16. LIST STRUCTURE EDITOR

Many important objects such as function definitions, property lists, and variable values are represented as list structures. The Interlisp-D environment includes a list structure editor to allow the user to rapidly and conveniently modify list structures.

The list structure editor is most often used to edit function definitions. Editing function definitions "in core" is a facility not offered by many lisp systems, where typically the user edits external text files containing function definitions, and then loads them into the environment. In Interlisp, function definitions are edited in the environment, and written to an external file using the file package (page 17.1), which provides a complex set of tools for managing function definitions.

Early implementations of Interlisp using primitive terminals offered a teletype-oriented editor, which included a large set of cryptic commands for printing different parts of a list structure, searching a list, replacing elements, etc. Interlisp-D includes an extended, display-oriented version of the teletype list structure editor, called DEdit. 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 15.6) calls the teletype editor to insert a function break within the body of a function. DEdit also provides facilities for using the complex teletype editor commands from within DEdit. By calling the function **EDITMODE** (page 16.4) it is possible to set the "default editor" (**TELETYPE** or **DISPLAY**) called by Masterscope, the break package, etc.

This chapter documents both DEdit and the teletype list structure editor (sometimes referred to as "Edit"). The first part documents DEdit, the most commonly used editor of the two. Then, there are a large number of sections describing the commands of the older teletype editor. Most users will only need to reference the DEdit documentation.

16.1 DEdit

DEdit is a structure oriented, modeless, display based editor for objects represented as list structures, such as functions, property

lists, data values, etc. DEdit is an integral part of the standard Interlisp-D environment.

DEdit is designed to be the user's primary editor for programs and data. To that end, it has incorporated the interfaces of the (older) teletype oriented Interlisp editor so the two can be used interchangeably. In addition, the full power of the teletype editor, and indeed the full Interlisp system itself, is easily accessible from within DEdit.

DEdit is structure, rather than character, oriented to facilitate selecting and operating on pieces of structure as objects in their own right, rather than as collections of characters. However, for the occasional situation when character oriented editing is appropriate, DEdit provides access to the Interlisp-D text editing facilities. DEdit is modeless, in that all commands operate on previously selected arguments, rather than causing the behavior of the interface to change during argument specification.

16.1.1 Calling DEdit

DEdit is normally called using the following functions:

(DF *FN NEW?***)** [NLambda NoSpread Function]

Calls DEdit on the definition of the function *FN*. **DF** handles exceptional cases (the function is broken or advised, the expr definition is on the property list, the function needs to be loaded from a file, etc.) the same as **EDITF** (see page 16.68).

If **DF** is called on a name with no function definition, the user is prompted with "**No FNS defn for** *FN*. **Do you wish to edit a dummy defn?**". If the user confirms by typing **Yes**, a "blank" definition (stored on the variable **DUMMY-EDIT-FUNCTION-BODY**) is displayed in the Dedit window. If any changes are made, on exit from the editor, the definition will be installed as the name's function definition. Exiting the editor with the **STOP** command will prevent any changes to the function definition.

If **DF** is called with a second arg of **NEW**, as in **(DF** *FNNAME* **NEW)**, a blank definition will be edited whether the function already has a definition or not.

(DV *VAR***)** [NLambda NoSpread Function]

Calls DEdit on the value of the variable *VAR*.

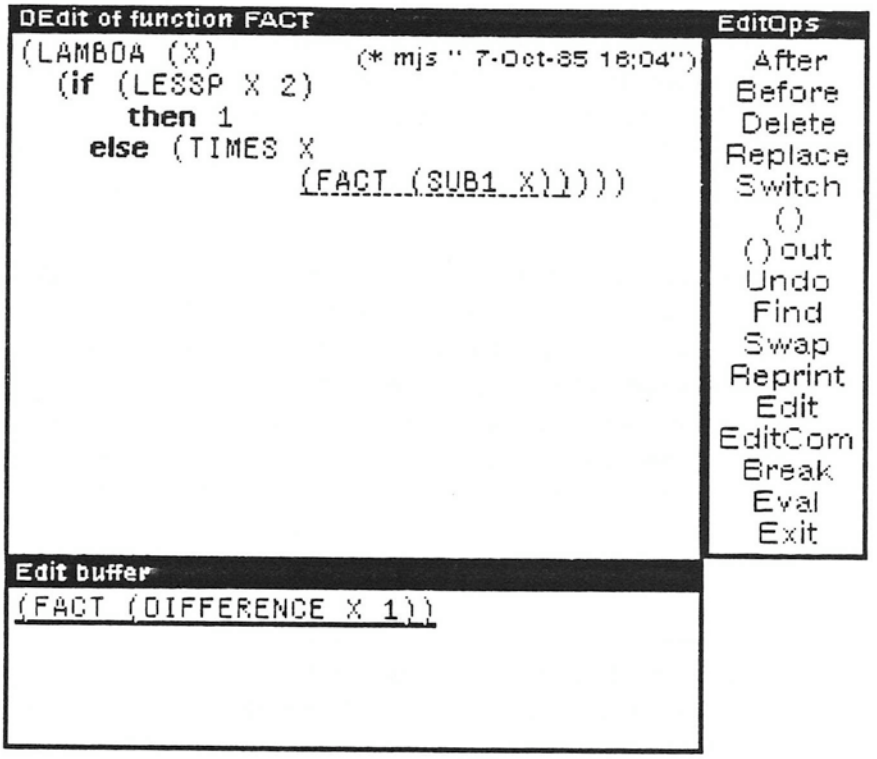
(DP *NAME PROP***)** [NLambda NoSpread Function]

Calls DEdit on the property *PROP* of the atom *NAME*. If *PROP* is not given, the whole property list of *NAME* is edited.

(DC *FILE*) [NLambda NoSpread Function]

Calls DEdit on the file package commands (or filecoms, see page 17.32) for the file *FILE*.

When DEdit is called for the first time, it prompts for an edit window, which is preserved and reused for later DEdits, and pretty prints the expression to be edited therein. (Note: The DEdit pretty printer ignores user **PRETTYPRINTMACROS** because they do not provide enough structural information during printing to enable selection.) The expression being edited can be scrolled by using a standard Interlisp-D scroll bar on the left edge of the window. DEdit adds an edit command menu, which remains active throughout the edit, on the right edge of the edit window. If anything is typed by the user, an "edit buffer" window is positioned below the edit window. Below is a picture of a Dedit window, displaying the function definition for **FACT**:



While Dedit is running, it yields control so that background activities, such as mouse commands in other windows, continue to be performed.

(RESETDEDIT) [Function]

Completely reinitializes DEdit. Closes all DEdit windows, so that the user must specify the window the next time DEdit is envoked. **RESETDEDIT** is also used to make DEdit recognize the new values

of variables such as **DEDITTYPEINCOMS** (page 16.12), when the user changes them.

DEdit is normally installed as the default editor for all editing operations, including those invoked by other subsystems, such as the Programmer's Assistant and Masterscope. DEdit provides functions **EF**, **EV** and **EP** (analogous to the corresponding **D***x* functions) for conveniently accessing the teletype editor from within a DEdit context, e.g. from under a call to DEdit or if DEdit is installed as the default editor.

The default editor may be set with **EDITMODE**:

(EDITMODE *NEWMODE***)** [Function]

If *NEWMODE* is non-**NIL**, sets the default editor to be DEdit (if *NEWMODE* is **DISPLAY**), or the teletype editor (if *NEWMODE* is **TELETYPE**). Returns the previous setting.

DEdit operates by providing an alternative, plug-compatible definition of **EDITL** (**DEDITL**). The normal user entries operate by redefining **EDITL** and then calling the corresponding teletype editor function (i.e., **DF** calls **EDITF** etc). Thus, the normal teletype editor file package, spelling correction, etc. behavior is obtained.

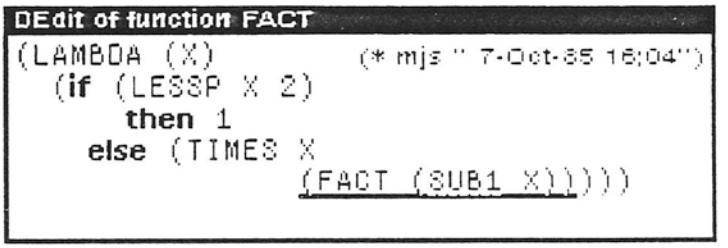
If teletype editor commands are specified in a call to **DEDITL** (e.g., in calls to the editor from Masterscope), **DEDITL** will pass those commands to **EDITL**, after having placed a **TTY:** entry on **EDITMACROS** which will cause DEdit to be invoked if any interaction with the user is called for. In this way, automatic edits can be made completely under program control, yet DEdit's interactive interface is available for direct user interaction.

16.1.2 Selecting Objects and Lists

Selection in a DEdit window is as follows: the **LEFT** button selects the object being directly pointed at; the **MIDDLE** button selects the containing list; and the **RIGHT** button extends the current selection to the lowest common ancestor of that selection and the current position. The only things that may be pointed at are atomic objects (literal atoms, numbers, etc) and parentheses, which are considered to represent the list they delimit. White space is not selectable or editable.

When a selection is made, it is pushed on a selection stack which will be the source of operands for DEdit commands. As each new selection pushes down the selections made before it, this stack can grow arbitrarily deep, so only the top two selections on the stack are highlighted on the screen. This highlighting is done by

underscoring the topmost (most recent) selection with a solid black line and the second topmost selection with a dashed line. The patterns used were chosen so that their overlappings would be both visible and distinct, since selecting a sub-part of another selection is quite common. For example, in the picture below, the last selection is the list **(FACT (SUB1 X))**, and the previous selection is the single litatom **SUB1**:



Because one can invoke DEdit recursively, there may be several DEdit windows active on the screen at once. This is often useful when transferring material from one object to another (as when reallocating functionality within a set of programs). Selections may be made in any active DEdit window, in any order. When there is more than one DEdit window, the edit command menu (and the type-in buffer, see below) will attach itself to the most recently opened (or current) DEdit window.

16.1.3 Typing Characters to DEdit

Characters may be typed at the keyboard at any time. This will create a type-in buffer window which will position itself under the current DEdit window and do a **LISPXREAD** (which must be terminated by a right parenthesis or a return) from the keyboard. During the read, any character editing subsystem (such as **TTYIN**) that is loaded can be used to do character level editing on the typein. When the read is complete, the typein will become the current selection (top of stack) and be available as an operand for the next command. Once the read is complete, objects displayed in the type-in buffer can be selected from, scrolled, or even edited, just like those in the main window.

One can also give some editing commands directly into the typein buffer. Typing control-Z will interpret the rest of the line as a teletype editor command which will be interpreted when the line is closed. Likewise, "control-S *OLD NEW*" will substitute *NEW* for *OLD* and "control-F *X*" will find the next occurrence of *X*.

16.1.4 Copy-Selection

Often, significant pieces of what one wishes to type can be found in an active DEdit window. To aid in transferring the

keystrokes that these objects represent into the typein buffer, DEdit supports copy-selection. Whenever a selection is made in the DEdit window with either shift key down (or the **COPY** key on the Xerox 1108), the selection made is not pushed on the selection stack, but is instead *unread* into the keyboard input (and hence shows up in the typein buffer). A characteristically different highlighting is used to indicate when copy selection (as opposed to normal selection) is taking place.

Note that copy-selection remains active even when DEdit is not. Thus one can unread particularly choice pieces of text from DEdit windows into the typescript window.

16.1.5 DEdit Commands

A DEdit command is invoked by selecting an item from the edit command menu. This can be done either directly, using the **LEFT** mouse button in the usual way, or by selecting a subcommand. Subcommands are less frequently used commands than those on the main edit command menu and are grouped together in submenus "under" the command on the main menu to which they are most closely related. For example, the teletype editor defines six commands for adding and removing parentheses (defined in terms of transformations on the underlying list structure). Of these six commands, only two (inserting and removing parentheses as a pair) are commonly used, so DEdit provides the other four as subcommands of the common two. The subcommands of a command are accessed by selecting the command from the commands menu with the **MIDDLE** button. This will bring up a menu of the subcommand options from which a choice can be made. Subcommands are flagged in the list below with the name of the top level command of which they are options.

If one has a large DEdit window, or several DEdit windows active at once, the edit command window may be far away from the area of the screen in which one is operating. To solve this problem, the DEdit command window is a "snuggle up" menu. Whenever the **TAB** key is depressed, the command window will move over to the current cursor position and stay there as long as either the **TAB** key remains down or the cursor is in the command window. Thus, one can "pull" the command window over, slide the cursor into it and then release the **TAB** key (or not) while one makes a command selection in the normal way. This eliminates a great deal of mouse movement.

Whenever a change is made, the prettyprinter reprints until the printing stablizes. As the standard pretty print algorithm is used and as it leaves no information behind on how it makes its choices, this is a somewhat heuristic process. The **Reprint**

command can be used to tidy the result up if it is not, in fact, "pretty".

All commands take their operands from the selection stack, and may push a result back on. In general, the rule is to select *target* selections first and *source* selections second. Thus, a **Replace** command is done by selecting the thing to be replaced, selecting (or typing) the new material, and then buttoning the **Replace** command in the command menu. Using *TOP* to denote the topmost (most recent) element of the stack and *NXT* the second element, the DEdit commands are:

After [DEdit Command]

Inserts a copy of *TOP* after *NXT*.

Before [DEdit Command]

Inserts a copy of *TOP* before *NXT*.

Delete [DEdit Command]

Deletes *TOP* from the structure being edited. (A copy of) *TOP* remains on the stack and will appear, selected, in the edit buffer.

Replace [DEdit Command]

Replaces *NXT* with a copy of *TOP* obtained by substituting a copy of *NXT* wherever the value of the atom **EDITEMBEDTOKEN** (initially, the **&** character) appears in *TOP*. This provides a facility like the **MBD** edit command (page 16.36), see Idioms below.

Switch [DEdit Command]

Exchanges *TOP* and *NXT* in the structure being edited.

() [DEdit Command]

Puts parentheses around *TOP* and *NXT* (which can, of course, be the same element).

(in [DEdit Command]

Subcommand of **()**. Inserts **(** before *TOP* (like the **LI** Edit command, page 16.41)

) in [DEdit Command]

Subcommand of **()**. Inserts **)** after *TOP* (like the **RI** Edit command, page 16.41)

() out [DEdit Command]

Removes parentheses from *TOP*.

(out [DEdit Command]

Subcommand of **() out**. Removes (from before *TOP* (like the **LO** Edit command, page 16.41)

) out [DEdit Command]

Subcommand of **() out**. Removes) from after *TOP* (like the **RO** Edit command, page 16.41)

Undo [DEdit Command]

Undoes last command.

!Undo [DEdit Command]

Subcommand of **Undo**. Undoes all changes since the start of this call on DEdit. This command can be undone.

?Undo [DEdit Command]

&Undo [DEdit Command]

Subcommands of **Undo**. Allows selective undoing of other than the last command. Both of these commands bring up a menu of all the commands issued during this call on DEdit. When the user selects an item from this menu, the corresponding command (and if **&Undo**, all commands since that point) will be undone.

Find [DEdit Command]

Selects, in place of *TOP*, the first place after *TOP* which matches *NXT*. Uses the Edit subsystem's search routine, so supports the full wildcarding conventions of Edit.

Swap [DEdit Command]

Exchanges *TOP* and *NXT* on the stack, i.e. the stack is changed, the structure being edited isn't.

The following set of commands are grouped together as subcommands of **Swap** because they all affect the stack and the selections, rather than the structure being edited.

Center [DEdit Command]

Subcommand of **Swap**. Scrolls until *TOP* is visible in its window.

Clear [DEdit Command]

Subcommand of **Swap**. Discards all selections (i.e., "clears" the stack).

Copy [DEdit Command]

Subcommand of **Swap**. Puts a copy of *TOP* into the edit buffer and makes it the new *TOP*.

Pop [DEdit Command]

Subcommand of **Swap**. Pops *TOP* off the selection stack.

Reprint [DEdit Command]

Reprints *TOP*.

Edit [DEdit Command]

Runs DEdit on the definition of the atom *TOP* (or **CAR** of list *TOP*). Uses **TYPESOF** to determine what definitions exist for *TOP* and, if there is more than one, asks the user, via menu, which one to use. If *TOP* is defined and is a non-list, calls **INSPECT** on that value. **Edit** also has a variety of subcommands which allow choice of editor (DEdit, TTYEdit, etc.) and whether to invoke that editor on the definition of *TOP* or the form itself.

Note: DEdit caches each subordinate edit window in the window from which it was entered, for as long as the higher window is active. Thus, multiple DEdit commands do not incur the cost of repeatedly allocating a new window.

EditCom [DEdit Command]

Allows one to run arbitrary Edit commands on the structure being DEdited (there are far too many of these for them all to appear on the main menu). *TOP* should be an Edit command, which will be applied to *NXT* as the current Edit expression. On return to DEdit, the (possibly changed) current Edit expression will be selected as the new *TOP*. Thus, selecting some expression, typing **(R FOO BAZ)**, and buttoning **EditCom** will cause **FOO** to be replaced with **BAZ** in the expression selected.

In addition, a variety of common Edit commands are available as subcommands of **EditCom**. Currently, these include **? =** , **GETD**, **CL**, **DW**, **REPACK**, **CAP**, **LOWER**, and **RAISE**. ,

Break [DEdit Command]

Does a **BREAKIN AROUND** the current expression *TOP*. (See page 15.6.)

Eval [DEdit Command]

Evaluates *TOP*, whose value is pushed onto the stack in place of *TOP*, and which will therefore appear, selected, in the edit buffer.

Exit [DEdit Command]

Exits from DEdit (equivalent to Edit **OK**, page 16.49).

OK [DEdit Command]

Stop [DEdit Command]

Subcommands of **Exit**. **OK** exits without an error; **STOP** exits with an error. Equivalent to the Edit commands with the same names.

16.1.6 Multiple Commands

It is occasionally useful to be able to give several commands at once - either because one thinks of them as a unit or because the intervening reprettyprinting is distracting. The stack architecture of DEdit makes such multiple commands easy to construct - one just pushes whatever arguments are required for the complete suite of commands one has in mind. Multiple commands are specified by holding down the **CONTROL** key during command selection. As long as the **CONTROL** key is down, commands selected will not be executed, but merely saved on a list. Finally, when a command is selected without the **CONTROL** key down, the command sequence is terminated with that command being the last one in the sequence.

One rarely constructs long sequences of commands in this fashion, because the feedback of being able to inspect the intermediate results is usually worthwhile. Typically, just two or three step idioms are composed in this fashion. Some common examples are given in the next section.

16.1.7 DEdit Idioms

As with any interactive system, there are certain common idioms on which experienced users depend heavily. Not only is discovering the idioms of a new system tiresome, but in places the designer may have assumed familiarity with one or more of them, so not knowing them can make life quite unbearable. In the case of DEdit, many of these idioms concern easy ways to achieve the effects of specific commands from the Edit system, with which many users are already familiar. The DEdit idioms described below are the result of the experience of the early users of the system and are by no means exhaustive. In addition to those that each user will develop to fit his or her own particular style, there are many more to be discovered and you are encouraged to share your discoveries.

Because of the novel argument specification technique (postfix; target first) many of the DEdit idioms are very simple, but opaque until one has absorbed the "target-source-command" way of looking at the world. Thus, one selects where typein is to go before touching the keyboard. After typing, the target will be selected second and the typein selected on top, so that an **After**, **Before** or **Replace** will have the desired effect. If the order is switched, the command will try to change the typein (which may or may not succeed), or will require tiresome **Swap**ping or reselection. Although this discipline seems strange at first, it comes easily with practice.

Segment selection and manipulation are handled in DEdit by first making them into a sublist, so they can be handled in the usual way. Thus, if one wants to remove the three elements between **A** and **E** in the list **(A B C D E)**, one selects **B**, then **D** (either order), then makes them into a sublist with the "**()**" command (pronounced "both in"). This will leave the sublist **(B C D)** selected, so a subsequent **Delete** will remove it. This can be issued as a single "**(); Delete**" command using multiple command selection, as described above, in which case the intermediate state of **(A (B C D) E)** will not show on the screen.

Inserting a segment proceeds in a similar fashion. Once the location of the insertion is selected, the segment to be inserted is typed as a list (if it is a list of atoms, they can be typed without parentheses and the **READ** will make them into a list, as one would expect). Then, the command sequence "**After** (or **Before** or **Replace**); **() out**" (given either as a multiple command or as two separate commands) will insert the typein and splice it in by removing its parentheses.

Moving an expression to another place in the structure being edited is easily accomplished by a delete followed by an insert. Select the location where the moved expression is to go to; select the expression to be moved; then give the command sequence "**Delete**; **After** (or **Before** or **Replace**)". The expression will first be deleted into the edit buffer where it will remain selected. The subsequent insertion will insert it back into the structure at the selected location.

Embedding and extracting are done with the **Replace** command. Extraction is simply a special case of replacing something with a subpiece of itself: select the thing to be replaced; select the subpart that is to replace it; **Replace**. Embedding also uses **Replace**, in conjunction with the "embed token" (the value of **EDITEMBEDTOKEN**, initially the single character atom **&**). Thus, to embed some expression in a **PROG**, select the expression; type "**(PROG** *VARSLST* **&)**"; **Replace**.

Switch can also be used to generate a whole variety of complex moves and embeds. For example, switching an expression with

typein not only replaces that expression with the typein, but provides a copy of the expression in the buffer, from where it can be edited or moved to somewhere else.

Finally, one can exploit the stack structure on selections to queue multiple arguments for a sequence of commands. Thus, to replace several expressions by one common replacement, select each of the expressions to be replaced (any number), then the replacing expression. Now hit the **Replace** command as many times as there are replacements to be done. Each **Replace** will pop one selection off the stack, leaving the most recently replaced expression selected. As the latter is now a copy of the original source, the next Replace will have the desired effect, and so on.

16.1.8 DEdit Parameters

There are several global variables that can be used to affect various aspects of DEdit's operation. Although most have been alluded to above, they are summarized here for reference.

EDITEMBEDTOKEN [Variable]

Initially **&**. Used in both DEdit and the teletype editor to indicate the special atom used as the "embed token".

DEditLinger [Variable]

Initially **T**. The default behavior of the topmost DEdit window is to remain active on the screen when exited. This is occasionally inconvenient for programs that call DEdit directly, so it can be made to close automatically when exited by setting this variable to **NIL**.

DEDITTYPEINCOMS [Variable]

Defines the control characters recognized as commands during DEdit typein. The elements of this list are of the form (*LETTER COMMANDNAME FN*), where *LETTER* is the alphabetic corresponding to the control character desired (e.g., **A** for control-A), *COMMANDNAME* is a litatom used both as a prompt and internal tag, and *FN* is a function applied to the expressions typed as arguments to the command. See the current value of **DEDITTYPEINCOMS** for examples. **DEDITTYPEINCOMS** is only accessed when DEdit is initialized, so DEdit should be reinitialized with **RESETDEDIT** (page 16.3) if it is changed.

DT.EDITMACROS [Variable]

Defines the behavior of the **Edit** command when invoked on a form that is not a list or litatom, thus telling DEdit how to edit

instances of certain datatypes. **DT.EDITMACROS** is an association list keyed by datatype name; entries are of the form (*DATATYPE MAKESOURCEFN INSTALLEDITFN*). When told to **Edit** an object of type *DATATYPE*, DEdit calls *MAKESOURCEFN* with the object as its argument. *MAKESOURCEFN* can either do the editing itself, in which case it should return **NIL**, or it should "destructure" the object into an editable list and return that list. In the latter case, DEdit is then invoked recursively on the list; when that edit is finished, DEdit calls *INSTALLEDITFN* with two arguments, the original object and the edited list. If *INSTALLEDITFN* causes an error, the recursive Dedit is invoked again, and the process repeats until the user either exits the lower editor with **STOP**, or exits with an expression that *INSTALLEDITFN* accepts.

For example, suppose the user has a datatype declared by **(DATATYPE FOO (NAME AGE SEX))**. To make instances of **FOO** editable, an entry **(FOO DESTRUCTUREFOO INSTALLFOO)** is added to **DT.EDITMACROS**, where the functions are defined by

```
(DESTRUCTUREFOO (OBJECT)
 (LIST (fetch NAME of OBJECT)
    (fetch AGE of OBJECT)
    (fetch SEX of OBJECT)))

(INSTALLFOO (OBJECT CONTENTS)
 (if (EQLENGTH CONTENTS 3)
  then (replace NAME of OBJECT with (CAR CONTENTS))
    (replace AGE of OBJECT with (CADR CONTENTS))
    (replace SEX of OBJECT with (CADDR CONTENTS))
  else (ERROR "Wrong number of fields for FOO" CONTENTS)))
```

16.2 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 first determines if the current expression is a tail of the next higher expression. If it is, **UP** is finished. Otherwise, **UP** computes **(MEMB** *CURRENT-EXPRESSION NEXT-HIGHER-EXPRESSION***)** to obtain a tail beginning with the current expression. The current expression should *always* be either a tail or an element of the next higher expression. If it is neither, for example the user has directly (and incorrectly) manipulated the edit chain, **UP** generates an error. If there are no other instances of the current expression in the next higher expression, this tail is the correct one. Otherwise **UP** uses **LASTAIL** to select the correct tail.

Occasionally the user can get the edit chain into a state where **LASTAIL** cannot resolve the ambiguity, for example if there were two non-atomic structures in the same expression that were **EQ**, and the user descended more than one level into one of them and then tried to come back out using **UP**. In this case, **UP** 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 effects. The **LASTAIL** solution is cheap and resolves almost all of the ambiguities.

N **(*N*> = 1)** [Editor Command]

Adds the *N*th 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 *N*th 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:

```
*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]

Effectively 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 first 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 *N*th element of the current expression (or *N*th 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 first printing "**>**".)

Control-X [Editor Command]

Control-X 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 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 different control characters for those macros, the function **SETTERMCHARS** is provided (see page 16.75).

16.3 Commands That Search

All of the editor commands that search use the same pattern matching routine (the function **EDIT4E**, page 16.72). 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 **$**s (escapes), each **$** can match an indefinite 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 litatom **$** (escape) matches only with itself.

(7) If *PAT* is a literal atom or string ending in **$$** (escape, escape), *PAT* matches with the atom or string *X* if it is "close" to *PAT*, in the

sense used by the spelling corrector (page 20.15). E.g. **CONSS$$** matches with **CONS**, **CNONC$$** with **NCONC** or **NCONC1**.

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 **VERS** 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 16.27.

(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))**.

16.3.1 Search Algorithm

Searching begins with the current expression and proceeds in print order. Searching usually means find the next instance of this pattern, and consequently a match is not attempted that would leave the edit chain unchanged. 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 first in the **CAR** direction, and then in the **CDR** direction, i.e., if the element under examination is a list, the search descends into that list before attempting to match with other elements (or tails) at the same level. Note: A find 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 **CAR**s and **CDR**s descended into) allowed to exceed the value of the variable **MAXLEVEL**. At that point, the search of that element or tail is abandoned, exactly as though the element or tail had been completely searched without finding a match, and the search continues with the 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 infinity. **MAXLEVEL** is initially set to 300.

If a successful match is not found in the current expression, the search automatically ascends to the next higher expression, and continues searching there on the next expression after the expression it just finished searching. If there is none, it ascends again, etc. This process continues until the entire edit chain has been searched, at which point the search fails, and an error is generated. If the search fails (or is aborted by control-E), the edit chain is not changed (nor are any **CONS**es performed).

If the search is successful, i.e., an expression is found that the pattern matches, the edit chain is set to the value it would have had had the user reached that expression via a sequence of integer commands.

If the expression that matched was a list, it will be the final link in the edit chain, i.e., the new current expression. If the expression that matched is not a list, e.g., is an atom, the current expression will be the tail beginning with that atom, 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 effectively does an **UP** (unless **UPFINDFLG** = **NIL** (initially **T**). See "Form Oriented Editing", page 16.34).

16.3.2 Search Commands

All of the commands below set **LASTAIL** for use by **UP**, set **UNFIND** for use by \ (page 16.28), and do not change the edit chain or perform any **CONS**es 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 interpreted as a pattern. This is the most common and useful form of the find command. If successful, the edit chain always changes, i.e., **F** *PATTERN* means find the next instance of *PATTERN*.

If **(MEMB** *PATTERN CURRENT-EXPRESSION***)** is true, **F** does not proceed with a full recursive search. If the value of the **MEMB** is **NIL**, **F** invokes the search algorithm described on page 16.20.

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

F *PATTERN* **N** [Editor Command]

Same as **F** *PATTERN*, i.e., Finds the **N**ext 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 *N*th 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 find **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 find the **COND** inside the **SETQ**, whereas **(F (COND --))** will find the top level **COND**, i.e., the second one.

(FS $PATTERN_1$ ... $PATTERN_N$**)** [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 16.18).

(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 16.18).

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.

16.3.3 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**. 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.

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 **COND**s, 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 16.60) in conjunction with the ## function (page 16.59) 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 16.60.

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 differs from **BF** in that it does not search *inside* of each link, it simply ascends. Thus in the above example, **F CADR** followed by **BF COND** would find **(COND (FLG (RETURN L)))**, not the higher **COND**.

(BELOW *COM X*) [Editor Command]

Ascends the edit chain looking for a link specified 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 specified point, and then backs off *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 find 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 **MARK**ing (see page 16.28) 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 16.18, 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**.

16.4 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 **MARK**s 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 first 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 first **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 flip 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.

16.5 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 *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.

(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 16.64).

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 26.40). 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 |.

16.5.1 Implementation

Note: Since all commands that insert, replace, delete or attach structure use the same low level editor functions, the remarks made here are valid for all structure changing commands.

For all replacement, insertion, and attaching at the end of a list, unless the command was typed in directly to the editor, *copies* of the corresponding structure are used, because of the possibility that the exact same command, (i.e., same list structure) might be used again. 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**. The user can circumvent this by using the **I** command (page 16.58), 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.

Note: 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.

The rest of this section is included for applications wherein the editor is used to modify a data structure, and pointers into that data structure are stored elsewhere. In these cases, the actual mechanics of structure modification must be known in order to predict the effect that various commands may have on these

outside pointers. For example, if the value of **FOO** is **CDR** of the current expression, what will the commands **(2)**, **(3)**, **(2 X Y Z)**, **(-2 X Y Z)**, etc. do to **FOO**?

Deletion of the first element in the current expression is performed by replacing it with the second element and deleting the second element by patching around it. Deletion of any other element is done by patching around it, i.e., the previous tail is altered. Thus if **FOO** is **EQ** to the current expression which is **(A B C D)**, and **FIE** is **CDR** of **FOO**, after executing the command **(1)**, **FOO** will be **(B C D)** (which is **EQUAL** but not **EQ** to **FIE**). However, under the same initial conditions, after executing **(2)** **FIE** will be unchanged, i.e., **FIE** will still be **(B C D)** even though the current expression and **FOO** are now **(A C D)**.

A general solution of the problem 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.

Both replacement and insertion are accomplished by smashing both **CAR** and **CDR** of the corresponding tail. Thus, if **FOO** were **EQ** to the current expression, **(A B C D)**, after **(1 X Y Z)**, **FOO** would be **(X Y Z B C D)**. Similarly, if **FOO** were **EQ** to the current expression, **(A B C D)**, then after **(-1 X Y Z)**, **FOO** would be **(X Y Z A B C D)**.

The **N** command is accomplished by smashing the last **CDR** of the current expression a la **NCONC**. Thus if **FOO** were **EQ** to any tail of the current expression, after executing an **N** command, the corresponding expressions would also appear at the end of **FOO**.

In summary, the only situation in which an edit operation will *not* change an external pointer occurs when the external pointer is to a *proper tail* of the data structure, i.e., to **CDR** of some node in the structure, and the operation is deletion. If all external pointers are to *elements* of the structure, i.e., to **CAR** of some node, or if only insertions, replacements, or attachments are performed, the edit operation will *always* have the same effect on an external pointer as it does on the current expression.

16.5.2 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 *N*th 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 16.23. 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** (see warning about **INSERT**, page 16.33). 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 off. Similarly, if the 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, find the first **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.

16.5.3 Form Oriented Editing and the Role of UP

The **UP** that is performed before **A**, **B**, and **:** commands (and therefore in **INSERT**, **CHANGE**, **REPLACE**, and **DELETE** commands after the location portion of the operation has been performed)

makes these operations form-oriented. For example, if the user types **F SETQ**, and then **DELETE**, or simply **(DELETE SETQ)**, he will delete the entire **SETQ** expression, whereas **(DELETE X)** if **X** is a variable, deletes just the variable **X**. In both cases, the operation is performed on the corresponding *form*, and in both cases is probably what the user intended. Similarly, if the user types **(INSERT (RETURN Y) BEFORE SETQ)**, he means before the **SETQ** expression, not before the atom **SETQ**. A consequent of this procedure is that a pattern of the form **(SETQ Y --)** can be viewed as simply an elaboration and further refinement of the pattern **SETQ**. Thus **(INSERT (RETURN Y) BEFORE SETQ)** and **(INSERT (RETURN Y) BEFORE (SETQ Y --))** perform the same operation (assuming the next **SETQ** is of the form **(SETQ Y --)**) and, in fact, this is one of the motivations behind making the current expression after **F SETQ**, and **F (SETQ Y --)** be the same.

Note: There is some ambiguity in **(INSERT** *EXPR* **AFTER** *FUNCTIONNAME***)**, as the user might mean make *EXPR* be the function's first argument. Similarly, the user cannot write **(REPLACE 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)**.

Occasionally, however, a user may have a data structure in which no special significance or meaning is attached to the position of an atom in a list, as Interlisp attaches to atoms that appear as **CAR** of a list, versus those appearing elsewhere in a list. In general, the user may not even *know* whether a particular atom is at the head of a list or not. Thus, when he writes **(INSERT** *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 --)**.

16.5.4 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 . @)** (see warning about **INSERT**, page 16.33). 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]

Performs **(LC . $@_2$)** and then **(XTR . $@_1$)** (see warning about **INSERT**, page 16.33). The current edit chain is not changed, but **UNFIND** is set to the edit chain after the **XTR** was performed.

Note: $@_1$ is the *segment* between **EXTRACT** and **FROM**.

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.

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 first 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 specification:

(EMBED @ IN . *X*) [Editor Command]

(**@** is the segment between **EMBED** and **IN**.) Does **(LC . @)** and then **(MBD . *X*)** (see warning about **INSERT**, page 16.33). 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 **&**.

16.5.5 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 . $@_1$)** (see warning about **INSERT**, page 16.33), 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 & &))
*

*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

*
```

16.5.6 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 *N*th 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 *N*th element and after the *M*th element in the current expression. Generates an error if the *M*th element is not contained in the *N*th tail, i.e., the *M*th element must be "to the right" of the *N*th 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 *N*th element. Generates an error if *N*th 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 *N*th 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 *N*th element. *All elements following the Nth element are deleted.* Generates an error if *N*th 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 *M*th element of the *N*th element. The rest of the *N*th 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 *N*th element *in* to after its *N*th element."

(RO *N*) [Editor Command]

"Right Out". Removes the right parenthesis from the *N*th element, moving it to the end of the current expression. All

elements following the *N*th element are moved inside of the *N*th element. Generates an error if *N*th 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 *N*th element *out* to the end of the current expression."

16.5.7 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 a.. internal editor flag informing the above commands that the element they are operating on is actually a segment, and that the extra pair of parentheses should be removed when the operation is complete. Thus:

***P**
(PROG (& & 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]
*
```

16.5.8 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 16.20, and *X* can employ any of the patterns on page 16.18. 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 **$**s (escapes), **$**s appearing in *Y* stand for the characters matched by the corresponding **$** in *X*. For example, **(R FOO$ FIE$)** means for all atoms or strings that begin with **FOO**, replace the characters "**FOO**" by "**FIE**". Applied to the list **(FOO FOO2 XFOO1)**, *(R FOO$ 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**.

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.

Note that the **$** (escape) 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 not have a mate in *Y*, the characters matched by the **$** are effectively deleted. For example, **(R $/$ $)** will change **AND/OR** to **AND**. There is no similar operation for changing **AND/OR** to **OR**, since the first **$** in *Y* always corresponds to the first **$** in *X*, the second **$** in *Y* to the second in *X*, etc. *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 $))** changes every **(SETQ X &)** to **(PRINT (SETQ X &))**. If *X* is a pattern containing an **$** 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$)**.

Since **(R $*X*$ $*Y*$)** is a frequently used operation for **R**eplacing **C**haracters, 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 *N*th and *M*th 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 *N*th and *M*th 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.

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

16.6 Commands That Print

PP [Editor Command]

Prettyprints the current expression.

P [Editor Command]

Prints the current expression as though **PRINTLEVEL** (page 25.11) 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 26.43). **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 first resets ****COMMENT**FLG** to **NIL** (see page 26.43).

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 14.7). For example,

```
*P
(STRPOS "A0???" X N (QUOTE ?) T)
*?=
X = "A0???"
Y = X
```

```
START = N
SKIP = (QUOTE ?)
ANCHOR = T
TAIL =
```

The command **MAKE** (page 16.57) 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 16.28). All can be aborted with control-E.

16.7 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 16.51) 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 (namely, the size of the history list, which can be changed with **CHANGESLICE**, page 13.21) 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:

```
←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.

16.8 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.

16.9 Manipulating the Characters of an Atom or String

RAISE [Editor Command]

An edit macro defined as **UP** followed by **(I 1 (U-CASE (## 1)))**, i.e., it raises to upper-case the current expression, or if a tail, the first 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 first character, i.e., the first 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 first element. If the lower editor is exited successfully, i.e., via **OK** as opposed to **STOP**, the list of atoms is made into a single atom or string, which replaces the atom or string being "repacked." The new atom or string is always printed.

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$)**.

16.10 Manipulating Predicates and Conditional Expressions

JOINC [Editor Command]

Used to join two neighboring **COND**'s together, e.g. **(COND** $CLAUSE_1$ $CLAUSE_2$**)** followed by **(COND** $CLAUSE_3$ $CLAUSE_4$**)** becomes **(COND** $CLAUSE_1$ $CLAUSE_2$ $CLAUSE_3$ $CLAUSE_4$**)**. **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** $CLAUSE_1$ $CLAUSE_2$ $CLAUSE_3$ $CLAUSE_4$**)** into **(COND** $CLAUSE_1$ $CLAUSE_2$**) (COND** $CLAUSE_3$ $CLAUSE_4$**)**. 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 3.20).

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.

16.11 History commands in the editor

All of the user's inputs to the editor are stored on the history list **EDITHISTORY** (see page 13.43, 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**.

16.12 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 21.22).

DW [Editor Command]

Dwimifies the current expression (see page 21.18).

IFY [Editor Command]

If the current statement is a **COND** statement (page 9.4), replaces it with an eqivalent **IF** statement (page 9.5).

GET* [Editor Command]

If the current expression is a comment pointer (see page 26.44), 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]
```

then **(* CALL FACT RECURSIVELY)** would insert **(* CALL FACT RECURSIVELY)** *before* the **ITIMES** expression. 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.

Warning: When expanding a function definition or open lambda expression, **GETD** does a simple substitution of the actual arguments for the formal arguments. Therefore, if any of the function arguments are used in other ways in the function definition (as functions, as record fields, etc.), they will simply be replaced with the actual arguments.

(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**.

16.13 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 definition 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 first element of the current expression, otherwise replaces the first element by the value of **FOO**.

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

EVAL [Editor Command]

Does an **EVAL** of the current expression.

Note that **EVAL**, line-feed, and the **GO** command together effectively allow the user to "single-step" a program through its symbolic definition.

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.

Note: 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 16.34). Thus, **(INSERT (## 3 2) AFTER 1)** is equivalent to **(I INSERT (COPY (## 3 2)) 'AFTER 1)**.

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. The editor command **NIL** is a no-op (page 16.55).

(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)))**.

16.14 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))))**.

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)))
        COMS1)
       (T COMS2))))
```

For example, the command **(IF (READP T) NIL (P))** will print the current expression provided the input buffer 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. The form **(## 3)** will cause an error if the edit command **3** causes an error, thereby selecting **((N T))** as

the list of commands to be executed. The **IF** could also be written as **(IF (CDDR (##)) NIL ((N T)))**.

When an error occurs, **LP** prints *N* **OCCURRENCES** where *N* is the number of times 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 infinity. 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 13.8).

(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).

(EXAM *X***)** [Editor Command]

Like **SHOW** except calls the editor recursively (via the **TTY:** command, see page 16.51) 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 finished. Otherwise, **ORR** restores the edit chain to its original value, and continues by executing $COMS_2$, etc. If none of the command lists execute without errors, i.e., the **ORR** "drops off the end", **ORR** generates an error. Otherwise, the edit chain is left as of the completion of the first command list which executes without 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)))**.

16.15 Edit Macros

Many of the more sophisticated branching commands in the editor, such as **ORR**, **IF**, etc., are most often used in conjunction with edit macros. The macro feature permits the user to define new commands and thereby expand the editor's repertoire, or redefine existing commands (to refer to the original definition of a built-in command when redefining it via a macro, use the **ORIGINAL** command, page 16.64).

Macros are defined by using the **M** command:

(M *C* $COMS_1$ ... $COMS_N$**)** [Editor Command]

For *C* an atom, **M** defines *C* as an atomic command. If a macro is redefined, its new definition replaces its old. Executing *C* is then the same as executing the list of commands $COMS_1$ ... $COMS_N$.

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

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

(M (*C***) (***ARG_1 ... ARG_N***)** $COMS_1$ ... $COMS_M$**)** [Editor Command]

C an atom. **M** defines *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 $COMS_1$ ... $COMS_M$.

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 16.24), 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 $COMS_1$ ... $COMS_N$. **BIND** uses a **PROG** to make these bindings, so they are only in effect while the

commands are being executed and **BIND**s can be used recursively; the variables **#1**, **#2**, and **#3** will be rebound each time **BIND** is invoked.

Thus, we can 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 17.34), is available for dumping all or selected user macros.

16.16 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 13.13. (Note that if the **I** command was typed directly to the editor, **/NCONC** would automatically be substituted for **NCONC** as described in page 13.27.)

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. 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 13.6). 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.

16.17 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. 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.

If a location specification is being executed, an internal flag informs **EDITDEFAULT** to treat the command as though it had been preceded by an **F**.

If the command is a list, an attempt is made to perform spelling correction on the **CAR** of the command (unless **DWIMFLG** = **NIL**) using **EDITCOMSL**, a list of all list edit commands. If spelling correction is successful, the correct command name is **RPLACA**ed 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.

Note: 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**.

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. The line is read using **READLINE** (page 13.36), so the line can be terminated by a square bracket, or by a carriage return not preceded by a space. 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)**.

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.

(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 &))
*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 first *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.

16.18 Editor Functions

(EDIT *NAME* **—)** [Function]

General purpose function for calling the editor. Figures out what type of definition *NAME* has (function, variable, macro, etc.), and calls the editor to edit it. If *NAME* has more than one definition of different types, the user is prompted for which type of definition to edit.

(EDITF *NAME* COM_1 COM_2 ... COM_N**)** [NLambda NoSpread Function]

Nlambda, nospread function for **EDIT**ing a **F**unction. *NAME* is the name of the function, COM_1, COM_2, ..., COM_n are (optional) edit commands. **EDITF** returns *NAME*.

If *NAME* is **NIL**, it defaults to the value of **LASTWORD** (page 20.18), the last function or variable referred to by the user.

Note: **EDITF** initially calls **HASDEF** (page 17.26), which does spelling correction on *NAME* using the spelling list **USERWORDS** (unless **DWIMFLG = NIL**).

The action of **EDITF** is somewhat complicated, because the function may be broken or advised, the expr definition of the function may be saved on the property list of *NAME*, the

function may need to be loaded from a file, etc. There are many special cases that have to be handled differently. When **EDITF** is called, it tries the following, in order:

(1) In the most common case, if the definition of *NAME* is an expr definition (not as a result of its being broken or advised), **EDITE** (page 16.71) is called to edit the function definition.

(2) If *NAME* has an expr definition by virtue of its being broken or advised, and the original definition is also an expr definition, then the broken/advised definition is given to **EDITE** to be edited (since any changes there will also affect the original definition because all changes are destructive). However, a warning message (e.g. "**Note: you are editing a BROKEN definition**") is printed to alert the user that the function definition is surrounded by a call to **BREAK1** or **ADV-PROG**.

(3) If *NAME* has an expr definition by virtue of its being broken or advised, the original definition is not an expr definition, there is no **EXPR** property, and the file package "knows" which file *NAME* is contained in (see **EDITLOADFNS?**, page 16.73), then the expr definition of *NAME* is loaded onto its property list as described below, and the editor proceeds to the next possibility. Otherwise, a warning message is printed (e.g. "**Note: you are editing a BROKEN compiled definition**"), and the edit proceeds, e.g., the user may have called the editor to examine the advice on a compiled function.

(4) If *NAME* has an expr definition by virtue of its being broken or advised, the original definition 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 the editor proceeds to the next possibility.

(5) If *NAME* does not have an expr definition, but has an **EXPR** property, **EDITF** prints **prop**, and calls **EDITE** (page 16.71 to edit this saved expr definition. 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 10.10) is **PROP**, **EDITE** prints "**changed, but not unsaved.**" Otherwise if changes were made, **EDITE** prints **unsaved** and does an **UNSAVEDEF** (page 17.28).

(6) If *NAME* neither has an expr definition nor an **EXPR** property, and the file package "knows" which file *NAME* is contained in (see **EDITLOADFNS?**, page 16.73), the expr definition of *NAME* is automatically loaded (using **LOADFNS**, page 17.6) onto the **EXPR** property, and **EDITE** proceeds as described above. Because of the existence of file maps (page 17.55), this operation is extremely fast, essentially requiring only the time to perform the **READ** to obtain the actual definition. 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.

The editor's behaviour in this case is controlled by the value of **EDITLOADFNSFLG**, which is a dotted pair of two flags. 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 flag means "load but ask first," 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.

(7) If *NAME* has neither an expr definition nor an **EXPR** property, but it does have a macro definition, that definition is edited.

(8) If *NAME* has neither an expr definition nor an **EXPR** property nor a macro definition, the user is prompted with "**No FNS defn for** *NAME*. **Do you wish to edit a dummy defn?**". If the user confirms by typing **Yes**, a "blank" definition (stored on the variable **DUMMY-EDIT-FUNCTION-BODY**) is edited. If any changes are made, on exit from the editor, the definition will be installed as the name's function definition. Exiting the editor with the **STOP** command will prevent any changes to the function definition.

(9) Otherwise, the editor 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 16.76). 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. If *NAME* is atomic, and its value is not a list, and it is the name of a file, **(FILEFNSLST '***NAME***)** will be used as the list of functions to be edited.

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 (page 14.21), 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. *NAME* is the name of the variable, COM_1, COM_2, ..., COM_n are (optional) edit commands.

If *NAME* is **NIL**, it defaults to the value of **LASTWORD** (page 20.18), the last function or variable referred to by the user.

If *NAME* is bound as a variable on the stack, **EDITV** edits its value, otherwise if *NAME* has a top-level variable binding, **EDITV** edits the top-level value. **EDITV** returns *NAME* if it is bound or has a top-level value, **NIL** otherwise.

EDITV calls **EDITE** (page 16.71) to edit the value of the variable. Note that if the value of the variable is not a list, this causes an error: "*EXPR* **not editable.**"

Note: **EDITV** initially calls **HASDEF** (page 17.26), which does spelling correction on *NAME* using the spelling list **USERWORDS** (unless **DWIMFLG = NIL**).

(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** (unless **DWIMFLG = NIL**). Then **EDITP** calls **EDITE** on the property list of *NAME*, (or the corrected spelling thereof), with *TYPE* = **PROPLST**.

EDITP returns 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: "*EXPR* **not editable.**"

ATM and *TYPE* are for use in conjunction with the file 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 17.17.) For example, if *EXPR* is the definition 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, and the value of *IFCHANGEDFN* is not **NIL**, the value of *IFCHANGEDFN* is applied to the arguments *ATM*, *EXPR*, *TYPE*, and a flag 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 17.17) 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* will be **NIL**.)

Note: For *TYPE* = **FNS** or *TYPE* = **PROP**, i.e., calls from **EDITF**, **EDITE** performs some additional operations as described earlier under **EDITF**.

(EDITL *L COMS ATM MESS EDITCHANGES***)** [Function]

EDITL *is* the editor. Its first argument is the edit chain, and its value is an edit chain, namely the value of *L* at the time **EDITL** is exited. *L* is a **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.

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 read table. Exit occurs only via an **OK**, **STOP**, or **SAVE** command.

If *COMS* is *not* **NIL**, no message is typed, and each member of *COMS* is treated as a command and executed. If an error occurs in the execution of one of the commands, no error message is printed, the rest of the commands are ignored, and **EDITL** exits with an error, i.e., the effect is the same as though a **STOP** command had been executed. If all commands execute successfully, **EDITL** returns the current value of *L*.

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 16.18 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 ERRORFLG CHARFLG***)** [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 *ERRORFLG* = **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** if no file is found.

EDITLOADFNS? performs **(WHEREIS** *FN* **'FNS** *FILES***)** to obtain the name of the file(s) containing *FN*, if any (see page 17.14). If **WHEREIS** returns more than one file, **EDITLOADFNS?** asks the user to indicate which file to use.

If the file has been **LOAD**ed or **LOADFROM**ed, the file name saved on the **FILEDATES** property (page 17.20) of the file is checked by calling **INFILEP**. If not found, **FINDFILE** is called to find the file. If a file is found, the file date (see **FILEDATE**, page 17.52) is compared to the file date saved on the **FILEDATES** property of the file, to determine whether this file is the one that was *originally* loaded. If not, **EDITLOADFNS?** prints "*** note: *FILENAME* dated *DATE* isn't current version; *FILENAME* dated *DATE* is." and then uses the file found.

In the case that *FILES* = **T** and the WHEREIS library package has been loaded, files(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.

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* = **NIL** and *STR* = **NIL**, **EDITLOADFNS?** prints "**loading definition of** *FN* **from** *FILENAME*."

If *ASKFLG* = **T**, **EDITLOADFNS?** calls **ASKUSER** (page 26.12) giving **(LIST** *FN STR FILENAME***)** as the message to be printed. If **ASKUSER** returns **Y**, **EDITLOADFNS?** returns the filename.

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

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.

(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, 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 at which 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.

EDITCALLERS uses **GETDEF** (page 17.25) to obtain the "definition" for each object. When **EDITE** returns, if a change was made, **PUTDEF** is called to store the changed object.

(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 [Variable]

This variable 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** (initially undefined) 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 25.42). *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, **(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.

16.19 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 (page 12.1), 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 12.2).

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** *OLDATE INITLS*) which returns a new time-stamp comment. If *OLDATE* is a time-stamp comment, it will be reused.