CHAPTER 8

THE PROGRAMMER'S ASSISTANT

8.1 INTRODUCTION

With any interactive computer language, the user interacts with the system through an ‘‘executive’’, which interprets and executes typed-in commands. In most implementations of Lisp, the executive is a simple ‘‘read-eval-print’’ loop, which repeatedly reads a Lisp expression, evaluates it, and prints out the value of the expression. Interlisp has an executive which allows a much greater range of inputs, other than just regular Interlisp expressions.

In particular, the Interlisp executive implements a facility known as the ‘‘programmer's assistant’’ (or ‘‘p.a.’’). The central idea of the programmer's assistant is that the user is addressing an active intermediary, namely his assistant. Normally, the assistant is invisible to the user, and simply carries out the user's requests. However, the assistant remembers what the user has done, so the user can give commands to repeat a particular operation or sequence of operations, with possible modi cations, or to undo the e ect of speci ed operations. Like DWIM, the programmer's assistant embodies an approach to system design whose ultimate goal is to construct an environment that ‘‘cooperates’’ with the user in the development of his programs, and frees him to concentrate more fully on the conceptual di culties and creative aspects of the problem at hand.

We will rst discuss the various input formats, then the use of commands to the programmer's assistant, and nally how to modify the programmer's assistant for specialized uses.

8.1.1 Input Formats

The Interlisp executive accepts inputs in the following formats:

(1) A single litatom, followed by a carriage-return. The value of the litatom is returned. For the purposes of this discussion, we will call this EVALV-format.

(2) A regular Interlisp expression, beginning with a left parenthesis or square bracket and terminated by a matching right parenthesis or square bracket. A right bracket matches any number of left parentheses, back to the last left bracket or the entire expression. Such an input is known as an ‘‘EVAL-format’’ input, since the form is simply passed to `EVAL` for evaluation. Notice that it is not necessary to type a carriage return at the end of such a form; Interlisp will supply one automatically. If a carriage-return is typed before the nal matching right parenthesis or bracket, it is treated as a space, and input continues. The following examples are all interpreted the same:

```
_(PLUS 1 (TIMES 2 3))
```

```
_(PLUS 1 (TIMES 2 3]
_(PLUS 1 (TIMES cr
2 3]
```

(3) Often, the user, typing at the keyboard, calls functions with constant argument values, which would have to be quoted if the user typed it in ‘‘EVAL-format’’. For convience, if the user types a litatom immediately followed by a list form, the litatom is `APPLY`ed to the elements within the list, unevaluated. For example, typing `LOAD(FOO)` is equivalent to typing `(LOAD 'FOO)`, and `GETPROP(X COLOR)` is equivalent to `(GETPROP 'X 'COLOR)`. The input is terminated by the matching right parenthesis or bracket. We will call such input ‘‘APPLY-format.’’ APPLY-format input is useful in some situations, but note that it may produce unexpected results when an *nlambda* function is called that explicitly evaluates its arguments. For example, typing `SETQ(FOO BAR)` will set `FOO` to the *value* of `BAR`, not to `BAR` itself.

However, there are times when a user does not want to terminate the input when a closing parenthesis is typed especially when giving a command to the programmer's assistant. This leads us to our fourth format.

(4) A sequence of litatoms and lists *beginning with* a litatom and a space (to distinguish it from APPLY-format), terminated by a carriage return or an extra right parenthesis or bracket. If a list is terminated then Interlisp will type a carriage-return and `"..."` to indicate that further input will be accepted. The user can type further expressions or terminate the whole expression by a carriage-return.

Once the input is terminated, the programmer's assistant decides how to evaluate the expression. This determination relies on a heuristic that says ‘‘If there is only expression, then assume EVALV-format. If there are two expressions, then assume APPLY-format. If there are three or more expressions, then assume EVAL-format.’’ The following inputs are examples of this rule:

```
_FOO<space> cr
```
same as FOO^{cr} EVALV- format

```
_LIST (A B)
... cr
```
same as LIST(A B) APPLY- format

```
_PLUS (TIMES 2 3)
...1 cr
```
same as (PLUS (TIMES 2 3) 1) EVAL- format

8.1.2 Examples

So far, we have dealt only with how the executive instructs Interlisp to evaluate input. However, the same scheme also allows the user to give commands directly to the programmer's assistant. In fact, in each of the above cases, it is rst determined whether the initial litatom is a command to the programmer's assistant. If so, the normal lisp evaluation process is bypassed. Note that this means that a function or variable with the same name as a programmer's assistant command will not be evaluated (in the normal lisp sense) if it is the rst litatom of an expression input to the executive.

The programmer's assistant facility features the use of memory structures called ‘‘history lists.’’ A history list is a list of the information associated with each of the individual ‘‘events’’ that have occurred in the system, where each event corresponds to one user input. Associated with each event on the history list is the input and its value, plus other optional information such as side-e ects, formatting information, etc.

The following dialogue, taken from an actual session at the terminal, contains illustrative (but not necessarily useful) examples and gives the avor of the programmer’s assistant facility in Interlisp. The number before each prompt is the ‘‘event number’’ (see page 8.26).

```
12_(SETQ FOO 5)
5
13_(SETQ FOO 10)
(FOO reset)
10
```

The p.a. notices that the user has reset the value of FOO and informs the user.

```
14_UNDO
SETQ undone.
15_FOO cr
5
```

This is the rst example of direct communication with the p.a. The user has said to UNDO the previous input to the executive.

```
.
.
.

25_SET(LST1 (A B C))
(A B C)
26_(SETQ LST2 ’(D E F))
(D E F)
27_(FOR X IN LST1 DO (REMPROP X ’MYPROP]
NIL
```

The user asked to remove the property MYPROP from the atoms A, B, and C. Now lets assume that is not what he wanted to do, but rather use the elements of LST2.

```
28_UNDO FOR
FOR undone.
```

First he undoes the REMPROP, by undoing the iterative statement. Notice the UNDO accepted an ‘‘argument,’’ although in this case UNDO by itself would be su cient.

```
29_USE LST2 FOR LST1 IN 27
NIL
```

The user just instructed to go back to event number 27 and substitute LST2 for LST1 and then reexecute the expression. The user could have also speci ed -2 instead of 27 to specify a relative address.

```
.
.
.
```

Examples

```
47_(PUTHASH ’FOO (MKSTRING ’FOO) MYHASHARRAY)
"FOO"
```

If MKSTRING was a computationally expensive function (which it is not), then the user might be cacheing its value for later use.

```
48_USE FIE FUM FOE FOR FOO IN MKSTRING
"FIE"
"FUM"
"FOE"
```

The user now decides he would like to redo the PUTHASH several times with di erent values. He speci es the event by ‘‘IN MKSTRING’’ rather than PUTHASH.

```
49_?? USE

48.      USE FIE FUM FOE FOR FOO IN MKSTRING
     _(PUTHASH (QUOTE FIE) (MKSTRING (QUOTE FIE)) MYHASHARRAY)
     "FIE"
     _(PUTHASH (QUOTE FUM) (MKSTRING (QUOTE FUM)) MYHASHARRAY)
     "FUM"
     _(PUTHASH (QUOTE FOE) (MKSTRING (QUOTE FOE)) MYHASHARRAY)
     "FOE"
```

Here we see the user ask the p.a. (using the `??` *command) what it has on its history list for the last input to the executive. Since the event corresponds to a programmer’s assistant command that evaluates several forms, these forms are saved as the input, although the user’s actual input, the p.a. command, is also saved in order to clarify the printout of that event.*

As stated earlier, the most common interaction with the programmer’s assistant occurs at the top level read-eval-print loop, or in a break, where the user types in expressions for evaluation, and sees the values printed out. In this mode, the assistant acts much like a standard Lisp executive, except that before attempting to evaluate an input, the assistant rst stores it in a new entry on the history list. Thus if the operation is aborted or causes an error, the input is still saved and available for modi cation and/or reexecution. The assistant also notes new functions and variables to be added to its spelling lists to enable future corrections. Then the assistant executes the computation (i.e., evaluates the form or applies the function to its arguments), saves the value in the entry on the history list corresponding to the input, and prints the result, followed by a prompt character to indicate it is again ready for input.

If the input typed by the user is recognized as a p.a. command, the assistant takes special action. Commands such as `UNDO` and `??` are immediately performed. Commands that involved reexecution of previous inputs, such as `REDO` and `USE`, are achieved by computing the corresponding input expression(s) and then *unreading* them. The e ect of this unreading operation is to cause the assistant’s input routine, `LISPXREAD`, to act exactly as though these expressions were typed in by the user. These expressions are processed exactly as though they had been typed, except that they are not saved on new and separate entries on the history list, but associated with the history command that generated them.

The net e ect of this implementation of the programmer’s assistant is to provide a facility which is easily inserted at many levels, and embodies a consistent set of commands and conventions for talking about past events. This gives the user the subjective feeling that a single agent is watching everything he does and says, and is always available to help.

8.2 PROGRAMMER'S ASSISTANT COMMANDS

The programmer's assistant recognizes a number of commands, which usually refer to past events on the history list. These commands are treated specially; for example, they may not be put on the history list.

Note: If the user de nes a function by the same name as a p.a. command, a warning message is printed to remind him that the p.a. command interpretation will take precedence for type-in.

All programmer's assistant commands use the same conventions and syntax for indicating which event or events on the history list the command refers to, even though di erent commands may be concerned with di erent aspects of the corresponding event(s), e.g., side-e ects, value, input, etc. Therefore, before discussing the various p.a. commands, the following section describes the types of event speci cations currently implemented.

8.2.1 Event Speci cation

An event address identi es one event on the history list. It consists of a sequence of ‘‘commands’’ for moving an imaginary cursor up or down the history list, much in the manner of the arguments to the `@` break command (see page 9.3). The event identi ed is the one ‘‘under’’ the imaginary cursor when there are no more commands. (If any command fails, an error is generated and the history command is aborted.) For example, the event address `42` refers to the event with event number 42, `42 FOO` refers to the rst event (searching back from event 42) whose input contains the word `FOO`, and `42 FOO -1` refers to the event preceeding that event. Usually, an event address will contain only one or two commands.

Most of the event address commands perform searches for events which satisfy some condition. Unless the _ command is given (see below), this search always goes backwards through the history list, from the most recent event speci ed to the oldest. Note that each search skips the current event. For example, if `FOO` refers to event `N`, `FOO FIE` will refer to some event before event `N`, even if there is a `FIE` in event `N`.

The event address commands are interpreted as follows:

`N` (an integer)	If `N` is the rst command in an event address, refers to the event with event number `N`. Otherwise, refers to the event `N` events forward (in direction of increasing event number). If `N` is negative, it always refers to the event `-N` events backwards. For example, `-1` refers to the previous event, `42` refers to event number 42 (if the rst command in an event address), and `42 3` refers to the event with event number 45.
`_LITATOM`	Speci es the last event with an `APPLY`-format input whose *function* matches `LITATOM`. Note: There must not be a space between _ and `LITATOM`.
_	Speci es that the next search is to go forward instead of backward. If given as the rst event address command, the next search begins with last (oldest) event on the history list.
`F`	Speci es that the next object in the event address is to be searched for, regardless

of what it is. For example, `F -2` looks for an event containing `-2`.

`=`	Speci es that the next object (presumably a pattern) is to be matched against the *values* of events, instead of the inputs.
`\`	Speci es the event last located.
`SUCHTHAT PRED`	Speci es an event for which the function `PRED` returns true. `PRED` should be a function of two arguments, the input portion of the event, and the event itself. See page 8.25 for a discussion of the format of events on the history list.
`PAT`	Any other event address command speci es an event whose input contains an expression that matches `PAT` as described in page 17.13.

The matching is performed by the function `HISTORYMATCH` (page 8.33), which is initially de ned to call `EDITFINDP` but can be advised or rede ned for specialized applications.

Note: Symbols used below of the form $\texttt{EventAddress}_i$ refer to event addresses, described above. Since an event address may contain multiple words, the event address is parsed by searching for the words which delimit it. For example, in `FROM` $\texttt{EventAddress}_1$ `THRU` $\texttt{EventAddress}_2$, the symbol $\texttt{EventAddress}_1$ corresponds to all words between `FROM` and `THRU` in the event speci cation, and $\texttt{EventAddress}_2$ to all words from `THRU` to the end of the event speci cation.

`FROM` $\texttt{EventAddress}_1$ `THRU` $\texttt{EventAddress}_2$
$\texttt{EventAddress}_1$ `THRU` $\texttt{EventAddress}_2$

Speci es the sequence of events from the event with address $\texttt{EventAddress}_1$ through the event with address $\texttt{EventAddress}_2$. For example, `FROM 47 THRU 49` speci es events 47, 48, and 49. $\texttt{EventAddress}_1$ can be more recent than $\texttt{EventAddress}_2$. For example, `FROM 49 THRU 47` speci es events 49, 48, and 47 (note reversal of order).

`FROM` $\texttt{EventAddress}_1$ `TO` $\texttt{EventAddress}_2$
$\texttt{EventAddress}_1$ `TO` $\texttt{EventAddress}_2$

Same as `THRU` but does not include event $\texttt{EventAddress}_2$.

`FROM` $\texttt{EventAddress}_1$	Same as `FROM` $\texttt{EventAddress}_1$ `THRU -1`. For example, if the current event is number 53, then `FROM 49` speci es events 49, 50, 51, and 52.
`THRU` $\texttt{EventAddress}_2$	Same as `FROM -1 THRU` $\texttt{EventAddress}_2$. For example, if the current event is number 53, then `THRU 49` speci es events 52, 51, 50, and 49 (note reversal of order).
`TO` $\texttt{EventAddress}_2$	Same as `FROM -1 TO` $\texttt{EventAddress}_2$.
`ALL` $\texttt{EventAddress}_1$	Speci es all events satisfying $\texttt{EventAddress}_1$. For example, `ALL LOAD`, `ALL SUCHTHAT FOO`.
empty	If nothing is speci ed, it is the same as specifying `-1`.

Note: In the special case that the last event was an `UNDO`, it is the same as specifying `-2`. For example, if the user types `(NCONC FOO FIE)`, he can then type `UNDO`, followed by `USE NCONC1`.

EventSpec$_1$ AND EventSpec$_2$ AND … AND EventSpec$_N$
: Each of the EventSpec$_i$ is an event specification. The lists of events are concatenated. For example, FROM 30 THRU 32 AND 35 THRU 37 is the same as 30 AND 31 AND 32 AND 35 AND 36 AND 37.

@ LITATOM
: If LITATOM is the name of a command defined via the NAME command (page 8.12), specifies the event(s) defining LITATOM.

@@ EventSpec
: EventSpec is an event specification interpreted as above, but with respect to the archived history list (see page 8.13).

If no events can be found that satisfy the event specification, spelling correction on each word in the event specification is performed using LISPXFINDSPLST as the spelling list. For example, REDO 3 THRUU 6 will work correctly. If the event specification still fails to specify any events after spelling correction, an error is generated.

8.2.2 Commands

All programmer's assistant commands can be input as list forms, or as lines (see page 8.30). For example, typing REDO 5^{cr} and (REDO 5) are equivalent.

EventSpec is used to denote an event specification. Unless specified otherwise, omitting EventSpec is the same as specifying EventSpec = -1. For example, REDO and REDO -1 are the same.

REDO EventSpec [Prog. Asst. Command]
: Redoes the event or events specified by EventSpec. For example, REDO FROM -3 redoes the last three events.

REDO EventSpec N TIMES [Prog. Asst. Command]
: Redoes the event or events specified by EventSpec N times. For example, REDO 10 TIMES redoes the last event ten times.

REDO EventSpec WHILE FORM [Prog. Asst. Command]
: Redoes the specified events as long as the value of FORM is true. FORM is evaluated before each iteration so if its initial value is NIL, nothing will happen.

REDO EventSpec UNTIL FORM [Prog. Asst. Command]
: Same as REDO EventSpec WHILE (NOT FORM).

REPEAT EventSpec [Prog. Asst. Command]
: Same as REDO EventSpec WHILE T. The event(s) are repeated until an error occurs, or the user types control-E or control-D.

REPEAT EventSpec WHILE FORM [Prog. Asst. Command]
REPEAT EventSpec UNTIL FORM [Prog. Asst. Command]
: Same as REDO.

For all history commands that perform multiple repetitions, the variable REDOCNT is initialized to 0 and incremented each iteration. If the event terminates gracefully, i.e., is not aborted by an error or control-D, the number of iterations is printed.

`RETRY` EventSpec [Prog. Asst. Command]

Similar to `REDO` except sets `HELPCLOCK` (page 9.11) so that any errors that occur while executing EventSpecwill cause breaks.

`USE` EXPRS `FOR` AR GS `IN` EventSpec [Prog. Asst. Command]

Substitutes EXPRS for AR GS in EventSpec, and redoes the result. Substitution is done by `ESUBST` (page 17.57), and is carried out as described below. EXPRS and AR GS can include non-atomic members.

For example, `USE LOG (MINUS X) FOR ANTILOG X IN -2 AND -1` will substitute `LOG` for every occurrence of `ANTILOG` in the previous two events, and substitute `(MINUS X)` for every occurrence of `X`, and reexecute them. Note that these substitutions do not change the information saved about these events on the history list.

Any expression to be substituted can be preceded by a `!`, meaning that the expression is to be substituted as a *segment*, e.g., `LIST(A B C)` followed by `USE ! (X Y Z) FOR B` will produce `LIST(A X Y Z C)`, and `USE ! NIL FOR B` will produce `LIST(A C)`.

If `IN` EventSpecis omitted, the rst member of AR GS is used for EventSpec. For example, `USE PUTD FOR @UTD` is equivalent to `USE PUTD FOR @UTD IN F @UTD`. The `F` is inserted to handle correctly the case where the rst member of AR GS could be interpreted as an event address command.

`USE` EXPRS `IN` EventSpec [Prog. Asst. Command]

If AR GS are omitted, and the event referred to was itself a `USE` command, the arguments and expression substituted into are the same as for the indicated `USE` command. In e ect, this `USE` command is thus a continuation of the previous `USE` command. For example, following `USE X FOR Y IN 50`, typing `USE Z IN -1` is equivalent to `USE Z FOR Y IN 50`.

If AR GS are omitted and the event referred to was *not* a `USE` command, substitution is for the ‘‘operator’’ in that command. For example `ARGLIST(FF)` followed by `USE CALLS IN -1` is equivalent to `USE CALLS FOR ARGLIST IN -1`.

If `IN` EventSpecis omitted, it is the same as specifying `IN -1`.

`USE` EXPRS $_1$ `FOR` AR GS $_1$ `AND` `AND` EXPRS $_N$ `FOR` AR GS $_N$ `IN` EventSpec [Prog. Asst. Command]

More general form of `USE` command. See description of the substitution algorithm below.

Note: The `USE` command is parsed by a small nite state parser to distinguish the expressions and arguments. For example, `USE FOR FOR AND AND AND FOR FOR` will be parsed correctly.

Every `USE` command involves three pieces of information: the expressions to be substituted, the arguments to be substituted for, and an event speci cation, which de nes the input expression in which the substitution takes place. If the `USE` command has the same number of expressions as arguments, the substitution procedure is straightforward. [1] For example, USE X Y FOR U V means substitute X for U and Y for V, and is equivalent to USE X FOR U AND Y FOR V. However, the USE command also permits distributive substitutions, for substituting several expressions for the same argument. For example, USE A B C FOR X means rst substitute A for X then substitute B for X (in a new copy of the expression), then substitute C for X. The e ect is the same as three separate USE commands. Similarly, USE A B C FOR D AND X Y Z FOR W is equivalent to USE A FOR D AND X FOR W, followed by USE B FOR D AND Y FOR W, followed by USE C FOR D AND Z FOR W. USE A B C FOR D AND X FOR Y also corresponds to three substitions, the rst with A for D and X for Y, the second with B for D, and X for Y, and the third with C for D, and again X for Y. However, USE A B C FOR D AND X Y FOR Z is ambiguous and will cause an error. Essentially, the USE command operates by proceeding from left to right handling each ‘‘AND’’ separately. Whenever the number of expressions exceeds the number of expressions available, multiple USE expressions are generated. Thus USE A B C D FOR E F means substitute A for E at the same time as substituting B for F, then in another copy of the indicated expression, substitute C for E and D for F. Note that this is also equivalent to USE A C FOR E AND B D FOR F.

... VARS [Prog. Asst. Command]

Similar to USE except substitutes for the (rst) *operand*.

For example, EXPRP(FOO) followed by ... FIE FUM is equivalent to USE FIE FUM FOR FOO.

Note: In the following discussion, $ is used to represent the character <esc>, since this is how <esc> is echoed.

$ X FOR Y IN EventSpec [Prog. Asst. Command]

$ is a special form of the USE command for conveniently specifying *character* substitutions in litatoms or strings. In addition, it has a number of useful properties in connection with events that involve errors (see below).

Equivalent to USE X FOR Y IN EventSpec, which will do a character substitution of the characters in X for the characters in Y.

For example, if the user types MOVD(FOO FOOSAVE T), he can then type $ FIE FOR FOO IN MOVD to perform MOVD(FIE FIESAVE T). Note that USE FIE FOR FOO would perform MOVD(FIE FOOSAVE T).

$ Y X IN EventSpec [Prog. Asst. Command]
$ Y TO X IN EventSpec [Prog. Asst. Command]
$ Y = X IN EventSpec [Prog. Asst. Command]
$ Y -> X IN EventSpec [Prog. Asst. Command]

Abbreviated forms of the $ command: the same as $ X FOR Y IN EventSpec, which changes Ys to Xs.

$ does event location the same as the USE command, i.e., if IN EventSpec is not speci ed, $ searches for Y. However, unlike USE, $ can only be used to specify one substitution at a time. After $ nds the event, it looks to see if an error was involved in that event, and if the indicated character substitution can be performed in the object of the error message, called the o ender. If so, $ assumes the substitution refers

[1] Except when one of the arguments and one of the expressions are the same, e.g., USE X Y FOR Y X, or USE X FOR Y AND Y FOR X. This situation is noticed when parsing the command, and handled correctly.

to the o ender, performs the indicated character substitution in the o ender only, and then substitutes the result for the original o ender throughout the event. For example, suppose the user types `(PRETTYDEF FOOFNS 'FOO FOOOVARS)` causing a `U.B.A. FOOOVARS` error message. The user can now type `$ OO O`, which will change `FOOOVARS` to `FOOVARS`, but *not* change `FOOFNS` or `FOO`.

If an error did occur in the speci ed event, the user can also omit specifying the object of the substitution, `Y`, in which case the o ender itself is used. Thus, the user could have corrected the above example by simply typing `$ FOOVARS`. Since `ESUBST` is used for performing the substitution (see page 17.57), `$` can be used in `X` to refer to the characters in `Y`. For example, if the user types `LOAD(PRSTRUC PROP)`, causing the error `FILE NOT FOUND PRSTRUC`, he can request the le to be loaded from `LISP`'s directory by simply typing `$ <LISP>$`. This is equivalent to performing `(R PRSTRUC <LISP>$)` on the event, and therefore replaces `PRSTRUC` by `<LISP>PRSTRUC`.

Note that `$` never *searches* for an error. Thus, if the user types `LOAD(PRSTRUC PROP)` causing a `FILE NOT FOUND` error, types `CLOSEALL()`, and *then* types `$ <LISP>$`, `LISPX` will complain that there is no error in `CLOSEALL()`. In this case, the user would have to type `$ <LISP>$ IN LOAD`, or `$ PRS <LISP>PRS` (which would cause a search for `PRS`).

Note also that `$` operates on *input*, not on programs. If the user types `FOO()`, and within the call to `FOO` gets a `U.D.F. CONDD` error, he *cannot* repair this by `$ COND`. `LISPX` will type `CONDD NOT FOUND IN FOO()`.

`FIX EventSpec` [Prog. Asst. Command]

Envokes the default program editor (Dedit or the teletype editor) on a copy of the input(s) for `EventSpec` Whenever the user exits via `OK`, the result is unread and reexecuted exactly as with `REDO`.

`FIX` is provided for those cases when the modi cations to the input(s) are not simple substitutions of the type that can be speci ed by `USE`. For example, if the default editor is the teletype editor, then:

```
_(DEFINEQ FOO (LAMBDA (X) (FIXSPELL SPELLINGS2 X 70]
INCORRECT DEFINING FORM
FOO
_FIX
EDIT
*P
(DEFINEQ FOO (LAMBDA & &))
*(LI 2)
*P
(DEFINEQ (FOO &))
*OK
(FOO)
_
```

The user can also specify the edit command(s) to `LISPX`, by typing `-` followed by the command(s) after the event speci cation, e.g., `FIX - (LI 2)`. In this case, the editor will not type `EDIT`, or wait for an `OK` after executing the commands.

Note: `FIX` calls the editor on the ‘‘input sequence’’ of an event, adjusting the editor so it is initially editing the expression typed. However, the entire input sequence is being edited, so it is possible to give editor commands that examine this structure further. For more information on the format of an event's input, see page 8.25.

`?? EventSpec` [Prog. Asst. Command]

Prints the speci ed events from the history list. If `EventSpec`is omitted, `??` prints the entire history list, beginning with most recent events. Otherwise `??` prints only those events speci ed in `EventSpec`(in the order speci ed). For example, `?? -1`, `?? 10 THRU 15`, etc.

For each event speci ed, `??` prints the event number, the prompt, the input line(s), and the value(s). If the event input was a p.a. command that ‘‘unread’’ some other input lines, the p.a. command is printed without a preceding prompt, to show that they are not stored as the input, and the input lines are printed with prompts.

Events are initially stored on the history list with their value eld equal to the character ‘‘bell’’ (control-G). Thefore, if an operation fails to complete for any reason, e.g., causes an error, is aborted, etc., `??` will print a bell as its ‘‘value’’.

`??` commands are not entered on the history list, and so do not a ect relative event numbers. In other words, an event speci cation of `-1` typed following a `??` command will refer to the event immediately preceding the `??` command.

`??` is implemented via the function `PRINTHISTORY`, page 8.35, which can also be called directly by the user. Printing is performed via the function `SHOWPRIN2` (page 6.17), so that if the value of `SYSPRETTYFLG = T`, events will be prettyprinted.

`UNDO EventSpec` [Prog. Asst. Command]

Undoes the side e ects of the speci ed events. For each event undone, `UNDO` prints a message: `RPLACA UNDONE`, `REDO UNDONE` etc. If nothing is undone because nothing was saved, `UNDO` types `NOTHING SAVED`. If nothing was undone because the event(s) were already undone, `UNDO` types `ALREADY UNDONE`.

If `EventSpec`is not given, `UNDO` searches back for the last event that contained side e ects, was not undone, and itself was not an `UNDO` command. Note that the user can undo `UNDO` commands themselves by specifying the corresponding event address, e.g., `UNDO -7` or `UNDO UNDO`.

In order to restore all pointers correctly, the user should `UNDO` events in the reverse order from which they were executed. For example, to undo all the side e ects of the last ve events, perform `UNDO THRU -5`, *not* `UNDO FROM -5`. Undoing out of order may have unforseen e ects if the operations are *dependent*. For example, if the user performed `(NCONC1 FOO FIE)`, followed by `(NCONC1 FOO FUM)`, and then undoes the `(NCONC1 FOO FIE)`, he will also have undone the `(NCONC1 FOO FUM)`. If he then undoes the `(NCONC1 FOO FUM)`, he will cause the `FIE` to reappear, by virtue of restoring `FOO` to its state before the execution of `(NCONC1 FOO FUM)`. For more details, see page 8.23.

`UNDO EventSpec :` X_1 X_N [Prog. Asst. Command]

Each X_i is a pattern that is matched to a message printed by DWIM in the event(s) speci ed by `EventSpec` The side e ects of the corresponding DWIM corrections, and only those side e ects, are undone.

For example, if DWIM printed the message `PRINTT [IN FOO] -> PRINT`, then `UNDO : PRINTT` or `UNDO : PRINT` would undo the correction.

Some portions of the messages printed by DWIM are strings, e.g., the message `FOO UNSAVED` is printed by printing `FOO` and then `" UNSAVED"`. Therefore, if the user types UNDO : UNSAVED, the DWIM correction will not be found. He should instead type UNDO : FOO or UNDO : $UNSAVED$ (<esc>UNSAVED<esc>, see R command in editor, page 17.35).

NAME LITATOM EventSpec [Prog. Asst. Command]

Saves the event(s) (including side e ects) speci ed by EventSpecon the property list of LITATOM (under the property HISTORY). For example, NAME FOO 10 THRU 15. NAME commands are undoable.

Events saved on a litatom can be retrieved with the event speci cation @ LITATOM . For example, ?? @ FOO, REDO @ FOO, etc.

Commands de ned by NAME can also be typed in directly as though they were built- in commands, e.g., FOO^{cr} is equivalent to REDO @ FOO. However, if FOO is the name of a variable, it would be evaluated, i.e., FOO^{cr} would return the value of FOO.

Commands de ned by NAME can also be de ned to take arguments:

NAME LITATOM (ARG$_1$ ARG$_N$) : EventSpec [Prog. Asst. Command]
NAME LITATOM ARG$_1$ ARG$_N$: EventSpec [Prog. Asst. Command]

The arguments ARG$_i$ are interpreted the same as the arguments for a USE command. When LITATOM is invoked, the argument values are substituted for ARG$_1$ ARG$_N$ using the same substitution algorithm as for USE.

NAME FOO EventSpecis equivalent to NAME FOO : EventSpec In either case, if FOO is invoked *with* arguments, an error is generated.

For example, following the event (PUTD 'FOO (COPY (GETPROP 'FIE 'EXPR))), the user types NAME MOVE FOO FIE : PUTD. Then typing MOVE TEST1 TEST2 would cause (PUTD 'TEST1 (COPY (GETPROP 'TEST2 'EXPR))) to be executed, i.e., would be equivalent to typing USE TEST1 TEST2 FOR FOO FIE IN MOVE. Typing MOVE A B C D would cause two PUTD's to be executed. Note that !'s and $'s can also be employed the same as with USE. For example, if following

```
_PREPINDEX(<MANUAL>14LISP.XGP)
_FIXFILE(<MANUAL>14LISP.XGPIDX)
```

the user performed NAME FOO 14 : -2 AND -1, then FOO 15 would perform the indicated two operations with 14 replaced by 15.

RETRIEVE LITATOM [Prog. Asst. Command]

Retrieves and reenters on the history list the events named by LITATOM . Causes an error if LITATOM was not named by a NAME command.

For example, if the user performs NAME FOO 10 THRU 15, and at some time later types RETRIEVE FOO, 6 *new* events will be recorded on the history list (whether or not the corresponding events have been forgotten yet). Note that RETRIEVE does *not* reexecute the events, it simply retrieves them. The user can then REDO, UNDO, FIX, etc. any or all of these events. Note that the user can combine the e ects of a RETRIEVE and a subsequent history command in a single operation, e.g., REDO FOO is equivalent to RETRIEVE FOO, followed by an appropriate REDO. Actually, REDO FOO is better than RETRIEVE followed by REDO since in the latter case, the corresponding events would be entered on the history list *twice*, once for the RETRIEVE and once for the REDO. Note that UNDO FOO and ?? FOO are permitted.

BEFORE LITATOM [Prog. Asst. Command]

Undoes the effects of the events named by LITATOM.

AFTER LITATOM [Prog. Asst. Command]

Undoes a BEFORE LITATOM.

BEFORE and AFTER provide a convenient way of flipping back and forth between two states, namely the state *before* a specified event or events were executed, and that state *after* execution. For example, if the user has a complex data structure which he wants to be able to interrogate before and after certain modifications, he can execute the modifications, name the corresponding events with the NAME command, and then can turn these modifications off and on via BEFORE or AFTER commands. Both BEFORE and AFTER are no-ops if the LITATOM was already in the corresponding state; both generate errors if LITATOM was not named by a NAME command.

The alternative to BEFORE and AFTER for repeated switching back and forth involves typing UNDO, UNDO of the UNDO, UNDO of that etc. At each stage, the user would have to locate the correct event to undo, and furthermore would run the risk of that event being ‘‘forgotten’’ if he did not switch at least once per time-slice.

Note: Since UNDO, NAME, RETRIEVE, BEFORE, and AFTER are recorded as inputs they can be referenced by REDO, USE, etc. in the normal way. However, the user must again remember that the context in which the command is reexecuted is different than the original context. For example, if the user types NAME FOO DEFINEQ THRU COMPILE, then types ... FIE, the input that will be reread will be NAME FIE DEFINEQ THRU COMPILE as was intended, but both DEFINEQ and COMPILE, will refer to the most recent event containing those atoms, namely the event consisting of NAME FOO DEFINEQ THRU COMPILE.

ARCHIVE EventSpec [Prog. Asst. Command]

Records the events specified by EventSpecon a permanent history list. This history list can be referenced by preceding a standard event specification with @@. For example, ?? @@ prints the archived history list, REDO @@ -1 will recover the corresponding event from the archived history list and redo it, etc.

The user can also provide for automatic archiving of selected events by appropriately defining ARCHIVEFN, or by putting the property *ARCHIVE*, value T, on the event. Events that are referenced by history commands are automatically marked for archiving in this fashion (See page 8.19).

FORGET EventSpec [Prog. Asst. Command]

Permanently erases the record of the side effects for the events specified by EventSpec If EventSpecis omitted, forgets side effects for entire history list.

FORGET is provided for users with space problems. For example, if the user has just performed SETs, RPLACAs, RPLACDs, PUTD, REMPROPs, etc. to release storage, the old pointers would not be garbage collected until the corresponding events age sufficiently to drop off the end of the history list and be forgotten. FORGET can be used to force immediate forgetting (of the side-effects only). FORGET is not undoable (obviously).

REMEMBER EventSpec [Prog. Asst. Command]

Instructs the file package to ‘‘remember’’ the events specified by EventSpec These events will be marked as changed objects of file package type EXPRESSIONS, which can be written out via the le package command P. For example, after the user types:

```
_MOVD?(DELFILE /DELFILE)
DELFILE
_REMEMBER -1
(MOVD? (QUOTE DELFILE) (QUOTE /DELFILE))
_
```

If the user calls FILES?, MAKEFILES, or CLEANUP, the command (P (MOVD? (QUOTE DELFILE) (QUOTE /DELFILE))) will be constructed by the le package and added to the lecoms indicated by the user, unless the user has already explicitly added the corresponding expression to some P command himself.

Note that ‘‘remembering’’ an event like (PUTPROP 'FOO 'CLISPTYPE EXPRESSION) will *not* result in a (PROP CLISPTYPE FOO) command, because this will save the current (at the time of the MAKEFILE) value for the CLISPTYPE property, which may or may not be EXPRESSION . Thus, even if there is a PROP command which saves the CLISPTYPE property for FOO in some FILECOMS, remembering this event will still require a (P (PUTPROP 'FOO 'CLISPTYPE EXPRESSION)) command to appear.

PL LITATOM [Prog. Asst. Command]

‘‘Print Property List.’’ Prints out the property list of LITATOM in a nice format, with PRINTLEVEL reset to (2 . 3). For example,

```
_PL +
CLISPTYPE:  12
ACCESSFNS:  (PLUS IPLUS FPLUS)
```

PL is implemented via the function PRINTPROPS.

PB LITATOM [Prog. Asst. Command]

‘‘Print Bindings.’’ Prints the value of LITATOM with PRINTLEVEL reset to (2 . 3). If LITATOM is not bound, does not attempt spelling correction or generate an error. PB is implemented via the function PRINTBINDINGS.

PB is also a break command (page 9.5). As a break command, it ascends the stack and, for each frame in which LITATOM is bound, prints the frame name and value of LITATOM . If typed in to the programmer’s assistant when not at the top level, e.g. in the editor, a lower USEREXEC, etc., PB will also ascend the stack as it does with a break. However, as a programmer’s assistant command, it is primarily used to examine the top level value of a variable that may or may not be bound, or to examine a variable whose value is a large list.

; FORM [Prog. Asst. Command]

Allows the user to type a line of text without having the programmer’s assistant process it. Useful when linked to other users, or to annotate a dribble le (page 6.12).

SHH FORM [Prog. Asst. Command]

Allows the user to evaluate an expression without having the programmer’s assistant process it or record it on a history list. Useful when one wants to bypass a programmer’s assistant command or to keep the evaluation o the history list.

EXEC [Prog. Asst. Command]

(Interlisp-10) Calls SUBSYS (page 22.21) to descend to lower exec.

Rather than start up a new fork each time the user types EXEC, the EXEC command will save the old fork handle upon return from an EXEC command, and, if the fork handle is still active, reuse it for the next EXEC command, i.e. an EXEC followed by another EXEC is equivalent to an EXEC followed by a CONTIN.

CONTIN [Prog. Asst. Command]

(Interlisp-10) Performs (SUBSYS T) to continue the last call to SUBSYS (page 22.21).

TYPE-AHEAD [Prog. Asst. Command]

A command that allows the user to type-ahead an inde nite number of inputs.

The assistant responds to TYPE-AHEAD with a prompt character of >. The user can now type in an inde nite number of lines of input, under ERRORSET protection. The input lines are saved and unread when the user exits the type-ahead loop with the command $GO (<esc>GO). While in the type-ahead loop, ?? can be used to print the type-ahead, FIX to edit the type-ahead, and $Q (<esc>Q) to erase the last input (may be used repeatedly). The TYPE-AHEAD command may be aborted by $STOP (<esc>STOP); control-E simply aborts the current line of input.

For example:

```
_TYPE-AHEAD
>SYSOUT(TEM)
>MAKEFILE(EDIT)
>BRECOMPILE((EDIT WEDIT))
>F
>$Q
\\F
>$Q
\\BRECOMPILE
>LOAD(WEDIT PROP)
>BRECOMPILE((EDIT WEDIT))
>F
>MAKEFILE(BREAK)
>LISTFILES(EDIT BREAK)
>SYSOUT(CURRENT)
>LOGOUT]
>??
     >SYSOUT(TEM)
     >MAKEFILE(EDIT)
     >LOAD(WEDIT PROP)
     >BRECOMPILE((EDIT WEDIT))
     >F
     >MAKEFILE(BREAK)
     >LISTFILES(EDIT BREAK)
     >SYSOUT(CURRENT)
     >LOGOUT]
>FIX
EDIT
*(R BRECOMPILE BCOMPL)
*P
((LOGOUT) (SYSOUT &) (LISTFILES &) (MAKEFILE &) (F) (BCOMPL &)
(LOAD &) (MAKEFILE &) (SYSOUT &))
*(DELETE LOAD)
*OK
>$GO
```

Note that type-ahead can be addressed to the compiler, since it uses LISPXREAD for input. Type-ahead can also be directed to the editor, but type-ahead to the editor and to LISPX cannot be intermixed.

The following are some useful functions and variables:

(VALUEOF LINE) [NLambda NoSpread Function]

An nlambda function for obtaining the value of a particular event, e.g., (VALUEOF -1), (VALUEOF _FOO -2). The value of an event consisting of several operations is a list of the values for each of the individual operations.

Note: The value eld of a history entry is initialized to bell (control-G). Thus a value of bell indicates that the corresponding operation did not complete, i.e., was aborted or caused an error (or else it returned bell).

Note: Although the input for VALUEOF is entered on the history list before VALUEOF is called, (VALUEOF -1) still refers to the value of the expression immediately before the VALUEOF input, because VALUEOF e ectively backs the history list up one entry when it retrieves the speci ed event. Similarly, (VALUEOF FOO) will nd the rst event before this one that contains a FOO.

IT [Variable]

The value of the variable IT is always the value of the last event executed, i.e. (VALUEOF -1). For example,

```
_(SQRT 2)
1.414214
_(SQRT IT)
1.189207
```

If the last event was a multiple event, e.g. REDO -3 THRU -1, IT is set to value of the last of these events. Following a ?? command, IT is set to value of the last event printed. In other words, in all cases, IT is set to the last value printed on the terminal.

control-U When typed in at any point during an input being read by LISPXREAD, permits the user to edit the input before it is returned to the calling function.

Note: control-N for Interlisp on TOPS-20.

This feature is useful for correcting mistakes noticed in typing *before* the input is executed, instead of waiting till after execution and then performing an UNDO and a FIX. For example, if the user types

‘‘`(DEFINEQ FOO (LAMBDA (X) (FIXSPELL X`’’ and at that point notices the missing left parenthesis, instead of completing the input and allowing the error to occur, and then xing the input, he can simply type control-U, and nish typing normally. Control-U can be typed at any point, even in the middle of an atom; it simply sets a variable checked by `LISPXREAD`.

When the line is nished, the editor is called on `(DEFINEQ FOO (LAMBDA (X) (FIXSPELL X ]`, which the user can then x. If the user exits from the editor via `OK`, the (corrected) expression will be returned to whoever called `LISPXREAD` exactly as though it had been typed. If the user exits via `STOP`, the expression is returned so that it can be stored on the history list. However it will *not* be executed. In other words, the e ect is the same as though the user had typed control-E at exactly the right instant.

Control-U also works for calls to `READLINE` (page 8.30), i.e., for line commands.

8.2.3 P.A. Commands Applied to P.A. Commands

Programmer’s assistant commands that unread expressions, such as `REDO`, `USE`, etc. do not appear in the input portion of events, although they are stored elsewhere in the event. They do not interfere with or a ect the searching operations of event speci cations. As a result, p.a. commands themselves cannot be recovered for execution in the normal way. For example, if the user types `USE A B C FOR D` and follows this with `USE E FOR D`, he will not produce the e ect of `USE A B C FOR E`, but instead will simply cause `E` to be substituted for `D` in the last event containing a `D`. To produce the desired e ect, the user should type `USE D FOR E IN USE`. The appearance of the word `REDO`, `USE` or `FIX` in an event address speci es a search for the corresponding programmer’s assistant command. It also speci es that the text of the programmer’s assistant command itself be treated as though it were the input. However, the user must remember that the *context* in which a history command is reexecuted is that of the current history, not the original context. For example, if the user types `USE FOO FOR FIE IN -1`, and then later types `REDO USE`, the `-1` will refer to the event before the `REDO`, not before the `USE`.

The one exception to the statement that programmer’s assistant commands ‘‘do not interfere with or a ect the searching operations of event speci cations’’ occurs when a p.a. command fails to produce any input. For example, suppose the user types `USE LOG FOR ANTILOG AND ANTILOG FOR LOGG`, mispelling the second `LOG`. This will cause an error, `LOGG ?`. Since the `USE` command did not produce any input, the user can repair it by typing `USE LOG FOR LOGG`, without having to specify `IN USE`. This latter `USE` command will invoke a search for `LOGG`, which *will* nd the bad `USE` command. The programmer’s assistant then performs the indicated substitution, and unreads `USE LOG FOR ANTILOG AND ANTILOG FOR LOG`. In turn, this `USE` command invokes a search for `ANTILOG`, which, because it was not typed in but reread, ignores the bad `USE` command which was found by the earlier search for `LOGG`, and which is still on the history list. In other words, p.a. commands that fail to produce input are visible to searches arising from event speci cations typed in by the user, but not to secondary event speci cations.

In addition, if the most recent event is a history command which failed to produce input, a secondary event speci cation will e ectively back up the history list one event so that relative event numbers for that event speci cation will not count the bad p.a. command. For example, suppose the user types `USE LOG FOR ANTILOG AND ANTILOG FOR LOGG IN -2 AND -1`, and after the p.a. types `LOGG ?`, the user types `USE LOG FOR LOGG`. He thus causes the command `USE LOG FOR ANTILOG AND ANTILOG FOR LOG IN -2 AND -1` to be constructed and unread. In the normal case, `-1` would refer to the last event, i.e., the ‘‘bad’’ `USE` command, and `-2` to the event before it. However, in this case, `-1` refers to the event before the bad `USE` command, and the `-2` to the event before that. In short, the caveat above that ‘‘the user must remember that the context in which a history command is reexecuted is that of the current history, not the original context'' does not apply if the correction is performed immediately.

8.3 CHANGING THE PROGRAMMER'S ASSISTANT

(CHANGESLICE N HISTORY _) [Function]

Changes the time-slice of the history list HISTORY to N (see page 8.25). If HISTORY is NIL, changes both the top level history list LISPXHISTORY and the edit history list EDITHISTORY.

Note: The e ect of *increasing* the time-slice is gradual: the history list is simply allowed to grow to the corresponding length before any events are forgotten. *Decreasing* the time-slice will immediately remove a su cient number of the older events to bring the history list down to the proper size. However, CHANGESLICE is undoable, so that these events are (temporarily) recoverable. Therefore, if the user wants to recover the storage associated with these events without waiting N more events until the CHANGESLICE event drops o the history list, he must perform a FORGET command (page 8.13).

PROMPT#FLG [Variable]

When this variable is set to T, the current event number to be printed before each prompt character. See PROMPTCHAR, page 8.31. PROMPT#FLG is initially T.

PROMPTCHARFORMS [Variable]

The value of PROMPTCHARFORMS is a list of expression which are evaluated each time PROMPTCHAR (page 8.31) is called to print the prompt character. If PROMPTCHAR is going to print something, it rst maps down PROMPTCHARFORMS evaluating each expression under an ERRORSET.

These expressions can access the special variables HISTORY (the current history list), ID (the prompt character to be printed), and PROMPTSTR, which is what PROMPTCHAR will print before ID, if anything. When PROMPT#FLG is T, PROMPTSTR will be the event number. The expressions on PROMPTCHARFORMS can change the shape of a cursor, update a clock, check for mail, etc. or change what PROMPTCHAR is about to print by resetting ID and/or PROMPTSTR. After the expressions on PROMPTCHARFORMS have been evaluated, PROMPTSTR is printed if it is (still) non-NIL, and then ID is printed, if it is (still) non-NIL.

HISTORYSAVEFORMS [Variable]

The value of HISTORYSAVEFORMS is a list of expressions that are evaluated under errorset protection each time HISTORYSAVE (page 8.32) creates a new event. This happens each time there is an interaction with the user, but not when performing an operation that is being redone.

The expressions on HISTORYSAVEFORMS are presumably executed for e ect, and can access the special variables HISTORY (the current history list), ID (the current prompt character), and EVENT (the current event which HISTORYSAVE is going to return).

Note that PROMPTCHARFORMS and HISTORYSAVEFORMS together enable bracketing each interaction with the user. These can be used to measure how long the user takes to respond, to use a di erent readtable or terminal table, etc.

RESETFORMS [Variable]

The value of RESETFORMS is a list of forms that are evaluated at each RESET, i.e. when user types control-D, calls function RESET, or types control-C followed by START.

ARCHIVEFN [Variable]

If the *value* of ARCHIVEFN is T, and an event is about to drop o the end of the history list and be forgotten, ARCHIVEFN is called as a function with two arguments: the input portion of the event, and the entire event (see page 8.25 for the format of events). If ARCHIVEFN returns T, the event is archived on a permanent history list (see page 8.13). Note that ARCHIVEFN must be *both* set and de ned. ARCHIVEFN is initially NIL and unde ned.

For example, de ning ARCHIVEFN as (LAMBDA (X Y) (EQ (CAR X) 'LOAD)) will keep a record of all calls to LOAD.

ARCHIVEFLG [Variable]

If the value of ARCHIVEFLG is non-NIL, the system automatically marks all events that are referenced by history commands so that they will be archived when they drop o the history list. ARCHIVEFLG is initially T, so once an event is redone, it is guaranteed to be saved.

An event is ''marked for archiving'' by putting the property *ARCHIVE*, value T, on the event (see page 8.25). The user could do this by means of an appropriately de ned LISPXUSERFN (see below).

LISPXMACROS [Variable]

LISPXMACROS provides a macro facility that allows the user to de ne his own programmer's assistant commands. It is a list of elements of the form (COMMAND DEF). Whenever COMMAND appears as the rst expression on a line in a LISPX input, the variable LISPXLINE is bound to the rest of the line, the event is recorded on the history list, DEF is evaluated, and DEF 's value is stored as the value of the event. Similarly, whenever COMMAND appears as CAR of a form in a LISPX input, the variable LISPXLINE is bound to CDR of the form, the event is recorded, and DEF is evaluated.

An element of the form (COMMAND NIL DEF) is interpreted to mean bind LISPXLINE and evaluate DEF as described above, except do *not* save the event on the history list.

LISPXHISTORYMACROS [Variable]

LISPXHISTORYMACROS allows the user to de ne programmer's assistant commands that re-execute other events. LISPXHISTORYMACROS is interpreted the same as LISPXMACROS, except that the result of evaluating DEF is treated as a list of expressions to be *unread*, exactly as though the expressions had been retrieved by a REDO command, or computed by a USE command. Note that returning NIL means nothing else is done. This provides a mechanism for de ning LISPX commands which are executed for e ect only.

Changing The Programmer's Assistant

Many programmer's assistant commands, such as RETRIEVE, BEFORE, AFTER, etc. are implemented through LISPXMACROS or LISPXHISTORYMACROS.

Note: De nitions of commands on LISPXMACROS or LISPXHISTORYMACROS can be saved on les with the le package command LISPXMACROS (see page 11.24).

LISPXUSERFN [Variable]

When LISPXUSERFN is set to T, it is applied as a function to all inputs not recognized as a programmer's assistant command, or on LISPXMACROS or LISPXHISTORYMACROS. If LISPXUSERFN decides to handle this input, it simply processes it (the event was already stored on the history list before LISPXUSERFN was called), sets LISPXVALUE to the value for the event, and returns T. The programmer's assistant will then know not to call EVAL or APPLY, and will simply store LISPXVALUE into the value slot for the event, and print it. If LISPXUSERFN returns NIL, EVAL or APPLY is called in the usual way. Note that LISPXUSERFN must be both set and de ned.

LISPXUSERFN is given two arguments: X and LINE. X is the rst expression typed, and LINE is the rest of the line, as read by READLINE (page 8.30). For example, if the user typed FOO(A B C), X= FOO, and LINE= ((A B C)); if the user typed (FOO A B C), X= (FOO A B C), and LINE= NIL; and if the user typed FOO A B C, X= FOO and LINE= (A B C).

By appropriately de ning (and setting) LISPXUSERFN, the user can with a minimum of e ort incorporate the features of the programmer's assistant into his own executive (actually it is the other way around). For example, LISPXUSERFN could be de ned to parse all input (other than p.a. commands) in an alternative way. Note that since LISPXUSERFN is called for each input (except for p.a. commands), it can also be used to monitor some condition or gather statistics.

(LISPXPRINT X Y Z NODOFL G) [Function]
(LISPXPRIN1 X Y Z NODOFL G) [Function]
(LISPXPRIN2 X Y Z NODOFL G) [Function]
(LISPXSPACES X Y Z NODOFL G) [Function]
(LISPXTERPRI X Y Z NODOFL G) [Function]
(LISPXTAB X Y Z NODOFL G) [Function]
(LISPXPRINTDEF EXPR FILE LEFT DEF TAIL NODOFL G) [Function]

In addition to saving inputs and values, the programmer's assistant saves most system messages on the history list. For example, FILE CREATED , (FN REDEFINED), (VAR RESET), output of TIME, BREAKDOWN, STORAGE, DWIM messages, etc. When ?? prints the event, the output is also printed. This facility is implemented via these functions.

These functions print exactly the same as their non-LISPX counterparts. Then, they put the output on the history list under the property *LISPXPRINT* (see page 8.25).

If NODOFL G is non-NIL, these fuctions do not print, but only put their output on the history list.

To perform output operations from user programs so that the output will appear on the history list, the program needs simply to call the corresponding LISPX printing function.

(USERLISPXPRINT X FILE Z NODOFLG) [Function]

The function USERLISPXPRINT is available to permit the user to de ne additional LISPX printing functions. If the user has a function FN that takes three or fewer arguments, and the second argument is the le name, he can de ne a LISPX printing function by simply giving LISPXFN the de nition of USERLISPXPRINT, for example, with MOVD(USERLISPXPRINT LISPXFN). USERLISPXPRINT is de ned to look back on the stack, nd the name of the calling function, strip o the leading "LISPX", perform the appropriate saving information, and then call the function to do the actual printing.

LISPXPRINTFLG [Variable]

If LISPXPRINTFLG = NIL, the LISPX printing functions will not store their output on the history list. LISPXPRINTFLG is initially T.

8.4 STATISTICS

The programmer's assistant keeps various statistics about system usage, e.g., number of user inputs, number of undo saves, number of calls to editor, number of edit commands, number of p.a. commands, cpu time, console time, etc. These can be viewed via the function LISPXSTATS. The user can de ne add new statistics to the p.a. statistics via the function ADDSTATS, and increment them with LISPXWATCH.

Note: The collection of programmer's assistant statistics is not supported in Interlisp- D. ADDSTATS and LISPXWATCH are de ned with null de nitions, so programs can be transferred.

(LISPXSTATS RETURNVALUESFLG) [Function]

Prints programmer's assistant statistics. If RETURNVALUESFLG = T, returns the statistics as a list of elements of the form (VALUE . EXPLANATION).

(ADDSTATS $STAT_1$ … $STAT_N$) [NLambda NoSpread Function]

Each $STAT_i$ is a list of the form (STAT-NAME . MESSAGE). Each STAT-NAME is de ned as the name of a new statistic.

For example, (ADDSTATS (EDITCALLS CALLS TO EDITOR) (UNDOSTATS CHANGES UNDONE) will de ne two new statistics, named EDITCALLS and UNDOSTATS.

(LISPXWATCH STAT N) [Function]

Increments the statistic with name STAT by N (or 1 if N = NIL).

LISPXWATCH has a BLKLIBRARYDEF (see page 12.14).

The user can save his statistics for loading into a new system by performing MAKEFILE(DUMPSTATS). After the le DUMPSTATS is loaded, the statistics printed by LISPXSTATS will be the same as those that would be printed following the MAKEFILE.

8.5 UNDOING

Note: This discussion only applies to undoing under the executive and break; the editors handles undoing itself in a slightly di erent fashion.

The UNDO capability of the programmer's assistant is implemented by requiring that each operation that is to be undoable be responsible itself for saving on the history list enough information to enable reversal of its side e ects. In other words, the assistant does not ‘‘know’’ when it is about to perform a destructive operation, i.e., it is not constantly checking or anticipating. Instead, it simply executes operations, and any undoable changes that occur are automatically saved on the history list by the responsible functions. The UNDO command, which involves recovering the saved information and performing the corresponding inverses, works the same way, so that the user can UNDO an UNDO, and UNDO that etc.

At each point, until the user speci cally requests an operation to be undone, the assistant does not know, or care, whether information has been saved to enable the undoing. Only when the user attempts to undo an operation does the assistant check to see whether any information has been saved. If none has been saved, and the user has speci cally named the event he wants undone, the assistant types NOTHING SAVED. (When the user simply types UNDO, the assistant searches for the last undoable event, ignoring events already undone as well as UNDO operations themselves.)

This implementation minimizes the overhead for undoing. Only those operations which actually make changes are a ected, and the overhead is small: two or three cells of storage for saving the information, and an extra function call. However, even this small price may be too expensive if the operation is su ciently primitive and repetitive, i.e., if the extra overhead may seriously degrade the overall performance of the program. Hence not every destructive operation in a program should necessarily be undoable; the programmer must be allowed to decide each case individually.

Therefore for each primitive destructive function, Interlisp has de ned an undoable version which always saves information. By convention, the name of the undoable version of a function is the function name, preceeded by ‘‘/.’’ For example, there is RPLACA and /RPLACA, REMPROP and /REMPROP, etc. The ‘‘slash’’ functions that are currently implemented can be found as the value of /FNS.

The various system packages use the appropriate undoable functions. For example, BREAK uses /PUTD and /REMPROP so as to be undoable, and DWIM uses /RPLACA and /RPLACD, when it makes a correction.[2] Similarly, the user can simply use the corresponding / function if he wants to make a destructive operation in his own program undoable. When the / function is called, it will save the UNDO information in the current event on the history list.

The programmer's assistant cannot know whether e ciency and overhead are serious considerations for the execution of an expression in a user *program*, so the user must decide if he wants these operations undoable by explicitly calling /MAPCONC, etc. However, *typed-in* expressions rarely involve iterations or lengthy computations *directly*. Therefore, before evaluating the user input, the programmer's assistant substitutes the corresponding undoable function for any destructive function (see LISPX/, page 8.34). For example, if the user types (MAPCONC NASDIC), it is actually (/MAPCONC NASDIC) that is evaluated. Obviously, with a more sophisticated analysis of both user input and user programs, the

[2]The e ects of the following functions are always undoable: DEFINE, DEFINEQ, DEFC (used to give a function a compiled code de nition), DEFLIST, LOAD, SAVEDEF, UNSAVEDEF, BREAK, UNBREAK, REBREAK, TRACE, BREAKIN, UNBREAKIN, CHANGENAME, EDITFNS, EDITF, EDITV, EDITP, EDITE, EDITL, ESUBST, ADVISE, UNADVISE, READVISE, plus any changes caused by DWIM.

decision concerning which operations to make undoable could be better advised. However, we have found the con guration described here to be a very satisfactory one. The user pays a very small price for being able to undo what he types in, and if he wishes to protect himself from malfunctioning in his own programs, he can have his program explicitly call undoable functions.

8.5.1 Undoing Out of Order

/RPLACA operates undoably by saving (on the history list) the list cell that is to be changed and its original CAR. Undoing a /RPLACA simply restores the saved CAR. This implementation can produce unexpected results when multiple /RPLACAs are done on the same list cell, and then undone out of order. For example, if the user types (RPLACA FOO 1), followed by (RPLACA FOO 2), then undoes both events by undoing the most recent event rst, then undoing the older event, FOO will be restored to its state before either RPLACA operated. However if the user undoes the rst event, *then* the second event, (CAR FOO) will be 1, since this is what was in CAR of FOO before (RPLACA FOO 2) was executed. Similarly, if the user types (NCONC1 FOO 1), followed by (NCONC1 FOO 2), undoing just (NCONC1 FOO 1) will remove both 1 and 2 from FOO. The problem in both cases is that the two operations are not ‘‘independent.’’ In general, operations are always independent if they a ect di erent lists or di erent sublists of the same list. Undoing in reverse order of execution, or undoing independent operations, is always guaranteed to do the ‘‘right’’ thing. However, undoing dependent operations out of order may not always have the predicted e ect.

Property list operations, (i.e., PUTPROP, ADDPROP and REMPROP) are handled specially, so that operations that a ect di erent properties on the same property list are always independent. For example, if the user types (PUTPROP 'FOO 'BAR 1) then (PUTPROP 'FOO 'BAZ 2), then undoes the rst event, the BAZ property will remain, even though it may not have been on the property list of FOO at the time the rst event was executed.

8.5.2 SAVESET

Typed-in SETs are made undoable by substituting a call to SAVESET. SETQ is made undoable by substituting SAVESETQ, and SETQQ by SAVESETQQ, both of which are implemented in terms of SAVESET.

In addition to saving enough information on the history list to enable undoing, SAVESET operates in a manner analogous to SAVEDEF (page 11.18) when it resets a top level value: when it changes a top level binding from a value other than NOBIND to a new value that is not EQUAL to the old one, SAVESET saves the old value of the variable being set on the variable's property list under the property VALUE, and prints the message (VARIABLE RESET). The old value can be restored via the function UNSET, which also saves the current value (but does not print a message). Thus UNSET can be used to ip back and forth between two values.

Of course, UNDO can be used as long as the event containing this call to SAVESET is still active. Note however that the old value will remain on the property list, and therefore be recoverable via UNSET, even after the original event has been forgotten.

RPAQ and RPAQQ are implemented via calls to SAVESET. Thus old values will be saved and messages printed for any variables that are reset as the result of loading a le.

For top level variables, SAVESET also adds the variable to the appropriate spelling list, thereby noticing variables set in les via `RPAQ` or `RPAQQ`, as well as those set via type-in.

`(SAVESET NAME VALUE TOPFLG FLG)` [Function]

An undoable `SET`. `SAVESET` scans the stack looking for the last binding of `NAME`, sets `NAME` to `VALUE`, and returns `VALUE`.

If the binding changed was a top level binding, `NAME` is added to the spelling list `SPELLINGS3` (see page 15.14). Furthermore, if the old value was not `NOBIND`, and was also not `EQUAL` to the new value, `SAVESET` calls the le package to update the necessary le records. Then, if `DFNFLG` is not equal to `T`, `SAVESET` prints `(NAME RESET)`, and saves the old value on the property list of `NAME`, under the property `VALUE`.

If `TOPFLG` = `T`, `SAVESET` operates as above except that it always uses `NAME`'s top-level value cell. When `TOPFLG` is `T`, and `DFNFLG` is `ALLPROP` and the old value was not `NOBIND`, `SAVESET` simply stores `VALUE` on the property list of `NAME` under the property `VALUE`, and returns `VALUE`. This option is used for loading les without disturbing the current value of variables (see page 5.9).

If `FLG` = `NOPRINT`, `SAVESET` saves the old value, but does not print the message. This option is used by `UNSET`.

If `FLG` = `NOSAVE`, `SAVESET` does *not* save the old value on the property list, nor does it add `NAME` to `SPELLINGS3`. However, the call to `SAVESET` is still undoable. This option is used by `/SET`.

If `FLG` = `NOSTACKUNDO`, `SAVESET` is undoable only if the binding being changed is a top-level binding, i.e. this says when resetting a variable that has been rebound, don't bother to make it undoable. This option is used by `RPAQ`, `RPAQQ`, and `ADDTOVAR`.

`(UNSET NAME)` [Function]

If `NAME` does not contain a property `VALUE`, `UNSET` generates an error. Otherwise `UNSET` calls `SAVESET` with `NAME`, the property value, `TOPFLG` = `T`, and `FLG` = `NOPRINT`.

8.5.3 UNDONLSETQ and RESETUNDO

The function `UNDONLSETQ` provides a limited form of backtracking: if an error occurs under the `UNDONLSETQ`, all undoable side e ects executed under the `UNDONLSETQ` are undone. `RESETUNDO`, used in conjunction with `RESETLST` and `RESETSAVE` (page 9.19), provides a more general undo capability where the user can specify that the side e ects be undone after the speci ed computation nishes, is aborted by an error, or by a control-D.

`(UNDONLSETQ UNDOFORM _)` [NLambda Function]

An nlambda function similar to `NLSETQ` (page 9.15). `UNDONLSETQ` evaluates `UNDOFORM`, and if no error occurs during the evaluation, returns `(LIST (EVAL UNDOFORM))` and passes the undo information from `UNDOFORM` (if any) upwards. If an error does occur, the `UNDONLSETQ` returns `NIL`, and any undoable changes made during the evaluation of `UNDOFORM` are undone.

Any undo information is stored directly on the history event (if `LISPXHIST` is not NIL), so that if the user control-D's out of the UNDONLSETQ, the event is still undoable.

UNDONLSETQ will operate correctly if #UNDOSAVES is or has been exceeded for this event, or is exceeded while under the scope of the UNDONLSETQ.

Note: Caution must be exercised in using coroutines or other non-standard means of exiting while under an UNDONLSETQ. See discussion in page 9.19.

(RESETUNDO X STOPFL G) [Function]

For use in conjunction with RESETLST (page 9.19). (RESETUNDO) initializes the saving of undo information and returns a value which when given back to RESETUNDO undoes the intervening side e ects. For example, (RESETLST (RESETSAVE (RESETUNDO)) . FORMS) will undo the side e ects of FORMS on normal exit, or if an error occurs or a control-D is typed.

If STOPFL G = T, RESETUNDO stops accumulating undo information it is saving on X. Note that this has no bearing on the saving of undo information on higher RESETUNDO's, or on being able to undo the entire event.

For example,

```
(RESETLST
   (SETQ FOO (RESETUNDO))
   (RESETSAVE NIL (LIST 'RESETUNDO FOO))
   (ADVISE    )
   (RESETUNDO FOO T)
   . FORMS )
```

would cause the advice to be undone, but *not* any of the side e ects in FORMS .

8.6 FORMAT AND USE OF THE HISTORY LIST

The system currently uses three history lists, LISPXHISTORY for the top-level Interlisp executive, EDITHISTORY for the editors, and ARCHIVELST for archiving events (see page 8.13). All history lists have the same format, use the same functions, HISTORYSAVE, for recording events, and use the same set of functions for implementing commands that refer to the history list, e.g., HISTORYFIND, PRINTHISTORY, UNDOSAVE, etc.

Each history list is a list of the form (L EVENT q SIZE MOD), where L is the list of events with the most recent event rst, EVENT q is the event number for the most recent event on L, SIZE is the size of the time-slice (below), i.e., the maximum length of L, and MOD is the highest possible event number. LISPXHISTORY and EDITHISTORY are both initialized to (NIL 0 100 100). Setting LISPXHISTORY or EDITHISTORY to NIL disables all history features, so LISPXHISTORY and EDITHISTORY act like ags as well as repositories of events.

Each history list has a maximum length, called its ''time-slice.'' As new events occur, existing events are aged, and the oldest events are ''forgotten.'' For e ciency, the storage used to represent the forgotten event is reused in the representation of the new event, so the history list is actually a ring bu er. The time-slice of a history list can be changed with the function CHANGESLICE, page 8.18. Larger time-slices enable longer ‘‘memory spans,’’ but tie up correspondingly greater amounts of storage. Since the user seldom needs really ‘‘ancient history,’’ and a facility is provided for saving and remembering selected events (see NAME and RETRIEVE, page 8.12), a relatively small time-slice such as 30 events is more than adequate, although some users prefer to set the time-slice as large as 100 events.

If PROMPT#FLG (page 8.18) is set to T, an ‘‘event number’’ will be printed before each prompt. More recent events have higher numbers. When the event number of the current event is 100, the next event will be given number 1. If the time-slice is greater than 100, the ‘‘roll-over’’ occurs at the next highest hundred, so that at no time will two events ever have the same event number. For example, if the time-slice is 150, event number 1 will follow event number 200.

Each individual event on L is a list of the form (INPUT ID VALUE . PROPS). ID is the prompt character for this event, e.g., _, :, *, etc. VALUE is the value of the event, and is initialized to bell.[3] PROPS is a property list used to associate other information with the event (described below).

INPUT is the input sequence for the event. Normally, this is just the input that the user typed-in. For an APPLY format input, this is a list consisting of two expressions; for an EVAL format input, this is a list of just one expression; for an input entered as list of atoms, INPUT is simply that list. For example,

User Input	INPUT is:
PLUS[1 1]	(PLUS (1 1))
(PLUS 1 1)	((PLUS 1 1))
PLUS 1 1^{cr}	(PLUS 1 1)

If the user types in a programmer’s assistant command that ‘‘unreads’’ and reexecutes other events (REDO, USE,, etc.), INPUT contains a ‘‘sequence’’ of the inputs from the redone events. Speci cally, the INPUT elds from the speci ed events are concatenated into a single list, seperated by special markers called ‘‘pseudo-carriage returns,’’ which print out as the string "<c.r.>".[4] When the result of this concatenation is ‘‘reread,’’ the pseudo-carriage-returns are treated by LISPXREAD and READLINE exactly as real carriage returns, i.e., they serve to distinguish between APPLY and EVAL formats on inputs to LISPX, and to delimit line commands to the editor.

The same convention is used for representing multiple inputs when a USE command involves sequential substitutions. For example, if the user types GETD(FOO) and then USE FIE FUM FOR FOO, the input sequence that will be constructed is (GETD (FIE) "<c.r.>" GETD (FUM)), which is the result of substituting FIE for FOO in (GETD (FOO)) concatenated with the result of substituting FUM for FOO in (GETD (FOO)).

Note that once a multiple input has been entered as the input portion of a new event, that event can be treated exactly the same as one resulting from type-in. In other words, no special checks have to be made when *referencing* an event, to see if it is simple or multiple. This implementation permits an

[3]On EDITHISTORY, this eld is used to save the side e ects of each command. See page 8.35.

[4]The value of the variable HISTSTR0 is used to represent a pseudo-carriage return. This is initially the string "<c.r.>". Note that the functions that recognize pseudo-carriage returns compare them to HISTSTR0 using EQ, so this marker will never be confused with a string that was typed in by the user.

event speci cation to refer to a single simple event, or to several events, or to a single event originally constructed from several events (which may themselves have been multiple input events, etc.) without having to treat each case separately.

REDO, RETRY, USE, ..., and FIX commands, i.e., those commands that reexecute previous events, are not stored as inputs, because the input portion for these events are the expressions to be ''reread''. The history commands UNDO, NAME, RETRIEVE, BEFORE, and AFTER *are* recorded as inputs, and ?? prints them exactly as they were typed.

PROPS is a property list of the form (PROPERTY$_1$ VALUE$_1$ PROPERTY$_2$ VALUE$_2$), that can be used to associate arbitrary information with a particular event. Currently, the following properties are used by the programmer's assistant:

SIDE	A list of the side e ects of the event. See UNDOSAVE, page 8.33.
PRINT	Used by the ?? command when special formatting is required, for example, when printing events corresponding to the break commands OK, GO, EVAL, and ?=.
USE-ARGS ...ARGS	The USE-ARGS and ...ARGS properties are used to save the arguments and expression for the corresponding history command.
ERROR *CONTEXT*	*ERROR* and *CONTEXT* are used to save information when errors occur for subsequent use by the $ command. Whenever an error occurs, the o ender is automatically saved on that event's entry in the history list, under the *ERROR* property.
LISPXPRINT	Used to record calls to LISPXPRINT, LISPXPRIN1, etc. (see page 8.20).
ARCHIVE	The property *ARCHIVE* on an event causes the event to be automatically archived when it ''falls o the end'' of the history list (see page 8.13).
GROUP *HISTORY*	The *HISTORY* and *GROUP* properties are used for commands that reexecute previous events, i.e., REDO, RETRY, USE, ..., and FIX. The value of the *HISTORY* property is the history command that the user actually typed, e.g., REDO FROM F. This is used by the ?? command when printing the event. The value of the *GROUP* property is a structure containing the side e ects, etc. for the individual inputs being reexecuted. This structure is described below.

When LISPX is given an input, it calls HISTORYSAVE (page 8.32) to record the input in a new event.[5] Normally, HISTORYSAVE creates and returns a new event. LISPX binds the variable LISPXHIST to the value of HISTORYSAVE, so that when the operation has completed, LISPX knows where to store the value. Note that by the time it completes, the operation may no longer correspond to the most recent event on the history list. For example, all inputs typed to a lower break will appear later on the

[5]The commands ??, FORGET, TYPE-AHEAD, $BUFS, and ARCHIVE are executed immediately, and are not recorded on the history list.

history list. After binding LISPXHIST, LISPX executes the input, stores its value in the value eld of the LISPXHIST event, prints the value, and returns.

When the input is a REDO, RETRY, USE, ..., or FIX command, the procedure is similar, except that the event is also given a *GROUP* property, initially NIL, and a *HISTORY* property, and LISPX simply unreads the input and returns. When the input is ‘‘reread’’, it is HISTORYSAVE, not LISPX, that notices this fact, and nds the event from which the input originally came.[6] HISTORYSAVE then adds a new (INPUT ID VALUE . PROPS) entry to the *GROUP* property for this event, and returns this entry as the ‘‘new event.’’ LISPX then proceeds exactly as when its input was typed directly, i.e., it binds LISPXHIST to the value of HISTORYSAVE, executes the input, stores the value in CADDR of LISPXHIST, prints the value, and returns. In fact, LISPX never notices whether it is working on freshly typed input, or input that was reread. Similarly, UNDOSAVE will store undo information on LISPXHIST the same as always, and does not know or care that LISPXHIST is not the entire event, but one of the elements of the *GROUP* property. Thus when the event is nished, its entry will look like:

```
(INPUT ID VALUE
       *HISTORY*
            COMMAND
       *GROUP*
            ((INPUT1 ID1 VALUE1 SIDE SIDE1)
             (INPUT2 ID2 VALUE2 SIDE SIDE2)
                ))
```

In this case, the value eld of the event with the *GROUP* property is not being used; VALUEOF instead returns a list of the values from the *GROUP* property. Similarly, UNDO operates by collecting the SIDE properties from each of the elements of the *GROUP* property, and then undoing them in reverse order.

This implementation removes the burden from the function calling HISTORYSAVE of distinguishing between new input and reexecution of input whose history entry has already been set up.

8.7 PROGRAMMER'S ASSISTANT FUNCTIONS

(LISPX LISPXX LISPXID LISPXXMACROS LISPXXUSERFN LISPXFLG) [Function]

LISPX is the primary function of the programmer's assistant. LISPX takes one user input, saves it on the history list, evaluates it, saves its value, and prints and returns it. LISPX also interpretes p.a. commands, LISPXMACROS, LISPXHISTORYMACROS, and LISPXUSERFN.

If LISPXX is a list, it is interpreted as the input expression. Otherwise, LISPX calls READLINE, and uses LISPXX plus the value of READLINE as the input for the event. If LISPXX is a list CAR of which is LAMBDA or NLAMBDA, LISPX calls LISPXREAD to obtain the arguments.

LISPXID is the prompt character to print before accepting user input. A user can call LISPX specifying any prompt character as LISPXID except for *, since in

[6]If HISTORYSAVE cannot nd the event, for example if a user program unreads the input directly, and not via a history command, HISTORYSAVE proceeds as though the input were typed.

certain cases LISPX must use the value of LISPXID to tell whether or not it was called from the editor.

If LISPXXMA CR OS is not NIL, it is used as the list of LISPX macros, otherwise the top level value of the variable LISPXMACROS is used.

If LISPXXUSERFN is not NIL, it is used as the LISPXUSERFN. In this case, it is not necessary to both set and de ne LISPXUSERFN as described on page 8.20.

LISPXFL G is used by the E command in the editor (see page 8.35).

Note that the history is *not* one of the arguments to LISPX, i.e., the editor must bind (reset) LISPXHISTORY to EDITHISTORY before calling LISPX to carry out a history command. LISPX will continue to operate as an EVAL/ APPLY function if LISPXHISTORY is NIL. Only those functions and commands that involve the history list will be a ected.

LISPX performs spelling corrections using LISPXCOMS, a list of its commands, as a spelling list whenever it is given an unbound atom or unde ned function, before attempting to evaluate the input.

LISPX is responsible for rebinding HELPCLOCK, used by BREAKCHECK (page 9.10) for computing the amount of time spent in a computation, in order to determine whether to go into a break if and when an error occurs.

(USEREXEC LISPXID LISPXXMA CR OS LISPXXUSERFN) [Function]

Repeatedly calls LISPX under errorset protection specifying LISPXXMA CR OS and LISPXXUSERFN , and using LISPXID (or _ if LISPXID= NIL) as a prompt character. USEREXEC is exited via the command OK, or else with a RETFROM.

(LISPXEVAL LISPXF ORM LISPXID) [Function]

Evaluates LISPXF ORM (using EVAL) the same as though it were typed in to LISPX, i.e., the event is recorded, and the evaluation is made undoable by substituting the slash functions for the corresponding destructive functions (see page 8.22). LISPXEVAL returns the value of the form, but does not print it.

When LISPX recieves an ‘‘input,’’ it may come from the user typing it in, or it may be an input that has been ‘‘unread.’’ LISPX handles these two cases by getting inputs with LISPXREAD and READLINE, described below. These functions use the variable READBUF to store the expressions that have been unread. When READBUF is not NIL, READLINE and LISPXREAD ‘‘read’’ expressions from READBUF until READBUF is NIL, or until they read a pseudo- carriage return (see page 8.26). Both functions return a list of the expressions that have been ‘‘read.’’ (The pseudo- carriage return is not included in the list.)

When READBUF is NIL, both LISPXREAD and READLINE actually obtain their input by performing (APPLY* LISPXREADFN FILE), where LISPXREADFN is initially set to READ. The user can make LISPX, the editor, break, etc. do their reading via a di erent input function by simply setting LISPXREADFN to the name of that function (or an appropriate LAMBDA expression).

Note: The user should only add expressions to READBUF using the function LISPXUNREAD (page 8.31), which knows about the format of READBUF.

(READLINE RDTBL _ _) [Function]

Reads a line from the terminal, returning it as a list. If (READP T) is NIL, READLINE returns NIL. Otherwise it reads expressions by performing (APPLY* LISPXREADFN T) (LISPXREADFN is initially set to READ) until it encounters either:

a carriage-return (typed by the user) that is not preceded by any spaces, e.g.,

```
A B C^cr
```

and READLINE returns (A B C)

a list terminating in a "]", in which case the list is included in the value of READLINE, e.g.,

```
A B (C D]
```

and READLINE returns (A B (C D)).

an unmatched right parentheses or right square bracket, which is not included in the value of READLINE, e.g.,

```
A B C]
```

and READLINE returns (A B C).

In the case that one or more spaces precede a carriage-return, or a list is terminated with a ")", READLINE will type "..." and continue reading on the next line, e.g.,

```
A B C^cr
...(D E F)
...(X Y Z]
```

and READLINE returns (A B C (D E F) (X Y Z)).

If the user types another carriage-return after the "...", the line will terminate, e.g.,

```
A B C^cr
...^cr
```

and READLINE returns (A B C).

Note that carriage-return, i.e., the EOL character, can be rede ned with SETSYNTAX (page 6.34). READLINE actually checks for the EOL character, whatever that may be. The same is true for right parenthesis and right bracket.

When READLINE is called from LISPX, it operates di erently in two respects:

(1) If the line consists of a single) or], READLINE returns (NIL) instead of NIL, i.e., the) or] *is* included in the line. This permits the user to type FOO) or FOO], meaning call the function FOO with no arguments, as opposed to FOO^{cr}

(FOO<carriage-return>), meaning evaluate the variable FOO.

(2) If the rst expression on the line is a list that is not preceded by any spaces, the list terminates the line regardless of whether or not it is terminated by]. This permits the user to type EDITF(FOO) as a single input.

Note that if any spaces are inserted between the atom and the left parentheses or bracket, READLINE will assume that the list does not terminate the line. This is to enable the user to type a line command such as USE (FOO) FOR FOO. Therefore, if the user accidentially puts an extra space between a function and its arguments, he will have to complete the input with another carriage return, e.g.,

```
_EDITF (FOO)
...cr
EDIT
*
```

(LISPXREAD FILE RDTBL) [Function]

A generalized READ. If READBUF= NIL, LISPXREAD performs (APPLY* LISPXREADFN FILE), which it returns as its value. If READBUF is not NIL, LISPXREAD ''reads'' and returns the next expression on READBUF.

Note: If the user types control-U during the call to READ, LISPXREAD calls the editor and returns the edited value.

LISPXREAD also sets REREADFLG to NIL when it reads via READ, and sets REREADFLG to the value of READBUF when rereading.

(LISPXREADP FLG) [Function]

A generalized READP. If FLG = T, LISPXREADP returns T if there is any input waiting to be ''read'', in the manner of LISPXREAD. If FLG = NIL, LISPXREADP returns T only if there is any input waiting to be ''read'' *on this line.* In both cases, leading spaces are ignored, i.e., skipped over with READC, so that if only spaces have been typed, LISPXREADP will return NIL.

(LISPXUNREAD LST _) [Function]

Unreads LST, a list of expressions.

(PROMPTCHAR ID FLG HISTORY) [Function]

Called by LISPX to print the prompt character ID before each input. PROMPTCHAR will not print anything when the next input will be ''reread'', i.e., when READBUF is not NIL.

PROMPTCHAR will not print when (READP)= T, unless FLG is T. The editor calls PROMPTCHAR with FLG= NIL so that extra *'s are not printed when the user types several commands on one line. However, EVALQT calls PROMPTCHAR with FLG= T, since it always wants the _ printed (except when ''rereading'').

If PROMPT#FLG (page 8.18) is T and HISTORY is not NIL, PROMPTCHAR prints the current event number (of HISTORY) before printing ID.

The value of PROMPTCHARFORMS (page 8.18) is a list of expressions that are evaluated by PROMPTCHAR before, and if, it does any printing.

(HISTORYSAVE HISTORY ID INPUT1 INPUT2 INPUT3 PROPS) [Function]

Records one event on HISTORY.

If INPUT1 is not NIL, the input is of the form (INPUT $_1$ INPUT $_2$. INPUT $_3$). If INPUT $_1$ is NIL, and INPUT $_2$ is not NIL, the input is of the form (INPUT $_2$. INPUT $_3$). Otherwise, the input is just INPUT $_3$.

HISTORYSAVE creates a new event with the corresponding input, ID, value eld initialized to bell, and PROPS. If the HISTORY has reached its full size, the last event is removed and cannibalized.

The value of HISTORYSAVE is the new event. However, if REREADFLG is not NIL, and the most recent event on the history list contains the history command that produced this input, HISTORYSAVE does not create a new event, but simply adds an (INPUT ID bell . PROPS) entry to the *GROUP* property for that event and returns that entry. See discussion on page 8.28.

HISTORYSAVEFORMS (page 8.18) is a list of expressions that are evaluated under errorset protection each time HISTORYSAVE creates a new event.

(LISPXSTOREVALUE EVENT VALUE) [Function]

Used by LISPX for storing the value of an event. Can be advised by user to watch for particular values or perform other monitoring functions.

(LISPXFIND HISTORY LINE TYPE BACKUP _) [Function]

LINE is an event speci cation, TYPE speci es the format of the value to be returned by LISPXFIND, and can be either ENTRY, ENTRIES, COPY, COPIES, INPUT, or REDO. LISPXFIND parses LINE, and uses HISTORYFIND to nd the corresponding events. LISPXFIND then assembles and returns the appropriate structure.

LISPXFIND incorporates the following special features:

(1) if BACKUP = T, LISPXFIND interprets LINE in the context of the history list *before* the current event was added. This feature is used, for example, by VALUEOF, so that (VALUEOF -1) will not refer to the VALUEOF event itself.

(2) if LINE = NIL and the last event is an UNDO, the next to the last event is taken. This permits the user to type UNDO followed by REDO or USE.

(3) LISPXFIND recognizes @@, and substitutes ARCHIVELST for HISTORY (see page 8.13).

(4) LISPXFIND recognizes @, and retrieves the corresponding event(s) from the property list of the atom following @ (see page 8.12).

(HISTORYFIND LST INDEX MOD EVENTADDRESS _) [Function]

Searches LST and returns the tails of LST beginning with the event corresponding to EVENTADDRESS. LST, INDEX, and MOD are the rst three elements of a ‘‘history list’’ structure (see page 8.25). EVENTADDRESS is an event address (see page 8.5) e.g., (43), (-1), (FOO FIE), (LOAD _ FOO), etc. If HISTORYFIND cannot nd EVENTADDRESS, it generates an error.

(HISTORYMATCH INPUT PAT EVENT) [Function]

Used by HISTORYFIND for ‘‘matching’’ when EVENT ADDRESS speci es a pattern. Matches PAT against INPUT , the input portion of the history event EVENT , as matching is de ned on page 17.13. Initially de ned as (EDITFINDP INPUT PAT T), but can be advised or rede ned by the user.

(ENTRY# HIST X) [Function]

HIST is a history list (see page 8.25). X is EQ to one of the events on HIST. ENTRY# returns the event number for X.

(UNDOSAVE UNDOF ORM HISTENTR Y) [Function]

UNDOSAVE adds the ‘‘undo information’’ UNDOF ORM to the SIDE property of the history event HISTENTR Y. If there is no SIDE property, one is created. If the value of the SIDE property is NOSAVE, the information is not saved.

HISTENTR Y speci es an event. If HISTENTR Y = NIL, the value of LISPXHIST is used. If both HISTENTR Y and LISPXHIST are NIL, UNDOSAVE is a no-op. Note that HISTENTR Y (or LISPXHIST) can either be a ‘‘real’’ event, or an event within the *GROUP* property of another event (see page 8.28).

The form of UNDOF ORM is (FN . AR GS).[7] Undoing is done by performing (APPLY (CAR UNDOF ORM) (CDR UNDOF ORM)). For example, if the de nition of FOO is DEF , (/PUTD FOO NEWDEF) will cause a call to UNDOSAVE with UNDOF ORM = (/PUTD FOO DEF).

CAR of the SIDE property of an event is a count of the number of UNDOF ORM s saved for this event. Each call to UNDOSAVE increments this count. If this count is set to -1, then it is never incremented, and any number of UNDOF ORM s can be saved. If this count is a positive number, UNDOSAVE restricts the number of UNDOF ORM s saved to the value of #UNDOSAVES, described below. LOAD initializes the count to -1, so that regardless of the value of #UNDOSAVES, no message will be printed, and the LOAD will be undoable.

#UNDOSAVES [Variable]

The value of #UNDOSAVES is the maximum number of UNDOF ORM s to be saved for a single event. When the count of UNDOF ORM s reaches this number, UNDOSAVE prints the message CONTINUE SAVING?, asking the user if he wants to continue saving. If the user answers NO or defaults, UNDOSAVE discards the previously saved information for this event, and makes NOSAVE be the value of the property SIDE, which disables any further saving for this event. If the user answers YES, UNDOSAVE changes the count to -1, which is then never incremented, and continues saving. The purpose of this feature is to avoid tying up large quantities of storage for operations that will never need to be undone.

If #UNDOSAVES is negative, then when the count reaches -#UNDOSAVES, UNDOSAVE simply stops saving without printing any messages or interacting with the

[7] In the special case of /RPLNODE and /RPLNODE2, the format of UNDOF ORM is (X OLDCAR . OLDCDR). When UNDOF ORM is undone, this form is recognized and handled specially. This implementation saves space.

user. #UNDOSAVES = NIL is equivalent to #UNDOSAVES =in nity. #UNDOSAVES is initially NIL.

(NEW/FN FN) [Function]

NEW/FN performs the necessary housekeeping operations to make FN be translated to the undoable version /FN when typed- in. For example, RADIX can be made undoable when typed- in by performing:

```
_ (DEFINEQ (/RADIX (X)
          (UNDOSAVE (LIST '/RADIX (RADIX X))
(/RADIX)
_ (NEW/FN 'RADIX)
```

(LISPX/ X FN VARS) [Function]

LISPX/ performs the substitution of / functions for destructive functions that are typed- in. If FN is not NIL, it is the name of a function, and X is its argument list. If FN is NIL, X is a form. In both cases, LISPX/ returns X with the appropriate substitutions. VARS is a list of bound variables (optional).

LISPX/ incorporates information about the syntax and semantics of Interlisp expressions. For example, it does not bother to make undoable operations involving variables bound in X. It does not perform substitution inside of expressions CAR of which is an nlambda function (unless CAR of the form has the property INFO value EVAL, see page 5.4). For example, (BREAK PUTD) typed to LISPX, will break on PUTD, not /PUTD. Similarly, substitution *should* be performed in the arguments for functions like MAPC, RPTQ, etc., since these contain expressions that will be evaluated or applied. For example, if the user types (MAPC '(FOO1 FOO2 FOO3) 'PUTD) the PUTD must be replaced by /PUTD.

(UNDOLISPX LINE) [Function]

LINE is an event speci cation. UNDOLISPX is the function that executes UNDO commands by calling UNDOLISPX1 on the appropriate entry(s).

(UNDOLISPX1 EVENT FLG _) [Function]

Undoes one event. UNDOLISPX1 returns NIL if there is nothing to be undone. If the event is already undone, UNDOLISPX1 prints ALREADY UNDONE and returns T. Otherwise, UNDOLISPX1 undoes the event, prints a message, e.g., SETQ UNDONE, and returns T.

If FLG = T and the event is already undone, or is an undo command, UNDOLISPX1 takes no action and returns NIL. UNDOLISPX uses this option to search for the last event to undo. Thus when LINE = NIL, UNDOLISPX simply searches history until it nds an event for which UNDOLISPX1 returns T.

Undoing an event consists of mapping down (CDR of) the property value for SIDE, and for each element, applying CAR to CDR, and then marking the event undone by attaching (with /ATTACH) a NIL to the front of its SIDE property. Note that the undoing of each element on the SIDE property will usually cause undosaves to be added to the *current* LISPXHIST, thereby enabling the e ects of UNDOLISPX1 to be undone.

(PRINTHISTORY HISTOR Y LINE SKIPFN NO VAL UES FILE) [Function]

LINE is an event speci cation. PRINTHISTORY prints the events on HISTOR Y speci ed by LINE, e.g., (-1 THRU -10). Printing is performed via the function SHOWPRIN2, so that if the value of SYSPRETTYFLG = T, events will be prettyprinted.

SKIPFN is an (optional) functional argument that is applied to each event before printing. If it returns non-NIL, the event is skipped, i.e., not printed.

If NO VAL UES = T, or NO VAL UES applied to the corresponding event is true, the value is not printed. For example, NO VAL UES is T when printing events on EDITHISTORY.

For example, the following LISPXMACRO will de ne ??′ as a command for printing the history list while skipping all ''large events'' and not printing any values.

```
(??′ (PRINTHISTORY
        LISPXHISTORY
        LISPXLINE
        (FUNCTION (LAMBDA (X)
           (IGREATERP (COUNT (CAR X)) 5)))
        T
        T))
```

8.8 THE EDITOR AND THE PROGRAMMER'S ASSISTANT

As mentioned earlier, all of the remarks concerning ''the programmer's assistant'' apply equally well to user interactions with EVALQT, BREAK or the editor. The di erences between the editor's implementation of these features and that of LISPX are mostly obvious or inconsequential. However, for completeness, this section discusses the editor's implementation of the programmer's assistant.

The editor uses PROMPTCHAR to print its prompt character, and LISPXREAD, LISPXREADP, and READLINE for obtaining inputs. When the editor is given an input, it calls HISTORYSAVE to record the input in a new event on its history list, EDITHISTORY.[8] EDITHISTORY follows the same conventions and format as LISPXHISTORY. However, since edit commands have no value, the editor uses the value eld for saving side e ects, rather than storing them under the property SIDE.

The editor recognizes and processes the four commands DO, !E, !F, and !N which refer to previous events on EDITHISTORY. The editor also processes UNDO itself, as described below. All other history

[8]Except that the atomic commands OK, STOP, SAVE, P, ?, PP and E are not recorded. In addition, number commands are grouped together in a single event. For example, 3 3 -1 is considered as one command for changing position.

commands[9] are simply given to LISPX for execution, after first binding (resetting) LISPXHISTORY to EDITHISTORY. The editor also calls LISPX when given an E command (page 17.45). In this case, the editor uses the fifth argument to LISPX, LISPXFLG, to specify that any history commands are to be executed by a recursive call to LISPX, rather than by unreading. For example, if the user types E REDO in the editor, he wants the last event on LISPXHISTORY processed as LISPX input, and not to be unread and processed by the editor.

The major implementation difference between the editor and LISPX occurs in undoing. EDITHISTORY is a list of only the last N commands, where N is the value of the time-slice. However the editor provides for undoing *all* changes made in a single editing session, even if that session consisted of more than N edit commands. Therefore, the editor saves undo information independently of the EDITHISTORY on a list called UNDOLST, (although it also stores each entry on UNDOLST in the field of the corresponding event on EDITHISTORY.) Thus, the commands UNDO, !UNDO, and UNBLOCK, are not dependent on EDITHISTORY, and in fact will work if EDITHISTORY= NIL, or even in a system which does not contain LISPX at all. For example, UNDO specifies undoing the last command on UNDOLST, even if that event no longer appears on EDITHISTORY. The only interaction between UNDO and the history list occurs when the user types UNDO followed by an event specification. In this case, the editor calls LISPXFIND to find the event, and then undoes the corresponding entry on UNDOLST. Thus the user can only undo a *specified* command within the scope of the EDITHISTORY. (Note that this is also the only way UNDO commands themselves can be undone, that is, by using the history feature, to specify the corresponding event, e.g., UNDO UNDO.)

The implementation of the actual undoing is similar to the way it is done in LISPX: each command that makes a change in the structure being edited does so via a function that records the change on a variable. After the command has completed, this variable contains a list of all the pointers that have been changed and their original contents. Undoing that command simply involves mapping down that list and restoring the pointers.

[9] as indicated by their appearance on HISTORYCOMS, a list of the history commands. EDITDEFAULT interrogates HISTORYCOMS before attempting spelling correction. (All of the commands on HISTORYCOMS are also on EDITCOMSA and EDITCOMSL so that they can be corrected if misspelled in the editor.) Thus if the user defines a LISPXMACRO and wishes it to operate in the editor as well, he need simply add it to HISTORYCOMS. For example, RETRIEVE is implemented as a LISPXMACRO and works equally well in LISPX and the editor.