16. SEdit - The Structure Editor

Medley's code editors are "structure" editors—they know how to take advantage of Lisp code being represented as lists. One is a display editor named SEdit and the other is a TTY-based editor.

Starting the Editor

The editor is normally called using the following functions:

(DF *FN*) [NLambda NoSpread Function]

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

If you call DF with a name that has no function definition, you are prompted with a choise of definers to use.

(DV *VAR*) [NLambda NoSpread Function]

Edit the value of the variable *VAR*.

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

Edit property *PROP* of the symbol *NAME*. If *PROP* is not given, the whole property list of *NAME* is edited.

(DC *FILE*) [NLambda NoSpread Function]

Edit the file package commands (or "filecoms," see Chapter 17) for the file *FILE*.

(ED *NAME OPTIONS*) [Function]

This function finds out what kind of definiton *NAME* has and lets you edit it. If NAME has more than one definition (e.g., it's both a function and a macro), you will be prompted for the right one. If NAME has no definition, you'll be asked what kind of definition to create.

Choosing Your Editor

The default editor may be set with EDITMODE:

(EDITMODE *NEWMODE*) [Function]

If *NEWMODE* is DISPLAY, sets the default editor to be SEdit; or the teletype editor (if *NEWMODE* is TELETYPE). Returns the previous setting. If *NEWMODE* is NIL, returns the previous setting without setting a new editor.

SEdit - The Structure Editor

SEdit is a structure editor. You use a structure editor when you want to edit objects instead of text. SEdit is a part of the environment and operates directly on objects in the system you are running. SEdit behaves differently depending on the type of objects you are editing.

Common Lisp definitions:	SEdit always edits a copy of a Common Lisp definition. The changes made while you edit a function will not be installed until the edit session is complete. For example, when you edit a Common Lisp function, you edit the definition of the function and not the executable version of the function. When you end the session the comments will be stripped of the definition and the definition will be installed as the executable version of the function.
Interlisp functions and macros:	SEdit edits the actual structure that will be run, except editing the source for a compiled function. In this case, changes are made and the function is unsaved when you complete the edit session.
All other structures:	Variables, property lists and other structures are edited directly in place, i.e. SEdit installs all changes as they are made.

If you make a severe editing error, you can abort the edit session with an Abort command (see Command Keys, below). This command undoes all changes from the beginning of the edit session and exits from SEdit without changing your environment.

If you change the definition of an object that is being edited in an SEdit window, Medley will ask you if you want to throw away the changes made there.

SEdit supports the standard Copy-Select mechanism in Medley.

An SEdit Session

Whenever you call SEdit, a new SEdit window is created. This SEdit window has its own process. You can make edits in the window, shrink it while you do something else, expand it and edit some more, and finally close the window when you are done.

Throughout an edit session, SEdit remembers everything that you do IN a change history. You can

undo and redo edits sequentially. When you end the edit session, SEdit forgets this information and installs the changes in the system.

You signal the end of the session in the following ways:

- Close the window.
- Shrink the window. If you expand the window again, you can continue editing.
- Issue a Completion Command, see below.

SEdit Carets

There are two carets in SEdit, the edit caret and the structure caret. The edit caret appears when characters are edited within a single symbol, string, or comment. Anything you type will appear at the edit caret as part of the item it's in. The edit caret looks like this:

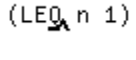
The structure caret appears when the edit point is between symbols (or strings or comments), so that anything you type will go into a new one. It looks like this:

SEdit changes the caret frequently, depending on where the caret is positioned. The left mouse button positions the edit caret. The middle mouse button positions the structure caret.

The Mouse

The left mouse button selects parts of Lisp structures. The middle mouse button selects whole Lisp structures.

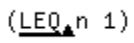
For example; select the `Q` in `LEQ` below by pressing the left mouse button when the pointer is over the `Q`.



Any characters you type in now will be appended to the symbol `LEQ`.

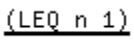
Selecting the same letter with the middle mouse button selects the whole symbol (this matches TEdit's

character/word selection convention), and sets a structure caret between the LEQ and the n:



Any characters you type in now will form a new symbol between the LEQ and the n.

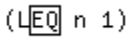
Larger structures can be selected in two ways. Use the middle mouse button to position the mouse cursor on the parenthesis of the list you want to edit. Press the mouse button multiple times, without moving the mouse, extends the selection. In the previous example, if the middle button was pressed twice, the list (LEQ ...) would be selected:



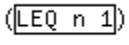
Press the button a third time and you will select the list containing the (LEQ n 1) to be selected.

The right mouse button positions the mouse cursor for selecting sequences of structures or substructures. Extended selections are indicated by a box enclosing the structures selected. The selection extends in the same mode as the original selection. That is, if the original selection was a character selection, the right button will be used to select more characters in the same atom. Extended selections also have the property of being marked for pending deletion. That is, the selection takes the place of the caret, and anything typed in is inserted in place of the selection.

For example, selecting the E by pressing the left mouse button and selecting the Q by pressing the right mouse button will produce:



Similarly, pressing the middle mouse button and then selecting with the right mouse button extends the selection by whole structures. In our example, pressing the middle mouse button to select LEQ and pressing the right mouse button to select the 1 will produce:



This is not the same as selecting the entire list, as above. Instead, the elements in the list are collectively selected, but the list itself is not.

Gaps

SEdit requires that everything edited must have an underlying Lisp structure at all times. Some

characters, such as single quote "'" have no meaning by themselves, but must be followed by something more. Whenyou type such a character, SEdit puts a "gap" where the rest of the input should go. When you type, the gap is automatically replaced.

A gap looks like:

After you type a quote, the gap looks like this: with the gap marked for pending deletion.

Broken Atoms

When you type an atom (a symbol or a number), SEdit saves the characters you type until you are finished. Typing any character that cannot belong to an atom, like a space or open parenthesis, ends the atom. SEdit then tries to create an atom with the characters you just typed, just as if they were read by the Lisp reader. The atom then becomes part of the structure you're editing.

If an error occurs when SEdit reads the atom, SEdit creates a structure called a Broken-Atom. A Broken-Atom looks and behaves just like a normal atom, but is printed in italics to tell you that something is wrong.

SEdit creates a Broken-Atom when the characters typed don't make a legal atom. For example, the characters "DECLARE:" can't be a symbol because the colon is a package specifier, but the form is not correct for a package-qualified symbol. Similarly, the characters "#b123" cannot represent an integer in base two, because 2 and 3 aren't legal digits in base two, so SEdit would make a Broken-Atom that looks like *#b123*.

You can edit Broken-Atoms just like real atoms. Whenever you finish editing a Broken-Atom, SEdit again tries to create an atom from the characters. If SEdit succeeds, it reprints the atom in SEdit's default font, rather than in italics. Be sure to correct any Broken-Atoms you create before exiting SEdit, since Broken-Atoms do not behave in any useful way outside SEdit.

Special Characters

Some characters have special meanings in Lisp, and are therefor treated specially by SEdit. SEdit must always have a complete structure to work on at any level of the edit. This means that SEdit needs a special way to type in structures such as lists, strings, and quoted objects. In most instances these structures can be typed in just as they would be to a regular Exec, but in the following cases this is not possible.

Lists: **( )** Lists begin with an open parenthesis character "(". Typing an open parenthesis gives a balanced list. SEdit inserts both an open and a close parenthesis. The structure caret is placed between the two parentheses. List elements can be typed in at the structure caret. When a close parenthesis, ")" is typed, the caret will be moved outside the list, effectively

finishing the list. Square bracket characters, "[" and "]", have no special meaning in SEdit, as they have no special meaning in Common Lisp.

Single Quote:	`'`	
Backquote:	`` ` ``	
Comma:	`,`	
At Sign:	`,@`	
Dot:	`,.`	
Hash Quote:	`#'`	All these characters are special macro characters in Common Lisp. When you type one, SEdit will echo the character followed by a gap, which you should then fill in.
Dotted Lists:	`( . )`	Use period to enter dotted pairs. After you type a dot, SEdit prints a dot and a gap to fill in for the tail of the list. To dot an existing list, point the cursor between the last and second to last elements, and type a dot. To undot a list, select the tail of the list before the dot while holding down the SHIFT key.
Single escape:	`\` or `%`	Use the single escape characters to make symbols with special characters. The single escape character for Interlisp is "%". The single escape character for Common Lisp is "\". When you want to create a symbol with a special character in it you have to type a single escape character before you type the character itself. SEdit does not echo the single escape character until you type the following character. For example; create the Common Lisp symbol `APAREN-(`. Since SEdit normally will treat the "`(`" as the start of a new list you have to tell SEdit to treat it as an ordinary character. You do this by typing a "\" before you type the "`(`".
Multiple Escape:	`|`	Use the multiple escape character when you enter symbols with many special characters. SEdit always balances multiple escape characters. When you type one, SEdit adds another, with the caret between them. If you type a second vertical bar, the caret moves after it, but is still inthe same symbol, so you can add more unescaped characters.
Comment:	`;`	A semicolon starts a comment. When you type a semicolon, an empty comment is inserted with the caret in position to type the comment. Comments can be edited like strings. There are three levels of comments supported by SEdit: single-, double-, and triple-semicolon. Single-semicolon comments are formatted at the comment column, about three-quarters of the way across the window. Double-semicolon comments are formatted at the current

indentation of the code they are in. Triple semicolon comments are formatted against the left margin. The level of a comment can be increased or decreased by pointing after the semicolon, and either typing another semicolon, or backspacing over the preceding semicolon. Comments can be placed anywhere in your Common Lisp code. However, in Interlisp code, they must follow the placement rules for Interlisp comments.

String: **"** Enter strings in SEdit by typing a double quote. SEdit balances the double quotes: When one is typed, SEdit produces a second, with the caret between the two. If you type a double-quote in the middle of a string, SEdit breaks the string in two, leaving the caret between them.

SEdit Commands

Enter SEdit commands either from the keyboard or from the SEdit menu. When possible, SEdit uses a named key on the keyboard, e.g., the DELETE key. Other commands are combinations of Meta, Control, and alphabetic keys. For the alphabetic command keys, either uppercase or lowercase will work.

There are two menus available, as an alternative means of invoking commands. They are the middle button popup menu, and the attached command menu. These menus are described in more detail below.

Meta-A Abort the session. Throw away the changes made to the form.

Meta-B Change the Print Base. Prompts for entry of the desired Print Base, in decimal. SEdit redisplays fixed point numbers in this new base.

Control-C Tell SEdit that this session is complete and compiles the definition being edited. The variable `*COMPILE-FN*` determines which function to use as compiler. See the Options section below.

Control-Meta-C Signals the system that this edit is complete, compiles the definition being editing, and closes the window.

DELETE Deletes the current selection.

Meta-E Evaluate the current selection. If the result is a structure, the inspector is called on it, allowing the user to choose how to look at the result. Otherwise, the result is printed in the SEdit prompt window. The evaluation is done in the process from which the edit session was started. Thus, while editing a function from a break window, evaluations are done in the context of the break.

FIND
Meta-F Find a specified structure, or sequence of structures. If there is a current selection, SEdit looks for the next occurrence of the selected structure. If there is no selection, SEdit prompts for the structure to find, and searches forward from the position of the caret. The found structure will be selected, so the Find

command can be used to easily find the same structure again.

If a sequence of structures are selected, SEdit will look for the next occurrence of the same sequence. Similarly, when SEdit prompts for the structure to find, you can type a sequence of structures to look for.

The variable *WRAP-SEARCH* controls whether or not SEdit wraps around from the end of the structure being edited and continues searching from the beginning.

Control-Meta-F Find a specified structure, searching in reverse from the position of the caret.

HELP
Meta-H Show the argument list for the function currently selected, or currently being typed in, in the SEdit prompt window. If the argument list will not fit in the SEdit prompt window, it is displayed in the main Prompt Window.

Meta-I Inspect the current selection.

Meta-J Join any number of sequential Lisp objects of the same type into a single object of that type. Join is supported for atoms, strings, lists, and comments. In addition, SEdit permits joining of a sequence of atoms and strings, since either type can easily be coerced into the other. In this case, the result of the Join will be an atom if the first object in the selection is an atom, otherwise the result will be a string.

Control-L Redisplay the structure being edited.

SKIP-NEXT
Meta-N Select next gap in the structure.

Meta-O Edit the definition of the current selection. If the selected name has more than one type of definition, SEdit asks for the type to edit. If the selection has no definition, a menu pops up. This menu lets you specify the type of definition to create.

Control-Meta-O Perform a fast edit by calling ED with the CURRENT option.

Meta-P Change the current package for this edit. Prompt the user for a new package name. SEdit will redisplay atoms with respect to that package.

AGAIN
Meta-R Redo the edit change that was just undone. Redo only works directly following an Undo. Any number of Undo commands can be sequentially redone.

SHIFT-FIND
Meta-S Substitute a structure, or sequence of structures within the current selection. SEdit prompts you in the SEdit prompt window for the structures to replace, and the structures to replace with. The selection to substitute within must be a structure selection.

Control-Meta-S Remove all occurences of a structure or sequence of structures within the current selection. SEdit prompts you for the structures to delete.

UNDO
Meta-U Undo the last edit. All changes in the the edit session are remembered, and can be undone sequentially.

Control-W Delete the previous atom or structure. If the caret is in the middle of an atom,

deletes backward to the beginning of the atom only.

Control-X Tell SEdit that this session is complete. The SEdit window remains open.

EXPAND
Meta-X Replaces the current selection with its definition. This command can be used to expand macros and translate CLISP.

Control-Meta-X Tell SEdit that this session is complete Close the SEdit window.

Meta-Z Mutate. Prompt for a function and call this function with the current selection as the argument. The result is inserted into SEdit and made the current selection.

For example, you can replace a structure with its value by selecting it and mutating by `EVAL`.

Meta-; Convert old style comments in the selected structure to new style comments. The converter notices any list that begins with the symbol IL:* as an old style comment. Section 16.1.18, Options, describes the converter options .

Control-Meta-; Put the contents of a structure selection into a comment. This provides an easy way to "comment out" a chunk of code. The Extract command can be used to reverse this process, returning the comment to the structures contained therein.

Meta-/ Extract one level of structure from the current selection. If there is no selection, but there is a structure caret, the list containing the caret is used. This command can be used to strip the parentheses off a list, or to unquote a quoted structure, or to replace a comment with the contained structures.

Meta-'
Meta-`
Meta-,
Meta-.
Meta-@ or **Meta-2**
Meta-# or **Meta-3**
Meta-. Quote the current selection with the specified kind of quote.

Meta-Space
Meta-Return Scroll the current selection to the center of the window. Similarly, the Space or Return key can be used to normalize the caret.

Meta-)
Meta-0 Parenthesize the current selection, position the caret after the new list.

Meta-(
Meta-9 ParenthesizE the current selection, position the caret at the beginning of the new list.

Meta-M Attach a menu of common commands to the top of the SEdit window. Each SEdit window can have its own menu.

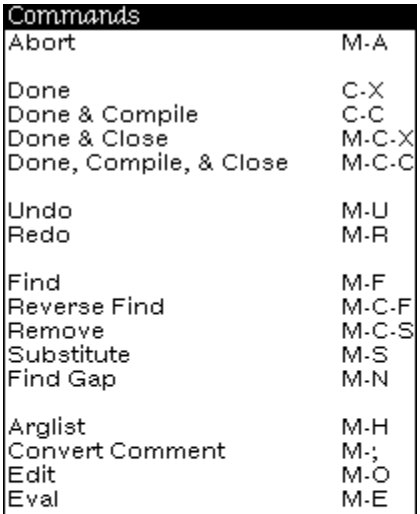
SEdit Command Mnemonics

Abort Meta-A

Change Print Base	Meta-B
Complete	Control-X
Compile & Complete	Control-C
Close, Compile & Complete	Control-Meta-C
Convert Comment	Meta-;
Make Selection Comment	Control-Meta-;
Previous Delete	Control-W
Selection Delete	DELETE
Selection Dot Comma	Meta-.
Selection At Comma	Meta-@
Edit	Meta-O
Fast Edit	Control-Meta-O
Selection Eval	Meta-E
Macro Expand	Meta-X
Forward Find	Meta-F
Reverse Find	Control-Meta-F
Next Gap	Meta-N
Arglist Help	Meta-H
Inspect	Meta-I
Join	Meta-J
Attach Menu	Meta-M
Expression Mutate	Meat-Z
Change Package	Meta-P
Selection Left Parenthesize	Meta-(
Selection Right Parenthesize	Meta-)
Selection Pop	Meta-/
Selection Back Quote	Meta-‘
Selection Hash Quote	Meta-#
Selection Quote	Meta-’
Redisplay	Control-L
Redo	Meta-R
Remove	Control-Meta-S
Substitute	Meta-S
Undo	Meta-U

SEdit Command Menu

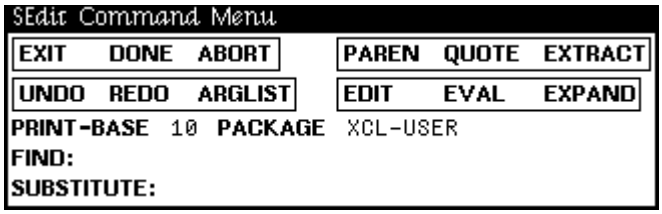
When the mouse cursor is in the SEdit title bar and you press middle mouse button, a Help Menu of commands pops up. The menu looks like this:



The Help Menu lists each command and its corresponding Command Key. (C- stands for Control, M- for Meta.) The menu pops up with the mouse cursor next to the last command you used from the menu. This makes it easy to repeat a command.

SEdit Attached Menu

SEdit's Attached Command Menu contains the commonly used commands. Use the Meta-M keyboard command to bring up this menu. The menu can be closed, independently of the SEdit window. The menu looks like:



Menu commands work like the corresponding keyboard commands, except for Find and Substitute.

For Find, SEdit prompts in the menu window, next to the Find button, for the structures to find. Type in the structures then select Find again. The search begins from the caret position in the SEdit window.

Similarly, Substitute prompts next to the Find button for the structures to find, and next to the Substitute button for the structures to replace them with. After both have been typed in, selecting Substitute replaces all occurrences of the Find structures with the Substitute structures, within the current selection.

To selectively substitute, use Find to find the next potential substitution target. If you want to replace it, select Substitute. Otherwise, select Find again to go on.

Selecting either Find or Substitute with the right mouse button erases the old structure to find or substitute from the menu, and prompts for a new one.

SEdit Programmer's Interface

The following sections describe SEdit's programmer's interface. All symbols are external in the package `SEDIT`.

SEdit Window Region Manager

SEdit provides user redefinable functions which control how SEdit chooses the region for a new edit window. In the follwin thext there are a few concepts that you will have to be familliar with. They are:

The region stack. This is a stack of old used regions. The reason to keep these around is

that the user probably was comfortable with the old position of the window, so when he starts a new SEdit it is a good bet that he will be happy with the old placement.

SEdit uses the respective value of the symbols SEDIT::DEFAULT-FONT, SEDIT::ITALIC-FONT, SEDIT::KEYWORD-FONT, SEDIT::COMMENT-FONT, and SEDIT::BROKEN-ATOM-FONT when displaying an expression. The value of these symbols have to be font descriptors.

(**GET-WINDOW-REGION** *context reason name type*) [Function]

This function is called when SEdit wants to know where to place a window it is about to open. This happens whenever the user starts a new SEdit or expands an Sedit icon. The default behavior is to pop a window region off SEdit's stack of regions that have been used in the past. If the stack is empty, SEdit prompts for a new region.

context is the current editor context.

reason is one of :CREATE or :EXPAND depending on what action prompted the call to GET-WINDOW-REGION

name is the name of the structure to be edited.

type is the edit type of the calling context.

(**SAVE-WINDOW-REGION** *context reason name type region*) [Function]

This function is called whenever SEdit is finished with a region and wants to make the region available for other SEdits. This happens whenever an SEdit window is closed or shrunk, or when an SEdit Icon is closed. The default behavior is simply to push the region onto SEdit's stack of regions.

context is the current editor context.

reason is one of :CLOSE, :SHRINK, or :CLOSE-ICON or depending on what action prompted the call to SAVE-WINDOW-REGION

name is the name of the structure to be edited.

type is the edit type of the calling context.

region is the region to be pushed onto the region stack. If region is NIL the old region of the SEdit will be pushed top the region stack.

KEEP-WINDOW-REGION [Variable]

Default T. This flag determines the behavior of the default SEdit region manager, explained above, for shrinking and expanding windows. When set to T, shrinking an SEdit window will not give up that window's region; the icon will always expand back into the same region. When set to NIL, the window's region is made available for other SEdits when the window is shrunk. Then when an SEdit icon is expanded, the window will be reshaped to the next available region.

This variable is only used by the default implementations of the functions get-window-region and save-window-region. If these functions are redefined, this flag is no longer used.

Options

The following parameters can be set as desired.

WRAP-PARENS [Variable]

This SEdit pretty printer flag determines whether or not trailing close parenthesis characters, `)`, are forced to be visible in the window without scrolling. By default it is set to `NIL`, meaning that close parens are allowed to "fall off" the right edge of the window. If set to `T`, the pretty printer will start a new line before the structure preceding the close parens, so that all the parens will be visible.

WRAP-SEARCH [Variable]

This flag determines whether or not SEdit find will wrap around to the top of the structure when it reaches the end, or vice versa in the case of reverse find. The default is `NIL`.

CLEAR-LINEAR-ON-COMPLETION [Variable]

This flag determines whether or not SEdit completely re-pretty prints the structure being edited when you complete the edit. The default value is `NIL`, meaning that SEdit reuses the pretty printing.

IGNORE-CHANGES-ON-COMPLETION [Variable]

Sometimes the structure that you are editing is changed by the system upon completion. Editdates are an example of this behavior. When this flag is `NIL`, the default, SEdit will redisplay the new structure, capturing the changes. When `T`, SEdit will ignore the fact that changes were made by the system and keep the old structure.

CONVERT-UPGRADE [Variable]

Default 100. When using Meta-; to convert old-style single- asterisk comments, if the length of the comment exceeds convert-upgrade characters, the comment is converted into a double semicolon comment. Otherwise, the comment is converted into a single semicolon comment.

Old-style double-asterisk comments are always converted into new-style triple-semicolon comments.

Control Functions

(**RESET**) [Function]

This function recomputes the SEdit edit environment. Any changes made in the font profile, or any changes made to SEdit's commands are captured by resetting. Close all SEdit windows before calling this function.

(**ADD-COMMAND** *key-code form &optional scroll? key-name command-name help-string*) [Function]

This function allows you to write your own SEdit keyboard commands. You can add

commands to new keys, or you can redefine keys that SEdit already uses as command keys. If you mistakenly redefine an SEdit command, the funtion Reset-Commands will remove all user-added commands, leaving SEdit with its default set of commands.

key-code can be a character code, or any form acceptible to il:charcode.

form determines the function to be called when the key command is typed. It can be a symbol naming a function, or a list, whose first element is a symbol naming a function and the rest of the elements are extra arguments to the function. When the command is invoked, SEdit will apply the function to the edit context (SEdit's main data structure), the charcode that was typed, and any extra arguments supplied in form. The extra arguments do not get evaluated, but are useful as keywords or flags, depending on how the command was invoked. The command function must return T if it handled the command. If the function returns NIL, SEdit will ignore the command and insert the character typed.

The first optional argument, *scroll?*, determines whether or not SEdit scrolls the window after running the command. This argument defaults to NIL, meaning don't scroll. If the value of *scroll?* is T, SEdit will scroll the window to ensure that the caret is visible.

The rest of the optional arguments are used to add this command to SEdit's middle button menu. When the item is selected from the menu, the command function will be called as described above, with the charcode argument set to NIL.

key-name is a string to identify the key (combination) to be typed to invoke the command. For example "M-A" to represent the Meta-A key combination, and "C-M-A" for Control-Meta-A.

command-name is a string to identify the command function, and will appear in the menu next to the key-name.

help-string is a string to be printed in the prompt window when a mouse button is held down over the menu item.

After adding all the commands that you want, you must call Reset-Commands to install them.

For example:

```
(ADD-COMMAND "↑U" (MY-CHANGE-CASE T))

(ADD-COMMAND "↑Y" (MY-CHANGE-CASE NIL))

(ADD-COMMAND "1,R" MY-REMOVE-NIL
  "M-R" "REMOVE NIL"
  "REMOVE NIL FROM THE SELECTED STRUCTURE"))

(RESET-COMMANDS)
```

will add three commands.

Suppose MY-CHANGE-CASE takes the arguments *context*, *charcode*, and *upper-case?*. *upper-case?* will be set to T when MY-CHANGE-CASE is called from Control-U, and NIL when called from Control-Y. MY-REMOVE-NIL will be called with only *context* and *charcode*

arguments when you type Meta-R.

(**RESET-COMMANDS**) [Function]

This function installs all commands added by add-command. SEdits which are open at the time of the reset-commands will not see the new commands; only new SEdits will have the new commands available.

(**DEFAULT-COMMANDS**) [Function]

This function removes all commands added by add-command, leaving SEdit with its default set of commands. As in reset-commands, open SEdits will not be changed; only new SEdits will have the user commands removed.

(**GET-PROMPT-WINDOW** *context*) [Function]

Returns the attached prompt window for a particular SEdit.

(**GET-SELECTION** *context*) [Function]

This function returns two values: the selected structure, and the type of selection, one of `NIL`, `T`, or `:SUB-LIST`. The selection type `NIL` means there is not a valid selection (in this case the structure is meaningless). `T` means the selection is one complete structure. `:SUB-LIST` means a series of elements in a list is selected, in which case the structure returned is a list of the elements selected.

(**REPLACE-SELECTION** *context structure selection-type*) [Function]

This function replaces the current selection with a new structure, or multiple structures, by deleting the selection and then inserting the new structure(s). The selection-type argument must be one of `T` or `:SUB-LIST`. If `T`, the structure is inserted as one complete structure. If `:SUB-LIST`, the structure is treated as a list of elements, each of which is insertd.

EDIT-FN [Variable]

This function is called with the selected structure and the edit specified as arguments to Sedit options as its arguments from the Edit (M-O) command. It should start the editor as appropriate, or generate an error if the selection is not editable.

COMPILE-FN [Variable]

This function is called with the arguments *`name`*, *`type`*, and *`body`*, from the compile/completion commands. It should compile the definition, *`body`*, and install the code as appropriate.

(**SEDIT** *`structure props options`*) [Function]

This function provides a means of starting SEdit directly. *`structure`* is the structure to be edited.

`props` is a property list, which may specify the following properties:

`:NAME` - the name of the object being edited

`:TYPE` - the file manager type of the object being edited. If NIL, SEdit will not call the file

manager when it tries to refetch the definition it is editing. Instead, it will just continue to use the structure that it has.

:COMPLETION-FN - the function to be called when the edit session is completed. This function is called with the *context*, *structure*, and *changed?* arguments. *context* is SEdits main data structure. *structure* is the structure being edited. *changed?* specifies if any changes have been made, and is one of NIL, T, or :ABORT, where :ABORT means the user is aborting the edit and throwing away any changes made. If the value of this property is a list, the first element is treated as the function, and the rest of the elements are extra arguments for the function.

:ROOT-CHANGED-FN - the function to be called when the entire structure being edited is replaced with a new structure. This function is called with the new structure as its argument. If the value of this property is a list, the first element is treated as the function, and the rest of the elements are extra arguments that the function is applied to following the structure argument.

options is one or a list of any number of the followng keywords:

:CLOSE-ON-COMPLETION - This option specifies that SEdit cannot remain active for multiple completions. That is, the SEdit window cannot be shrunk, and the completion commands that normally leave the window open will in this case close the window and terminate the edit.

:COMPILE-ON-COMPLETION - This option specifies that SEdit should call the *COMPILE-FN* to compile the definition being edited upon completion, regardless of the completion command used.

The TTY Editor

This editor the main code editor in pre-window-system versions of Interlisp. For that task, it has been replaced by SEdit.

However, the TTY Editor provides an excellent language for manipulating list structure and making large-scale code changes. For example, several tools for cleaning up code are written using TTY Editor calls to do the actual work.

TTY Editor 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 you perform 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 yopu have 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 you 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 you 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. To facilitate using different control characters for those macros, the function SETTERMCHARS is provided (see below).

Commands That Search

All of the editor commands that search use the same pattern matching routine (the function EDIT4E, below). 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 $$ (escape, escape), *PAT* matches with the atom or string *X* if it is "close" to *PAT*, in the sense used by the spelling corrector (see Chapter 20). For example, CONSS$$ matches with CONS, CNONC$$ with NCONC or NCONC1.

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

8. If (CAR *PAT*) is the atom **--**, *PAT* matches *X* if (CDR *PAT*) matches with some tail of *X*. For example, (A -- (&)) will match with (A B C (D)), but not (A B C D), or (A B C (D) E). However, note that (A -- (&) --) will match with (A B C (D) E). In other words, -- can match any interior segment of a list.

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

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

   This pattern is for use by programs that call the editor as a subroutine, since any non-atomic expression in a command *typed* in by you 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 below.
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 you 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))`.

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 CARs and CDRs descended into) allowed to exceed the value of the variable MAXLEVEL. At that point, the search of that element or tail is abandoned, exactly as though the element or tail had been completely searched without finding a match, and the search continues with the element or tail for which the recursive depth is below MAXLEVEL. This feature is designed to enable you 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 CONSes performed).

If the search is successful, i.e., an expression is found that the pattern matches, the edit chain is set to the value it would have had had you 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" in this chapter).

Search Commands

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

F *PATTERN* [Editor Command]

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

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. 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 Next instance of *PATTERN*, except that the MEMB check of F *PATTERN* is not performed.

F *PATTERN* T [Editor Command]

Similar to F *PATTERN*, except that it may succeed without changing the edit chain, and it does not perform the MEMB check. For example, if the current expression is (COND ...), F COND will look for the next COND, but (F COND T) will "stay here".

(F *PATTERN* N) [Editor Command]

(*N* >= 1) Finds the *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 above).

(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 above).

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.

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 you could always write F COND followed by 2 and 3 for (COND 2 3) if you were not sure whether or not COND was the name of an atomic command.

Secondly, if an error occurs while evaluating one of the commands in the location specification, and the edit chain had been changed, i.e., was not the same as it was at the beginning of that execution of the location specification, the location operation will continue. In other words, the location operation keeps going unless it reaches a state where it detects that it is "looping", at which point it gives up. Thus, if (COND 2 3) is being located, and the first clause of the next COND contained only two elements, the execution of the command 3 would cause an error. The search would then continue by looking for the next COND. However, if a point were reached where there were no further CONDs, then the first command, COND, would cause the error; the edit chain would not have been changed, and so the entire location operation would fail, and cause an error.

The IF command (see above) in conjunction with the ## function (see below) 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.

If *PATTERN* is of the form (IF *EXPRESSION*), *EXPRESSION* is evaluated at each link, and if its value is NIL, or the evaluation causes an error, the ascent continues.

For example:

```
*PP
[PROG NIL
   (COND
      [(NULL (SETQ L (CDR L)))
        (COND
           (FLG (RETURN L]
      ([NULL (CDR (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 you are editing a list of lists, and want 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 you are deep inside of a SELECTQ clause, you can advance to the next clause with (NEX SELECTQ).

NEX [Editor Command]

Same as (NEX _).

The atomic form of NEX is useful if you will be performing repeated executions of (NEX *COM*). By simply MARKing (see the next section) the chain corresponding to *COM*, you 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]

For example, (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 above, 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, you can write commands of the form (COND .. (RETURN .. COND)), which will locate the first COND that contains a RETURN that contains a COND.

Commands That Save and Restore the Edit Chain

Several facilities are available for saving the current edit chain and later retrieving it: MARK, which marks the current chain for future reference, _, which returns to the last mark without destroying it, and __, which returns to the last mark and also erases it.

MARK [Editor Command]

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

← [Editor Command]

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

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

←← [Editor Command]

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

If you have two chains marked, and wish to return to the first chain, you must perform ←←, which removes the second mark, and then _. However, the second mark is then no longer accessible. If you want to be able to return to either of two (or more) chains, you can use the following generalized MARK:

(MARK *SYMBOL*) [Editor Command]

Sets *SYMBOL* to the current edit chain,

(\ SYMBOL) [Editor Command]

Makes the current edit chain become the value of *SYMBOL*.

If you did not prepare in advance for returning to a particular edit chain, you 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 you type F COND, and then F CAR, \ would take you back to the COND. Another \ would take you 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 you type P followed by 3 2 1 P, \P returns to the first P, i.e., would be equivalent to 0 0 0. Another \P would then take you back to the second P, i.e., you could use \P to flip back and forth between the two edit chains.

If you had typed P followed by F COND, you could use *either* \ or \P to return to the P, i.e., the action of \ and \P are independent.

S *SYMBOL* @ [Editor Command]

Sets *SYMBOL* (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.

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 .

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 (Chapter 26). 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 |.

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. You can circumvent this by using the I command, which computes the structure to be used. In the above example, the form of the command would be (I 1 FOO), which would replace the first element with the value of FOO itself.

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.

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, you 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 you perform -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 you perform 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 above. For example, if `@ = (COND 3)` and the next `COND` does not have a thirdelement, the search stops and the `INSERT` fails. You can always write `(LC COND 3)` if you intend 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` above). 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 you perform (DELETE 1), the tail, (C), will have been cut off. Similarly, if the current expression is (CDR Y) and you perform (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.

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 you type F SETQ, and then DELETE, or simply (DELETE SETQ), you 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 you intended. Similarly, if you type (INSERT (RETURN Y) BEFORE SETQ), you mean 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 you might mean make *EXPR* be the function's first argument. Similarly, you cannot write (REPLACE SETQ WITH SETQQ) meaning change the name of the

function. You must in these cases write (INSERT *EXPR* AFTER *FUNCTIONNAME* 1), and (REPLACE SETQ 1 WITH SETQQ).

Occasionally, however, you 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, you may not even *know* whether a particular atom is at the head of a list or not. Thus, when you write (INSERT *EXPR* BEFORE FOO), you mean before the atom FOO, whether or not it is CAR of a list. By setting the variable UPFINDFLG to NIL (initially T), you 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 you intend the operation to refer to the list which FOO heads, use the pattern (FOO --) instead .

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 above). 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). 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). 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 &.

The MOVE Command

The MOVE command allows you to specify the expression to be moved, the place it is to be moved to, and 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), 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 & &))
```

In the last example, you 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, you also perform 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 you 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

*
```

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."

EXTRACT, EMBED, DELETE, REPLACE, and MOVE can be made to operate on several contiguous elements, i.e., a segment of a list, by using in their respective location specifications the TO or THRU command.

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

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

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

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

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

If both @$_1$ and @$_2$ are numbers, and @$_2$ is greater than @$_1$, then @$_2$ counts from the beginning of the current expression, the same as @$_1$. In other words, if the current expression is (A B C D E F G), (3 THRU 5) means (C THRU E) not (C THRU G). In this case, the corresponding BI command is (BI 1 @$_2$-@$_1$+1).

THRU and TO are not very useful commands by themselves; they are intended to be used in conjunction with EXTRACT, EMBED, DELETE, REPLACE, and MOVE. After THRU and TO have operated, they set an internal editor 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.,you had first performed F COND, you 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]
*
```

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 in the Search Algorithm section above, and *X* can employ any of the patterns shown in the Commands That Search section above. 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 you wanted to replace every `D` by `A`, you could perform `(LP (R $D$ $A$))`.

You 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. 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. *X* will never match with a number, i.e., `(R $1 $2)` will not change 11 to 12.

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 you type `(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]

Equivalent to (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))).
```

Commands That Print

`PP` [Editor Command]

Prettyprints the current expression.

`P` [Editor Command]

Prints the current expression as though PRINTLEVEL (Chapter 25) 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 Chapter 26). 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 Chapter 26).

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 (Chapter 14). For example,

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

The command MAKE (see below) is an imperative form of ?=. It allows you 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 (above). All can be aborted with Control-E.

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 (see next section) that you want to abort.

Since all of the commands in the editor are errorset protected, you must exit from the editor via a command. STOP provides a way of distinguishing between a successful and unsuccessful (from your standpoint) editing session. For example, if you are executing (MOVE 3 TO AFTER COND TTY:), and you exitsfrom the lower editor with an OK, the MOVE command will then complete its operation. If you want to abort the MOVE command,

you must make the TTY: command generate an error. Do 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 you are 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 you are 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, Chapter 13) 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 you can always continue editing, including undoing changes from a previous editing session, if one of the following occurs:

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).
2. That session was "sufficiently" recent.
3. It was ended with a SAVE command.

Nested Calls to Editor

TTY: [Editor Command]

Calls the editor recursively. You can then type in commands, and have them executed. The

TTY: command is completed when you exit from the lower editor (see OK and STOP above).

The TTY: command is extremely useful. It enables you 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 you to interact, in effect, *within* the MOVE command. You can then verify for yourself 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 you exit from the lower editor, any attention changing commands you execute only affect the lower editor's edit chain. Of course, if you perform 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.

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 uppercase 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.

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 lowercase *X* to uppercase in the

current expression.

(LOWER *X*) [Editor Command]

Similar to RAISE, except performs (I R *X* (L-CASE *X*)).

In both (RAISE *X*) and (LOWER *X*), *X* should be typed in uppercase.

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"
*
```

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$).

Manipulating Predicates and Conditional Expressions

JOINC [Editor Command]

Used to join two neighboring CONDs 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., you 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 (Chapter 3).

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.

History Commands in the Editor

All of your inputs to the editor are stored on the history list EDITHISTORY (see Chapter 13, 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 you to supply the command name when it was omitted.

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

For example, suppose you want to perform (-2 (SETQ X (LIST Y Z))) but instead types just (SETQ X (LIST Y Z)). The editor will type SETQ ?, whereupon you can type DO -2. The effect is the same as though you 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 you type `(SETQ X &)` and then `!F`, the effect is the same as though you had typed `F (SETQ X &)`, not `(F (SETQ X &))`.

`!E` [Editor Command]

Same as `DO E`.

`!N` [Editor Command]

Same as `DO N`.

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 Chapter 21).

`DW` [Editor Command]

Dwimifies the current expression (see Chapter 21).

`IFY` [Editor Command]

If the current statement is a `COND` statement (Chapter 9), replaces it with an eqivalent `IF` statement.

`GET*` [Editor Command]

If the current expression is a comment pointer (see Chapter 26), 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 N1 N2`) [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.

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)
:
```

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; you 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 third 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.

EVAL, line-feed, and the GO command together effectively allows you 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 the A,B, and : Commands section above). 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 (see the Miscellaneous Commands section above).

(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 you want to compute an entire list of commands for evaluation, as opposed to computing each command one at a time as does the COMS command. You would then write (COMS (CONS 'COMSQ *X*)) where *X* computed the list of commands, e.g., (COMS (CONS 'COMSQ (GETP FOO 'COMMANDS))).

(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, you can simply continue the LP command with REDO (see Chapter 13).

(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 FOOs 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 above) on each instance of the indicated espression(s) so that you 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))).

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 you 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, below).

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 (see the Location Specification section above), could be defined as a macro by (M (2ND) X (ORR ((LC . X) (LC . X)))).

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.

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, your 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 BINDs 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 you to "advise" edit commands.

User macros are stored on a list USERMACROS. The file package command USERMACROS (Chapter 17) is available for dumping all or selected user macros.

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 third element but also restore FOO. Similarly, undoing an S command will undo the set. See the discussion of UNDO in Chapter 13. (If the I command was typed directly to the editor, /NCONC would automatically be substituted for NCONC as described in Chapter 13.)

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 you perform an INSERT, and then an MBD, the first UNDO will undo the MBD, and the second will undo the INSERT. However, you can also specify precisely which commands you want undone by identifying the corresponding entry. In this case, you 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 you *continue* 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 you enter the editor continuing a session, and immediately execute an UNDO or !UNDO, the editor will type BLOCKED instead of NOTHING SAVED. Similarly, if you execute several commands and then undo 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., you 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 Chapter 13). 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, you do 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 your accidentally undoing a random command by giving the wrong event specification.

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, you cannot advise or redefine it as a means of augmenting or extending the editor. However, you 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 RPLACAed into the command, and the editor continues by executing the command. In other words, if you type (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 (see Chapter 13), so the line can be terminated by a square bracket, or by a carriage return not preceded by a space. You 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. For example:

```
*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).

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 you forgot to type E and intend 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 you 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
          *
```

7. Otherwise, generate an error.

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 (see Chapter 12), the function SETINITIALS is called. SETINITIALS searches INITIALSLST, a list of elements of the form (*USERNAME . INITIALS*) or (*USERNAME FIRSTNAME INITIALS*). If your name is found, INITIALS is set accordingly. If your username 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, you must either include an entry of the form (*USERNAME*) on INITIALSLST, or set DEFAULTINITIALS to NIL before greeting, i.e. in your user profile, or else, *after* greeting, explicitly

set INITIALS to NIL.

If you want your functions to be time stamped with your initials when edited, include a file package command command of the form (ADDVARS (INITIALSLST (*USERNAME* . *INITIALS*))) in your INIT.LISP file (see Chapter 12).

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.

Warning with Declarations

CAUTION: There is a feature of the BYTECOMPILER that is not supported by SEdit or the XCL compiler. It is possible to insert a comment at the beginning of your function that looks like

(* DECLARATIONS: --)

The tail, or -- section, of this comment is taken as a set of local record declarations which are then used by the compiler in that function just as if they had been declared globally. See the "Compiler" section in Chapter 3 of these Notes for additional behavior in XCL.

SEdit does not recognize such declarations. Thus, if the "Expand" command is used, the expansion will not be done with these record declarations in effect. The code that you see in SEdit will not be the same code compiled by the BYTECOMPILER.

[This page intentionally left blank]