CHAPTER 17

THE TELETYPE EDITOR

The Interlisp teletype 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 a property list, via EDITP, or to edit an arbitrary expression, via EDITE. It is an important feature which allows good on-line interaction in the Interlisp system.

In Interlisp-D, most editing is done using the display editor DEdit (page 20.1), which is an extended, display-oriented version of the teletype editor. The teletype editor is still available, as it offers a facility for doing complex modifications of program structure under program control. For example, BREAKIN (page 10.5) calls the teletype editor to insert a function break within the body of a function. By calling the function EDITMODE (page 20.2) it is possible to set the ''default editor'' (TELETYPE or DISPLAY) called by Masterscope, the break package, etc.

This chapter begins with a lengthy introduction intended for the new user. The reference portion begins on page 17.9.

17.1 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 prompt character. This signifies that the editor is ready to accept commands. In the examples in this chapter, 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 as though printlevel (page 6.18) were set to (2 . 20), i.e., sublists of sublists are printed as &, tails of long lists printed as --. The command ? will print the current expression as though printlevel were 1000.

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

and the command PP will prettyprint 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 e ect, but counting begins from the end of the current expression and proceeds backward, i.e., -1 refers to the last element in the current 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 Interlisp function errors, i.e., they never cause breaks or even go through the error machinery but are direct calls to ERROR! 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. Note that typing control-E while a command is being executed aborts the command exactly as though it had caused an error. *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 modi cation 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, the same as `EVALQT`. Thus no command is actually seen by the editor, or executed, until the line is terminated, either by a carriage return, or a matching right parenthesis. The user can thus use control-A and control-Q for line-editing edit commands, the same as he does for inputs to the Interlisp executive.

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, 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 no-op. 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)` [Editor Command]
Deletes the corresponding element from the current expression.

`(N` E_1 ... E_M`) (N 1)` [Editor Command]
Replaces the `N`th element in the current expression with E_1 ... E_M.

`(-N` E_1 ... E_M`) (N 1)` [Editor Command]
Inserts E_1 ... E_M before the `N`th element in the current expression.

Thus:

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

```
(LAMBDA (X Y) (COND & &))
*
```

All structure modi cation 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 N th 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 N th element from the end of the current expression, or insertion before the N th 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 NUL, and then replace it with NULL, e.g., 3 2 1 (1 NULL), we will use F, the nd command, to nd NUL:

```
*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, [1] and the chain will be the same as one resulting from the appropriate sequence of ascent and descent

[1]If the search is for an atom, e.g., F NUL, the current expression will be the structure containing the atom.

commands. If the search is not successful, an error occurs, and neither the current expression nor the chain is changed: [2]

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

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

Here the search failed to nd 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 E_1 E_2) will replace all occurrences of E_1 in the current expression by E_2. 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

[2] F is never a no-op, 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 nd successive instances of EXPR .

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 e ectively inserts and/or removes left and right parentheses. There are six of these commands: BI (''Both In''), BO (''Both Out''), LI (''Left In''), LO (''Left Out''), RI (''Right In''), and RO (''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, LI, LO, RI, and RO 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 nished 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((A B) (C D E))
(A B C D E)
*
```

The E command causes the next input to be evaluated by Interlisp. If there is another input following it, as in the above example, the rst will be applied (with APPLY) to the second. Otherwise, the input is evaluated (with EVAL).

We prettyprint APPEND, and leave the editor.

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

APPEND
—

17.2 COMMANDS FOR THE NEW USER

As mentioned earlier, the Interlisp 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 e ort. 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 [Editor Command]
Undoes the last modi cation 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 con dence to experiment with any and all editing commands, no matter how complex, because he can always reverse the e ect of the command.

BK [Editor Command]
Like NX, except makes the expression immediately *before* the current expression become current.

BF [Editor Command]
Backwards Find. Like F, except searches backwards, i.e., in inverse print order.

\ [Editor Command]
Restores the current expression to the expression before the last ‘‘big jump’’, e.g., a nd 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 [Editor Command]
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 rst P, i.e., would be equivalent to 0 0 0. Another \P would then take him back to the second P. Thus the user can use \P to ip back and forth between two current expressions.

The search expression given to the F or BF command need not be a literal 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 de nition of APPEND used earlier, F (NUL &) could have been used to nd (NUL X), and F (CDR --) or F (CDR & &), but not F (CDR &), to nd (CDR X Y).

Note that & and -- can be nested arbitrarily deeply in the pattern. For example, if there are many places where the variable X is set, F SETQ may not nd 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.

$ (<esc>) is equivalent to -- at the character level, e.g., VER$ will match with VERYLONGATOM, as will $ATOM, $LONG$, (but not $LONG) and VNM. $ can be nested inside of a 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 E_1 ... E_M) or (-N E_1 ... E_M), 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 E_1 ... E_M) or (-1 E_1 ... E_M). UP is provided for this purpose.

UP [Editor Command]

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 proper tail (page 2.19) of the next higher expression:

```
*F APPEND P
(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 E_1 ... E_M) [Editor Command]

Inserts E_1 ... E_M before the current expression, i.e., does an UP and then a (-1 E_1 ... E_M).

(A E_1 ... E_M) [Editor Command]

Inserts E_1 ... E_M after the current expression, i.e., does an UP and then either a (-2 E_1 ... E_M) or an (N E_1 ... E_M), if the current expression is the last one in the next higher expression.

(: E_1 ... E_M) [Editor Command]

Replaces the current expression by E_1 ... E_M, i.e., does an UP and then a (1 E_1 ... E_M).

DELETE [Editor Command]

Deletes the current expression; equivalent to (:).

Earlier, we introduced the RI command in the APPEND example. The rest of the commands in this family: BI, BO, 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 (page 17.28) 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 could accomplish this by (LI 1), (-1 FLG), (LI 1), and (-1 COND), or by a single MBD command.

XTR (page 17.27) 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), (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.

17.3 LOCAL ATTENTION-CHANGING COMMANDS

This section describes commands that change the current expression (i.e., change the edit chain) thereby ''shifting the editor's attention.'' These commands depend only on the *structure* of the edit chain, as compared to the search commands (presented later), which search the contents of the structure.

UP [Editor Command]

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.

If a P command would cause the editor to type ... before typing the current expression, ie., the current expression is a tail of the next higher expression, UP has no effect.

For Example:

```
*PP
(COND ((NULL X) (RETURN Y)))
*1 P
COND
*UP P
(COND (& &))
*-1 P
((NULL X) (RETURN Y))
*UP P
... ((NULL X) (RETURN Y))
*UP P
```

```
... ((NULL X) (RETURN Y)))
*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 rst determines if the current expression is a tail of the next higher expression. If it is, UP is nished. Otherwise, UP computes (MEMB CURRENT- EXPRESSION NEXT- HIGHER- EXPRESSION) to obtain a tail beginning with the current expression.[3] 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.[4]

N (N 1) [Editor Command]

Adds the Nth element of the current expression to the front of the edit chain, thereby making it be the new current expression. Sets LASTAIL for use by UP. Generates an error if the current expression is not a list that contains at least N elements.

-N (N 1) [Editor Command]

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 [Editor Command]

Sets the edit chain to CDR of the edit chain, thereby making the next higher expression be the new current 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:

---

[3]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.

[4]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 prints LOCATION UNCERTAIN and generates an error. 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 e ects. The LASTAIL solution is cheap and resolves 99% of the ambiguities.

```
*P
(A B C D E F B)
*3 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 [Editor Command]

Does repeated 0's until it reaches a point where the current expression is *not* a tail of the next higher expression, i.e., always goes back to the next higher left parenthesis.

^ [Editor Command]

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

NX [Editor Command]

E ectively does 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 [Editor Command]

Makes the current expression be the previous expression in the next higher expression. Generates an error if the current expression is the rst expression in a list.

For example,

```
*PP
(COND ((NULL X) (RETURN Y)))
*F RETURN P
(RETURN Y)
*BK P
(NULL X)
```

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.

(NX N) [Editor Command]

(N 1) Equivalent to N NX commands, except if an error occurs, the edit chain is not changed.

(BK N) [Editor Command]

(N 1) 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 [Editor Command]

Makes the 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 ((L L)
       (UF L))
  LP  (COND
         ((NULL (SETQ L (CDR L)))
            (ERROR!))
         ([NULL (CDR (FMEMB (CAR L) (CADR L]
            (GO LP)))
       (EDITCOM (QUOTE NX))
       (SETQ UNFIND UF)
       (RETURN L))
*F CDR P
(CDR L)
*NX

NX   ?
*!NX P
(ERROR!)
*!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) [Editor Command]

(N 0) Equivalent to N followed by UP, i.e., causes the list starting with the Nth element of the current expression (or Nth from the end if N < 0) to become the current expression. Causes an error if current expression does not have at least N elements.

(NTH 1) is a no-op, as is (NTH -L) where L is the length of the current expression.

line-feed [Editor Command]
Moves to the ''next'' expression and prints it, i.e. performs a NX if possible, otherwise performs a !NX. (The latter case is indcated by rst printing ''>''.)

control-X [Editor Command]
Control-X[5] moves to the ''previous'' thing and then prints it, i.e. performs a BK if possible, otherwise a !0 followed by a BK.

control-Z [Editor Command]
Control-Z[6] moves to the last expression and prints it, i.e. does -1 followed by P.

Line-feed, control-X, and control-Z are implemented as *immediate* read macros; as soon as they are read, they abort the current printout. They thus provide a convenient way of moving around in the editor. In order to facilitate using di erent control characters for those macros, the function SETTERMCHARS is provided (see page 17.59).

17.4 COMMANDS THAT SEARCH

All of the editor commands that search use the same pattern matching routine (the function EDIT4E, page 17.57). We will therefore begin our discussion of searching by describing the pattern match mechanism. A pattern PAT matches with X if any of the following conditions are true:

(1) If PAT is EQ to X.

(2) If PAT is &.

(3) If PAT is a number and EQP to X.

(4) If PAT is a string and (STREQUAL PAT X) is true.

(5) If (CAR PAT) is the atom *ANY*, (CDR PAT) is a list of patterns, and one of the patterns on (CDR PAT) matches X.

(6) If PAT is a literal atom or string containing one or more $s (<esc>s), each $ can match an inde nite number (including 0) of contiguous characters in the atom or string X, e.g., VER$ matches both VERYLONGATOM and "VERYLONGSTRING" as do $LONG$ (but not $LONG), and VLT. Note: the atom $ (<esc>) matches only with itself.

(7) If PAT is a literal atom or string ending in *two* <esc>s, PAT matches with the atom or string X if it is ''close'' to PAT, in the sense used by the spelling corrector (page 15.13). E.g. CONSS$$ matches with CONS, CNONC$$ with NCONC or NCONC1.

---

[5] Control-A in Interlisp on TOPS-20.

[6] Control-L in Interlisp on TOPS-20.

The pattern matching routine always types a message of the form =MATCHING-ITEM to inform the user of the object matched by a pattern of the above two types, unless EDITQUIETFLG = T. For example, if VER$ matches VERYLONGATOM, the editor would print =VERYLONGATOM.

(8) If (CAR PAT) is the atom --, PAT matches X if (CDR PAT) matches with some tail of X. For example, (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, -- can match any interior segment of a list.

If (CDR PAT)= NIL, i.e., PAT= (--), then it matches any tail of a list. Therefore, (A --) matches (A), (A B C) and (A . B).

(9) If (CAR PAT) is the atom ==, PAT matches X if and only if (CDR PAT) is EQ to X.

This pattern is for use by programs that call the editor as a subroutine, since any non-atomic expression in a command *typed* in by the user obviously cannot be EQ to already existing structure.

(10) If (CADR PAT) is the atom .. (two periods), PAT matches X if (CAR PAT) matches (CAR X) and (CDDR PAT) is contained in X, as described on page 17.20.

(11) Otherwise if X is a list, PAT matches X if (CAR PAT) matches (CAR X), and (CDR PAT) matches (CDR X).

When the editor is searching, the pattern matching routine is called to match with *elements* in the structure, unless the pattern begins with ... (three periods), in which case CDR of the pattern is matched against proper tails in the structure. Thus,

```
*P
(A B C (B C))
*F (B --)
*P
(B C)
*0 F (... B --)
*P
... B C (B C))
```

Matching is also attempted with atomic tails (except for NIL). Thus,

```
*P
(A (B . C))
*F C
*P
... . C)
```

Although the current expression is the atom C after the final command, it is printed as ... . C) to alert the user to the fact that C is a *tail*, not an element. Note that the pattern C will match with either instance of C in (A C (B . C)), whereas (... . C) will match only the second C. The pattern NIL will only match with NIL as an element, i.e., it will not match in (A B), even though CDDR of (A B) is NIL. However, (... . NIL) (or equivalently (...)) may be used to specify a NIL *tail*, e.g., (...

. NIL) will match with CDR of the third subexpression of ((A . B) (C . D) (E)).

17.4.1 Search Algorithm

Searching begins with the current expression and proceeds in print order. Searching usually means nd the next instance of this pattern, and consequently a match is not attempted that would leave the edit chain unchanged. At each step, the pattern is matched against the next element in the expression currently being searched, unless the pattern begins with ... (three periods) in which case it is matched against the next tail of the expression.

If the match is not successful, the search operation is recursive rst 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. Note: A nd command of the form (F PATTERN NIL) will only attempts matches at the top level of the current expression, i.e., it 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 nding a match, and the search continues with the 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 can also be set to NIL, which is equivalent to in nity. 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 nished 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 (or is aborted by control-E), 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 nal 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, unless the atom is a tail, e.g., B in (A . B). In this case, the current expression will be B, but will print as ... . B). In other words, the search e ectively does an UP.[7]

17.4.2 Search Commands

All of the commands below set LASTAIL for use by UP, set UNFIND for use by \ (page 17.21), and do not change the edit chain or perform any CONSes if they are unsuccessful or aborted.

F PATTERN [Editor Command]

Actually two commands: the F informs the editor that the *next* command is to be

[7]Unless UPFINDFLG= NIL (initially set to T). For discussion, see ‘‘Form Oriented Editing’’, page 17.26.

interpreted as a pattern. This is the most common and useful form of the nd command. If successful, the edit chain always changes, i.e., F PATTERN means nd 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 on page 17.15.

Note that if the current expression is (PROG NIL LP (COND (-- (GO LP1))) LP1), then F LP1 will nd the PROG label, not the LP1 inside of the GO expression, even though the latter appears rst (in print order) in the current expression. Note that typing 1 (making the atom PROG be the current expression) followed by F LP1 *would* nd the rst LP1.

F PATTERN N [Editor Command]

Same as F PATTERN , i.e., Finds the Next instance of PATTERN , except that the MEMB check of F PATTERN is not performed.

F PATTERN T [Editor Command]

Similar to F PATTERN , except that it may succeed without changing the edit chain, and it does not perform the MEMB check.

For example, if the current expression is (COND), F COND will look for the next COND, but (F COND T) will ''stay here''.

(F PATTERN N) [Editor Command]

(N 1) 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 PATTERN does not have to match with N identical expressions; it just has to match N times. Thus if the current expression is (FOO1 FOO2 FOO3), (F FOO$ 3) will nd FOO3.

If PATTERN does not match successfully N times, an error is generated and the edit chain is unchanged (even if PATTERN matched N-1 times).

(F PATTERN) [Editor Command]
F PATTERN NIL [Editor Command]

Similar to F PATTERN , except that it 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 the edit chain.

For example, if the current expression is (PROG NIL (SETQ X (COND & &)) (COND &) ...), the command F COND will nd the COND inside the SETQ, whereas (F (COND --)) will nd the top level COND, i.e., the second one.

(FS PATTERN $_1$ PATTERN $_N$) [Editor Command]

Equivalent to F PATTERN $_1$ followed by F PATTERN $_2$ followed by F PATTERN $_N$, so that if F PATTERN $_M$ fails, the edit chain is left at the place PATTERN $_{M-1}$ matched.

(F= EXPRESSION X) [Editor Command]

Equivalent to (F (== . EXPRESSION) X), i.e., searches for a structure EQ to EXPRESSION (see page 17.13).

(ORF PATTERN 1 PATTERN N) [Editor Command]

Equivalent to (F (*ANY* PATTERN 1 PATTERN N) N), i.e., searches for an expression that is matched by either PATTERN 1, PATTERN 2, or PATTERN N (see page 17.13).

BF PATTERN [Editor Command]

''Backwards Find''. Searches in reverse print order, beginning with the 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 --) --)) --),

the command 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 [Editor Command]

Similar to BF PATTERN , except that the search always includes the current expression, i.e., starts at the 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) [Editor Command]

BF PATTERN NIL [Editor Command]

Same as BF PATTERN .

(GO LABEL) [Editor Command]

Makes the current expression be the first thing after the PROG label LABEL , i.e. goes where an executed GO would go.

17.4.3 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.[8]

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 (page 17.46) in conjunction with the ## function (page 17.46) provide a way of using arbitrary predicates applied to elements in the current expression. IF and ## will be described in detail later in the chapter, along with examples illustrating 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 . @) [Editor Command]

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

(LCL . @) [Editor Command]

Same as LC except the search is confined to the current expression, i.e., the edit chain is rebound during the search so that it looks as though 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.

(2ND . @) [Editor Command]

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

(3ND . @) [Editor Command]

Similar to 2ND.

(_ PATTERN) [Editor Command]

Ascends the edit chain looking for a link which matches PATTERN . In other words, it keeps doing 0's until it gets to a specified point. If PATTERN is atomic, it is matched with the first element of each link, otherwise with the entire link. If no match is found, an error is generated, and the edit chain is unchanged.

Note: 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. See page 17.46.

[8]Note that the user could always write F COND followed by 2 and 3 for (COND 2 3) if he were not sure whether or not COND was the name of an atomic command.

For example:

```
*PP
[PROG NIL
   (COND
      [(NULL (SETQ L (CDR L)))
        (COND
           (FLG (RETURN L]
      ([NULL (CDR (FMEMB (CAR L)
        (CADR L]]
*F CADR
*(_ COND)
*P
(COND (& &) (& &))
*
```

Note that this command di ers 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 nd (COND (FLG (RETURN L))), not the higher COND.

(BELOW COM X) [Editor Command]

Ascends the edit chain looking for a link speci ed by COM , and stops X links below that (only links that are elements are counted, not tails). In other words BELOW keeps doing 0's until it gets to a speci ed point, and then backs o X 0's.

Note that X is evaluated, so one can type (BELOW COM (IPLUS X Y)).

(BELOW COM) [Editor Command]

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.

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 nd a sublist that contains a FOO (at any depth). He simply executes F FOO (BELOW \).

(NEX COM) [Editor Command]

Same as (BELOW COM) 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 [Editor Command]

Same as (NEX _).

The atomic form of NEX is useful if the user will be performing repeated executions of (NEX COM). By simply MARKing (see page 17.21) the chain corresponding to COM , he can use NEX to step through the

sublists.

`(NTH COM )` [Editor Command]

Generalized `NTH` command. Effectively performs `(LCL . COM )`, followed by `(BELOW \)`, followed by `UP`.

If the search is unsuccessful, `NTH` generates an error and the edit chain is not changed.

Note that `(NTH NUMBER )` is just a special case of `(NTH COM )`, and in fact, no special check is made for `COM` a number; both commands are executed identically.

In other words, `NTH` locates `COM` , 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))
*
```

`PATTERN .. @` [Editor Command]

E.g., `(COND .. RETURN)`. Finds a `COND` that contains a `RETURN`, at any depth. Equivalent to (but more efficient than) `(F PATTERN N)`, `(LCL . @)` followed by `(_ PATTERN )`.

An infix command, ''`..`'' is not a meta-symbol, it *is* the name of the command. `@` is `CDDR` of the command. Note that `(PATTERN .. @)` can also be used directly as an edit pattern as described on page 17.13, e.g. `F (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 *always* 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`.

17.5 COMMANDS THAT SAVE AND RESTORE THE EDIT CHAIN

Several facilities are available for saving the current edit chain and later retrieving it: `MARK`, which marks

the current chain for future reference, _, which returns to the last mark without destroying it, and __, which returns to the last mark and also erases it.

MARK [Editor Command]
Adds the current edit chain to the front of the list MARKLST.

_ [Editor Command]
Makes the new edit chain be (CAR MARKLST). Generates an error if MARKLST is NIL, i.e., no MARKs have been performed, or all have been erased.

This is an atomic command; do not confuse it with the list command (_ PATTERN).

__ [Editor Command]
Similar to _ but also erases the last MARK, i.e., performs (SETQ MARKLST (CDR MARKLST)).

Note that if the user has two chains marked, and wishes to return to the rst chain, he must perform __, which removes the second mark, and then _. However, the second mark is then no longer accessible. If the user wants to be able to return to either of two (or more) chains, he can use the following generalized MARK:

(MARK LITATOM) [Editor Command]
Sets LITATOM to the current edit chain,

(\ LITATOM) [Editor Command]
Makes the current edit chain become the value of LITATOM .

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.

\ [Editor Command]
Makes the edit chain be the value of UNFIND. Generates an error if UNFIND= NIL.

UNFIND is set to the current edit chain by each command that makes a ‘‘big jump’’, i.e., a command that usually performs more than a single ascent or descent, namely ^, _, __, !NX, all commands that involve a search, e.g., F, LC, .., BELOW, et al and \ and \P themselves. One exception is 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 [Editor Command]
Restores the edit chain to its state as of the last print operation, i.e., P, ?, or PP. If the edit chain has not changed since the last printing, \P restores it to its state as of the printing before that one, i.e., two chains are always saved.

For example, if the user types P followed by 3 2 1 P, \P will return to the rst P, i.e., would be equivalent to 0 0 0. Another \P would then take him back to the second P, i.e., the user could use \P to ip back and forth between the two edit chains.

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.

S LITATOM @ [Editor Command]
Sets LITATOM (using SETQ) to the current expression after performing (LC . @). The edit chain is not changed.

Thus (S FOO) will set FOO to the current expression, and (S FOO -1 1) will set FOO to the first element in the last element of the current expression.

17.6 COMMANDS THAT MODIFY STRUCTURE

The basic structure modification commands in the editor are:

(N) (N 1) [Editor Command]
Deletes the corresponding element from the current expression.

(N E_1 E_M) (N 1) [Editor Command]
Replaces the Nth element in the current expression with E_1 E_M.

(-N E_1 E_M) (N 1) [Editor Command]
Inserts E_1 E_M before the Nth element in the current expression.

(N E_1 E_M) [Editor Command]
Attaches E_1 E_M at the end of the current expression.

As mentioned earlier: *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.* However, all structure modification is undoable, see UNDO (page 17.50).

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.

If the value of CHANGESARRAY is a hash array, the editor will mark all structures that are changed by doing (PUTHASH STRUCTURE FN CHANGESARRAY), where FN is the name of the function. The algorithm used for marking is as follows: (1) If the expression is inside of another expression already marked as being changed, do nothing. (2) If the change is an insertion of or replacement with a list, mark the list as changed. (3) If the change is an insertion of or replacement with an atom, or a deletion, mark the parent as changed.

CHANGESARRAY is primarily for use by PRETTYPRINT (page 6.47). When the value of CHANGECHAR is non-NIL, PRETTYPRINT, when printing to a file or display terminal, prints CHANGECHAR in the right margin while printing an expression marked as having been changed. CHANGECHAR is initially |.

17.6.1 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,[9] *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. Thus if a program constructs the command `(1 (A B C))` e.g., 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`.[10]

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)`.[11]

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

---

[9] 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.

[10] The user can circumvent this by using the `I` command (page 17.45), 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.

[11] 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.

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.

17.6.2 The A, B, and : Commands

In the (N), (N E_1 ... E_M), and (-N E_1 ... E_M) 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 E_1 ... E_M) [Editor Command]
Inserts E_1 ... E_M before the current expression. Equivalent to UP followed by (-1 E_1 ... E_M).

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

(A E_1 ... E_M) [Editor Command]
Inserts E_1 ... E_M after the current expression. Equivalent to UP followed by (-2 E_1 ... E_M) or (N E_1 ... E_M), whichever is appropriate.

(: E_1 ... E_M) [Editor Command]
Replaces the current expression by E_1 ... E_M. Equivalent to UP followed by (1 E_1 ... E_M).

DELETE [Editor Command]
(:) [Editor Command]
Deletes the current expression.

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.

If the current expression is a tail, then B, A, :, and DELETE all 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 E_1 E_M BEFORE . @) [Editor Command]
(@ is (CDR (MEMBER 'BEFORE COMMAND))) Similar to (LC .@) followed by (B E_1 E_M).

Warning: If @ causes an error, the location process does *not* continue as described on page 17.17. For example if @= (COND 3) and the next COND does not have a 3rd element, the search stops and the INSERT fails. Note that the user can always write (LC COND 3) if he intends the search to continue.

```
*P
(PROG (& & X) **COMMENT** (SELECTQ ATM & NIL) (OR & &) (PRIN1 & T)
(PRIN1 & T) (SETQ X &

*(INSERT LABEL BEFORE PRIN1)
*P
(PROG (& & X) **COMMENT** (SELECTQ ATM & NIL) (OR & &) LABEL
(PRIN1 & T) (       user typed control-E

*
```

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 E_1 E_M AFTER . @) [Editor Command]
Similar to INSERT BEFORE except uses A instead of B.

(INSERT E_1 E_M FOR . @) [Editor Command]
Similar to INSERT BEFORE except uses : for B.

(REPLACE @ BY E_1 E_M) [Editor Command]
(REPLACE @ WITH E_1 E_M) [Editor Command]
Here @ is the *segment* of the command between REPLACE and WITH. Same as (INSERT E_1 E_M FOR . @).

Example: (REPLACE COND -1 WITH (T (RETURN L)))

(CHANGE @ TO E_1 E_M) [Editor Command]
Same as REPLACE WITH.

(DELETE . @) [Editor Command]
Does a (LC . @) followed by DELETE.[12] The current edit chain is not changed, but UNFIND is set to the edit chain after the DELETE was performed.

Note: the edit chain will be changed if 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 o. Similarly, if the

[12]See warning about INSERT, page 17.25.

current expression is (CDR Y) and the user performs (REPLACE WITH (CAR X)).

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

Note: if @ is NIL *(i.e., empty), the corresponding operation is performed on the current edit chain.*

For example, (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: @ 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, nd the rst PROG, and insert a (RETURN) at its end, and not change the current edit chain.

The A, B, and : commands, commands, (and consequently INSERT, REPLACE, and CHANGE), all make special checks in E_1 thru E_M 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) AFTER 3) 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. Note that this is not the same as (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.

17.6.3 Form Oriented Editing and the Role of UP

The UP that is performed before A, B, and : commands [13] 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.[14] A consequent of this procedure is that a pattern of the form (SETQ Y --) can be viewed as simply an elaboration and further re nement of the pattern SETQ. Thus (INSERT (RETURN Y) BEFORE SETQ) and (INSERT (RETURN Y) BEFORE (SETQ Y --)) perform the same operation [15] 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 signi cance or meaning is attached to the position of an atom in a list, as Interlisp attaches to atoms that appear as CAR of a list,

---

[13] and therefore in INSERT, CHANGE, REPLACE, and DELETE commands after the location portion of the operation has been performed.

[14] There is some ambiguity in (INSERT EXPR AFTER FUNCTIONNAME), as the user might mean make EXPR be the function's rst argument. Similarly, the user cannot write (REPLACE SETQ WITH SETQQ) meaning change the name of the function. The user must in these cases write (INSERT EXPR AFTER FUNCTIONNAME 1), and (REPLACE SETQ 1 WITH SETQQ).

[15] assuming the next SETQ is of the form (SETQ Y --).

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 EXPR BEFORE FOO), he means before the atom FOO, whether or not it is CAR of a list. By setting the variable UPFINDFLG to NIL (initially T), the user can suppress the implicit UP that follows searches for atoms, and thus achieve the desired effect. With UPFINDFLG= NIL, 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 --).

17.6.4 Extract and Embed

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

(XTR . @) [Editor Command]

Replaces the original current expression with the expression that is current after performing (LCL . @).[16] If the current expression after (LCL . @) is a *tail* of a higher expression, its first element is used.

If the extracted expression is a list, then after XTR has finished, the current expression will be that list. If the extracted expression is not a list, the new current expression will be a tail whose first element is that non-list.

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. The current expression after the XTR would be (PRINT Y).

If the current expression is (COND ((NULL X) Y) (T Z)), then (XTR Y) will replace the COND with Y, even though the current expression after performing (LCL Y) is ... Y). The current expression after the XTR would be ... Y followed by whatever followed the 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 if the current expression is ... (COND ((NULL X) (PRINT Y))) (RETURN Z)), then (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 . $@_2$) [Editor Command]

($@_1$ is the *segment* between EXTRACT and FROM.) Performs (LC . $@_2$)[17] and then (XTR . $@_1$). The current edit chain is not changed, but UNFIND is set to the edit chain after the XTR was performed.

For 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), and (EXTRACT 2 -1 FROM 2) will all produce the same result.

[16] See warning about INSERT, page 17.25.

[17] See warning about INSERT, page 17.25.

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

(MBD E_1 ... E_M) [Editor Command]

MBD substitutes the current expression for all instances of the atom & in E_1 ... E_M, and replaces the current expression with the result of that substitution. As with SUBST, a fresh copy is used for each substitution.

If & does not appear in E_1 ... E_M, the MBD is interpreted as (MBD (E_1 ... E_M &)).

MBD leaves the edit chain so that the larger expression is the new current expression.

Examples:

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))).

If the current expression is (RETURN X), (MBD (PRINT Y) (AND FLG &)) would replace it with the *two* expressions (PRINT Y) and (AND FLG (RETURN X)) i.e., if the (RETURN X) appeared in the cond clause (T (RETURN X)), after the MBD, the clause would be (T (PRINT Y) (AND FLG (RETURN X))).

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

If the current expression *initially* is a tail, embedding works exactly the same as though the current expression were the rst element in that tail. Thus if the current expression were ... (PRINT Y) (PRINT Z)), (MBD SETQ X) would replace (PRINT Y) with (SETQ X (PRINT Y)).

The embed command can also incorporate a location speci cation:

(EMBED @ IN . x) [Editor Command]

(@ is the segment between EMBED and IN.) Does (LC . @)[18] and then (MBD . x). Edit chain is not changed, but UNFIND is set to the edit chain after the MBD was performed.

Examples: (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))).

EDITEMBEDTOKEN [Variable]

The special atom used in the MBD and EMBED commands is the value of this variable, initially &.

[18]See warning about INSERT, page 17.25.

17.6.5 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 . @$_2$) [Editor Command]

(@$_1$ is the segment between MOVE and TO.) COM is BEFORE, AFTER, or the name of a list command, e.g., :, N, etc. Performs (LC . @$_1$),[19] and obtains the current expression there (or its first element, if it is a tail), which we will call EXPR ; MOVE then goes back to the original edit chain, performs (LC . @$_2$) followed by (COM EXPR) (setting an internal flag so EXPR is not copied), then goes back to @$_1$ and deletes EXPR . The edit chain is not changed. UNFIND is set to the edit chain after (COM EXPR) was performed.

If @$_2$ specifies a location *inside of the expression to be moved,* a message is printed and an error is generated, e.g., (MOVE 2 TO AFTER X), where X is contained inside of the second element.

For example, if the current expression is (A B C D), (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 L)) (EDLOC (CDDR C)) (RETURN (CAR L)))
*(MOVE 3 TO : CAR)
*?
(PROG ((L 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 & &))
*
```

[19] See warning about INSERT, page 17.25.

```
*P
((NULL X) **COMMENT** (COND & &))
*(-3 (GO NXT]
*(MOVE 4 TO N (_ PROG))
*P
((NULL X) **COMMENT** (GO NXT))
*\ P
(PROG (&) **COMMENT** (COND & & &) (COND & & &) (COND & &))
*(INSERT NXT BEFORE -1)
*P
(PROG (&) **COMMENT** (COND & & &) (COND & & &) NXT (COND & &))
```

Note that in the last example, the user could have added the PROG label NXT and moved the COND in one operation by performing (MOVE 4 TO N (_ PROG) (N NXT)). 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 &) **COMMENT** (SETQ CL &) (EDITSMASH CL & &))
*MOVE 4 TO N 0 (N (T)) -1]
*P
((CDR &) **COMMENT** (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
(PROG (& & & ATM IND VAL) (OR & &)   **COMMENT**   (OR & &)
(PRIN1 & T) (
PRIN1 & T) (SETQ IND        user typed control-E

*(MOVE * TO BEFORE HERE)
*P
(PROG (& & & ATM IND VAL) (OR & &) (OR & &) (PRIN1 &

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

```
*
```

Finally, if @1 is NIL, the MOVE command allows the user to specify where 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 (FRPLACA DFPRP &) (FRPLACD DFPRP
&) (SELECTQ        user typed control-E

*
```

17.6.6 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 (NTH COM) 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) [Editor Command]

''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) [Editor Command]

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) [Editor Command]

''Both Out''. Removes both parentheses from the Nth element. Generates an error if Nth element is not a list.

Example: If the current expression is (A B (C D E) F G), then (BO D) will modify it to be (A B C D E F G).

(LI N) [Editor Command]

''Left In''. Inserts a left parenthesis before the Nth element (and a matching right parenthesis at the end of the current expression), i.e. equivalent to (BI N -1).

Example: if the current expression is (A B (C D E) F G), then (LI 2) will modify it to be (A (B (C D E) F G)).

(LO N) [Editor Command]

''Left Out''. Removes a left parenthesis from the Nth element. *All elements following the Nth element are deleted.* Generates an error if Nth element is not a list.

Example: If the current expression is (A B (C D E) F G), then (LO 3) will modify it to be (A B C D E).

(RI N M) [Editor Command]

''Right In''. Inserts a right parenthesis after the Mth element of the Nth element. The rest of the Nth element is brought up to the level of the current expression.

Example: If the current expression is (A (B C D E) F G), (RI 2 2) will modify it to be (A (B C) D E F G). Another way of thinking about RI is to read it as ''move the right parenthesis at the end of the Nth element *in* to after its Nth element.''

(RO N) [Editor Command]

''Right Out''. Removes the right parenthesis from the Nth element, moving it to the end of the current expression. All elements following the Nth element are moved inside of the Nth element. Generates an error if Nth element is not a list.

Example: If the current expression is (A B (C D E) F G), (RO 3) will modify it to be (A B (C D E F G)). Another way of thinking about RO is to read it as ''move the right parenthesis at the end of the Nth element *out* to the end of the current expression.''

17.6.7 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 in their respective location specifications the TO or THRU command.

(@$_1$ THRU @$_2$) [Editor Command]

Does a (LC . @$_1$), followed by an UP, and then a (BI 1 @$_2$), thereby grouping the segment into a single element, and finally does a 1, making the final current expression be that element.

For example, if the current expression is (A (B (C D) (E) (F G H) I) J K), following (C THRU G), the current expression will be ((C D) (E) (F G H)).

(@$_1$ TO @$_2$) [Editor Command]

Same as THRU except the 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 5) means (C THRU E) not (C THRU G). In this case, the corresponding BI command is (BI 1 @$_2$-@$_1$+1).

THRU and TO are not very useful commands by themselves; they are intended to be used in conjunction with EXTRACT, EMBED, DELETE, REPLACE, and MOVE. After THRU and TO have operated, they set an internal editor ag 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 (& & ATM IND VAL WORD) (PRIN1 & T) (PRIN1 & T) (SETQ IND &)
(SETQ VAL &) **COMMENT** (SETQQ      user typed control-E

*(MOVE (3 THRU 4) TO BEFORE 7)
*P
(PROG (& & ATM IND VAL WORD) (SETQ IND &) (SETQ VAL &) (PRIN1 & T)
(PRIN1 & T) **COMMENT**      user typed control-E

*

*P
(* FAIL RETURN FROM EDITOR. USER SHOULD NOTE THE VALUES OF SOURCEXPR
AND CURRENTFORM.  CURRENTFORM IS THE LAST FORM IN SOURCEXPR WHICH WILL
HAVE BEEN TRANSLATED, AND IT CAUSED THE ERROR.)
*(DELETE (USER THRU CURR$))
=CURRENTFORM.
*P
(* FAIL RETURN FROM EDITOR.  CURRENTFORM IS      user typed control-E

*

*P
... LP (SELECTO & & & & NIL) (SETQ Y &) OUT (SETQ FLG &) (RETURN Y))
*(MOVE (1 TO OUT) TO N HERE]
*P
... OUT (SETQ FLG &) (RETURN Y) LP (SELECTQ & & & & NIL) (SETQ Y &))
*

*PP
[PROG (RF TEMP1 TEMP2)
   (COND
      ((NOT (MEMB REMARG LISTING))
         (SETQ TEMP1 (ASSOC REMARG NAMEDREMARKS)) **COMMENT**
         (SETQ TEMP2 (CADR TEMP1))
         (GO SKIP))
      (T    **COMMENT**
         (SETQ TEMP1 REMARG)))
   (NCONC1 LISTING REMARG)
   (COND
      ((NOT (SETQ TEMP2 (SASSOC
```

```
*(EXTRACT (SETQ THRU CADR) FROM COND)
*P
(PROG (RF TEMP1 TEMP2) (SETQ TEMP1 &) **COMMENT** (SETQ TEMP2 &) (NCONC1 LISTING
REMARG) (COND & &      user typed control-E

*
```

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) [Editor Command]
(@$_1$ THRU) [Editor Command]
Both are the same as (@$_1$ THRU -1), i.e., from @$_1$ through the end of the list.

Examples:

```
*P
(VALUE (RPLACA DEPRP &) (RPLACD &) (RPLACA VARSWORD &) (RETURN))
*(MOVE (2 TO) TO N (_ PROG))
*(N (GO VAR))
*P
(VALUE (GO VAR))

*P
(T **COMMENT** (COND &) **COMMENT** (EDITSMASH CL & &) (COND &))
*(-3 (GO REPLACE))
*(MOVE (COND TO) TO N ^ PROG (N REPLACE))
*P
(T **COMMENT** (GO REPLACE))
*\ P
(PROG (&) **COMMENT** (COND & & &) (COND & & &) DELETE (COND & &) REPLACE
(COND &) **COMMENT** (EDITSMASH CL & &) (COND &))
*

*PP
[LAMBDA (CLAUSALA X)
   (PROG (A D)
      (SETQ A CLAUSALA)
  LP  (COND
         ((NULL A)
            (RETURN)))
      (SERCH X A)
      (RUMARK (CDR A))
      (NOTICECL (CAR A))
      (SETQ A (CDR A))
      (GO LP]
*(EXTRACT (SERCH THRU NOT$) FROM PROG)
=NOTICECL
*P
```

```
(LAMBDA (CLAUSALA X) (SERCH X A) (RUMARK &) (NOTICECL &))
*(EMBED (SERCH TO) IN (MAP CLAUSALA (FUNCTION (LAMBDA (A) *]
*PP
[LAMBDA (CLAUSALA X)
   (MAP CLAUSALA
        (FUNCTION (LAMBDA (A)
                     (SERCH X A)
                     (RUMARK (CDR A))
                     (NOTICECL (CAR A]
*
```

17.6.8 The R Command

`(R X Y)` [Editor Command]

Replaces all instances of `X` by `Y` in the current expression, e.g., `(R CAADR CADAR)`. Generates an error if there is not at least one instance.

The `R` command operates in conjunction with the search mechanism of the editor. The search proceeds as described on page 17.15, and `X` can employ any of the patterns on page 17.13. Each time `X` matches an element of the structure, the element is replaced by (a copy of) `Y`; each time `X` matches a tail of the structure, the tail is replaced by (a copy of) `Y`.

For example, if the current expression is `(A (B C) (B . C))`,

`(R C D)` will change it to `(A (B D) (B . D))`,

`(R (... . C) D)` will change it to `(A (B C) (B . D))`,

`(R C (D E))` will change it to `(A (B (D E)) (B D E))`, and

`(R (... . NIL) D)` will change it to `(A (B C . D) (B . C) . D)`.

If `X` is an atom or string containing `$`s (<esc>s), `$`s appearing in `Y` stand for the characters matched by the corresponding `$` in `X`. For example, `(R FOO$ FIE$)` means for all atoms or strings that begin with `FOO`, replace the characters "`FOO`" by "`FIE`".[20] Applied to the list `(FOO FOO2 XFOO1)`, `(R FOO FIE)` would produce `(FIE FIE2 XFOO1)`, and `(R $FOO$ $FIE$)` would produce `(FIE FIE2 XFIE1)`. Similarly, `(R $D$ $A$)` will change `(LIST (CADR X) (CADDR Y))` to `(LIST (CAAR X) (CAADR))`. Note that `CADDR` was *not* changed to `CAAAR`, i.e., `(R $D$ $A$)` does not mean replace every `D` with `A`, but replace the first `D` in every atom or string by `A`. If the user wanted to replace every `D` by `A`, he could perform `(LP (R $D$ $A$))`.

The user will be informed of all such `$` replacements by a message of the form `X->Y`, e.g., `CADR->CAAR`.

Note that the `$` feature can be used to delete or add characters, as well as replace them. For example, `(R $1 $)` will delete the terminating `1`'s from all literal atoms and strings. Similarly, if an `$` in `X` does

[20] If `X` matches a string, it will be replaced by a string. Note that it does not matter whether `X` or `Y` themselves are strings, i.e. `(R $D$ $A$)`, `(R "$D$" $A$)`, `(R $D$ "$A$")`, and `(R "$D$" "$A$")` are equivalent. Note also that `X` will never match with a number, i.e., `(R $1 $2)` will not change 11 to 12.

not have a mate in Y, the characters matched by the $ are effectively deleted. For example, (R $/$ $) will change AND/OR to AND.[21] Y can also be a list containing $s, e.g., (R $1 (CAR $)) will change FOO1 to (CAR FOO), FIE1 to (CAR FIE).

If X does not contain $s, $ appearing in Y refers to the *entire* expression matched by X, e.g., (R LONGATOM '$) changes LONGATOM to 'LONGATOM, (R (SETQ X &) (PRINT $)) changes every (SETQ X &) to (PRINT (SETQ X &)).[22]

Since (R X Y) is a frequently used operation for Replacing Characters, the following command is provided:

(RC X Y) [Editor Command]
Equivalent to (R X Y)

R and RC change all instances of X to Y. The commands R1 and RC1 are available for changing just one, (i.e., the first) instance of X to Y.

(R1 X Y) [Editor Command]
Find the first instance of X and replace it by Y.

(RC1 X Y) [Editor Command]
(R1 X Y).

In addition, while R and RC only operate within the current expression, R1 and RC1 will continue searching, a la the F command, until they find an instance of x, even if the search carries them beyond the current expression.

(SW N M) [Editor Command]
Switches the Nth and Mth elements of the current expression.

For example, if the current expression is (LIST (CONS (CAR X) (CAR Y)) (CONS (CDR X) (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, i.e., (SW 3 2) and (SW 2 3) are equivalent.

SW uses the generalized NTH command (NTH COM) 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.

(SWAP $@_1$ $@_2$) [Editor Command]
Like SW except switches the expressions specified by $@_1$ and $@_2$, not the corresponding elements of the current expression, i.e. $@_1$ and $@_2$ can be at different levels in current expression, or one or both be outside of current expression.

---

[21] There is no similar operation for changing AND/OR to OR, since the first $ in Y always corresponds to the first $ in X, the second $ in Y to the second in X, etc.

[22] If X is a pattern containing an $ pattern somewhere *within* it, the characters matched by the $s are not available, and for the purposes of replacement, the effect is the same as though X did not contain any $s. For example, if the user types (R (CAR F$) (PRINT $)), the second $ will refer to the entire expression matched by (CAR F$).

Thus, using the previous example, (SWAP CAR CDR) would result in (LIST (CONS (CDR X) (CAR Y)) (CONS (CAR X) (CDR Y))).

17.7 COMMANDS THAT PRINT

PP [Editor Command]
Prettyprints the current expression.

P [Editor Command]
Prints the current expression as though PRINTLEVEL (page 6.18) were set to 2.

(P M) [Editor Command]
Prints the M th element of the current expression as though PRINTLEVEL were set to 2.

(P 0) [Editor Command]
Same as P.

(P M N) [Editor Command]
Prints the M th element of the current expression as though PRINTLEVEL were set to N.

(P 0 N) [Editor Command]
Prints the current expression as though PRINTLEVEL were set to N.

? [Editor Command]
Same as (P 0 100).

Both (P M) and (P M N) use the generalized NTH command (NTH COM) to obtain the corresponding element, so that M does not have to be a number, e.g., (P COND 3) will work. PP causes all comments to be printed as **COMMENT** (see page 6.50). P and ? print as **COMMENT** only those comments that are (top level) elements of the current expression. Lower expressions are not really seen by the editor; the printing command simply sets PRINTLEVEL and calls PRINT.

PP* [Editor Command]
Prettyprints current expression, *including* comments.

PP* is equivalent to PP except that it rst resets **COMMENT**FLG to NIL (see page 6.50).

PPV [Editor Command]
Prettyprints the current expression as a variable, i.e., no special treatment for LAMBDA, COND, SETQ, etc., or for CLISP.

PPT [Editor Command]
Prettyprints the current expression, printing CLISP translations, if any.

?= [Editor Command]
Prints the argument names and corresponding values for the current expression. Analagous to the ?= break command (page 9.5). For example,

```
*P
(STRPOS "A0???" X N (QUOTE ?) T)
*?=
X = "A0???"
Y = X
START = N
SKIP = (QUOTE ?)
ANCHOR = T
TAIL =
```

The command MAKE (page 17.44) is an imperative form of ?=. It allows the user to specify a change to the element of the current expression that corresponds to a particular argument name.

All printing functions print to the terminal, regardless of the primary output file. All use the readtable T. No printing function ever changes the edit chain. All record the current edit chain for use by \P (page 17.21). All can be aborted with control-E.

17.8 COMMANDS FOR LEAVING THE EDITOR

OK [Editor Command]

Exits from the editor.

STOP [Editor Command]

Exits from the editor with an error. Mainly for use in conjunction with TTY: commands (page 17.40) that the user wants to abort.

Since all of the commands in the editor are errorset 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.

Actually, it is also possible to exit the editor by typing control-D. STOP is preferred even if the user is editing at the EVALQT level, as it will perform the necessary ''wrapup'' to insure that the changes made while editing will be undoable.

SAVE [Editor Command]

Exits from the editor and saves the ''state of the edit'' on the property list of the function or 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 or 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 and 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 LISPX inputs
  .                    except for calls to the editor
_EDITF(FOO)
EDIT
*P
(LAMBDA (X) (PROG & & LP & & & &))
*\ P
(COND & &)
*
```

Furthermore, as a result of the history feature, if the editor is called on the same expression within a certain number of LISPX inputs,[23] the state of the edit of that expression is restored, regardless of how many other expressions may have been edited in the meantime. For example:

[23]Namely, the size of the history list, which can be changed with CHANGESLICE, (page 8.18).

```
_EDITF(FOO)
EDIT
*
  .
  .
  .
*P
(COND (& &) (& &) (&) (T &))
*OK
FOO
  .                    a small number of LISPX inputs,
  .                    including editing
  .
_EDITF(FOO)
EDIT
*\ P
(COND (& &) (& &) (&) (T &))
*
```

Thus 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 aborted via control-D or exited via STOP will not affect the editor's memory); or (2) That session was ''sufficiently'' recent; or (3) It was ended with a SAVE command.

17.9 NESTED CALLS TO EDITOR

TTY: [Editor Command]

Calls the editor recursively. The user can then type in commands, and have them executed. The TTY: command is completed when the user exits from the lower editor. (see OK and STOP above).

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.

EF [Editor Command]
EV [Editor Command]
EP [Editor Command]

Calls EDITF or EDITV or EDITP on CAR of current expression.

17.10 MANIPULATING THE CHARACTERS OF AN ATOM OR STRING

`RAISE` [Editor Command]

An edit macro de ned as `UP` followed by `(I 1 (U-CASE (## 1)))`, i.e., it raises to upper- case the current expression, or if a tail, the rst element of the current expression.

`LOWER` [Editor Command]

Similar to `RAISE`, except uses `L-CASE`.

`CAP` [Editor Command]

First does a `RAISE`, and then lowers all but the rst character, i.e., the rst character is left capitalized.

Note: `RAISE`, `LOWER`, and `CAP` are all no-ops if the corresponding atom or string is already in that state.

`(RAISE x)` [Editor Command]

Equivalent to `(I R (L-CASE x) x)`, i.e., changes every lower-case `x` to upper-case in the current expression.

`(LOWER x)` [Editor Command]

Similar to `RAISE`, except performs `(I R x (L-CASE x))`.

Note that in both `(RAISE x)` and `(LOWER x)`, `x` should be typed in upper case.

`REPACK` [Editor Command]

Permits the ‘‘editing’’ of an atom or string.

`REPACK` operates by calling the editor recursively on `UNPACK` of the current expression, or if it is a list, on `UNPACK` of its rst 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.

Example:

```
*P
... "THIS IS A LOGN STRING")
*REPACK
*EDIT
P
(T H I S %  I S %  A %  L O G N %  S T R I N G)
*(SW G N)
*OK
"THIS IS A LONG STRING"
*
```

Note that this could also have been accomplished by `(R $GN$ $NG$)` or simply `(RC GN NG)`.

`(REPACK @)` [Editor Command]

Does `(LC . @)` followed by `REPACK`, e.g. `(REPACK THIS$)`.

17.11 MANIPULATING PREDICATES AND CONDITIONAL EXPRESSIONS

JOINC [Editor Command]

Used to join two neighboring COND's together, e.g. (COND CLA USE$_1$ CLA USE$_2$) followed by (COND CLA USE$_3$ CLA USE$_4$) becomes (COND CLA USE$_1$ CLA USE$_2$ CLA USE$_3$ CLA USE$_4$). JOINC does an (F COND T) first so that you don't have to be at the first COND.

(SPLITC X) [Editor Command]

Splits one COND into two. X specifies the last clause in the first COND, e.g. (SPLITC 3) splits (COND CLA USE$_1$ CLA USE$_2$ CLA USE$_3$ CLA USE$_4$) into (COND CLA USE$_1$ CLA USE$_2$) (COND CLA USE$_3$ CLA USE$_4$). Uses the generalized NTH command (NTH COM), so that X does not have to be a number, e.g., the user can say (SPLITC RETURN), meaning split after the clause containing RETURN. SPLITC also does an (F COND T) first.

NEGATE [Editor Command]

Negates the current expression, i.e. performs (MBD NOT), except that is smart about simplifying. For example, if the current expression is: (OR (NULL X) (LISTP X)), NEGATE would change it to (AND X (NLISTP X)).

NEGATE is implemented via the function NEGATE (page 14.2).

SWAPC [Editor Command]

Takes a conditional expression of the form (COND (A B)(T C)) and rearranges it to an equivalent (COND ((NOT A) C)(T B)), or (COND (A B) (C D)) to (COND ((NOT A) (COND (C D))) (T B)).

SWAPC is smart about negations (uses NEGATE) and simplifying CONDs. It always produces an equivalent expression. It is useful for those cases where one wants to insert extra clauses or tests.

17.12 HISTORY COMMANDS IN THE EDITOR

As described on page 8.35, all of the user's inputs to the editor are stored on EDITHISTORY, the editor's history list, and all of the programmer's assistant commands for manipulating the history list, e.g. REDO, USE, FIX, NAME, etc., are available for use on events on EDITHISTORY. In addition, the following four history commands are recognized specially by the editor. They always operate on the last, i.e. most recent, event.

DO COM [Editor Command]

Allows the user to supply the command name when it was omitted.

USE is useful when a command name is *incorrect*.

For example, suppose the user wants to perform (-2 (SETQ X (LIST Y Z))) but instead types just (SETQ X (LIST Y Z)). The editor will type SETQ ?, whereupon the user can type DO -2. The effect is the same as though the user had typed FIX, followed by (LI 1), (-1 -2), and OK, i.e., the command (-2 (SETQ X (LIST Y Z))) is executed. DO also works if the command is a line

command.

`!F` [Editor Command]

Same as `DO F`.

In the case of `!F`, the previous command is always treated as though it were a line command, e.g., if the user types `(SETQ X &)` and then `!F`, the effect is the same as though he had typed `F (SETQ X &)`, not `(F (SETQ X &))`.

`!E` [Editor Command]

Same as `DO E`.

`!N` [Editor Command]

Same as `DO N`.

17.13 MISCELLANEOUS COMMANDS

`NIL` [Editor Command]

Unless preceded by `F` or `BF`, is always a no-op. Thus extra right parentheses or square brackets at the ends of commands are ignored.

`CL` [Editor Command]

Clispifies the current expression (see page 16.17).

`DW` [Editor Command]

Dwimifies the current expression (see page 16.14).

`GET*` [Editor Command]

If the current expression is a comment pointer (see page 6.51), reads in the full text of the comment, and replaces the current expression by it.

`(* . x)` [Editor Command]

`x` is the text of a comment. `*` ascends the edit chain looking for a ''safe'' place to insert the comment, e.g., in a `COND` clause, after a `PROG` statement, etc., and inserts `(* . x)` *after* that point, if possible, otherwise before. For example, if the current expression is `(FACT (SUB1 N))` in

```
[COND
   ((ZEROP N) 1)
   (T (ITIMES N (FACT (SUB1 N]
```

`(* CALL FACT RECURSIVELY)` would insert `(* CALL FACT RECURSIVELY)` *before* the `ITIMES` expression.[24]

[24] If inserted after the `ITIMES`, the comment would then be (incorrectly) returned as the value of the `COND`. However, if the `COND` was itself a `PROG` statement, and hence its value was not being used, the comment could be (and would be) inserted after the `ITIMES` expression.

* does not change the edit chain, but UNFIND is set to where the comment was actually inserted.

GETD [Editor Command]

Essentially ''expands'' the current expression in line: (1) if (CAR of) the current expression is the name of a macro, expands the macro in line; (2) if a CLISP word, translates the current expression and replaces it with the translation; (3) if CAR is the name of a function for which the editor can obtain a symbolic definition, either in-core or from a file, substitutes the argument expressions for the corresponding argument names in the body of the definition and replaces the current expression with the result; (4) if CAR of the current expression is an open lambda, substitutes the arguments for the corresponding argument names in the body of the lambda, and then removes the lambda and argument list.

(MAKEFN (FN . ACTUALARGS) ARGLIST N_1 N_2) [Editor Command]

The inverse of GETD: makes the current expression into a function. FN is the function name, ARGLIST its arguments. The argument names are substituted for the corresponding argument values in ACTUALARGS, and the result becomes the body of the function definition for FN. The current expression is then replaced with (FN . ACTUALARGS).

If N_1 and N_2 are supplied, (N_1 THRU N_2) is used rather than the current expression; if just N_1 is supplied, (N_1 THRU -1) is used.

If ARGLIST is omitted, MAKEFN will make up some arguments, using elements of ACTUALARGS, if they are literal atoms, otherwise arguments selected from (X Y Z A B C ...), avoiding duplicate argument names.

Example: If the current expression is (COND ((CAR X) (PRINT Y T)) (T (HELP))), then (MAKEFN (FOO (CAR X) Y) (A B)) will define FOO as (LAMBDA (A B) (COND (A (PRINT B T)) (T (HELP)))) and then replace the current expression with (FOO (CAR X) Y).

(MAKE ARGNAME EXP) [Editor Command]

Makes the value of ARGNAME be EXP in the call which is the current expression, i.e. a ?= command following a MAKE will always print ARGNAME = EXP. For example:

```
*P
(JSYS)
*?=
JSYS[N;AC1,AC2,AC3,RESULTAC]
*(MAKE N 10)
*(MAKE RESULTAC 3)
*P
(JSYS 10 NIL NIL NIL 3)
```

Q [Editor Command]

Quotes the current expression, i.e. MBD QUOTE.

D [Editor Command]

Deletes the current expression, then prints new current expression, i.e. (:) I P.

17.14 COMMANDS THAT EVALUATE

E [Editor Command]

Causes the editor to call the Interlisp executive LISPX giving it the next input as argument. Example:

```
*E BREAK(FIE FUM)
(FIE FUM)
*E (FOO)

(FIE BROKEN)
:
```

Note: E only works when when typed in, e.g, (INSERT D BEFORE E) will treat E as a pattern, and search for E.

(E x) [Editor Command]

Evaluates x, i.e., performs (EVAL x), and prints the result on the terminal.

(E x T) [Editor Command]

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 x_1 ... x_N) [Editor Command]

Executes the *editor command* (c y_1 ... y_N) where y_i = (EVAL x_i). If c is not an atom, c is evaluated also.

Examples:

(I 3 (GETD 'FOO)) will replace the 3rd element of the current expression with the de nition of FOO.

(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.

(I (COND ((NULL FLG) '-1) (T 1)) FOO), if FLG is NIL, inserts the value of FOO before the rst element of the current expression, otherwise replaces the rst element by the value of FOO.

The I command sets an internal ag to indicate to the structure modi cation commands *not* to copy expression(s) when inserting, replacing, or attaching.

EVAL [Editor Command]

Does an EVAL of the current expression.

Note that EVAL, line-feed, and the GO command together e ectively allow the user to ''single-step'' a program through its symbolic de nition.

GETVAL [Editor Command]

Replaces the current expression by the result of evaluating it.

(## COM_1 COM_2 ... COM_N) [NLambda NoSpread Function]

An nlambda, nospread function (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. [25]

Example: (I R '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 X_1 ... X_M) [Editor Command]

Each X_i is evaluated and its value is 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. [26]

(COMSQ COM_1 ... COM_N) [Editor Command]

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 'COMSQ X)) where X computed the list of commands, e.g., (COMS (CONS 'COMSQ (GETP FOO 'COMMANDS))).

17.15 COMMANDS THAT TEST

(IF X) [Editor Command]

Generates an error *unless* the value of (EVAL X) is true. In other words, 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 (IPLUS (E (OR (NUMBERP (## 3)) (ERROR!)) T)) specifies the first IPLUS 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 (IPLUS (IF (NUMBERP (## 3)))).

---

[25] The A, B, :, INSERT, REPLACE, and CHANGE commands make special checks for ## forms in the expressions used for inserting or replacing, and use a copy of ## form instead (see page 17.26). Thus, (INSERT (## 3 2) AFTER 1) is equivalent to (I INSERT (COPY (## 3 2)) 'AFTER 1).

[26] The editor command NIL is a no-op, see page 17.43.

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

(IF x COMS 1 COMS 2) [Editor Command]

If (EVAL x) is true, execute COMS 1; if (EVAL x) causes an error or is equal to NIL, execute COMS 2.

Thus IF is equivalent to

```
(COMS (CONS 'COMSQ
            (COND
               ((CAR (NLSETQ (EVAL X)))
                  COMS 1)
               (T  COMS 2))))
```

For example, the command (IF (READP T) NIL (P)) will print the current expression provided the input bu er is empty.

(IF x COMS 1) [Editor Command]

If (EVAL x) is true, execute COMS 1; otherwise generate an error.

(LP COMS 1 COMS N) [Editor Command]

Repeatedly executes COMS 1 COMS N 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. [27]

When an error occurs, LP prints N OCCURRENCES where N is the number of times the commands were successfully executed. The edit chain is left as of the last complete successful execution of COMS 1 COMS N.

(LPQ COMS 1 COMS N) [Editor Command]

Same as LP but does not print the message 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. MAXLOOP can be set to NIL, which is equivalent to setting it to in nity. Since the edit chain is left as of the last successful completion of the loop, the user can simply continue the LP command with REDO (page 8.7).

(SHOW x) [Editor Command]

x is a list of patterns. SHOW does a LPQ printing all instances of the indicated expression(s), e.g. (SHOW FOO (SETQ FIE &)) will print all FOO's and all (SETQ FIE &)'s. Generates an error if there aren't any instances of the expression(s).

[27] 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))).

(EXAM x) [Editor Command]

Like SHOW except calls the editor recursively (via the TTY: command, see page 17.40) on each instance of the indicated espression(s) so that the user can examine and/or change them.

(ORR COMS$_1$ COMS$_N$) [Editor Command]

ORR begins by executing COMS$_1$, a list of commands. If no error occurs, ORR is nished. 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 o the end'', ORR generates an error. Otherwise, the edit chain is left as of the completion of the rst command list which executes without an 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 above example 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))).

17.16 EDIT MACROS

Many of the more sophisticated branching commands in the editor, such as ORR, IF, etc., are most often used in conjunction with edit macros. The macro feature permits the user to de ne new commands and thereby expand the editor's repertoire, or rede ne existing commands.[28] Macros are de ned by using the M command:

(M c COMS$_1$ COMS$_N$) [Editor Command]

For c an atom, M de nes c as an atomic command. If a macro is rede ned, its new de nition replaces its old. Executing c is then the same as executing the list of commands COMS$_1$ COMS$_N$.

For example, (M BP BK UP P) will de ne BP as an atomic command which does three things, a BK, and UP, and a P. Macros can use commands de ned by macros as well as built in commands in their de nitions. For example, suppose Z is de ned by (M Z -1 (IF (READP T) NIL (P))), i.e., Z does a -1, and then if nothing has been typed, a P. Now we can de ne ZZ by (M ZZ -1 Z), and ZZZ by (M ZZZ -1 -1 Z) or (M ZZZ -1 ZZ).

Macros can also de ne list commands, i.e., commands that take arguments.

(M (c) (ARG$_1$ ARG$_N$) COMS$_1$ COMS$_M$) [Editor Command]

c an atom. M de nes c as a list command. Executing (c E$_1$ E$_N$) is then performed by substituting E$_1$ for ARG$_1$, E$_N$ for ARG$_N$ throughout COMS$_1$ COMS$_M$, and then executing COMS$_1$ COMS$_M$.

---

[28] To refer to the original de nition of a built-in command when rede ning it via a macro, use the ORIGINAL command (page 17.50).

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'', as with spread vs. nospread functions. 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) ARG COMS$_1$ ... COMS$_M$) [Editor Command]
If C, ARG are both atoms, this defines C as a list command. Executing (C E$_1$ ... E$_N$) is performed by substituting (E$_1$ ... E$_N$), i.e., CDR of the command, for ARG throughout COMS$_1$ ... COMS$_M$, and then executing COMS$_1$ ... COMS$_M$.

For example, the command 2ND (page 17.18), could 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 as atomic commands 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 it is preceded by an F. Thus (INSERT -- BEFORE BP) would not search for BP, but instead perform a BK, and 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))
```

Since this version of SW sets FOO and FIE, using SW may have undesirable side effects, especially when the editor was called from deep in a computation, we would have to 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$_1$ ... COMS$_N$) [Editor Command]
Binds three dummy variables #1, #2, #3, (initialized to NIL), and then executes the edit commands COMS$_1$ ... COMS$_N$. 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 entire 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)))
```

(ORIGINAL COMS$_1$ ... COMS$_N$) [Editor Command]

Executes COMS$_1$ ... COMS$_N$ without regard to macro definitions. Useful for redefining a built in command in terms of itself., i.e. effectively allows user to ''advise'' edit commands.

User macros are stored on a list USERMACROS. The file package command USERMACROS (page 11.24), is available for dumping all or selected user macros.

17.17 UNDO

Each command that causes structure modification automatically adds an entry to the front of UNDOLST that contains the information required to restore all pointers that were changed by that command.

UNDO [Editor Command]

Undoes the last, i.e., most recent, structure modification command that has not yet been undone, and prints the name of that command, e.g., MBD UNDONE. 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 [Editor Command]

Undoes all modifications performed during this editing session, i.e. this call to the editor. As each command is undone, its name is printed a la UNDO. If there is nothing to be undone, !UNDO prints NOTHING SAVED.

Undoing an event containing an I, E, or S command will also undo the side effects of the evaluation(s), e.g., undoing (I 3 (/NCONC FOO FIE)) will not only restore the 3rd element but also restore FOO. Similarly, undoing an S command will undo the set. See the discussion of UNDO in page 8.11. (Note that if the I command was typed directly to the editor, /NCONC would automatically be substituted for NCONC as described in page 8.22.)

Since UNDO and !UNDO cause 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 commands he wants undone by identifying the corresponding entry on the history list. 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.

Whenever the user *continues* an editing session, the undo information of the previous session 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, the editor will type BLOCKED instead of NOTHING SAVED. Similarly, if the user executes several commands and then undoes them all, another UNDO or !UNDO will also cause BLOCKED to be typed.

UNBLOCK [Editor Command]

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

TEST [Editor Command]

Adds an undo-block at the front of UNDOLST.

Note that TEST together with !UNDO provide a ''tentative'' mode for editing, i.e., the user can perform a number of changes, and then undo all of them with a single !UNDO command.

(UNDO EventSpec) [Editor Command]

EventSpec is an event specification (see page 8.5). Undoes the indicated event on the history list. In this case, the event does not have to be in the current editing session, even if the previous session has not been unblocked as described above. However, the user does have to be editing the same expression as was being edited in the indicated event.

If the expressions differ, the editor types the warning message ''different expression'', and does not undo the event. The editor enforces this to avoid the user accidentally undoing a random command by giving the wrong event specification.

17.18 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.[29] If a location specification is being

[29]Since EDITDEFAULT is part of the edit block, the user cannot advise or redefine it as a means of augmenting or extending the editor. However, the user can accomplish this via EDITUSERFN. If the value of the variable EDITUSERFN is T, EDITDEFAULT calls the function EDITUSERFN giving it the command as an argument. If EDITUSERFN returns a non-NIL value, its value is interpreted as a single command and executed. Otherwise, the error correction procedure described below is performed.

executed, an internal flag informs EDITDEFAULT to treat the command as though it had been preceded by an F.

If the command is a list, an attempt is made to perform spelling correction on CAR of the command [30] using EDITCOMSL, a list of all list edit commands. [31] If spelling correction is successful, the correct command name is RPLACAed into the command, and the editor continues by executing the command. In other words, if the user types (LP F PRINT (MBBD AND (NULL FLG))), only one spelling correction will be necessary to change MBBD to MBD. If spelling correction is not successful, an error is generated.

If the command is atomic, the procedure followed is a little more elaborate.

(1) If the command is one of the list commands, i.e., a member of EDITCOMSL, and there is additional input on the same terminal line, treat the entire line as a single list command. [32] Thus, the user may omit parentheses for any list command typed in at the top level (provided the command 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 terminal line, an error is generated, e.g.

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

MOVE ?
*
```

If the command is on EDITCOMSL, and *not* typed in directly, e.g., it appears as one of the commands in a LP command, the procedure is similar, with the rest of the command stream at that level being treated as ''the terminal line'', e.g. (LP F (COND (T &)) XTR 2 2). [33]

(2) If the command was typed in and the first character in the command is an 8, treat the 8 as a mistyped left parenthesis, and and the rest of the line as the arguments to the command, e.g.,

```
*P
(COND (& &) (T &))
```

---

[30] unless DWIMFLG= NIL.

[31] When a macro is defined via the M command, the command name is added to EDITCOMSA or EDITCOMSL, depending on whether it is an atomic or list command. The USERMACROS file package command is aware of this, and provides for restoring EDITCOMSA and EDITCOMSL.

[32] The line is read using READLINE (page 8.30). Thus the line can be terminated by a square bracket, or by a carriage return not preceded by a space.

[33] Note that if the command is being executed in location context, EDITDEFAULT does not get this far, e.g., (MOVE TO AFTER COND XTR 3) will search for XTR, *not* execute it. However, (MOVE TO AFTER COND (XTR 3)) will work.

```
*8-2 (Y (RETURN Z)))
=(-2
*P
(COND (Y &) (& &) (T &))
```

(3) If the command was typed in, is the name of a function, and is followed by NIL or a list CAR of which is not an edit command, assume the user forgot to type E and means to apply the function to its arguments, type =E and the function name, and perform the indicated computation, e.g.

```
*BREAK(FOO)
=E BREAK
(FOO)
*
```

(4) If the last character in the command is P, and the rst N-1 characters comprise a number, assume that the user intended two commands, e.g.,

```
*P
(COND (& &) (T &))
*0P
=0 P
(SETQ X (COND & &))
```

(5) Attempt spelling correction using EDITCOMSA, and if successful, execute the corrected command.

(6) If there is additional input on the same line, or command stream, spelling correct using EDITCOMSL as a spelling list, e.g.,

```
*MBBD SETQ X
=MBD
*
```

(6) Otherwise, generate an error.

17.19 EDITOR FUNCTIONS

(EDITF NAME COM$_1$ COM$_2$ ... COM$_N$) [NLambda NoSpread Function]

Nlambda, nospread function for EDITing a Function. NAME is the name of the function, COM$_1$, COM$_2$, ..., COM$_n$ are (optional) edit commands.

The value of EDITF is NAME .

The action of EDITF is somewhat complicated:

(1) In the most common case, if the de nition of NAME is an EXPR (not as a result of its being broken or advised), and EDITF simply performs (PUTD NAME (EDITE (GETD 'NAME) (LIST 'COM$_1$ 'COM$_2$ ... 'COM$_N$) 'NAME 'FNS)).

(2) If NAME is an EXPR by virtue of its being broken or advised, and the original de nition is also an EXPR, then the broken/advised de nition is given to EDITE to be edited (since any changes there will also a ect the original de nition because all changes are destructive). However, a warning message is printed to alert the user that he must rst position himself correctly before he can begin typing commands such as (-3 --), (N --), etc.

(3) If NAME is an EXPR by virtue of its being broken or advised, the original de nition is not an EXPR, there is no EXPR property, and the le package ''knows'' which le NAME is contained in (see EDITLOADFNS?, page 17.58), then the EXPR de nition of NAME is loaded onto its property list as described below, and the EDITF proceeds to the next possibility. Otherwise, a warning message is printed, and the edit proceeds, e.g., the user may have called the editor to examine the advice on a SUBR.

(4) If NAME is an EXPR by virtue of its being broken or advised, the original de nition is not an EXPR, and there is an EXPR property, then the function is unbroken/unadvised (latter only with user's approval, since the user may really want to edit the advice) and EDITF proceeds to the next possibility.

(5) If NAME is not an EXPR, but has an EXPR property, EDITF prints PROP, and performs (EDITE (GETPROP 'NAME 'EXPR) (LIST 'COM$_1$ 'COM$_2$ ... 'COM$_N$) 'NAME 'PROP). In this case, if the edit completes and no changes have been made, EDITE prints NOT CHANGED, SO NOT UNSAVED. If changes were made, but the value of DFNFLG (page 5.9) is PROP, EDITE prints CHANGED, BUT NOT UNSAVED. Otherwise if changes were made, EDITE prints UNSAVED and does an UNSAVEDEF.

(6) If NAME is neither an EXPR nor has an EXPR property, and the le package ''knows'' which le NAME is contained in (see EDITLOADFNS?, page 17.58), the EXPR de nition of NAME is automatically loaded (using LOADFNS) onto the EXPR property, and EDITE proceeds as described above.[34] In addition, if NAME is a member of a block, the user will be asked whether he wishes the rest of the functions in the block to be loaded at the same time.[35]

(7) If NAME is neither an EXPR nor has an EXPR property, but it does have a de nition, EDITF generates an NAME NOT EDITABLE error.

(8) If NAME is neither de ned, nor has an EXPR property, but its top level value is a list, EDITF assumes the user meant to call EDITV, prints =EDITV, calls EDITV and returns. Similarly, if NAME has a non-NIL property list, EDITF prints =EDITP, calls EDITP and returns.

---

[34]Because of the existence of the le map (see page 11.38), this operation is extremely fast, essentially requiring only the time to perform the READ to obtain the actual de nition.

[35]The editor's behaviour in this case is controlled by the value of EDITLOADFNSFLG, which is a dotted pair of two ags. The CAR of EDITLOADFNSFLG controls the loading of the function, and the CDR controls the loading of the block. A value of NIL for either ag means ''load but ask rst,'' a value of T means ''don't ask, just do it'' and anything else means ''don't ask, don't do it.'' The initial value of EDITLOADFNSFLG is (T . NIL), meaning to load the function without asking, and ask about loading the block.

(9) If NAME is neither a function, nor has an EXPR property, nor a top level value that is a list, nor a non-NIL property list, EDITF attempts spelling correction using the spelling list USERWORDS,[36] and, if successful, goes back to the beginning.

(10) Otherwise, EDITF generates an NAME NOT EDITABLE error.

In all cases, if a function is edited, and changes were made, the function is time-stamped (by EDITE), which consists of inserting a comment of the form (* USERS-INITIALS DATE) (see page 17.60). If the function was already time-stamped, then only the date is changed.

(EDITFNS NAME COM$_1$ COM$_2$ COM$_N$) [NLambda NoSpread Function]

An nlambda, nospread function, used to perform the same editing operations on several functions. NAME is evaluated to obtain a list of functions.[37] COM$_1$, COM$_2$, , COM$_N$ are (optional) edit commands. EDITFNS maps down the list of functions, prints the name of each function, and calls the editor (via EDITF) on that function. The value of EDITFNS is NIL.

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 ERRORSET protected, so that if the editing of one function causes an error, EDITFNS will proceed to the next function. In particular, if an error occurred while editing a function via its EXPR property, the function would not be unsaved. Thus in the above example, if one of the functions did not contain a FIE, the R command would cause an error, it would not be unsaved, and editing would continue with the next function.

(EDITV NAME COM$_1$ COM$_2$ COM$_N$) [NLambda NoSpread Function]

Similar to EDITF, for editing values of variables.

The value of EDITV is the name of the variable whose value was edited.

If NAME is a list, it is evaluated and its value given to EDITE, e.g., (EDITV (CDR (ASSOC 'FOO DICTIONARY))). In this case, the value of EDITV is T.

However, for most applications, NAME is a variable name, i.e., atomic, as in EDITV(FOO). If the value of this variable is NOBIND, EDITV checks to see if it is the name of a function, and if so, assumes the user meant to call EDITF, prints =EDITF, calls EDITF and returns. Otherwise, EDITV attempts spelling correction using the list USERWORDS.[38] Then EDITV will call EDITE on the value of NAME (or the corrected spelling thereof), and TYPE = VARS. Thus, if the value of FOO is NIL, and the user performs (EDITV FOO), no spelling correction will occur, since FOO is the name of a variable in the user's system, i.e., it has a value. However, EDITE will generate an error, since FOO's value is not a list, and hence

---

[36]Unless DWIMFLG= NIL. Spelling correction is performed using the function MISSPELLED? (page 15.18). If NAME = NIL, MISSPELLED? returns the last ''word'' referenced, e.g., by DEFINEQ, EDITF, PRETTYPRINT etc. Thus if the user de nes FOO and then types (EDITF), the editor will assume he meant FOO, type =FOO, and then type EDIT.

[37]If NAME is atomic, and its value is not a list, and it is the name of a le, (FILEFNSLST 'NAME) will be used as the list of functions to be edited.

[38]Unless DWIMFLG= NIL. MISSPELLED? is also called if NAME is NIL, so that (EDITV) will edit LASTWORD.

not editable. If the user performs (EDITV FOOO), where the value of FOOO is NOBIND, and FOO is on the user's spelling list, the spelling corrector will correct FOOO to FOO. Then EDITE will be called on the value of FOO. Note that this may still result in an error if the value of FOO is not a list.

(EDITP NAME COM$_1$ COM$_2$ ... COM$_N$) [NLambda NoSpread Function]

Similar to EDITF for editing property lists. If the property list of NAME is NIL, EDITP attempts spelling correction using USERWORDS. Then EDITP calls EDITE on the property list of NAME, (or the corrected spelling thereof), with TYPE = PROPLST. When (if) EDITE returns, EDITP calls SETPROPLIST on NAME with the value returned.

The value of EDITP is the atom whose property list was edited.

(EDITE EXPR COMS ATM TYPE IFCHANGEDFN) [Function]

Edits the expression, EXPR, by calling EDITL on (LIST EXPR) and returning the last element of the value returned by EDITL. Generates an error if EXPR is not a list.

ATM and TYPE are for use in conjunction with the le package. If supplied, ATM is the *name* of the object that EXPR is associated with, and TYPE describes the association (i.e., TYPE corresponds to the TYPE argument of MARKASCHANGED, page 11.11.) For example, if EXPR is the de nition of FOO, ATM = FOO and TYPE = FNS. When EDITE is called from EDITP, EXPR is the property list of ATM, and TYPE = PROPLST, etc..

EDITE calls EDITL to do the editing (described below). Upon return, if both ATM and TYPE are non-NIL, ADDSPELL is called to add ATM to the appropriate spelling list. Then, if EXPR was changed, [39] and the value of IFCHANGEDFN is not NIL, the value of IFCHANGEDFN is applied to the arguments ATM, EXPR, TYPE, and a ag which is T for normal edits from editor, NIL for calls that were aborted via control-D or STOP. Otherwise, if EXPR was changed, and the value of IFCHANGEDFN is NIL, and TYPE is not NIL, MARKASCHANGED (page 11.11) is called on ATM and TYPE. EDITE uses RESETSAVE to insure that IFCHANGEDFN and MARKASCHANGED are called if any change was made even if editing is subsequently aborted via control-D. (In this case, the fourth argument to IFCHANGEDFN wil be NIL.)

(EDITL L COMS ATM MESS EDITCHANGES) [Function]

EDITL *is* the editor. Its rst argument is the edit chain, and its value is an edit chain, namely the value of L at the time EDITL is exited. [40]

COMS is an optional list of commands. For interactive editing, coms is NIL. In this case, EDITL types EDIT (or MESS, if it not NIL) and then waits for input from terminal. All input is done with EDITRDTBL as the readtable. Exit occurs only via an OK, STOP, or SAVE command.

---

[39] For TYPE = FNS or TYPE = PROP, i.e., calls from EDITF, EDITE performs some additional operations as described earlier under EDITF.

[40] L is a SPECVAR, and so can be examined or set by edit commands. For example, ^ is equivalent to (E (SETQ L (LAST L)) T). However, the user should only manipulate or examine L directly as a last resort, and then with caution.

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.

ATM is optional. On calls from EDITF, it is the name of the function being edited; on calls from EDITV, the name of the variable, and calls from EDITP, the atom whose 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.

EDITCHANGES is used for communicating with EDITE.

(EDITL0 L COMS MESS _) [Function]

Like EDITL, except it does not rebind or initialize the editor's various state variables, such as LASTAIL, UNFIND, UNDOLST, MARKLST, etc. Should only be called when already under a call to EDITL.

(EDIT4E PAT X _) [Function]

The editor's pattern match routine. Returns T, if PAT matches X. See page 17.13 for definition of ''match''.

Note: Before each search operation in the editor begins, the entire pattern is scanned for atoms or strings containing $s (<esc>s). Atoms or strings containing $s are replaced by lists of the form ($), and atoms or strings ending in double $s are replaced by lists of the form ($$). Thus from the standpoint of EDIT4E, single and double $ patterns are detected by (CAR PAT) being the atom $ (<esc>) or the atom $$ (<esc><esc>). Therefore, if the user wishes to call EDIT4E directly, he must first convert any patterns which contain atoms or strings containing $s to the form recognized by EDIT4E. This is done with the function EDITFPAT:

(EDITFPAT PAT _) [Function]

Makes a copy of PAT with all atoms or strings containing $s (<esc>s) converted to the form expected by EDIT4E.

(EDITFINDP X PAT FLG) [Function]

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.

(ESUBST NEW OLD EXPR ERR ORFL G CHARFL G) [Function]

Equivalent to performing (R OLD NEW) with EXPR as the current expression, i.e., the order of arguments is the same as for SUBST. Note that OLD and/or NEW can employ $s (<esc>s). The value of ESUBST is the modified EXPR . Generates an error if OLD not found in EXPR . If ERR ORFL G = T, also prints an error message of the form OLD ?.

If CHARFLG = T and no $s (<esc>s) are specified in NEW or OLD, it is equivalent to (RC OLD NEW). In other words, if CHARFLG = T, and no $s appear, ESUBST will supply them.

ESUBST is always undoable.

(EDITLOADFNS? FN STR ASKFLG FILES) [Function]

FN is the name of a function. EDITLOADFNS? returns the name of file FN is contained in, or NIL.

EDITLOADFNS? performs (WHEREIS FN FNS FILES) to obtain the name of the file(s) containing FN, if any (see page 11.10). If there is more than one file, EDITLOADFNS? asks the user to indicate which file. It then checks the FILEDATES property for each file to see if the version that was originally loaded still exists.[41] If the file that was *originally* loaded no longer exists, but there is a different version of the file on that directory, EDITLOADFNS? prints ‘‘****can't find FILENAME ’’, and then uses the version that it could find. Similarly, if the original version *is* found, but a newer version is also found, EDITLOADFNS? prints "****Note: FILENAME is not the newest version" and then uses the newest version.

Having decided which file the function is on, if ASKFLG = NIL, EDITLOADFNS? prints the value of STR followed by the name of the file, and returns the name of the file. If ASKFLG = T, EDITLOADFNS? calls ASKUSER giving (LIST FN STR FILENAME) as MESS, the message to be printed. If ASKUSER returns Y, EDITLOADFNS? returns the filename. If STR = NIL, "loading from" is used.

EDITLOADFNS? is used by the editor, LOADFNS (when the file name is not supplied), by PRETTYPRINT, and by DWIM.

(CHANGENAME FN FROM TO) [Function]

Replaces all occurrences of FROM by TO in the definition of FN. If FN is an EXPR, CHANGENAME performs (NLSETQ (ESUBST TO FROM (GETD FN))). If FN is *compiled*, CHANGENAME searches the literals of FN (and all of its compiler generated subfunctions), replacing each occurrence of FROM with TO. This will succeed even if FROM is called from FN via a linked call. In this case, the call will also be relinked to call TO instead.

The value of CHANGENAME is FN if at least one instance of FROM was found, otherwise NIL.

CHANGENAME is used by BREAK and ADVISE for changing calls to FN_1 to calls to FN_1-IN-FN_2.

The function EDITCALLERS provides a way of rapidly searching a file or entire set of files, even files not loaded into Interlisp or ‘‘noticed’’ by the file package, for the appearance of one or more key words (atoms) anywhere in the file.

[41] In the case that FILES= T and the WHEREIS package has been loaded (page 23.40), file(s) may be found that have not been loaded or otherwise noticed, and thus will not have FILEDATES property. In this case, EDITLOADFNS? does not do any version checks, but simply uses the latest version.

(EDITCALLERS ATOMS FILES COMS) [Function]

Uses FFILEPOS to search the file(s) FILES for occurrences of the atom(s) ATOMS . It then calls EDITE on each of those objects,[42] performing the edit commands COMS . If COMS = NIL, then (EXAM . ATOMS) is used. Both ATOMS and FILES may be single atoms. If FILES is NIL, FILELST is used. Elements on ATOMS may contain $s (<esc>s).

EDITCALLERS prints the name of each file as it searches it, and when it finds an occurrence of one of ATOMS , it prints out either the name of the containing function or, if the atom occurred outside a function definition, it prints out the byte position that the atom was found.

EDITCALLERS will read in and use the filemap of the file. In the case that the editor is actually called, EDITCALLERS will LOADFROM the file if the file has not previously been noticed.

(FINDCALLERS ATOMS FILES) [Function]

Like EDITCALLERS, except does not call the editor, but instead simply returns the list of files that contain one of ATOMS .

(EDITRACEFN COM) [Function]

Is available to help the user debug complex edit macros, or subroutine calls to the editor. If EDITRACEFN is set to T, the function EDITRACEFN is called whenever a command that was not typed in by the user is about to be executed, giving it that command as its argument. However, the TRACE and BREAK options described below are probably sufficient for most applications.

If EDITRACEFN is set to TRACE, the name of the command and the current expression are printed. If EDITRACEFN = BREAK, the same information is printed, and the editor goes into a break. The user can then examine the state of the editor.

EDITRACEFN is initially NIL.

(SETTERMCHARS NEXTCHAR BKCHAR LASTCHAR UNQUOTECHAR 2CHAR PPCHAR) [Function]

Used to set up the immediate read macros used by the editor, as well as the control-Y read macro (page 6.39). NEXTCHAR , BKCHAR , LASTCHAR , 2CHAR and PPCHAR specify which control character should perform the edit commands NXP, BKP, -1P, 2P and PP*, respectively; UNQUOTECHAR corresponds to control-Y. For each non-NIL argument, SETTERMCHARS makes the corresponding control character have the indicated function. The arguments to SETTERMCHARS can be character codes, the control characters themselves, or the alphabetic letters corresponding to the control characters.

If an argument to SETTERMCHARS is currently assigned as an interrupt character, it cannot be a read macro (since the reader will never see it); SETTERMCHARS prints a message to that effect and makes no change to the control character. However, if SETTERMCHARS is given a list as one of its arguments, it uses CAR of the list even if the character is an interrupt. In this case, if CADR of the list is non-NIL, SETTERMCHARS reassigns the interrupt function to CADR. For example, if control-X is an interrupt,

[42] EDITCALLERS uses GETDEF (page 11.17) to obtain the ''definition'' for each object. When EDITE returns, if a change was made, PUTDEF is called to store the changed object.

(SETTERMCHARS '(X W)) assigns control-W the interrupt control-X had, and makes control-X be the NEXTCHAR operator.

As part of the greeting operation, SETTERMCHARS is applied to the value of EDITCHARACTERS, which is initially (J X Z Y N) in Interlisp-D and in Interlisp-10 under Tenex, (J A L Y K) under Tops-20 (control-J is line-feed). SETTERMCHARS is called *after* the user's init file is loaded, so it works to reset EDITCHARACTERS in the init file; alternatively, SETTERMCHARS can be called explicitly.

17.20 TIME STAMPS

Whenever a function is edited, and changes were made, the function is time-stamped (by EDITE), which consists of inserting a comment of the form (* USERS-INITIALS DATE). USERS-INITIALS is the value of the variable INITIALS. After greeting, or following a SYSIN, the function SETINITIALS is called. SETINITIALS searches INITIALSLST, a list of elements of the form (USERNAME . INITIALS) or (USERNAME FIRSTNAME INITIALS). If the user's name is found, INITIALS is set accordingly. If the user's name is *not* found on INITIALSLST, INITIALS is set to the value of DEFAULTINITIALS, initially edited:. Thus, the default is to always time stamp. To suppress time stamping, the user must either include an entry of the form (USERNAME) on INITIALSLST, or set DEFAULTINITIALS to NIL before greeting, i.e. in his user profile, or else, *after* greeting, explicitly set INITIALS to NIL.

If the user wishes his functions to be time stamped with his initials when edited, he should include a file package command command of the form (ADDVARS (INITIALSLST (USERNAME . INITIALS))) in the user's INIT.LISP file (see page 14.5).

The following three functions may be of use for specialized applications with respect to time-stamping: (FIXEDITDATE EXPR) which, given a lambda expression, inserts or smashes a time-stamp comment; (EDITDATE? COMMENT) which returns T if COMMENT is a time stamp; and (EDITDATE OLD ATE INITLS) which returns a new time-stamp comment. If OLD ATE is a time-stamp comment, it will be reused.