A))))
      CLAUSALA))
```

(R x y)

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.

(SW n m)

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.

Commands That Print

P

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 PRINTLEVEL were given a depth of N.

?

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, p. 2.37.

Commands That Evaluate

E

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.

Example:

```
#E (BREAK FIE FUM)
(FIE FUM)
#E (FOO)
(FIE BROKEN)
1:
```

(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.

(I c x1 ... xn)

Same as (c y1 ... yn) where yi=(EVAL xi).

Example: (I 3 (GETD (QUOTE FOO)) will replace the 3rd element of the current expression with the definition 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.

(## com[1] com[2] ... com[n])
is an FSUBR (not a command). Its value is what the current expression would be after executing the edit commands com[1] ... com[n] starting from the present edit chain. Generates an error if any of com[1] thru com[n] 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 (see p. 2.44). 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.

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 following two commands provide more general ways of computing commands.

(COMS x1 ... xn)
Each xi is evaluated and its value executed as a command.

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, see p. 2.71.)

(COMSQ com[1] ... com[n])
Executes com[1] ... com[n].

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)))).
```

Commands That Test

(IF x)

Generates an error unless the value of (EVAL x) is true, i.e., if (EVAL x) causes an error or (EVAL x)=NIL, IF will cause 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)))).

The IF command can also be used to select between two alternate lists of commands for execution.

(IF x coms1 coms2)

If (EVAL x) is true, execute coms1; if (EVAL x) causes an error or is equal to NIL, execute coms2.

For example, the command (IF (NULL A) NIL (P)) will print the current expression provided A=NIL.

(IF x coms1)

If (EVAL x) is true, execute coms1; otherwise generate an error.

(LP . coms)

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.

(LPQ . Coms)

Same as LP but does not print n OCCURRENCES.

In order to prevent non-terminating loops, both LP and LPQ terminate when the number of iterations reaches MAXLOOP, initially set to 30.

(ORR coms[1] ... Coms[n])

ORR begins by executing coms[1], 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 coms[2], 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))).

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.

(M c . coms)

> 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.

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).

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.

For example, 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.

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.

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.

(LPQ . Coms)

Same as LP but does not print n OCCURRENCES.

In order to prevent non-terminating loops, both LP and LPQ terminate when the number of iterations reaches MAXLOOP, initially set to 30.

(ORR coms[1] ... Coms[n])

ORR begins by executing coms[1], 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 coms[2], 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))).

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.

(M c . coms)

> 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.

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).

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.

For example, 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.

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)

Name, 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, the command SECOND, p. 2.31, 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.

Occasionally, the user will want to employ the S command in a macro to save some temporary result. For example, the SW command 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.

(BIND . coms)

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.)

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))).

User macros are stored on a list USERMACROS. (USERMACROS is initially NIL.) thus if the user wants to save his macros, he should save the value of USERMACROS. (The user probably should also save the value of EDITCOMSL).

Miscellaneous Commands

NIL

Unless preceded by F or BF, is always a NOP.

TTY:

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. (See OK and STOP below.)

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.

OK

Exits from the editor.

STOP

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.

SAVE

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.

For example:

```
#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. For example:

```
*(EDITF FOO)
EDIT
#P
(LAMBDA (X) (PROG & & LP & & & &))
   .
   .
   .
#P
(COND & &)
#OK
FOO
#
   .
   .            Any number of inputs except for
   .            calls to the editor.
*(EDITF FOO)
EDIT
#P
(LAMBDA (X) (PROG & & LP & & & &))
#< P
(COND & &)
#
```

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.

REPACK

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@).

(MAKEFN form args n m)

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).

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.

UNDO

Undoes the last, i.e., most recent, structure modification command that has not yet been 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 !UNDO command, or undo a command other than that most recently ← performed.) and prints the name of that command, e.g., MBD UNDONE. 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.

!UNDO

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.

Whenever the user continues an editing session as described on pages 2.72-2.73, 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.

UNBLOCK

Removes an undo-block. If executed at a non-blocked state, i.e., if UNDO or !UNDO could operate, types NOT BLOCKED.

TEST

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.

??

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 --)
#
```

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 READLINE. 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). For example:

```
#P
(COND (& &) (T &))
#(XTR 3 2)
#MOVE TO AFTER LP
#
```

If the command is on the list EDITCOMSL but no additional input is on the teletype line, an error is generated, e.g.,

```
#P
(COND (& &) (T &))
#MOVE


MOVE ?
#
```

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, e.g.,

```
#P
(COND (& &) (T &))
#2 NXP
(T (CONS X Y))
```

3)

Otherwise, generate an error.

Editor Functions

(EDITL L coms atm marklst mess)

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.

(EDITF x)

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 simply performs
(EDITE (CADR (GETL fn (QUOTE (FEXPR EXPR MACRO)))) coms fn)
and sets LASTWORD to fn.

(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.

(EDITE expr coms atm)

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.

(EDITV editvx)

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.

(EDITP x)

FSUBR function, 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.

(EDITFNS x)

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

(EDIT4E pat y)

Is the pattern match routine. Its value is T if pat matches y. See pp. 2.22-2.23 , 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.

(EDITFPAT pat flg)

Makes a copy of pat with all patterns of type 5 converted to the form expected by EDIT4E. Flg should be passed as NIL (flg=T is for internal use by the editor).

(EDITFINDP x pat flg)

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.

(EDITRACEFN com)

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.

For example, defineing 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.

EDITRACEFN is initially equal to NIL, and undefined.

EXTENDED INTERPRETATION OF LISP FORMS

Extended Lambda Expressions

When solving problems in LISP, it is very often convenient to have a function which executes more than one form but does not need the variable and label features of PROG. We have added this capability to UCI LISP by extending LAMBDA expressions to handle more than one form.

(LAMBDA "ARGUMENT-LIST" "FORM1" "FORM2" . . . "FORMn")

> When such a LAMBDA expression is applied to a list of arguments each FORM is evaluated in sequence and the value of the LAMBDA expression is FORMn (after the arguments are bound to the LAMBDA variables).

Examples:

```
((LAMBDA(X) (CAR X) (CDR X)) (QUOTE (A)))     = NIL
((LAMBDA(X Y) X Y (CONS X Y)) NIL T)     = (NIL . T)
```

> This means that functions defined by DF or DE evaluate all of forms in their definition, instead of just the first one as in Stanford's version. The value of the function is the value of the last form.

WARNING: This is not a PROG; GO and RETURN do not have the expected result.

The Functions PROG1 and PROGN

(PROG1 X1 X2 ... Xn) ,n<6

PROG1 evaluates all expressions X1 X2 ... Xn and returns X1 as its value.

(PROGN X1 X2 ... Xn)

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

Conditional Evaluation of Forms

(SELECTQ X "Y1" "Y2" ... "Yn" Z)

This very useful 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 (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 (not evaluated) 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.

Changes to the Handling of Errors

(ERRSET E "F")

ERRSET has been changed slightly. If F=NIL the error message is suppressed and the error will not cause a break to the Break Package. If F is not given then ERRSET assumes that F=T. If F=0 (i.e. zero) then the error message will be printed on the current output device, otherwise it will be printed on the teletype.

(ERR E)

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 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.

(ERROR E)

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.

Miscellania

(APPLY# FN ARGS)

APPLY# is similar to APPLY except that FN may be a function of any type including MACRO. Note that when either APPLY or APPLY# are given an EXPR as their 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.

Examples:
(APPLY# (QUOTE PLUS) (QUOTE (3 2 2))) = 7
(APPLY# (QUOTE CONS) (LIST (QUOTE A) (QUOTE B))) = (A . B)

(NILL "X1" "X2" ... "Xn") = 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.

EXTENSIONS TO THE STANDARD INPUT/OUTPUT FUNCTIONS

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 a device will cause the function to assume DSK: and use the list as the project-programmer number.

Saving Function Definitions, etc. on Disk Files

(DSKOUT "FILE" "EXPRSLIST")

DSKOUT is an FEXPR and is used to create an entire output file on disk file DSK: "FILE". It sets the linelength to LPTLENGTH, and evaluates all of the expressions in "EXPRSLIST". If an expression on "EXPRSLIST" is atomic, then that atom is given to GRINL instead of being evaluated directly.

For example, if FNLIST is a list of your functions, they can be saved on a disk file, FUNCS.LSP by:
(DSKOUT (FUNCS.LSP) FNLIST (PRINT (QUOTE END-OF-FILE)))

Reading Files Back In

(DSKIN "LIST OF FILE-NAMES")

READ-EVAL-PRINTs the contents of the given files. This is the function to use to read files created by DSKOUT.

Example:

(DSKIN (FUNCS.LSP) DTA0: (DATA.LSP))
Reads FUNCS.LSP from DSK: and DATA.LSP from DTA0:.

(DSKIN (667 2) (DSKLOG.LSP))
Reads DSKLOG.LSP from the disk area of [667,2].

Printing Circular or Deeply Nested Lists

(PRINTLEV EXPRESSION DEPTH)

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.

Spacing Control

(TAB N)

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

"Pretty Printing" Function Definitions and S-Expressions

(GRINDEF "F1" "F2" "F3" ... "FN")

GRINDEF is used to print the definitions of functions and the values of variables in a format suitable for reading back in to LISP, in what is known as DEFPROP format. GRINDEF uses SPRINT (see below) to print these s-expressions in a highly readable format, in which the levels of list structure (or parentheses levels) are indicated by indentation. GRINDEF prints all the properties of the identifiers F1, F2, ..., Fn which appear on the list GRINPROPS. If Fi is non-atomic, it will be SPRINTed.

GRINPROPS

The variable GRINPROPS contains the properties which will be printed by GRINDEF. This variable can be set by the user to print special properties which he has placed on atoms. The initial value of GRINPROPS is (EXPR FEXPR MACRO VALUE SPECIAL).

(GRINL "F1" "F2" ... "FN")

GRINL causes all of the atoms, "F1" "F2" ... "Fn", and all of the atoms on the lists which are the values of the atoms F1 F2 ... Fn to be GRINDEFed. GRINL correctly prints out read macros and is the only function which does. GRINDEF does not save the activation character for the read macros. Warning: Each Fi must be an atom.

(SPRINT EXPR IND)

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.

Reading Whole Lines

(LINEREAD)

LINEREAD reads a line, returning it as a list. If some expression takes more than one line or a line terminates in a comma, space 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.

Example:

```
*(LINEREAD)
*A B (C D
*E) F G

(A B (C D E) F G)

*(LINEREAD)
*A B (C D E),
*F G

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

Teletype and Prompt Character Control Functions

(CLRBFI)

CLRBFI clears the Teletype input buffer.

(TTYECHO)

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.

(PROMPT N)

The LISP READ routines type out a "prompt character" for the user when they expect to read from the teletype. This character is normally a "*". 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.

Example:

```
*(PROMPT 53)
52
+
```

(INITPROMPT N)

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.

(READP)

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

(UNTYI)

UNTYI unreads a character (such as a character input by a TYI or a READCH) and returns the ASCII code for that character.

```
*(DE PEEKC () (UNTYI (TYI)))

*(PROG () (CLRBFI) (PEEKC) (RETURN (TYI))

*A

101
```

READ MACROS - Extending the LISP READ ROUTINE

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.

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.

Functions for Defining Read Macros

(DRM "CHARACTER" "FUNCTION")

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.

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

(DSM "CHARACTER" "FUNCTION")

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

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

Using Read Macros

The use of Read Macros is best described with examples. The Read Macros defined above will be used for the examples.

Example 1

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

Example 2

If (F001 F002 *F003 F004) is read the apparent input is (F001 F002 (F003) F004).

In each case the associated function was evaluated and the result was returned as the next element of the input list.

Example 3

Reading (AT1 :(AT2 AT3) AT4) will result in (AT1 NIL AT2 AT3 AT4).

Example 4

If the input is (AA AB :AC) the result is (AA AB NIL . AC).

It can be seen that the effect of a Splice Macro is to place the result of the function evaluation into the input stream minus the outermost set of parentheses.

Modifying the READ Control Table

Since the LISP READ routines are table driven, it is possible to redefine the meaning of a character by changing its table entry. In each of the following functions CH is the ASCII representation of the character being modified.

(MODCHR CH N)

> The value of MODCHR is the old 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 56 (MODCHR 101 NIL)).

(SETCHR CH N)

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

The meaning of each of the fields in the table entry can be determined from the descriptive diagram of the LISP READ program in the appendix.

NEW FUNCTIONS ON S-EXPRESSIONS

S-Expression Building Functions

(TCONC PTR X)

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.

Example:

```
*(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.

(LCONC PTR X)

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)
```

S-Expression Transforming Functions

(NTH X N)

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).

(REMOVE X L)

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.

(COPY X)

The value of COPY is a copy of X. COPY is equivalent to: (SUBST 0 0 X).

(LSUBST X Y Z)

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)

S-Expression Modifying Functions

All these functions physically modify their arguments by changing appropriate CAR's and CDR's.

(DREMOVE X L)

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.

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.

(DREVERSE L)

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.

(DSUBST X Y Z)

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.

Mapping Functions with Several Arguments

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 < 7) 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
```

Mapping Functions Which Use NCONC

The functions MAPCON and MAPCAN produce lists by NCONC to splice together the values returned by repeated applications of their functional argument.

MAPCON and MAPCAN are especially useful in the case where the function returns NIL. Since NIL does not affect a list if NCONC'ed to it, the output from that function does not appear in the result returned from MAPCON or MAPCAN. 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.

(MAPCON FN ARG)

MAPCON calls 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 is each of the lists 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)
```

(MAPCAN FN ARG)
(MAPCONC FN ARG)

MAPCAN is similar to MAPCON except it calls FN with the CAR of ARG instead of the whole list.

S-Expression Searching and Substitution Functions

(SUBLIS ALST EXPR)

ALST is a list of pairs ((U1 . V1) (U2 . V2) ... (Un . Vn)) with each Ui atomic. The value of SUBLIS is the result of 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.

(SUBPAIR OLD NEW EXPR)

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.

(ASSOC# X Y)

Similar to ASSOC, but uses EQUAL instead of EQ.

(LDIFF X Y)

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.

Efficiently Working with Atoms as Character Strings

(FLATSIZEC L) = (LENGTH (EXPLODEC L))

(NTHCHAR X N) = (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)

Note: The above functions do not really perform the operations listed. They actually use far more efficient methods that require no CONSes, but the effects are as given.

(CHRVAL X)

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

NEW PREDICATES

Data Type Predicates

(CONSP X)

The value of CONSP is T iff X is not an atom.
CONSP is equivalent to:

```
(LAMBDA(X) (NOT (ATOM X)))
```

Examples:

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

(STRINGP X)

The value of STRINGP is T iff X is a string.

(PATOM X)

The value of PATOM is T iff X is an atom or X is a pointer outside of free storage.

(LITATOM X)

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

Alphabetic Ordering Predicate

(LEXORDER X Y)

The value of LEXORDER is T iff X is lexically less than or equal to Y. Note: Both arguments must be atoms and 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
```

Predicates that Return Useful Non-NIL Values

(MEMBER# X Y)

MEMBER# is the same as MEMBER except that it returns the tail of Y starting at the position where X is found.

Examples:

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

(MEMB X Y)
(MEMQ# X Y)

MEMQ# is the same as MEMQ except that it returns the tail of Y starting at the position where X is found.

Examples:

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

(TAILP X Y)

The value of TAILP is X iff X is a list and a tail of Y, i.e., X is EQ to some number of CDRs & Ø of Y.

(AND# X1 X2 ... Xn)
= Xn if all Xi are non-NIL
= NIL otherwise

(OR# X1 X2 ... Xn)
= The first non-NIL argument
= NIL if all Xi are NIL

As with AND and OR these functions only evaluate as many of their arguments as necessary to determine the answer (e.g. AND# stops evaluation after the first NIL argument).

Other Predicates

(NEQ X Y)

The value of NEQ is T iff X is not EQ to Y.
NEQ is equivalent to:

```
(LAMBDA(X Y) (NOT (EQ X Y)))
```

Examples:

```
(NEQ T T)          = NIL
(NEQ T NIL)        = T
(NEQ (QUOTE A) (QUOTE B)) = T
(NEQ 1 1.0)        = T
(NEQ 1 1)          = NIL
(NEQ 1.0 1.0)      = T
```

NEW NUMERIC FUNCTIONS

Minimum and Maximum

(*MIN X Y)	= Minimum of X and Y
(MIN X1 X2 ... Xn)	= Minimum of X1, X2, ... , Xn
(*MAX X Y)	= Maximum of X and Y
(MAX X1 X2 ... Xn)	= Maximum of X1, X2, ... , Xn

FORTRAN Functions In LISP

It is now possible to use the FORTRAN Math Functions in LISP. This allows the user to perform computations that previously were difficult to do in LISP. All functions return FLONUMs for values but may have either a FLONUM or a FIXNUM for an argument.

To load the Arithmetic Package execute the following at the top level of LISP:

```
*(INC(INPUT SYS: (ARITH.LSP)))
<SEQUENCE OF OUTPUT>
*(LOAD)SYS:ARITH$
<LOADER TYPES BACK>
*(ARITH)
```

The above will load the Arithmetic Package into expanded core. To load the package into BINARY PROGRAM SPACE type (LOAD T) instead of (LOAD).

Available Functions

Function Name	Meaning
SIN	Sine with argument in radians
SIND	Sine with argument in degrees
COS	Cosine with argument in radians
COSD	Cosine with argument in degrees
TAN	Tangent
ASIN	Arc Sine
ACOS	Arc Cosine
ATAN	Arc Tangent
SINH	Hyperbolic Sine
COSH	Hyperbolic Cosine
TANH	Hyperbolic Tangent
LOG	Log base e
EXP	Take e to a power
SQRT	Square Root
FLOAT	Convert to a FLONUM
RANDOM	Generates a random number between 0.0 and 1.0

FUNCTIONS FOR THE SYSTEM BUILDER

Loading Compiled Code into the High Segment

The UCI LISP System has a sharable high segment. This high segment contains the interpreter, EDITOR, BREAK package, and all of the utility functions. If the user wants to create his own system he must be able to load compiled code into the high segment. To allow the loading of code into the high segment, the user must both own the file and have write priveleges; to be write priveleged, the user must either be creating the system from UCILSP.REL (see the section on creating the system) or follow the procedure indicated in the function SETSYS. The following three functions are for the purpose of loading code into the high segment and will only work if the user is write priveleged.

(HGHCOR X)

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.

(HGHORG X)

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.

(HGHEND)

The value of HGHEND is the address of the last unused location in the high seg.

(SETSYS DEVICE FILE)

SETSYS enables the user to create his own sharable system. DEVICE is either a project-programmer number or a device name followed by a colon (i.e. DSK:). FILE is the name of the system the user is creating. 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.

Special variables

In order to print variable bindings in the backtraces, we have put a pointer to thje atom header in the CAR of the SPECIAL cell of all bound atoms not used free in compiled code. Unfortunately, for compiled code code to fun, the CAR of the SPECIAL cell of free variables must be NIL. This, when loading LAP code, special variables must be saved if they are to be printed properly in a backtrace. The necessary information is stored on LAPLST which contains the name and the special cell of each special variable in the system. Since this means a two word overhead for each special variable, there is a flag which controls the adding of items to LAPLST. Special variables are added to LAPLST iff the variable SPECIAL is non-NIL. The initial value of SPECIAL is T.

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) to make several NOCALL declarations. Like SPECIAL and DECLARE, when NOCALL is used outside of the compiler, it acts the same an NILL.

Miscellaneous Useful Functions

(UNBOUND)

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.

(SYSCLR)

Re-initializes LISP to read the user's INIT.LSP file when it returns to the top level, e.g. by a Control-G or a START, or a REENTER.

******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)

FREE returns the word X to the free storage list and returns NIL.

(FREELIST X)

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.

Initial System Generation

1) To Generate UCILSP.REL

```
.R MACRO
*UCILSP.REL/P/P/P/P/P/P/P/P/P/P←UCILSP.MAC
```

(Needs to be done only when UCILSP.MAC is changed.)

2) To Generate the LISP System (LISP.SHR and LISP.LOW)

```
R LOADER
*UCILSP.REL$
.CORE 15
.START
BIN. PROG. SP. = 100
(INC (INPUT DSK: LAP))
<RANDOM MESSAGES>
↑C
.SSA LISP
```

<The preceeding loads the following files:
UCILSP.REL, LAP, SYS1.LAP, SYS2.LSP, ERRORX.LSP, ERRORX.LAP,
BREAK.LAP, EDIT.LAP>

(Needs to be done whenever any of the above files are changed.)

3) To Generate LISP.SYM, the LISP LOADER SYMBOL TABLE

```
.RU L052A     (Version 52 of the DEC Loader.
               This file is included with the LISP System)
*UCILSP.REL/J,SYMMAK.REL$
.START
```

(Must be done whenever Step 1 is performed.)

4) To Generate COMPLR.SAV, The LISP COMPILER

```
.AS DSK SYS
.R LISP 36
FULL WORD SP. = 2000
BIN. PROG. SP. = 15000
*(INC (INPUT DSK: (COMPLR.LAP)))
 <RANDOM MESSAGES>
*(NOUUO NIL)
*(CINIT)
↑C
.SA COMPLR.SAV
.DEL COMPLR.HGH
```

(Must be done whenever Step 3 is performed.)

5) To Generate LISP.LOD, the LISP LOADER

```
.R LOADER
*LOADER.REL$
.START
```

(Needs to be done only when LOADER.MAC is changed.)

THE LISP EVALUATION CONTEXT STACK

The Contents of the Context Stack

Whenever a form is given to EVAL, it is pushed onto the top of the Special Pushdown List in the form of an Eval-Blip. This information is used for backtraces. An Eval-Blip entry has NIL in the left half (see SPDLFT) and the form being evaluated in the right half (see SPDLRT).

Also, variable bindings are saved on the Special Pushdown List. The left side of the entry contains a pointer to the special cell and the right side contains the value which was saved.

The only other items on the Special Pushdown List are used by the LISP interpreter, and always have a non-NIL atom in the left half.

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 chapter 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.)

Examining the Context Stack

(SPDLPT)

The value of SPDLPT is a stack pointer to the current top of the stack. (Returns an INUM).

(SPDLFT P)

The value of SPDLFT is the left side of the stack item pointed to by the stack pointer P.

(SPDLRT P)

The value of SPDLRT is the right side of the stack item pointed to by the stack pointer P.

(STKPTR)

The value of STKPTR is a true LISP pointer to a stack item.

(NEXTEV P)

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.

(PREVEV P)

PREVEV is similar to NEXTEV except that it moves up the stack instead of down it,

(STKCOUNT NAME P PEND)

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.

(STKNAME P)

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.

(STKNTH N P)

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.

(STKSRCH NAME P FLAG)

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.

(FNDBRKPT P)

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.

Controlling Evaluation Context

(OUTVAL P V)

OUTVAL adjusts P to an Eval-Blip and returns from that position with V.

(SPREDO P V)

SPREDO adjusts P to an Eval-Blip and re-evaluates from that point.

(SPREVAL P V)

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.

Note: OUTVAL, SPREDO and SPREVAL all use NEXTEV to adjust P to an Eval-Blip.

(EVALV A P)

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.

(RETFROM FN VAL)

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)).

Storage Allocation

When the LISP system is run with a core specification given (i.e., ".R LISP n", n>22), LISP types "ALLOC? (Y OR N)". If you type "N" or space (for no) then the system uses the current allocations. If you type "Y" (for yes) then the system allows you to specify for each area either an octal number followed by a space designating the number of words to added to that area, or a space designating an increase of zero words.

Example: (user input is underlined)

```
ALLOC? (Y OR N) Y
FULL WORD SP. = 200
BIN. PROG. SP. = 2000
REG. PDL. =
SPEC. PDL. = 1000
```

Any remaining storage is divided between the spaces as follows:

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

Reallocation of Storage

If you exhaust one of the storage areas it is possible to increase the size of that area by using the reallocation rocedure. First, expand core with the time sharing system command CORE and then reenter LISP with the REE command. For example, if the original core size was 22K, you could increase it by 4K as follows:

```
*↑C
.CORE 26
.REE
```

When you reenter LISP, the same allocation procedure is followed as described above.

Initial Allocations

The following are the initial allocations for the various storage areas when LISP is initially run.

FREE STORAGE	= 2200
FULL WORD SP.	= 700
BIN. PROG. SP.	= 100
REG. PDL.	= 1000
SPEC. PDL.	= 1000

INDEX