XEROX

Interlisp-D Reference Manual
Volume II: Environment

3101273
October, 1985

[This page intentionally left blank]

13. INTERLISP EXECUTIVE

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 modifications, or to undo the effect of specified 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 difficulties and creative aspects of the problem at hand.

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-effects, formatting information, etc.

The following dialogue, taken from an actual session at the terminal, contains illustrative examples and gives the flavor of the programmer's assistant facility in Interlisp. The number before each prompt is the "event number" (see page 13.31).

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 first 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 sufficient.

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 specified -2 instead of 27 to specify a relative address.

.

.

.

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 different values. He specifies 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 first 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 modification 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 effect 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.

13.1 Input Formats

The Interlisp-D executive accepts inputs in the following formats:

EVALV-format input

If the user types a single litatom, followed by a carriage-return, the value of the litatom is returned. For example, if the value of the variable **FOO** is the list **(A B C)**:

←FOO^{cr}
(A B C)

EVAL-format input

If the user types a regular Interlisp expression, beginning with a left parenthesis or square bracket and terminated by a matching right parenthesis or square bracket, the form is simply passed to **EVAL** for evaluation. A right bracket matches any number of left parentheses, back to the last left bracket or the entire expression. 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 final 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))
7
←(PLUS 1 (TIMES 2 3]
7
←(PLUS 1 (TIMES^{cr}
2 3]
7

APPLY-format input

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. The input is terminated by the matching right parenthesis or bracket. For example, typing **LOAD(FOO)** is equivalent to typing **(LOAD 'FOO)**, and **GETPROP(X COLOR)** is equivalent to **(GETPROP 'X 'COLOR)**.

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 (see page 10.2). For example, typing **SETQ(FOO BAR)** will set **FOO** to the *value* of **BAR**, not to the litatom **BAR** itself.

Other input

Sometimes, a user does not want to terminate the input when a closing parenthesis is typed. For example, some programmer's assistant commands take several arguments, some of which can be lists. If the user types 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, the Interlisp-D executive interprets it differently depending on the number of expressions typed.

If only one expression is typed (a litatom), it is interpreted as an EVALV-format input, and the value of the litatom is returned:

←FOO<space>^{cr}
(A B C)

If exactly two expressions are typed, it is interpreted as APPLY-format input:

←LIST (A B)^{cr}
(A B)

If three or more expressions are typed, it is interpreted as EVAL-format input. To warn the user, the full expression is printed out before it is executed. For example:

←PLUS (TIMES 2 3) 1^{cr}
= (PLUS (TIMES 2 3) 1)
7

Note: If **LISPXREADFN** (page 13.36) is set to **READ** (rather than the default, **TTYINREAD**), then whenever one of the elements typed is a list and the list is terminated by a right parenthesis or bracket, 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.

13.2 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 defines 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 different commands may be concerned with different aspects of the corresponding event(s), e.g., side-effects, value, input, etc. Therefore, before discussing the various p.a. commands, the following section describes the types of event specifications currently implemented.

13.2.1 Event Specification

An event address identifies 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 14.6). The event identified 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 first 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 specified 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 first 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 first command in an event address), and **42 3** refers to the event with event number 45.

←*LITATOM* — Specifies the last event with an **APPLY**-format input whose *function* matches *LITATOM*.

Note: There must not be a space between ← and *LITATOM*.

← — Specifies that the next search is to go forward instead of backward. If given as the first event address command, the next search begins with last (oldest) event on the history list.

F — Specifies 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**.

= — Specifies that the next object (presumably a pattern) is to be matched against the *values* of events, instead of the inputs.

\ — Specifies the event last located.

SUCHTHAT *PRED* — Specifies 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 13.31 for a discussion of the format of events on the history list.

PAT Any other event address command specifies an event whose input contains an expression that matches *PAT* as described in page 16.18.

The matching is performed by the function **HISTORYMATCH** (page 13.40), which is initially defined to call **EDITFINDP** but can be advised or redefined for specialized applications.

Note: Symbols used below of the form $EventAddress_i$ 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** $EventAddress_1$ **THRU** $EventAddress_2$, the symbol $EventAddress_1$ corresponds to all words between **FROM** and **THRU** in the event specification, and $EventAddress_2$ to all words from **THRU** to the end of the event specification.

FROM $EventAddress_1$ **THRU** $EventAddress_2$

$EventAddress_1$ **THRU** $EventAddress_2$ Specifies the sequence of events from the event with address $EventAddress_1$ through the event with address $EventAddress_2$. For example, **FROM 47 THRU 49** specifies events 47, 48, and 49. $EventAddress_1$ can be more recent than $EventAddress_2$. For example, **FROM 49 THRU 47** specifies events 49, 48, and 47 (note reversal of order).

FROM $EventAddress_1$ **TO** $EventAddress_2$

$EventAddress_1$ **TO** $EventAddress_2$ Same as **THRU** but does not include event $EventAddress_2$.

FROM $EventAddress_1$ Same as **FROM** $EventAddress_1$ **THRU -1**. For example, if the current event is number 53, then **FROM 49** specifies events 49, 50, 51, and 52.

THRU $EventAddress_2$ Same as **FROM -1 THRU** $EventAddress_2$. For example, if the current event is number 53, then **THRU 49** specifies events 52, 51, 50, and 49 (note reversal of order).

TO $EventAddress_2$ Same as **FROM -1 TO** $EventAddress_2$.

ALL $EventAddress_1$ Specifies all events satisfying $EventAddress_1$. For example, **ALL LOAD, ALL SUCHTHAT FOO**.

empty If nothing is specified, 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 13.14), specifies the event(s) defining *LITATOM*.

@@ *EventSpec* — *EventSpec* is an event specification interpreted as above, but with respect to the archived history list (see the **ARCHIVE** command, page 13.16).

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.

13.2.2 Commands

All programmer's assistant commands can be input as list forms, or as lines (see page 13.36). 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 14.14) so that any errors that occur while executing *EventSpec* will cause breaks.

USE *EXPRS* **FOR** *ARGS* **IN** *EventSpec* [Prog. Asst. Command]

Substitutes *EXPRS* for *ARGS* in *EventSpec*, and redoes the result. Substitution is done by **ESUBST** (page 16.73), and is carried out as described below. *EXPRS* and *ARGS* 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** *EventSpec* is omitted, the first member of *ARGS* 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 first member of *ARGS* could be interpreted as an event address command.

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

If *ARGS* 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 effect, 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 *ARGS* 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** *EventSpec* is omitted, it is the same as specifying **IN -1**.

USE *EXPRS*$_1$ **FOR** *ARGS*$_1$ **AND** ... **AND** *EXPRS*$_N$ **FOR** *ARGS*$_N$ **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 finite 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 specification, which defines 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. 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 first substitute **A** for **X** then substitute **B** for **X** (in a new copy of the expression), then substitute **C** for **X**. The effect 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 first 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**.

Note: Parsing the **USE** command gets more complicated 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.

... *VARS* [Prog. Asst. Command]

Similar to **USE** except substitutes for the (first) *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 "escape," since this is how this character is echoed.

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

$ (escape) 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 **$** (escape) command: the same as **$** *X* **FOR** *Y* **IN** *EventSpec*, which changes *Y*s to *X*s.

$ does event location the same as the **USE** command, i.e., if **IN** *EventSpec* is not specified, **$** searches for *Y*. However, unlike **USE**, **$** can only be used to specify one substitution at a time. After **$** finds the event, it looks to see if an error was involved in that event, and if the indicated character substitution can be performed in the object of the error message, called the offender. If so, **$** assumes the substitution refers to the offender, performs the indicated character substitution in the offender only, and then substitutes the result for the original offender 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 specified event, the user can also omit specifying the object of the substitution, *Y*, in which case the offender 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 16.73), **$** 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 file 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 modifications to the input(s) are not simple substitutions of the type that can be specified 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 specification, 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 13.31.

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

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

For each event specified, **??** 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 field 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 affect relative event numbers. In other words, an event specification of **-1** typed following a **??** command will refer to the event immediately preceding the **??** command.

?? is implemented via the function **PRINTHISTORY**, page 13.42, which can also be called directly by the user. Printing is performed via the function **SHOWPRIN2** (page 25.10), so that if the value of **SYSPRETTYFLG** = **T**, events will be prettyprinted.

UNDO *EventSpec* [Prog. Asst. Command]

Undoes the side effects of the specified 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 effects, 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 effects of the last five events, perform **UNDO THRU -5**, *not* **UNDO FROM -5**. Undoing out of order may have unforseen effects 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 13.27.

UNDO *EventSpec* : $X_1 \ldots X_N$ [Prog. Asst. Command]

Each X_i is a pattern that is matched to a message printed by DWIM in the event(s) specified by *EventSpec*. The side effects of the corresponding DWIM corrections, and only those side effects, 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 16.45).

NAME *LITATOM EventSpec* [Prog. Asst. Command]

Saves the event(s) (including side effects) specified by *EventSpec* on 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 specification **@** *LITATOM*. For example, **?? @ FOO**, **REDO @ FOO**, etc.

Commands defined 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 defined by **NAME** can also be defined 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 *EventSpec* is 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 effects 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 *EventSpec* on a permanent "archived" history list, **ARCHIVELST** (page 13.31). This history list can be referenced by preceding a standard event specification with **@@** (see page 13.8). 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** (page 13.23), or by putting the history list property ***ARCHIVE***, with value **T**, on the event (page 13.33). Events that are referenced by history commands are automatically marked for archiving in this fashion.

FORGET *EventSpec* [Prog. Asst. Command]

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

FORGET is provided for users with space problems. For example, if the user has just performed **SET**s, **RPLACA**s, **RPLACD**s, **PUTD**, **REMPROP**s, 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 file 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 file package and added to the filecoms 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 *FILE***COMS**, 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 14.8). 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, 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 file (page 30.12).

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 off the history list.

TYPE-AHEAD [Prog. Asst. Command]

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

The assistant responds to **TYPE-AHEAD** with a prompt character of **>**. The user can now type in an indefinite 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 **$GO** (escape-**GO**). While in the type-ahead loop, **??** can be used to print the type-ahead, **FIX** to edit the type-ahead, and **$Q** (escape-**Q**) to erase the last input (may be used repeatedly). The **TYPE-AHEAD** command may be aborted by **$STOP** (escape-**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 field 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** effectively backs the history list up one entry when it retrieves the specified event. Similarly, **(VALUEOF FOO)** will find the first 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.

13.2.3 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 affect the searching operations of event specifications. 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 effect 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 effect, the user should type **USE D FOR E IN USE**. The appearance of the word **REDO**, **USE** or **FIX** in an event address specifies a search for the corresponding programmer's assistant command. It also specifies 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 affect the searching operations of event specifications" 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* find 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 specifications typed in by the user, but not to secondary event specifications.

In addition, if the most recent event is a history command which failed to produce input, a secondary event specification will effectively back up the history list one event so that relative event numbers for that event specification 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.

13.3 Changing The Programmer's Assistant

(CHANGESLICE *N HISTORY* **—)** [Function]

Changes the time-slice of the history list *HISTORY* to *N* (see page 13.31). If *HISTORY* is **NIL**, changes both the top level history list **LISPXHISTORY** and the edit history list **EDITHISTORY**.

Note: The effect 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 sufficient 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 off the history list, he must perform a **FORGET** command (page 13.16).

PROMPT#FLG [Variable]

When this variable is set to **T**, the current event number to be printed before each prompt character. See **PROMPTCHAR**, page 13.38. **PROMPT#FLG** is initially **T**.

PROMPTCHARFORMS [Variable]

The value of **PROMPTCHARFORMS** is a list of expression which are evaluated each time **PROMPTCHAR** (page 13.38) is called to print the prompt character. If **PROMPTCHAR** is going to print something, it first 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 13.38) 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 effect, 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 different 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, or calls function **RESET**.

ARCHIVEFN [Variable]

If the *value* of **ARCHIVEFN** is **T**, and an event is about to drop off 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 13.31 for the format of events). If **ARCHIVEFN** returns **T**, the event is archived on a permanent history list (see page 13.16). Note that **ARCHIVEFN** must be *both* set and defined. **ARCHIVEFN** is initially **NIL** and undefined.

For example, defining **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 off 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 13.31). The user could do this by means of an appropriately defined **LISPXUSERFN** (see below).

LISPXMACROS [Variable]

LISPXMACROS provides a macro facility that allows the user to define his own programmer's assistant commands. It is a list of elements of the form (*COMMAND DEF*). Whenever *COMMAND* appears as the first 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 define 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 defining **LISPX** commands which are executed for effect only.

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

Note: Definitions of commands on **LISPXMACROS** or **LISPXHISTORYMACROS** can be saved on files with the file package command **LISPXMACROS** (see page 17.39).

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 defined.

LISPXUSERFN is given two arguments: *X* and *LINE*. *X* is the first expression typed, and *LINE* is the rest of the line, as read by **READLINE** (page 13.36). 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 defining (and setting) **LISPXUSERFN**, the user can with a minimum of effort incorporate the features of the programmer's assistant into his own executive (actually it is the other way around). For example, **LISPXUSERFN** could be defined 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 NODOFLG***)** [Function]

(LISPXPRIN1 *X Y Z NODOFLG***)** [Function]

(LISPXPRIN2 *X Y Z NODOFLG***)** [Function]

(LISPXSPACES *X Y Z NODOFLG***)** [Function]

(LISPXTERPRI *X Y Z NODOFLG***)** [Function]

(LISPXTAB *X Y Z NODOFLG***)** [Function]

(LISPXPRINTDEF *EXPR FILE LEFT DEF TAIL NODOFLG***)** [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 13.31).

If *NODOFLG* 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 define additional **LISPX** printing functions. If the user has a function *FN* that takes three or fewer arguments, and the second argument is the file name, he can define a **LISPX** printing function by simply giving **LISPX***FN* the definition of **USERLISPXPRINT**, for example, with **MOVD(USERLISPXPRINT LISPX***FN***)**. **USERLISPXPRINT** is defined to look back on the stack, find the name of the calling function, strip off 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**.

13.4 Undoing

Note: This discussion only applies to undoing under the executive and break; the editors handles undoing itself in a slightly different 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 effects. 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 specifically 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 specifically 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 affected, 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 sufficiently 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 defined 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. 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 effects of the following functions are always undoable: **DEFINE, DEFINEQ, DEFC** (used to give a function a compiled code definition), **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.

The programmer's assistant cannot know whether efficiency 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 (using **LISPX/**, page 13.41). 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 decision concerning which operations to make undoable could be better advised. However, we have found the configuration 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.

Note: The user can define new "slash" functions to be translated on type-in by calling **NEW/FN** (page 13.41).

13.4.1 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 **/RPLACA**s 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 first, then undoing the older event, **FOO** will be restored to its state before either **RPLACA** operated. However if the user undoes the first 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 affect different lists or different 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 effect.

Property list operations, (i.e., **PUTPROP**, **ADDPROP** and **REMPROP**) are handled specially, so that operations that affect different 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 first event, the **BAZ** property will remain, even though it may not have been on the property list of **FOO** at the time the first event was executed.

13.4.2 SAVESET

Typed-in **SET**s 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 17.27) 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 flip 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 file.

For top level variables, **SAVESET** also adds the variable to the appropriate spelling list, thereby noticing variables set in files 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 20.17). Furthermore, if the old value was not **NOBIND**, and was also not **EQUAL** to the new value, **SAVESET** calls the file package to update the necessary file 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 files without disturbing the current value of variables (see page 10.10).

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.

(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**.

13.4.3 UNDONLSETQ and RESETUNDO

The function **UNDONLSETQ** provides a limited form of backtracking: if an error occurs under the **UNDONLSETQ**, all undoable side effects executed under the **UNDONLSETQ** are undone. **RESETUNDO**, used in conjunction with **RESETLST** and **RESETSAVE** (page 14.24), provides a more general undo capability where the user can specify that the side effects be undone after the specified computation finishes, is aborted by an error, or by a control-D.

(UNDONLSETQ *UNDOFORM* —**)** [NLambda Function]

An nlambda function similar to **NLSETQ** (page 14.22). **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 14.24.

(RESETUNDO *X STOPFLG***)** [Function]

For use in conjunction with **RESETLST** (page 14.24). **(RESETUNDO)** initializes the saving of undo information and returns a value which when given back to **RESETUNDO** undoes the intervening side effects. For example, **(RESETLST (RESETSAVE (RESETUNDO))** . *FORMS***)** will undo the side effects of *FORMS* on normal exit, or if an error occurs or a control-D is typed.

If *STOPFLG* = **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 effects in *FORMS*.

13.5 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 13.16). 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# SIZE MOD*), where *L* is the list of events with the most recent event first, *EVENT#* 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 flags as well as repositories of events.

Note: One of the reasons why users may disable the history list facility is to allow the garbage collector to reclaim objects stored on the history list. Simply setting **LISPXHISTORY** to **NIL** will not necessarily remove all pointers to the history list. **GAINSPACE** (page 22.12) is a useful function when trying to reclaim memory space.

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 efficiency, 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 buffer. The time-slice of a history list can be changed with the function **CHANGESLICE** (page 13.21). 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 13.14), 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 13.22) 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. On

EDITHISTORY, this field is used to save the side effects of each command (see page 13.43). *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 (see page 13.4), 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. Specifically, the *INPUT* fields from the specified events are concatenated into a single list, seperated by special markers called "pseudo-carriage returns," which print out as the string **"<c.r.>"**. 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-format and EVAL-format inputs to **LISPX**, and to delimit line commands to the editor.

Note: 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.

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 event specification 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 effects of the event. See **UNDOSAVE**, page 13.40.

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 offender 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 13.25).

ARCHIVE The property ***ARCHIVE*** on an event causes the event to be automatically archived when it "falls off the end" of the history list (see page 13.16).

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 effects, etc. for the individual inputs being reexecuted. This structure is described below.

When **LISPX** is given an input, it calls **HISTORYSAVE** (page 13.38) to record the input in a new event (except for the commands ??, **FORGET**, **TYPE-AHEAD**, **$BUFS**, and **ARCHIVE**, that are executed immediately and are not recorded on the history list). 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 history list. After binding **LISPXHIST**, **LISPX** executes the input, stores its value in the value field 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 finds the event from which the input originally came. If **HISTORYSAVE** cannot find 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. **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 finished, its entry will look like:

(*INPUT ID VALUE*
  HISTORY
    COMMAND
  GROUP
    ((*INPUT*$_1$ *ID*$_1$ *VALUE*$_1$ **SIDE** *SIDE*$_1$)
     (*INPUT*$_2$ *ID*$_2$ *VALUE*$_2$ **SIDE** *SIDE*$_2$)
     ...))

In this case, the value field 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.

13.6 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. The user should be careful about using the prompt characters "←," "*," or ":," because in certain cases **LISPX** uses the value of *LISPXID* to tell whether or not it was called from the break package or editor.

If *LISPXXMACROS* 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 define **LISPXUSERFN** as described on page 13.24.

LISPXFLG is used by the **E** command in the editor (see page 13.43).

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 affected.

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

LISPX is responsible for rebinding **HELPCLOCK**, used by **BREAKCHECK** (page 14.13) 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 LISPXXMACROS LISPXXUSERFN***)** [Function]

Repeatedly calls **LISPX** under errorset protection specifying *LISPXXMACROS* 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 *LISPXFORM LISPXID***)** [Function]

Evaluates *LISPXFORM* (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 13.27). **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 following variable to store the expressions that have been unread:

READBUF [Variable]

This variable is used by **LISPXREAD** and **READLINE** 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 13.32). 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 **TTYINREAD** (page 26.28). The user can make **LISPX**, the editor, break, etc. do their reading via a different 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 13.38), 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 until it encounters either:

- an **EOL** (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 redefined with **SETSYNTAX** (page 25.37). **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 differently 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 first 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
*

Note: **READLINE** reads expressions by performing **(APPLY* LISPXREADFN T)**. **LISPXREADFN** (page 13.36) is initially set to **TTYINREAD** (page 26.28).

(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**.

LISPXREAD also sets **REREADFLG** (page 13.39) to **NIL** when it reads via **LISPXREADFN**, 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 13.22) is **T** and *HISTORY* is not **NIL**, **PROMPTCHAR** prints the current event number (of *HISTORY*) before printing *ID*.

The value of **PROMPTCHARFORMS** (page 13.22) 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 field 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 13.34.

HISTORYSAVEFORMS (page 13.22) 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 specification, *TYPE* specifies 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** (page 13.40) to find 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 searches the archived history list instead of *HISTORY* (see the **ARCHIVE** command, page 13.16).

(4) **LISPXFIND** recognizes @, and retrieves the corresponding event(s) from the property list of the atom following @ (see page 13.14).

(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 first three elements of a "history list" structure (see page 13.31). *EVENTADDRESS* is an event address (see page 13.6) e.g., **(43)**, **(-1)**, **(FOO FIE)**, **(LOAD ← FOO)**, etc. If **HISTORYFIND** cannot find *EVENTADDRESS*, it generates an error.

(HISTORYMATCH *INPUT PAT EVENT***)** [Function]

Used by **HISTORYFIND** for "matching" when *EVENTADDRESS* specifies a pattern. Matches *PAT* against *INPUT*, the input portion of the history event *EVENT*, as matching is defined on page 16.18. Initially defined as **(EDITFINDP** *INPUT PAT* **T)**, but can be advised or redefined by the user.

(ENTRY# *HIST X***)** [Function]

HIST is a history list (see page 13.31). *X* is **EQ** to one of the events on *HIST*. **ENTRY#** returns the event number for *X*.

(UNDOSAVE *UNDOFORM HISTENTRY***)** [Function]

UNDOSAVE adds the "undo information" *UNDOFORM* to the **SIDE** property of the history event *HISTENTRY*. If there is no **SIDE** property, one is created. If the value of the **SIDE** property is **NOSAVE**, the information is not saved.

HISTENTRY specifies an event. If *HISTENTRY* = **NIL**, the value of **LISPXHIST** is used. If both *HISTENTRY* and **LISPXHIST** are **NIL**, **UNDOSAVE** is a no-op. Note that *HISTENTRY* (or **LISPXHIST**) can either be a "real" event, or an event within the ***GROUP*** property of another event (see page 13.34).

The form of *UNDOFORM* is *(FN . ARGS)*. Undoing is done by performing **(APPLY (CAR** *UNDOFORM***) (CDR** *UNDOFORM***))**. For example, if the definition of **FOO** is *DEF*, **(/PUTD FOO** *NEWDEF***)** will cause a call to **UNDOSAVE** with *UNDOFORM* = **(/PUTD FOO** *DEF***)**.

Note: In the special case of **/RPLNODE** and **/RPLNODE2**, the format of *UNDOFORM* is *(X OLDCAR . OLDCDR)*. When *UNDOFORM* is undone, this form is recognized and handled specially. This implementation saves space.

CAR of the **SIDE** property of an event is a count of the number of *UNDOFORM*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 *UNDOFORM*s can be saved. If this count is a positive number, **UNDOSAVE** restricts the number of *UNDOFORM*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 *UNDOFORM*s to be saved for a single event. When the count of *UNDOFORM*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 user. **#UNDOSAVES** = **NIL** is equivalent to **#UNDOSAVES** = infinity. **#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 21.21). 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 specification. **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 finds 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 effects of **UNDOLISPX1** to be undone.

(PRINTHISTORY *HISTORY LINE SKIPFN NOVALUES FILE***)** [Function]

LINE is an event specification. **PRINTHISTORY** prints the events on *HISTORY* specified 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 *NOVALUES* = **T**, or *NOVALUES* applied to the corresponding event is true, the value is not printed. For example, *NOVALUES* is **T** when printing events on **EDITHISTORY**.

For example, the following **LISPXMACRO** will define **??'** 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))
```

13.7 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 differences 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**, 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. **EDITHISTORY** follows the same conventions and format as **LISPXHISTORY** (page 13.31). However, since edit commands have no value, the editor uses the value field for saving side effects, 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 commands 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 16.57). 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.

Note: The editor determines which history commands to pass to **LISPX** by looking at **HISTORYCOMS**, a list of the history commands. **EDITDEFAULT** (page 16.66) 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.

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.

[This page intentionally left blank]

14. ERRORS AND BREAKS

Occasionally, while a program is running, an error may occur which stops the computation. Errors can be caused in different ways. A coding mistake may have caused the wrong arguments to be passed to a function, or caused the function to try doing something illegal. For example, **PLUS** will cause an error if its arguments are not numbers. It is also possible to interrupt a computation at any time by typing one of the "interrupt characters," such as control-D or control-E (the Interlisp-D interrupt characters are listed on page 30.1). Finally, the programmer can specify that certain functions automatically cause an error whenever they are entered (see page 15.1). This facilitates debugging by allowing examination of the context within the computation.

When an error occurs, the system can either reset and unwind the stack, or go into a "break", an environment where the user can examine the state of the system at the point of the error, and attempt to debug the program. The mechanism that decides whether to unwind the stack or break can be modified by the user, and is described on page 14.13 of this chapter. Within a break, Interlisp offers an extensive set of "break commands" which assist with debugging.

This chapter explains what happens when errors occur. It also tells the user how to handle program errors using breaks and break commands. The debugging capabilities of break window facility are described, as well as the variables that control its operation. Finally, advanced facilities for modifying and extending the error mechanism are presented.

14.1 Breaks

One of the most useful debugging facilities in Interlisp is the ability to put the system into a "break", stopping a computation at any point and allowing the user to interrogate the state of the world and affect the course of the computation. When a break occurs, a "break window" (see page 14.3) is brought up near the tty window of the process that broke. The break window appears to the user like a top-level executive window, except that the prompt character ":" is used to indicate that the executive is ready to accept input, in the same way that "←" is

used at the top-level executive. However, a break saves the environment where the break occurred, so that the user may evaluate variables and expressions in the environment that was broken. In addition, the break program recognizes a number of useful "break commands", which provide an easy way to interrogate the state of the broken computation.

Breaks may be entered in several different ways. Some interrupt characters (page 30.1) automatically cause a break to be entered whenever they are typed. Function errors may also cause a break, depending on the depth of the computation (see page 14.13). Finally, Interlisp provides facilities which make it easy to "break" suspect functions so that they always cause a break whenever they are entered, to allow examination and debugging (see page 15.5).

Within a break the user has access to all of the power of Interlisp; he can do anything that he can do at the top-level executive. For example, the user can evaluate an expression, see that the value is incorrect, call the editor, change the function, and evaluate the expression again, all without leaving the break. The user can also type in commands to the programmer's assistant (page 13.1), e.g. to redo or undo previously executed events, including break commands.

Similarly, the user can prettyprint functions, define new functions or redefine old ones, load a file, compile functions, time a computation, etc. In short, anything that he can do at the top level can be done while inside of the break. In addition the user can examine the stack (see page 11.1), and even force a return back to some higher function via the function **RETFROM** or **RETEVAL**.

It is important to emphasize that once a break occurs, the user is in complete control of the flow of the computation, and the computation will not proceed without specific instruction from him. If the user types in an expression whose evaluation causes an error, the break is maintained. Similarly if the user aborts a computation initiated from within the break (by typing control-E), the break is maintained. Only if the user gives one of the commands that exits from the break, or evaluates a form which does a **RETFROM** or **RETEVAL** back out of **BREAK1**, will the computation continue. Also, **BREAK1** does not "turn off" control-D, so a control-D will force an immediate return back to the top level.

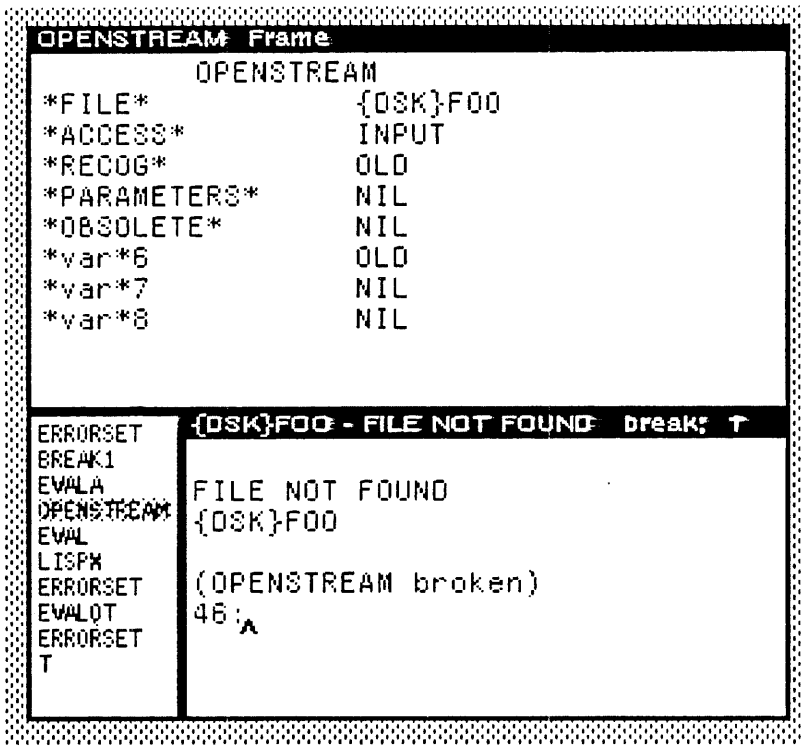
14.2 Break Windows

When a break occurs, a break window is brought up near the tty window of the process that broke and the terminal stream switched to it. The title of the break window is changed to give the name of the broken function and the reason for the break. If a break occurs under a previous break, a new break window is created.

Note: If a break is caused by a storage full error, the display break package will not try to open a new break window, since this would cause the error to occur repeatedly.

While in a break window, the clicking middle button brings up a menu of break commands: **!EVAL, EVAL, EDIT, revert, ↑, OK, BT, BT!,** and **?=**. Clicking on most of these commands is equivalent to typing the corresponding break command (page 14.5). Clicking **BT** and **BT!**, however, is different from the typed-in backtrace break commands.

The **BT** and **BT!** menu commands bring up a backtrace menu beside the break window showing the frames on the stack. **BT** shows frames for which **REALFRAMEP** is **T**; **BT!** shows all frames. When one of the frames is selected from the backtrace menu, it is grayed and the function name and the variables bound in that frame (including local variables and **PROG** variables) are printed in the "backtrace frame window." If the left button is used for the selection, only named variables are printed. If the middle button is used, all variables are printed (variables without names will appear as ***var****N*). The "backtrace frame" window is an inspect window (see page 26.1). In this window, the left button can be used to select the name of the function, the names of the variables or the values of the variables. For example, below is a picture of a break window with a backtrace menu created by **BT**. The **OPENSTREAM** stack frame has been selected, so its variables are shown in an inspect window on top of the break window:



After selecting an item, the middle button brings up a menu of commands that apply to the selected item. If the function name is selected, a choice of editing the function or seeing the compiled code with **INSPECTCODE** (page 26.2) will be given. If the function is edited in this way, the editor is called in the broken process, so variables evaluated in the editor will be in the broken process.

If a variable name is selected, the command **SET** will be offered. Selecting **SET** will **READ** a value and set the selected to the value read. (Note: The inspector will only allow the setting of named variables. Even with this restriction it is still possible to crash the system by setting variables inside system frames. It is recommended that you exercise caution in setting variables in other than your own code.) If the item selected is a value, the inspector will be called on the selected value.

The internal break variable **LASTPOS** (page 14.6) is set to the selected frame of the backtrace menu so that the normal break commands **EDIT**, **revert**, and **?=** work on the currently selected frame. The commands **EVAL**, **revert**, **↑**, **OK**, and **?=** in the break menu cause the corresponding commands to be "typed in." This means that these break commands will not have the intended effect if characters have already been typed in. Note also that the typed-in break commands **BT**, **BTV**, etc. use the value of **LASTPOS** to determine where to start listing the stack, so selecting a stack frame name in the backtrace menu will effect these commands.

14.3 Break Commands

The basic function of the break package is **BREAK1**. **BREAK1** is just another Interlisp function, not a special system feature like the interpreter or the garbage collector. It has arguments, and returns a value, the same as any other function. For more information on the function **BREAK1**, see page 14.16.

The value returned by **BREAK1** is called "the value of the break." The user can specify this value explicitly by using the **RETURN** break command (page 14.6). But in most cases, the value of a break is given implicitly, via a **GO** or **OK** command, and is the result of evaluating "the break expression." The break expression, stored in the variable **BRKEXP**, is an expression equivalent to the computation that would have taken place had no break occurred. For example, if the user breaks on the function **FOO**, the break expression is the body of the definition of **FOO**. When the user types **OK** or **GO**, the body of **FOO** is evaluated, and its value returned as the value of the break, i.e., to whatever function called **FOO**. **BRKEXP** is set up by the function that created the call to **BREAK1**. For functions broken with **BREAK** or **TRACE**, **BRKEXP** is equivalent to the body of the definition of the broken function (see page 15.5). For functions broken with **BREAKIN**, using **BEFORE** or **AFTER**, **BRKEXP** is **NIL**. For **BREAKIN AROUND**, **BRKEXP** is the indicated expression (see page 15.6).

BREAK1 recognizes a large set of break commands. These are typed in *without* parentheses. In order to facilitate debugging of programs that perform input operations, the carriage return that is typed to complete the **GO**, **OK**, **EVAL**, etc. commands is discarded by **BREAK1**, so that it will not be part of the input stream after the break.

GO [Break Command]

Evaluates **BRKEXP**, prints this value, and returns it as the value of the break. Releases the break and allows the computation to proceed.

OK [Break Command]

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

EVAL [Break Command]

Same as **OK** except that the break is maintained after the evaluation. The value of this evaluation is bound to the local variable **!VALUE**, which the user can interrogate. Typing **GO** or **OK** following **EVAL** will not cause **BRKEXP** to be reevaluated, but simply returns the value of **!VALUE** as the value of the break. Typing another **EVAL** will cause reevaluation. **EVAL** is useful

when the user is not sure whether the break will produce the correct value and wishes to examine it before continuing with the computation.

RETURN *FORM* [Break Command]

FORM is evaluated, and returned as the value of the break. For example, one could use the **EVAL** command and follow this with **RETURN (REVERSE !VALUE)**.

↑ [Break Command]

Calls **ERROR!** and aborts the break, making it "go away" without returning a value. This is a useful way to unwind to a higher level break. All other errors, including those encountered while executing the **GO**, **OK**, **EVAL**, and **RETURN** commands, maintain the break.

The following four commands refer to "the broken function." This is the function that caused the break, whose name is stored in the **BREAK1** argument **BRKFN**.

!EVAL [Break Command]

The broken function is first unbroken, then the break expression is evaluated (and the value stored in **!VALUE**), and then the function is rebroken. This command is very useful for dealing with recursive functions.

!GO [Break Command]

Equivalent to **!EVAL** followed by **GO**. The broken function is unbroken, the break expression is evaluated, the function is rebroken, and then the break is exited with the value typed.

!OK [Break Command]

Equivalent to **!EVAL** followed by **OK**. The broken function is unbroken, the break expression is evaluated, the function is rebroken, and then the break is exited.

UB [Break Command]

Unbreaks the broken function.

@ [Break Command]

Resets the variable **LASTPOS**, which establishes a context for the commands **?=**, **ARGS**, **BT**, **BTV**, **BTV***, **EDIT**, and **IN?** described below. **LASTPOS** is the position of a function call on the stack. It is initialized to the function just before the call to **BREAK1**, i.e., **(STKNTH -1 'BREAK1)**.

Note: When control passes from **BREAK1**, e.g. as a result of an **EVAL, OK, GO, REVERT,** ↑ command, or via a **RETFROM** or **RETEVAL** typed in by the user, **(RELSTK LASTPOS)** is executed to release this stack pointer.

@ treats the rest of the teletype line as its argument(s). It first resets **LASTPOS** to **(STKNTH -1 'BREAK1)** and then for each atom on the line, @ searches down the stack for a call to that atom. The following atoms are treated specially:

@ Do not reset **LASTPOS** to **(STKNTH -1 'BREAK1)** but leave it as it was, and continue searching from that point.

a number *N* If negative, move **LASTPOS** down the stack *N* frames. If positive, move **LASTPOS** up the stack *N* frames.

/ The next atom on the line (which should be a number) specifies that the *previous* atom should be searched for that many times. For example, "**@ FOO / 3**" is equivalent to "**@ FOO FOO FOO**".

= Resets **LASTPOS** to the *value* of the next expression, e.g., if the value of **FOO** is a stack pointer, "**@ = FOO FIE**" will search for **FIE** in the environment specified by (the value of) **FOO**.

For example, if the push-down stack looks like:

[9] **BREAK1**
[8] **FOO**
[7] **COND**
[6] **FIE**
[5] **COND**
[4] **FIE**
[3] **COND**
[2] **FIE**
[1] **FUM**

then "**@ FIE COND**" will set **LASTPOS** to the position corresponding to *[5]*; "**@ @ COND**" will then set **LASTPOS** to *[3]*; and "**@ FIE / 3 -1**" to *[1]*.

If @ cannot successfully complete a search for function *FN*, it searches the stack again from that point looking for a call to a function whose name is close to that of *FN*, in the sense of the spelling corrector (page 20.15). If the search is still unsuccessful, @ types **(*FN* NOT FOUND)**, and then aborts.

When @ finishes, it types the name of the function at **LASTPOS**, i.e., **(STKNAME LASTPOS)**.

@ can be used on **BRKCOMS** (see page 14.17). In this case, the *next* command on **BRKCOMS** is treated the same as the rest of the teletype line.

?= [Break Command]

This is a multi-purpose command. Its most common use is to interrogate the value(s) of the arguments of the broken

function. For example, if **FOO** has three arguments **(X Y Z)**, then typing **?=** to a break on **FOO** will produce:

```
:?=
X = value of X
Y = value of Y
Z = value of Z
:
```

?= operates on the rest of the teletype line as its arguments. If the line is empty, as in the above case, it operates on all of the arguments of the broken function. If the user types **?= X (CAR Y)**, he will see the value of **X**, and the value of **(CAR Y)**. The difference between using **?=** and typing **X** and **(CAR Y)** directly to **BREAK1** is that **?=** evaluates its inputs as of the stack frame **LASTPOS**, i.e., it uses **STKEVAL**. This provides a way of examining variables or performing computations *as of a particular point on the stack*. For example, **@ FOO / 2** followed by **?= X** will allow the user to examine the value of **X** in the previous call to **FOO**, etc.

?= also recognizes numbers as referring to the correspondingly numbered argument, i.e., it uses **STKARG** in this case. Thus

```
:@ FIE
FIE
:?= 2
```

will print the name and value of the second argument of **FIE**.

?= can also be used on **BRKCOMS** (page 14.17, in which case the next command on **BRKCOMS** is treated as the rest of the teletype line. For example, if **BRKCOMS** is **(EVAL ?= (X Y) GO)**, **BRKEXP** will be evaluated, the values of **X** and **Y** printed, and then the function exited with its value being printed.

Note: **?=** prints variable values using the function **SHOWPRINT** (page 25.10), so that if **SYSPRETTYFLG = T**, the value will be prettyprinted.

?= is a universal mnemonic for displaying argument names and their corresponding values. In addition to being a break command, **?=** is an edit macro which prints the argument names and values for the current expression (page 16.48), and a read macro (actually **?** is the read macro character) which does the same for the current level list being read.

PB [Break Command]

Prints the bindings of a given variable. Similar to **?=**, except ascends the stack starting from **LASTPOS**, and, for each frame in which the given variable is bound, prints the frame name and value of the variable (with **PRINTLEVEL** reset to **(2 . 3)**), e.g.

```
:PB FOO
@ FN1: 3
@ FN2: 10
@ TOP: NOBIND
```

PB is also a programmer's assistant command (page 13.17) that can be used when not in a break. **PB** is implemented via the function **PRINTBINDINGS**.

BT [Break Command]

Prints a backtrace of function names only starting at **LASTPOS**. The value of **LASTPOS** is changed by selecting an item from the backtrace menu page 14.15 or by the @ command. The several nested calls in system packages such as break, edit, and the top level executive appear as the single entries ****BREAK****, ****EDITOR****, and ****TOP**** respectively.

BTV [Break Command]

Prints a backtrace of function names *with* variables beginning at **LASTPOS**.

The value of each variable is printed with the function **SHOWPRINT** (page 25.10), so that if **SYSPRETTYFLG** = **T**, the value will be prettyprinted.

BTV + [Break Command]

Same as **BTV** except also prints local variables and arguments to **SUBRs**.

BTV* [Break Command]

Same as **BTV** except prints arguments to local variables and eval blips (see page 11.14).

BTV! [Break Command]

Same as **BTV** except prints *everything* on the stack.

BT, **BTV**, **BTV +**, **BTV***, and **BTV!** all take optional functional arguments. These arguments are used to choose functions to be *skipped* on the backtrace. As the backtrace scans down the stack, the name of each stack frame is passed to each of the arguments of the backtrace command. If any of these functions returns a non-**NIL** value, then that frame is skipped, and not shown in the backtrace. For example, **BT EXPRP** will skip all functions definied by expr definitions, **BTV (LAMBDA (X) (NOT (MEMB X FOOFNS)))** will skip all but those functions on **FOOFNS**. If used on **BRKCOMS** (page 14.17) the functional argument is no longer optional, i.e., the next element on **BRKCOMS** must either

be a list of functional arguments, or **NIL** if no functional argument is to be applied.

For **BT**, **BTV**, **BTV+**, **BTV***, and **BTV!**, if control-P is used to change a printlevel during the backtrace, the printlevel will be restored after the backtrace is completed.

The value of **BREAKDELIMITER**, initially the carriage return character, is printed to delimit the output of **?=** and backtrace commands. This can be reset (e.g. to the comma) for more linear output.

ARGS [Break Command]

Prints the names of the variables bound at **LASTPOS**, i.e., **(VARIABLES LASTPOS)** (page 11.7). For most cases, these are the arguments to the function entered at that position, i.e., **(ARGLIST (STKNAME LASTPOS))**.

REVERT [Break Command]

Goes back to position **LASTPOS** on stack and reenters the function called at that point with the arguments found on the stack. If the function is not already broken, **REVERT** first breaks it, and then unbreaks it after it is reentered.

REVERT can be given the position using the conventions described for **@**, e.g., **REVERT FOO -1** is equivalent to **@ FOO -1** followed by **REVERT**.

REVERT is useful for restarting a computation in the situation where a bug is discovered at some point *below* where the problem actually occurred. **REVERT** essentially says "go back there and start over in a break." **REVERT** will work correctly if the names or arguments to the function, or even its function type, have been changed.

ORIGINAL [Break Command]

For use in conjunction with **BREAKMACROS** (see page 14.17). Form is **(ORIGINAL . *COMS*)**. *COMS* are executed without regard for **BREAKMACROS**. Useful for redefining a break command in terms of itself.

The following two commands are for use only with unbound atoms or undefined function breaks.

= *FORM* [Break Command]

Can only be used in a break following an unbound atom error. Sets the atom to the value of *FORM*, exits from the break returning that value, and continues the computation, e.g.,

UNBOUND ATOM

(FOO BROKEN)
: = (COPY FIE)

sets **FOO** and goes on.

Note: *FORM* may be given in the form *FN*[*ARGS*].

-> *EXPR* [Break Command]

Can be used in a break following either an unbound atom error, or an undefined function error. Replaces the expression containing the error with *EXPR* (not the value of *EXPR*), and continues the computation. **->** does not just change **BRKEXP**; it changes the function or expression containing the erroneous form. In other words, the user does not have to perform any additional editing.

For example,

UNDEFINED CAR OF FORM

(FOO1 BROKEN)
:-> FOO

changes the **FOO1** to **FOO** and continues the computation. *EXPR* need not be atomic, e.g.,

UNBOUND ATOM

(FOO BROKEN)
:-> (QUOTE FOO)

For undefined function breaks, the user can specify a function *and* initial arguments, e.g.,

UNDEFINED CAR OF FORM
MEMBERX

(MEMBERX BROKEN)
:-> MEMBER X

Note that in the case of a undefined function error occurring immediately following a call to **APPLY** (e.g., **(APPLY X Y)** where the value of **X** is **FOO** and **FOO** is undefined), or a unbound atom error immediately following a call to **EVAL** (e.g., **(EVAL X)**, where the value of **X** is **FOO** and **FOO** is unbound), there *is* no expression containing the offending atom. In this case, **->** cannot operate, so **?** is printed and no action is taken.

EDIT [Break Command]

Designed for use in conjunction with breaks caused by errors. Facilitates editing the expression causing the break:

NON-NUMERIC ARG
NIL

```
. (IPLUS BROKEN)
:EDIT
IN FOO...
(IPLUS X Z)
EDIT
*(3 Y)
*OK
FOO
:
```

and the user can continue by typing **OK**, **EVAL**, etc.

This command is very simple conceptually, but its implementation is complicated by all of the exceptional cases involving interactions with compiled functions, breaks on user functions, error breaks, breaks within breaks, et al. Therefore, we shall give the following simplified explanation which will account for 90% of the situations arising in actual usage. For those others, **EDIT** will print an appropriate failure message and return to the break.

EDIT begins by searching up the stack beginning at **LASTPOS** (set by **@** command, initially position of the break) looking for a form, i.e., an internal call to **EVAL**. Then **EDIT** continues from that point looking for a call to an interpreted function, or to **EVAL**. It then calls the editor on either the **EXPR** or the argument to **EVAL** in such a way as to look for an expression **EQ** to the form that it first found. It then prints the form, and permits interactive editing to begin. Note that the user can then type successive **0**'s to the editor to see the chain of superforms for this computation.

If the user exits from the edit with an **OK**, the break expression is reset, if possible, so that the user can continue with the computation by simply typing **OK**. (Note that evaluating the new **BRKEXP** will involve reevaluating the form that causes the break, so that if **(PUTD (QUOTE (FOO))** *BIG-COMPUTATION*) were handled by **EDIT**, *BIG-COMPUTATION* would be reevaluated.) However, in some situations, the break expression cannot be reset. For example, if a compiled function **FOO** incorrectly called **PUTD** and caused the error **ARG NOT ATOM** followed by a break on **PUTD**, **EDIT** might be able to find the form headed by **FOO**, and also find *that* form in some higher interpreted function. But after the user corrected the problem in the **FOO**-form, if any, he would still not have informed **EDIT** what to do about the immediate problem, i.e., the incorrect call to **PUTD**. However, if **FOO** were *interpreted*, **EDIT** would find the **PUTD** form itself, so that when the user corrected that form, **EDIT** could use the new corrected form to reset the break expression.

IN? [Break Command]

Similar to **EDIT**, but just prints parent form, and superform, but does not call the editor, e.g.,

```
ATTEMPT TO RPLAC NIL
T
(RPLACD BROKEN)
:IN?
FOO: (RPLACD X Z)
```

Although **EDIT** and **IN?** were designed for error breaks, they can also be useful for user breaks. For example, if upon reaching a break on his function **FOO**, the user determines that there is a problem in the *call* to **FOO**, he can edit the calling form and reset the break expression with one operation by using **EDIT**.

14.4 Controlling When to Break

When an error occurs, the system has to decide whether to reset and unwind the stack, or go into a break. In the middle of a complex computation, it is usually helpful to go into a break, so that the user may examine the state of the computation. However, if the computation has only proceeded a little when the error occurs, such as when the user mistypes a function name, the user would normally just terminate a break, and it would be more convenient for the system to simply cause an error and unwind the stack in this situatuation. The decision over whether or not to induce a break depends on the depth of computation, and the amount of time invested in the computation. The actual algorithm is described in detail below; suffice it to say that the parameters affecting this decision have been adjusted empirically so that trivial type-in errors do not cause breaks, but deep errors do.

(BREAKCHECK *ERRORPOS ERXN***)** [Function]

BREAKCHECK is called by the error routine to decide whether or not to induce a break when a error occurs. *ERRORPOS* is the stack position at which the error occurred; *ERXN* is the error number. Returns **T** if a break should occur; **NIL** otherwise.

BREAKCHECK returns **T** (and a break occurs) if the "computation depth" is greater than or equal to **HELPDEPTH**. **HELPDEPTH** is initially set to 7, arrived at empirically by taking into account the overhead due to **LISPX** or **BREAK**.

If the depth of the computation is less than **HELPDEPTH**, **BREAKCHECK** next calculates the length of time spent in the

computation. If this time is greater than **HELPTIME** milliseconds, initially set to 1000, then **BREAKCHECK** returns **T** (and a break occurs), otherwise **NIL**.

BREAKCHECK determines the "computation depth" by searching back up the stack looking for an **ERRORSET** frame (**ERRORSET**s indicate how far back unwinding is to take place when an error occurs, see page 14.21). At the same time, it counts the number of internal calls to **EVAL**. As soon as the number of calls to **EVAL** exceeds **HELPDEPTH**, **BREAKCHECK** immediately stops searching for an **ERRORSET** and returns **T**. Otherwise, **BREAKCHECK** continues searching until either an **ERRORSET** is found or the top of the stack is reached. (Note: If the second argument to **ERRORSET** is **INTERNAL**, the **ERRORSET** is ignored by **BREAKCHECK** during this search.) **BREAKCHECK** then counts the number of function calls between the error and the last **ERRORSET**, or the top of the stack. The number of function calls plus the number of calls to **EVAL** (already counted) is used as the "computation depth".

BREAKCHECK determines the computation time by subtracting the value of the variable **HELPCLOCK** from the value of **(CLOCK 2)**, the number of milliseconds of compute time (see page 12.15). **HELPCLOCK** is rebound to the current value of **(CLOCK 2)** for each computation typed in to **LISPX** or to a break. The time criterion for breaking can be suppressed by setting **HELPTIME** to **NIL** (or a very big number), or by setting **HELPCLOCK** to **NIL**. Note that setting **HELPCLOCK** to **NIL** will not have any effect beyond the current computation, because **HELPCLOCK** is rebound for each computation typed in to **LISPX** and **BREAK**.

The user can suppress all error breaks by setting the top level binding of the variable **HELPFLAG** to **NIL** using **SETTOPVAL** (**HELPFLAG** is bound as a local variable in **LISPX**, and reset to the global value of **HELPFLAG** on every **LISPX** line, so just **SETQ**ing it will not work.) If **HELPFLAG = T** (the initial value), the decision whether to cause an error or break is decided based on the computation time and the computation depth, as described above. Finally, if **HELPFLAG = BREAK!**, a break will always occur following an error.

14.5 Break Window Variables

The appearance and use of break windows is controlled by the following variables:

(WBREAK *ONFLG*) [Function]

If *ONFLG* is non-**NIL**, break windows and trace windows are enabled. If *ONFLG* is **NIL**, break windows are disabled (break windows do not appear, but the executive prompt is changed to ":" to indicate that the system is in a break). **WBREAK** returns **T** if break windows are currently enabled; **NIL** otherwise.

MaxBkMenuWidth [Variable]

MaxBkMenuHeight [Variable]

The variables **MaxBkMenuWidth** (default 125) and **MaxBkMenuHeight** (default 300) control the maximum size of the backtrace menu. If this menu is too small to contain all of the frames in the backtrace, it is made scrollable in both vertical and horizontal directions.

AUTOBACKTRACEFLG [Variable]

This variable controls when and what kind of backtrace menu is automatically brought up. The value of **AUTOBACKTRACEFLG** can be one of the following:

NIL The backtrace menu is not automatically brought up (the default).

T On error breaks the **BT** menu is brought up.

BT! On error breaks the **BT!** menu is brought up.

ALWAYS The **BT** menu is brought up on both error breaks and user breaks (calls to functions broken by **BREAK**).

ALWAYS! On both error breaks and user breaks the **BT!** menu is brought up.

BACKTRACEFONT [Variable]

The backtrace menu is printed in the font **BACKTRACEFONT**.

CLOSEBREAKWINDOWFLG [Variable]

The system normally closes break windows after the break is exited. If **CLOSEBREAKWINDOWFLG** is **NIL**, break windows will not be closed on exit. Note that in this case, the user must close all break windows.

BREAKREGIONSPEC [Variable]

Break windows are positioned near the tty window of the broken process, as determined by the variable **BREAKREGIONSPEC**. The value of this variable is a region (page 27.1) whose **LEFT** and **BOTTOM** fields are an offset from the **LEFT** and **BOTTOM** of the tty window. The **WIDTH** and **HEIGHT** fields of **BREAKREGIONSPEC** determine the size of the break window.

TRACEWINDOW [Variable]

The trace window, **TRACEWINDOW**, is used for tracing functions. It is brought up when the first tracing occurs and stays up until the user closes it. **TRACEWINDOW** can be set to a particular window to cause the tracing formation to print there.

TRACEREGION [Variable]

The trace window is first created in the region **TRACEREGION**.

14.6 Creating Breaks with BREAK1

The basic function of the break package is **BREAK1**, which creates a break. A break appears to be a regular executive, with the prompt ":", but **BREAK1** also detects and interpretes break commands (page 14.5).

(BREAK1 *BRKEXP BRKWHEN BRKFN BRKCOMS BRKTYPE ERRORN***)** [NLambda Function]

If *BRKWHEN* (evaluated) is non-**NIL**, a break occurs and commands are then taken from *BRKCOMS* or the terminal and interpreted. All inputs not recognized by **BREAK1** are simply passed on to the programmer's assistant.

If *BRKWHEN* is **NIL**, *BRKEXP* is evaluated and returned as the value of **BREAK1**, without causing a break.

When a break occurs, if *ERRORN* is a list whose **CAR** is a number, **ERRORMESS** (page 14.20) is called to print an identifying message. If *ERRORN* is a list whose **CAR** is not a number, **ERRORMESS1** (page 14.21) is called. Otherwise, no preliminary message is printed. Following this, the message (*BRKFN* **broken**) is printed.

Since **BREAK1** itself calls functions, when one of these is broken, an infinite loop would occur. **BREAK1** detects this situation, and prints **Break within a break on** *FN*, and then simply calls the function without going into a break.

The commands **GO**, **!GO**, **OK**, **!OK**, **RETURN** and ↑ are the only ways to leave **BREAK1**. The command **EVAL** causes *BRKEXP* to be evaluated, and saves the value on the variable **!VALUE**. Other commands can be defined for **BREAK1** via **BREAKMACROS** (below).

BRKTYPE is **NIL** for user breaks, **INTERRUPT** for control-H breaks, and **ERRORX** for error breaks. For breaks when *BRKTYPE* is not **NIL**, **BREAK1** will clear and save the input buffer. If the break

returns a value (i.e., is not aborted via ↑ or control-D) the input buffer will be restored.

The fourth argument to **BREAK1** is *BRKCOMS*, a list of break commands that **BREAK1** interprets and executes as though they were keyboard input. One can think of *BRKCOMS* as another input file which always has priority over the keyboard. Whenever *BRKCOMS* = **NIL**, **BREAK1** reads its next command from the keyboard. Whenever *BRKCOMS* is not **NIL**, **BREAK1** takes **(CAR BRKCOMS)** as its next command and sets *BRKCOMS* to **(CDR BRKCOMS)**. For example, suppose the user wished to see the value of the variable **X** *after* a function was evaluated. He could set up a break with *BRKCOMS* = **(EVAL (PRINT X) OK)**, which would have the desired effect. Note that if *BRKCOMS* is not **NIL**, the value of a break command is not printed. If you desire to see a value, you must print it yourself, as in the above example. The function **TRACE** (page 15.5) uses *BRKCOMS*: it sets up a break with two commands; the first one prints the arguments of the function, or whatever the user specifies, and the second is the command **GO**, which causes the function to be evaluated and its value printed.

Note: If an error occurs while interpreting the *BRKCOMS* commands, *BRKCOMS* is set to **NIL**, and a full interactive break occurs.

The break package has a facility for redirecting ouput to a file. All output resulting from *BRKCOMS* will be output to the value of the variable **BRKFILE**, which should be the name of an open file. Output due to user typein is not affected, and will always go to the terminal. **BRKFILE** is initially **T**.

BREAKMACROS [Variable]

BREAKMACROS is a list of the form (($NAME_1$ COM_{11} ... COM_{1n}) ($NAME_2$ COM_{21} ... COM_{2n}) ...). Whenever an atomic command is given to **BREAK1**, it first searches the list **BREAKMACROS** for the command. If the command is equal to $NAME_i$, **BREAK1** simply appends the corresponding commands to the front of *BRKCOMS*, and goes on. If the command is not found on **BREAKMACROS**, **BREAK1** then checks to see if it is one of the built in commands, and finally, treats it as a function or variable as before.

If the command is not the name of a defined function, bound variable, or **LISPX** command, **BREAK1** will attempt spelling correction using **BREAKCOMSLST** as a spelling list. If spelling correction is unsuccessful, **BREAK1** will go ahead and call **LISPX** anyway, since the atom may also be a misspelled history command.

For example, the command **ARGS** could be defined by including on **BREAKMACROS** the form:

(ARGS (PRINT (VARIABLES LASTPOS T)))

(BREAKREAD *TYPE***)** [Function]

Useful within **BREAKMACROS** for reading arguments. If **BRKCOMS** is non-**NIL** (the command in which the call to **BREAKREAD** appears was not typed in), returns the next break command from **BRKCOMS**, and sets **BRKCOMS** to **(CDR BRKCOMS)**.

If **BRKCOMS** is **NIL** (the command was typed in), then **BREAKREAD** returns either the rest of the commands on the line as a list (if *TYPE* = **LINE**) or just the next command on the line (if *TYPE* is not **LINE**).

For example, the **BT** command is defined as **(BAKTRACE LASTPOS NIL (BREAKREAD 'LINE) 0 T)**. Thus, if the user types **BT**, the third argument to **BAKTRACE** will be **NIL**. If the user types **BT SUBRP**, the third argument will be **(SUBRP)**.

BREAKRESETFORMS [Variable]

If the user is developing programs that change the way a user and Interlisp normally interact (e.g., change or disable the interrupt or line-editing characters, turn off echoing, etc.), debugging them by breaking or tracing may be difficult, because Interlisp might be in a "funny" state at the time of the break. **BREAKRESETFORMS** is designed to solve this problem. The user puts on **BREAKRESETFORMS** expressions suitable for use in conjunction with **RESETFORM** or **RESETSAVE** (page 14.24). When a break occurs, **BREAK1** evaluates each expression on **BREAKRESETFORMS** *before* any interaction with the terminal, and saves the values. When the break expression is evaluated via an **EVAL**, **OK**, or **GO**, **BREAK1** first restores the state of the system with respect to the various expressions on **BREAKRESETFORMS**. When control returns to **BREAK1**, the expressions on **BREAKRESETFORMS** are *again* evaluated, and their values saved. When the break is exited with an **OK**, **GO**, **RETURN**, or ↑ command, by typing control-D, or by a **RETFROM** or **RETEVAL** typed in by the user, **BREAK1** again restores state. Thus the net effect is to make the break invisible with respect to the user's programs, but nevertheless allow the user to interact in the break in the normal fashion.

Note: All user type-in is scanned in order to make the operations undoable as described on page 13.27. At this point, **RETFROM**s and **RETEVAL**s are also noticed. However, if the user types in an expression which calls a function that then does a **RETFROM**, this **RETFROM** will not be noticed, and the effects of **BREAKRESETFORMS** will *not* be reversed.

As mentioned earlier, **BREAK1** detects "Break within a break" situations, and avoids infinite loops. If the loop occurs because of an error, **BREAK1** simply rebinds **BREAKRESETFORMS** to **NIL**, and calls **HELP**. This situation most frequently occurs when there is a bug in a function called by **BREAKRESETFORMS**.

Note: **SETQ** expressions can also be included on **BREAKRESETFORMS** for saving and restoring system parameters, e.g. **(SETQ LISPXHISTORY NIL)**, **(SETQ DWIMFLG NIL)**, etc. These are handled specially by **BREAK1** in that the current value of the variable is saved before the **SETQ** is executed, and upon restoration, the variable is set back to this value.

14.7 Signalling Errors

(ERRORX *ERXM***)** [Function]

The entry to the error routines. If *ERXM* = **NIL**, **(ERRORN)** is used to determine the error-message. Otherwise, **(SETERRORN (CAR** *ERXM***) (CADR** *ERXM***))** is performed, "setting" the error number and argument. Thus following either **(ERRORX '(10 T))** or **(PLUS T)**, **(ERRORN)** is **(10 T)**. **ERRORX** calls **BREAKCHECK**, and either induces a break or prints the message and unwinds to the last **ERRORSET** (page 14.13). Note that **ERRORX** can be called by any program to intentionally induce an error of any type. However, for most applications, the function **ERROR** will be more useful.

(ERROR *MESS1 MESS2 NOBREAK***)** [Function]

Prints *MESS1* (using **PRIN1**), followed by a space if *MESS1* is an atom, otherwise a carriage return. Then *MESS2* is printed (using **PRIN1** if *MESS2* is a string, otherwise **PRINT**). For example, **(ERROR "NON-NUMERIC ARG" T)** prints

NON-NUMERIC ARG
T

and **(ERROR 'FOO "NOT A FUNCTION")** prints **FOO NOT A FUNCTION**. If both *MESS1* and *MESS2* are **NIL**, the message printed is simply **ERROR**.

If *NOBREAK* = **T**, **ERROR** prints its message and then c; **ERROR!** (below). Otherwise it calls **(ERRORX '(17 (***MESS1* **.** *MESS2***)))**, i.e., generates error number 17, in which case the decision as to whether or not to break, and whether or not to print a message, is handled as per any other error.

If the value of **HELPFLAG** (page 14.14) is **BREAK!**, a break will always occur, irregardless of the value of *NOBREAK*.

Note: If **ERROR** causes a break, the "break expression" (page 14.5) will be **(ERROR MESS1 MESS2 NOBREAK)**. Using the **GO**, **OK**, or **EVAL** break commands (page 14.5) will simply call **ERROR** again. It is sometimes helpful to design programs that call **ERROR** such that if the call to **ERROR** returns (as the result of using the **RETURN** break command), the operation is tried again. This allows the use to fix any problems within the break environment, and try to continue the operation.

(HELP *MESS1 MESS2 BRKTYPE***)** [Function]

Prints *MESS1* and *MESS2* similar to **ERROR**, and then calls **BREAK1** passing *BRKTYPE* as the **BRKTYPE** argument. If both *MESS1* and *MESS2* are **NIL**, **Help!** is used for the message. **HELP** is a convenient way to program a default condition, or to terminate some portion of a program which the computation is theoretically never supposed to reach.

(SHOULDNT *MESS***)** [Function]

Useful in those situations when a program detects a condition that should never occur. Calls **HELP** with the message arguments *MESS* and **"Shouldn't happen!"** and a **BRKTYPE** argument of **'ERRORX**.

(ERROR!) [Function]

Programmable control-E; immediately returns from last **ERRORSET** or resets.

(RESET) [Function]

Programmable control-D; immediately returns to the top level.

(ERRORN) [Function]

Returns information about the last error in the form (*NUM EXP*) where *NUM* is the error number (page 14.27) and *EXP* is the expression which was printed out after the error message. For example, following **(PLUS T)**, **(ERRORN)** would return **(10 T)**.

(SETERRORN *NUM MESS***)** [Function]

Sets the value returned by **ERRORN** to (*NUM MESS*).

(ERRORMESS *U***)** [Function]

Prints message corresponding to an **ERRORN** that yielded *U*. For example, **(ERRORMESS '(10 T))** would print

NON-NUMERIC ARG
T

(ERRORMESS1 *MESS1 MESS2 MESS3***)** [Function]

Prints the message corresponding to a **HELP** or **ERROR** break.

(ERRORSTRING *X***)** [Function]

Returns as a new string the message corresponding to error number *X*, e.g., **(ERRORSTRING 10) = "NON-NUMERIC ARG"**.

14.8 Catching Errors

All error conditions are not caused by program bugs. For some programs, it is reasonable for some errors to occur (such as file not found errors) and it is possible for the program to handle the error itself. There are a number of functions that allow a program to "catch" errors, rather than abort the computation or cause a break.

(ERRORSET *FORM FLAG* **—)** [Function]

Performs **(EVAL** *FORM***)**. If no error occurs in the evaluation of *FORM*, the value of **ERRORSET** is a list containing one element, the value of **(EVAL** *FORM***)**. If an error did occur, the value of **ERRORSET** is **NIL**.

ERRORSET is a lambda function, so its arguments are evaluated *before* it is entered, i.e., **(ERRORSET X)** means **EVAL** is called with the *value* of **X**. In most cases, **ERSETQ** and **NLSETQ** (below) are more useful.

Performance Note: When a call to **ERSETQ** or **NLSETQ** is compiled, the form to be evaluated is compiled as a separate function. However, compiling a call to **ERRORSET** does not compile *FORM*. Therefore, if *FORM* performs a lengthy computation, using **ERSETQ** or **NLSETQ** can be much more efficient than using **ERRORSET**.

The argument *FLAG* controls the printing of error messages if an error occurs. Note that if a *break* occurs below an **ERRORSET**, the message is printed regardless of the value of *FLAG*.

If *FLAG* = **T**, the error message is printed; if *FLAG* = **NIL**, the error message is not printed (unless **NLSETQGAG** is **NIL**, see below).

If *FLAG* = **INTERNAL**, this **ERRORSET** is ignored for the purpose of deciding whether or not to break or print a message (see page 14.13). However, the **ERRORSET** is in effect for the purpose of flow of control, i.e., if an error occurs, this **ERRORSET** returns **NIL**.

If *FLAG* = **NOBREAK**, no break will occur, even if the time criterion for breaking is met (see page 14.13). Note that *FLAG* = **NOBREAK** will *not* prevent a break from occurring if the

error occurs more than **HELPDEPTH** function calls below the errorset, since **BREAKCHECK** will stop searching before it reaches the **ERRORSET**. To guarantee that no break occurs, the user would also either have to reset **HELPDEPTH** or **HELPFLAG** (page 14.13).

(ERSETQ *FORM***)** [NLambda Function]

Performs **(ERRORSET '***FORM* **T)**, evaluating *FORM* and printing error messages.

(NLSETQ *FORM***)** [NLambda Function]

Performs **(ERRORSET '***FORM* **NIL)**, evaluating *FORM* without printing error messages.

NLSETQGAG [Variable]

If **NLSETQGAG** is **NIL**, error messages will print, regardless of the *FLAG* argument of **ERRORSET**. **NLSETQGAG** effectively changes all **NLSETQ**s to **ERSETQ**s. **NLSETQGAG** is initially **T**.

Occasionally the user may want to treat certain types of errors differently from others, e.g., always break, never break, or perhaps take some corrective action. This can be accomplished via **ERRORTYPELST**:

ERRORTYPELST [Variable]

ERRORTYPELST is a list of elements, where each element is of the form (*NUM* $FORM_1$ ... $FORM_N$). *NUM* is one of the error numbers (page 14.27). During an error, after **BREAKCHECK** has been completed, but before any other action is taken, **ERRORTYPELST** is searched for an element with the same error number as that causing the error. If one is found, the corresponding forms are evaluated, and if the last one produces a non-**NIL** value, this value is substituted for the offender, and the function causing the error is reentered.

Note: **ERRORTYPELST** is accessed as a special variable (see page 18.5), so it can be rebound in a function argument list of **PROG** form to catch errors in a dynamic context.

Within **ERRORTYPELST** entries, the following variables may be useful:

ERRORMESS [Variable]

CAR is the error number, **CADR** the "offender", e.g., **(10 NIL)** corresponds to a **NON-NUMERIC ARG NIL** error.

ERRORPOS [Variable]

Stack pointer to the function in which the error occurred, e.g., **(STKNAME ERRORPOS)** might be **IPLUS**, **RPLACA**, **INFILE**, etc.

Note: If the error is going to be handled by a **RETFROM**, **RETTO**, or a **RETEVAL** in the **ERRORTYPELST** entry, it probably is a good idea to first release the stack pointer **ERRORPOS**, e.g. by performing **(RELSTK ERRORPOS)**.

BREAKCHK [Variable]

Value of **BREAKCHECK**, i.e., **T** means a break will occur, **NIL** means one will not. This may be reset within the **ERRORTYPELST** entry.

PRINTMSG [Variable]

If **T**, means print error message, if **NIL**, don't print error message, i.e., corresponds to second argument to **ERRORSET**. The user can force or suppress the printing of error messages for various error types by including on **ERRORTYPELST** an expression which explicitly sets **PRINTMSG**.

For example, putting

```
[10 (AND (NULL (CADR ERRORMESS))
    (SELECTQ (STKNAME ERRORPOS)
        ((IPLUS ADD1 SUB1) 0)
        (ITIMES 1)
        (PROGN (SETQ BREAKCHK T) NIL]
```

on **ERRORTYPELST** would specify that whenever a **NON-NUMERIC ARG - NIL** error occurred, and the function in question was **IPLUS**, **ADD1**, or **SUB1**, **0** should be used for the **NIL**. If the function was **ITIMES**, **1** should be used. Otherwise, always break. Note that the latter case is achieved not by the value returned, but by the *effect* of the evaluation, i.e., setting **BREAKCHK** to **T**. Similarly, **(16 (SETQ BREAKCHK NIL))** would prevent **END OF FILE** errors from ever breaking.

ERRORTYPELST is initially **((23 (SPELLFILE (CADR ERRORMESS) NIL NOFILESPELLFLG)))**, which causes **SPELLFILE** to be called in case of a **FILE NOT FOUND** error (see page 24.32). If **SPELLFILE** is successful, the operation will be reexecuted with the new (corrected) file name.

14.9 Changing and Restoring System State

In Interlisp, a computation can be interrupted/aborted at any point due to an error, or more forcefully, because a control-D was typed, causing return to the top level. This situation creates problems for programs that need to perform a computation with the system in a "different state", e.g., different radix, input file, readtable, etc. but want to be able to restore the state when the computation has completed. While program errors and control-E can be "caught" by **ERRORSET**s, control-D is not. Note that the program could redefine control-D as a user interrupt (page 30.3), check for it, reenable it, and call **RESET** or something similar. Thus the system may be left in its changed state as a result of the computation being aborted. The following functions address this problem.

Note that these functions cannot handle the situation where their environment is exited via anything other than a normal return, an error, or a reset. Therefore, a **RETEVAL**, **RETFROM**, **RESUME**, etc., will never be seen.

(RESETLST *FORM*$_1$ ... *FORM*$_N$**)** [NLambda NoSpread Function]

RESETLST evaluates its arguments in order, after setting up an **ERRORSET** so that any reset operations performed by **RESETSAVE** (see below) are restored when the forms have been evaluated (or an error occurs, or a control-D is typed). If no error occurs, the value of **RESETLST** is the value of *FORM*$_N$, otherwise **RESETLST** generates an error (after performing the necessary restorations).

RESETLST compiles open.

(RESETSAVE *X Y***)** [NLambda NoSpread Function]

RESETSAVE is used within a call to **RESETLST** to change the system state by calling a function or setting a variable, while specifying how to restore the original system state when the **RESETLST** is exited (normally, or with an error or control-D).

If *X* is atomic, resets the top level value of *X* to the value of *Y*. For example, **(RESETSAVE LISPXHISTORY EDITHISTORY)** resets the value of **LISPXHISTORY** to the value of **EDITHISTORY**, and provides for the original value of **LISPXHISTORY** to be restored when the **RESETLST** completes operation, (or an error occurs, or a control-D is typed).

Note: If the variable is simply rebound, the **RESETSAVE** will *not* affect the most recent binding but will change only the top level value, and therefore probably not have the intended effect.

If *X* is not atomic, it is a form that is evaluated. If *Y* is **NIL**, *X* must return as its value its "former state", so that the effect of evaluating the form can be reversed, and the system state can be restored, by applying **CAR** of *X* to the value of *X*. For example,

(RESETSAVE (RADIX 8)) performs **(RADIX 8)**, and provides for **RADIX** to be reset to its original value when the **RESETLST** completes by applying **RADIX** to the value returned by **(RADIX 8)**.

In the special case that **CAR** of *X* is **SETQ**, the **SETQ** is transparent for the purposes of **RESETSAVE**, i.e. the user could also have written **(RESETSAVE (SETQ X (RADIX 8)))**, and restoration would be performed by applying **RADIX**, not **SETQ**, to the previous value of **RADIX**.

If *Y* is not **NIL**, it is evaluated (before *X*), and its *value* is used as the restoring expression. This is useful for functions which do not return their "previous setting". For example,

[RESETSAVE (SETBRK ...) (LIST 'SETBRK (GETBRK]

will restore the break characters by applying **SETBRK** to the value returned by **(GETBRK)**, which was computed before the **(SETBRK ...)** expression was evaluated. Note that the restoration expression is "evaluated" by *applying* its **CAR** to its **CDR**. This insures that the "arguments" in the **CDR** are not evaluated again.

If *X* is **NIL**, *Y* is still treated as a restoration expression. Therefore,

(RESETSAVE NIL (LIST 'CLOSEF FILE))

will cause **FILE** to be closed when the **RESETLST** that the **RESETSAVE** is under completes (or an error occurs or a control-D is typed).

Note: **RESETSAVE** can be called when *not* under a **RESETLST**. In this case, the restoration will be performed at the next **RESET**, i.e., control-D or call to **RESET**. In other words, there is an "implicit" **RESETLST** at the top-level executive.

RESETSAVE compiles open. Its value is not a "useful" quantity.

(RESETVAR *VAR NEWVALUE FORM***)** [NLambda Function]

Simplified form of **RESETLST** and **RESETSAVE** for resetting and restoring global variables. Equivalent to **(RESETLST (RESETSAVE** *VAR NEWVALUE***)** *FORM***)**. For example, **(RESETVAR LISPXHISTORY EDITHISTORY (FOO))** resets **LISPXHISTORY** to the value of **EDITHISTORY** while evaluating **(FOO)**. **RESETVAR** compiles open. If no error occurs, its value is the value of *FORM*.

(RESETVARS *VARSLST* E_1 E_2 ... E_N**)** [NLambda NoSpread Function]

Similar to **PROG**, except that the variables in *VARSLST* are global variables. In a deep bound system (such as Interlisp-D), each variable is "rebound" using **RESETSAVE**.

In a shallow bound system (such as Interlisp-10) **RESETVARS** and **PROG** are identical, except that the compiler insures that

variables bound in a **RESETVARS** are declared as **SPECVARS** (see page 18.5).

RESETVARS, like **GETATOMVAL** and **SETATOMVAL** (page 2.4), is provided to permit compatibility (i.e. transportablility) between a shallow bound and deep bound system with respect to conceptually global variables.

Note: Like **PROG**, **RESETVARS** returns **NIL** unless a **RETURN** statement is executed.

(RESETFORM *RESETFORM FORM$_1$ FORM$_2$ ... FORM$_N$)* [NLambda NoSpread Function]

Simplified form of **RESETLST** and **RESETSAVE** for resetting a system state when the corresponding function returns as its value the "previous setting." Equivalent to **(RESETLST (RESETSAVE** *RESETFORM*) *FORM$_1$ FORM$_2$ ... FORM$_N$*). For example, **(RESETFORM (RADIX 8) (FOO))**. **RESETFORM** compiles open. If no error occurs, it returns the value returned by *FORM$_N$*.

For some applications, the restoration operation must be different depending on whether the computation completed successfully or was aborted somehow (e.g., by an error or by typing control-D). To facilitate this, while the restoration operation is being performed, the value of **RESETSTATE** will be bound to **NIL**, **ERROR**, **RESET**, or **HARDRESET** depending on whether the exit was normal, due to an error, due to a reset (i.e., control-D), or due to call to **HARDRESET** (page 23.1). As an example of the use of **RESETSTATE**,

```
(RESETLST
  (RESETSAVE (INFILE X)
    (LIST '[LAMBDA (FL)
          (COND ( (EQ RESETSTATE 'RESET)
              (CLOSEF FL)
              (DELFILE FL]
       X))
  FORMS)
```

will cause **X** to be closed and deleted only if a control-D was typed during the execution of *FORMS*.

When specifying complicated restoring expressions, it is often necessary to use the old value of the saving expression. For example, the following expression will set the primary input file (to **FL**) and execute some forms, but reset the primary input file only if an error or control-D occurs.

```
(RESETLST
  (SETQ TEM (INPUT FL))
  (RESETSAVE NIL
    (LIST '(LAMBDA (X) (AND RESETSTATE (INPUT X)))
      TEM))
```

FORMS)

So that you will not have to explicitely save the old value, the variable **OLDVALUE** is bound at the time the restoring operation is performed to the value of the saving expression. Using this, the previous example could be recoded as:

```
(RESETLST
  (RESETSAVE (INPUT FL)
    '(AND RESETSTATE (INPUT OLDVALUE)))
  FORMS)
```

As mentioned earlier, restoring is performed by applying **CAR** of the restoring expression to the **CDR**, so **RESETSTATE** and **(INPUT OLDVALUE)** will not be evaluated by the **APPLY**. This particular example works because **AND** is an nlambda function that explicitly evaluates its arguments, so **APPLY**ing **AND** to **(RESETSTATE (INPUT OLDVALUE))** is the same as **EVAL**ing **(AND RESETSTATE (INPUT OLDVALUE))**. **PROGN** also has this property, so you can use a lambda function as a restoring form by enclosing it within a **PROGN**.

The function **RESETUNDO** (page 13.30) can be used in conjunction with **RESETLST** and **RESETSAVE** to provide a way of specifying that the system be restored to its prior state by *undoing* the side effects of the computations performed under the **RESETLST**.

14.10 Error List

There are currently fifty-plus types of errors in the Interlisp system. Some of these errors are implementation dependent, i.e., appear in Interlisp-D but may not appear in other Interlisp systems. The error number is set internally by the code that detects the error before it calls the error handling functions. It is also the value returned by **ERRORN** if called subsequent to that type of error, and is used by **ERRORMESS** for printing the error message.

Most errors will print the offending expression following the message, e.g., **NON-NUMERIC ARG NIL** is very common. Error number 18 (control-B) always causes a break (unless **HELPFLAG** is **NIL**). All other errors cause breaks if **BREAKCHECK** returns **T** (see page 14.13).

The errors are listed below by error number:

0 **SYSTEM ERROR**

Low-level Interlisp system error. It is quite possible that random programs or data structures might have already been smashed.

Unless he is sure he knows what the problem is, the user is best advised to save any important information, and reload the Interlisp system as soon as possible.

1 No longer used.

2 **STACK OVERFLOW**

Occurs when computation is too deep, either with respect to number of function calls, or number of variable bindings. Usually because of a non-terminating recursive computation, i.e., a bug.

3 **ILLEGAL RETURN**

Call to **RETURN** when not inside of an interpreted **PROG**.

4 **ARG NOT LIST**

RPLACA called on a non-list.

5 **HARD DISK ERROR**

An error with the local disk drive.

6 **ATTEMPT TO SET NIL**

Via **SET** or **SETQ**

7 **ATTEMPT TO RPLAC NIL**

Attempt either to **RPLACA** or to **RPLACD NIL** with something other than **NIL**.

8 **UNDEFINED OR ILLEGAL GO**

GO when not inside of a **PROG**, or **GO** to nonexistent label.

9 **FILE WON'T OPEN**

From **OPENSTREAM** (page 24.2).

10 **NON-NUMERIC ARG**

A numeric function e.g., **PLUS**, **TIMES**, **GREATERP**, expected a number.

11 **ATOM TOO LONG**

Attempted to create a litatom (via **PACK**, or typing one in, or reading from a file) with too many characters. In Interlisp-D, the maximum number of characters in a litatom is 255.

12 **ATOM HASH TABLE FULL**

No room for any more (new) atoms.

13 **FILE NOT OPEN**

From an I/O function, e.g., **READ**, **PRINT**, **CLOSEF**.

14 **ARG NOT LITATOM**

SETQ, **PUTPROP**, **GETTOPVAL**, etc., given a non-atomic arg.

15 **TOO MANY FILES OPEN**

16 **END OF FILE**

From an input function, e.g., **READ**, **READC**, **RATOM**. After the error occurs, the file will still be left open.

Note: It is possible to use an **ERRORTYPELST** entry (page 14.22) to return a character as the value of the call to **ERRORX**, and the program will continue, e.g. returning "]" may be used to complete a read operation.

17 **ERROR**

Call to **ERROR** (page 14.19).

18 **BREAK**

Control-B was typed.

19 **ILLEGAL STACK ARG**

A stack function expected a stack position and was given something else. This might occur if the arguments to a stack function are reversed. Also occurs if user specified a stack position with a function name, and that function was not found on the stack. See page 11.1.

20 **FAULT IN EVAL**

Artifact of bootstrap process. Never occurs after **FAULTEVAL** is defined.

21 **ARRAYS FULL**

System will first initiate a garbage collection of array space, and if no array space is reclaimed, will then generate this error.

22 **FILE SYSTEM RESOURCES EXCEEDED**

Includes no more disk space, disk quota exceeded, directory full,etc.

23 **FILE NOT FOUND**

File name does not correspond to a file in the corresponding directory. Can also occur if file name is ambiguous.

Interlisp is initialized with an entry on **ERRORTYPELST** (page 14.22) to call **SPELLFILE** for this error. **SPELLFILE** will search alternate directories or perform spelling correction on the connected directory. If **SPELLFILE** fails, then the user will see this error.

24 **BAD SYSOUT FILE**

Date does not agree with date of **MAKESYS**, or file is not a sysout file at all (see page 12.8).

25 **UNUSUAL CDR ARG LIST**

A form ends in a non-list other than **NIL**, e.g., **(CONS T . 3)**.

26 **HASH TABLE FULL**

See hash array functions, page 6.1.

27 **ILLEGAL ARG**

Catch-all error. Currently used by **PUTD, EVALA, ARG, FUNARG,** etc.

28 **ARG NOT ARRAY**

ELT or **SETA** given an argument that is not a legal array (see page 5.1).

29 **ILLEGAL OR IMPOSSIBLE BLOCK**

(Interlisp-10) Not enough free blocks available (from **GETBLK** or **RELBLK**).

30 **STACK PTR HAS BEEN RELEASED**

A released stack pointer was supplied as a stack descriptor for a purpose other than as a stack pointer to be re-used (see page 11.10).

31 **STORAGE FULL**

Following a garbage collection, if not enough words have been collected, and there is no un-allocated space left in the system, this error is generated.

32 **ATTEMPT TO USE ITEM OF INCORRECT TYPE**

Before a field of a user data type is changed, the type of the item is first checked to be sure that it is the expected type. If not, this error is generated (see page 8.20).

33 **ILLEGAL DATA TYPE NUMBER**

The argument is not a valid user data type number (see page 8.20).

34 **DATA TYPES FULL**

All available user data types have been allocated (see page 8.20).

35 **ATTEMPT TO BIND NIL OR T**

In a **PROG** or **LAMBDA** expression.

36 **TOO MANY USER INTERRUPT CHARACTERS**

Attempt to enable a user interrupt character when all user channels are currently enabled (see page 30.3).

37 **READ-MACRO CONTEXT ERROR**

(Interlisp-10 only) Occurs when a **READ** is executed from within a read macro function and the next token is a) or a] (see page 25.39).

38 **ILLEGAL READTABLE**

The argument was expected to be a valid read table (see page 25.33).

39 **ILLEGAL TERMINAL TABLE**

The argument was expected to be a valid terminal table (see page 30.4).

40 **SWAPBLOCK TOO BIG FOR BUFFER**

(Interlisp-10) An attempt was made to swap in a function/array which is too large for the swapping buffer.

41 **PROTECTION VIOLATION**

Attempt to open a file that user does not have access to. Also reference to unassigned device.

42 **BAD FILE NAME**

Illegal character in file specification, illegal syntax, e.g. two ;'s etc.

43 **USER BREAK**

Error corresponding to user interrupt character. See page 30.3.

44 **UNBOUND ATOM**

This occurs when a variable (litatom) was used which had neither a stack binding (wasn't an argument to a function nor a **PROG** variable) nor a top level value. The "culprit" (**(CADR ERRORMESS)**) is the litatom. Note that if DWIM corrects the error, no error occurs and the error number is not set. However, if an error is going to occur, whether or not it will cause a break, the error number will be set.

45 **UNDEFINED CAR OF FORM**

Undefined function error. This occurs when a form is evaluated whose function position (**CAR**) does not have a definition as a function. Culprit is the form.

46 **UNDEFINED FUNCTION**

This error is generated if **APPLY** is given an undefined function. Culprit is **(LIST** *FN ARGS***)**

47 **CONTROL-E**

The user typed control-E.

48 **FLOATING UNDERFLOW**

Underflow during floating-point operation.

49 **FLOATING OVERFLOW**

Overflow during floating-point operation.

50 **OVERFLOW**

Overflow during integer operation.

51 **ARG NOT HARRAY**

Hash array operations given an argument that is not a hash array.

52 **TOO MANY ARGUMENTS**

Too many arguments given to a lambda-spread, lambda-nospread, or nlambda-spread function.

Note that Interlisp-D does not cause an error if more arguments are passed to a function than it is defined with. This argument occurs when more individual arguments are passed to a function than Interlisp-D can store on the stack at once. The limit is currently 80 arguments.

In addition, many system functions, e.g., **DEFINE**, **ARGLIST**, **ADVISE**, **LOG**, **EXPT**, etc, also generate errors with appropriate messages by calling **ERROR** (see page 14.19) which causes error number 17.

TABLE OF CONTENTS

[This page intentionally left blank]

15. BREAKING, TRACING, AND ADVISING

It is frequently useful to be able to modify the behavior of a function without actually editing its definition. Interlisp provides several different facilities for doing this. By "breaking" a function, the user can cause breaks to occur at various times in the running of an incomplete program, so that the program state can be inspected. "Tracing" a function causes information to be printed every time the function is entered or exited. These are very useful debugging tools.

"Advising" is a facility for specifying longer-term function modifications. Even system functions can be changed through advising.

15.1 Breaking Functions and Debugging

Debugging a collection of LISP functions involves isolating problems within particular functions and/or determining when and where incorrect data are being generated and transmitted. In the Interlisp system, there are three facilities which allow the user to (temporarily) modify selected function definitions so that he can follow the flow of control in his programs, and obtain this debugging information. All three redefine functions in terms of a system function, **BREAK1** (see page 14.16).

BREAK (page 15.5) modifies the definition of a function *FN*, so that whenever *FN* is called and a break condition (defined by the user) is satisfied, a function break occurs. The user can then interrogate the state of the machine, perform any computation, and continue or return from the call.

TRACE (page 15.5) modifies a definition of a function *FN* so that whenever *FN* is called, its arguments (or some other values specified by the user) are printed. When the value of *FN* is computed it is printed also. **TRACE** is a special case of **BREAK**.

BREAKIN (page 15.6) allows the user to insert a breakpoint *inside* an expression defining a function. When the breakpoint is reached and if a break condition (defined by the user) is satisfied, a temporary halt occurs and the user can again investigate the state of the computation.

The following two examples illustrate these facilities. In the first example, the user traces the function **FACTORIAL**. **TRACE** redefines **FACTORIAL** so that it print its arguments and value, and then goes on with the computation. When an error occurs on the fifth recursion, a full interactive break occurs. The situation is then the same as though the user had originally performed **(BREAK FACTORIAL)** instead of **(TRACE FACTORIAL)**, and the user can evaluate various Interlisp forms and direct the course of the computation. In this case, the user examines the variable **N**, and instructs **BREAK1** to return **1** as the value of this cell to **FACTORIAL**. The rest of the tracing proceeds without incident. The user would then presumably edit **FACTORIAL** to change **L** to **1**.

```
←PP FACTORIAL

(FACTORIAL
 [LAMBDA (N)
  (COND
    ((ZEROP N)
      L)
    (T (ITIMES N (FACTORIAL (SUB1 N])
FACTORIAL
←(TRACE FACTORIAL)
(FACTORIAL)
←(FACTORIAL 4)

FACTORIAL:
N = 4

  FACTORIAL:
  N = 3

    FACTORIAL:
    N = 2

      FACTORIAL:
      N = 1

        FACTORIAL:
        N = 0

UNBOUND ATOM
L
(FACTORIAL BROKEN)
:N
0
:RETURN 1
        FACTORIAL = 1
```

```
      FACTORIAL = 1
    FACTORIAL = 2
  FACTORIAL = 6
FACTORIAL = 24
24
←
```

In the second example, the user has constructed a non-recursive definition of **FACTORIAL**. He uses **BREAKIN** to insert a call to **BREAK1** just after the **PROG** label **LOOP**. This break is to occur only on the last two iterations, when **N** is less than **2**. When the break occurs, the user tries to look at the value of **N**, but mistakenly types **NN**. The break is maintained, however, and no damage is done. After examining **N** and **M** the user allows the computation to continue by typing **OK**. A second break occurs after the next iteration, this time with **N = 0**. When this break is released, the function **FACTORIAL** returns its value of **120**.

```
←PP FACTORIAL
(FACTORIAL
 [LAMBDA (N)
  (PROG ((M 1))
    LOOP (COND
           ((ZEROP N)
            (RETURN M)))
         (SETQ M (ITIMES M N))
         (SETQ N (SUB1 N))
         (GO LOOP])
FACTORIAL
←(BREAKIN FACTORIAL (AFTER LOOP) (ILESSP N 2]
SEARCHING...
FACTORIAL
←(FACTORIAL 5)

((FACTORIAL) BROKEN)
:NN
U.B.A.
NN
(FACTORIAL BROKEN AFTER LOOP)
:N
1
:M
120
:OK
(FACTORIAL)

((FACTORIAL) BROKEN)
:N
0
:OK
```

(FACTORIAL)
120
←

Note: **BREAK** and **TRACE** can also be used on CLISP words which appear as **CAR** of form, e.g. **FETCH**, **REPLACE**, **IF**, **FOR**, **DO**, etc., even though these are not implemented as functions. For conditional breaking, the user can refer to the entire expression via the variable **EXP**, e.g. **(BREAK (FOR (MEMB 'UNTIL EXP)))**.

(BREAK0 *FN WHEN COMS* — —**)** [Function]

Sets up a break on the function *FN*; returns *FN*. If *FN* is not defined, returns **(***FN* **NOT DEFINED)**.

The value of *WHEN*, if non-**NIL**, should be an expression that is evaluated whenever *FN* is entered. If the value of the expression is non-**NIL**, a break is entered, otherwise the function simply called and returns without causing a break. This provides the means of conditionally breaking a function.

The value of *COMS*, if non-**NIL**, should be a list of break commands, that are interpreted and executed if a break occurs. (See the *BRKCOMS* argument to **BREAK1**, page 14.17.)

BREAK0 sets up a break by doing the following: (1) it redefines *FN* as a call to **BREAK1** (page 14.16), passing an equivalent definition of *FN*, *WHEN*, *FN*, and *COMS* as the **BRKEXP**, *BRKWHEN*, *BRKFN*, and *BRKCOMS* arguments to **BREAK1**; (2) it defines a **GENSYM** (page 2.10) with the original definition of *FN*, and puts it on the property list of *FN* under the property **BROKEN**; (3) it puts the form **(BREAK0** *WHEN COMS***)** on the property list of *FN* under the property **BRKINFO** (for use in conjunction with **REBREAK**); and (4) it adds *FN* to the front of the list **BROKENFNS**.

If *FN* is non-atomic and of the form **(***FN1* **IN** *FN2***)**, **BREAK0** breaks every call to *FN1* from within *FN2*. This is useful for breaking on a function that is called from many places, but where one is only interested in the call from a specific function, e.g., **(RPLACA IN FOO)**, **(PRINT IN FIE)**, etc. It is similar to **BREAKIN** described below, but can be performed even when *FN2* is compiled or blockcompiled, whereas **BREAKIN** only works on interpreted functions. If *FN1* is not found in *FN2*, **BREAK0** returns the value **(***FN1* **NOT FOUND IN** *FN2***)**.

BREAK0 breaks one function *inside* another by first calling a function which changes the name of *FN1* wherever it appears inside of *FN2* to that of a new function, *FN1*-**IN**-*FN2*, which is initially given the same function definition as *FN1*. Then **BREAK0** proceeds to break on *FN1*-**IN**-*FN2* exactly as described above. In addition to breaking *FN1*-**IN**-*FN2* and adding *FN1*-**IN**-*FN2* to the list **BROKENFNS**, **BREAK0** adds *FN1* to the property value for the

property **NAMESCHANGED** on the property list of *FN2* and puts (*FN2* . *FN1*) on the property list of *FN1*-**IN**-*FN2* under the property **ALIAS**. This will enable **UNBREAK** to recognize what changes have been made and restore the function *FN2* to its original state.

If *FN* is nonatomic and not of the above form, **BREAK0** is called for each member of *FN* using the same values for *WHEN*, *COMS*, and *FILE*. This distributivity permits the user to specify complicated break conditions on several functions. For example,

```
(BREAK0 '(FOO1 ((PRINT PRIN1) IN (FOO2 FOO3)))
    '(NEQ X T)
    '(EVAL ? = (Y Z) OK) )
```

will break on **FOO1**, **PRINT-IN-FOO2**, **PRINT-IN-FOO3**, **PRIN1-IN-FOO2** and **PRIN1-IN-FOO3**.

If *FN* is non-atomic, the value of **BREAK0** is a list of the functions broken.

(BREAK *X***)** [NLambda NoSpread Function]

For each atomic argument, it performs **(BREAK0** *ATOM* **T)**. For each list, it performs **(APPLY 'BREAK0** *LIST***)**. For example, **(BREAK FOO1 (FOO2 (GREATERP N 5) (EVAL)))** is equivalent to **(BREAK0 'FOO1 T)** and **(BREAK0 'FOO2 '(GREATERP N 5) '(EVAL))**.

(TRACE *X***)** [NLambda NoSpread Function]

For each atomic argument, it performs **(BREAK0** *ATOM* **T '(TRACE ? = NIL GO))**. The flag **TRACE** is checked for in **BREAK1** and causes the message "*FUNCTION* :" to be printed instead of (*FUNCTION* **BROKEN**).

For each list argument, **CAR** is the function to be traced, and **CDR** the forms the user wishes to see, i.e., **TRACE** performs:

(BREAK0 (CAR *LIST***) T (LIST 'TRACE '? = (CDR** *LIST***) 'GO))**

For example, **(TRACE FOO1 (FOO2 Y))** will cause both **FOO1** and **FOO2** to be traced. All the arguments of **FOO1** will be printed; only the value of **Y** will be printed for **FOO2**. In the special case that the user wants to see *only* the value, he can perform **(TRACE (***FUNCTION***))**. This sets up a break with commands **(TRACE ? = (NIL) GO)**.

Note: the user can always call **BREAK0** himself to obtain combination of options of **BREAK1** not directly available with **BREAK** and **TRACE**. These two functions merely provide convenient ways of calling **BREAK0**, and will serve for most uses.

Note: **BREAK0**, **BREAK**, and **TRACE** print a warning if the user tries to modify a function on the list **UNSAFE.TO.MODIFY.FNS** (page 10.10).

(BREAKIN *FN WHERE WHEN COMS***)** [NLambda Function]

BREAKIN enables the user to insert a break, i.e., a call to **BREAK1** (page 14.16), at a specified location in the interpreted function *FN*. **BREAKIN** can be used to insert breaks before or after **PROG** labels, particular **SETQ** expressions, or even the evaluation of a variable. This is because **BREAKIN** operates by calling the editor and actually inserting a call to **BREAK1** at a specified point *inside* of the function. If *FN* is a compiled function, **BREAKIN** returns **(***FN* **UNBREAKABLE)** as its value.

WHEN should be an expression that is evaluated whenever the break is entered. If the value of the expression is non-**NIL**, a break is entered, otherwise the function simply called and returns without causing a break. This provides the means of creating a conditional break. Note: For **BREAKIN**, unlike **BREAK0**, if *WHEN* is **NIL**, it defaults to **T**.

COMS, if non-**NIL**, should be a list of break commands, that are interpreted and executed if a break occurs. (See the *BRKCONMS* argument to **BREAK1**, page 14.17.)

WHERE specifies where in the definition of *FN* the call to **BREAK1** is to be inserted. *WHERE* should be a list of the form **(BEFORE ...)**, **(AFTER ...)**, or **(AROUND ...)**. The user specifies where the break is to be inserted by a sequence of editor commands, preceded by one of the litatoms **BEFORE**, **AFTER**, or **AROUND**, which **BREAKIN** uses to determine what to do once the editor has found the specified point, i.e., put the call to **BREAK1 BEFORE** that point, **AFTER** that point, or **AROUND** that point. For example, **(BEFORE COND)** will insert a break before the first occurrence of **COND**, **(AFTER COND 2 1)** will insert a break after the predicate in the first **COND** clause, **(AFTER BF (SETQ X &))** after the *last* place **X** is set. Note that **(BEFORE TTY:)** or **(AFTER TTY:)** permit the user to type in commands to the editor, locate the correct point, and verify it, and exit from the editor with **OK**. **BREAKIN** then inserts the break **BEFORE**, **AFTER**, or **AROUND** that point.

Note: A **STOP** command typed to **TTY:** produces the same effect as an unsuccessful edit command in the original specification, e.g., **(BEFORE CONDD)**. In both cases, the editor aborts, and **BREAKIN** types **(NOT FOUND)**.

If *WHERE* is **(BEFORE ...)** or **(AFTER ...)**, the break expression is **NIL**, since the value of the break is irrelevant. For **(AROUND ...)**, the break expression will be the indicated form. In this case, the user can use the **EVAL** command to evaluate that form, and examine its value, before allowing the computation to proceed. For example, if the user inserted a break after a **COND** predicate, e.g., **(AFTER (EQUAL X Y))**, he would be powerless to alter the flow of computation if the predicate were not true, since the break would not be reached. However, by breaking **(AROUND**

(EQUAL X Y)), he can evaluate the break expression, i.e., **(EQUAL X Y)**, look at its value, and return something else if he wished.

If *FN* is interpreted, **BREAKIN** types **SEARCHING...** while it calls the editor. If the location specified by *WHERE* is not found, **BREAKIN** types **(NOT FOUND)** and exits. If it is found, **BREAKIN** puts **T** under the property **BROKEN-IN** and (*WHERE WHEN COMS*) under the the property **BRKINFO** on the property list of *FN*, and adds *FN* to the front of the list **BROKENFNS**.

Multiple break points, can be inserted with a single call to **BREAKIN** by using a list of the form **((BEFORE ...) ... (AROUND ...))** for *WHERE*. It is also possible to call **BREAK** or **TRACE** on a function which has been modified by **BREAKIN**, and conversely to **BREAKIN** a function which has been redefined by a call to **BREAK** or **TRACE**.

The message typed for a **BREAKIN** break is **((*FN*) BROKEN)**, where *FN* is the name of the function inside of which the break was inserted. Any error, or typing control-E, will cause the full identifying message to be printed, e.g., **(FOO BROKEN AFTER COND 2 1)**.

A special check is made to avoid inserting a break inside of an expression headed by any member of the list **NOBREAKS**, initialized to **(GO QUOTE *)**, since this break would never be activated. For example, if **(GO L)** appears before the label **L**, **BREAKIN (AFTER L)** will not insert the break inside of the **GO** expression, but skip this occurrence of **L** and go on to the next **L**, in this case the label **L**. Similarly, for **BEFORE** or **AFTER** breaks, **BREAKIN** checks to make sure that the break is being inserted at a "safe" place. For example, if the user requests a break **(AFTER X)** in **(PROG ... (SETQ X &) ...)**, the break will actually be inserted after **(SETQ X &)**, and a message printed to this effect, e.g., **BREAK INSERTED AFTER (SETQ X &)**.

(UNBREAK *X*) [NLambda NoSpread Function]

UNBREAK takes an indefinite number of functions modified by **BREAK**, **TRACE**, or **BREAKIN** and restores them to their original state by calling **UNBREAK0**. Returns list of values of **UNBREAK0**.

(UNBREAK) will unbreak all functions on **BROKENFNS**, in reverse order. It first sets **BRKINFOLST** to **NIL**.

(UNBREAK T) unbreaks just the first function on **BROKENFNS**, i.e., the most recently broken function.

(UNBREAK0 *FN* —) [Function]

Restores *FN* to its original state. If *FN* was not broken, value is **(NOT BROKEN)** and no changes are made. If *FN* was modified by **BREAKIN**, **UNBREAKIN** is called to edit it back to its original state.

If *FN* was created from (*FN1* **IN** *FN2*), (i.e., if it has a property **ALIAS**), the function in which *FN* appears is restored to its original state. All dummy functions that were created by the break are eliminated. Adds property value of **BRKINFO** to (front of) **BRKINFOLST**.

Note: (**UNBREAK0** '(*FN1* **IN** *FN2*)) is allowed: **UNBREAK0** will operate on (*FN1*-**IN**-*FN2*) instead.

(UNBREAKIN *FN*) [Function]

Performs the appropriate editing operations to eliminate all changes made by **BREAKIN**. *FN* may be either the name or definition of a function. Value is *FN*.

UNBREAKIN is automatically called by **UNBREAK** if *FN* has property **BROKEN-IN** with value **T** on its property list.

(REBREAK *X*) [NLambda NoSpread Function]

Nlambda nospread function for rebreaking functions that were previously broken without having to respecify the break information. For each function on *X*, **REBREAK** searches **BRKINFOLST** for break(s) and performs the corresponding operation. Value is a list of values corresponding to calls to **BREAK0** or **BREAKIN**. If no information is found for a particular function, returns (*FN* **- NO BREAK INFORMATION SAVED**).

(REBREAK) rebreaks everything on **BRKINFOLST**, so **(REBREAK)** is the inverse of **(UNBREAK)**.

(REBREAK T) rebreaks just the first break on **BRKINFOLST**, i.e., the function most recently unbroken.

(CHANGENAME *FN FROM TO*) [Function]

Replaces all occurrences of *FROM* by *TO* in the definition of *FN*. If *FN* is defined by an expr definition, **CHANGENAME** performs **(ESUBST** *TO FROM* **(GETD** *FN*)) (see page 16.73). If *FN* is compiled, **CHANGENAME** searches the literals of *FN* (and all of its compiler generated subfunctions), replacing each occurrence of *FROM* with *TO*.

Note that *FROM* and *TO* do not have to be functions, e.g., they can be names of variables, or any other literals.

CHANGENAME returns *FN* if at least one instance of *FROM* was found, otherwise **NIL**.

(VIRGINFN *FN FLG*) [Function]

The function that knows how to restore functions to their original state regardless of any amount of breaks, breakins, advising, compiling and saving exprs, etc. It is used by **PRETTYPRINT**, **DEFINE**, and the compiler.

If *FLG* = **NIL**, as for **PRETTYPRINT**, it does not modify the definition of *FN* in the process of producing a "clean" version of the definition; it works on a copy.

If *FLG* = **T**, as for the compiler and **DEFINE**, it physically restores the function to its original state, and prints the changes it is making, e.g., **FOO UNBROKEN, FOO UNADVISED, FOO NAMES RESTORED**, etc.

Returns the virgin function definition.

15.2 Advising

The operation of advising gives the user a way of modifying a function without necessarily knowing how the function works or even what it does. Advising consists of modifying the *interface* between functions as opposed to modifying the function definition itself, as in editing. **BREAK**, **TRACE**, and **BREAKDOWN**, are examples of the use of this technique: they each modify user functions by placing relevant computations *between* the function and the rest of the programming environment.

The principal advantage of advising, aside from its convenience, is that it allows the user to treat functions, his or someone else's, as "black boxes," and to modify them without concern for their contents or details of operations. For example, the user could modify **SYSOUT** to set **SYSDATE** to the time and date of creation by **(ADVISE 'SYSOUT '(SETQ SYSDATE (DATE)))**.

As with **BREAK**, advising works equally well on compiled and interpreted functions. Similarly, it is possible to effect a modification which only operates when a function is called from some other specified function, i.e., to modify the interface between two particular functions, instead of the interface between one function and the rest of the world. This latter feature is especially useful for changing the *internal* workings of a system function.

For example, suppose the user wanted **TIME** (page 22.8) to print the results of his measurements to the file **FOO** instead of the terminal. He could accomplish this by **(ADVISE '((PRIN1 PRINT SPACES) IN TIME) 'BEFORE '(SETQQ U FOO))**.

Note that advising **PRIN1**, **PRINT**, or **SPACES** directly would have affected all calls to these very frequently used function, whereas advising **((PRIN1 PRINT SPACES) IN TIME)** affects just those calls to **PRIN1**, **PRINT**, and **SPACES** from **TIME**.

Advice can also be specified to operate after a function has been evaluated. The value of the body of the original function can be obtained from the variable **!VALUE**, as with **BREAK1**.

15.2.1 Implementation of Advising

After a function has been modified several times by **ADVISE**, it will look like:

```
(LAMBDA arguments
  (PROG (!VALUE)
    (SETQ !VALUE
      (PROG NIL
        advice1
          .
          .  advice before
          .
        advicen
        (RETURN BODY)))
    advice1
      .
      .  advice after
      .
    advicem
    (RETURN !VALUE)))
```

where *BODY* is equivalent to the original definition. If *FN* was originally an expr definition, *BODY* is the body of the definition, otherwise a form using a **GENSYM** which is defined with the original definition.

Note that the structure of a function modified by **ADVISE** allows a piece of advice to bypass the original definition by using the function **RETURN**. For example, if **(COND ((ATOM X) (RETURN Y)))** were one of the pieces of advice **BEFORE** a function, and this function was entered with **X** atomic, **Y** would be returned as the value of the inner **PROG**, **!VALUE** would be set to **Y**, and control passed to the advice, if any, to be executed **AFTER** the function. If this same piece of advice appeared **AFTER** the function, **Y** would be returned as the value of the entire advised function.

The advice **(COND ((ATOM X) (SETQ !VALUE Y)))** **AFTER** the function would have a similar effect, but the rest of the advice **AFTER** the function would still be executed.

Note: Actually, **ADVISE** uses its own versions of **PROG**, **SETQ**, and **RETURN**, (called **ADV-PROG**, **ADV-SETQ**, and **ADV-RETURN**) in order to enable advising these functions.

15.2.2 Advise Functions

ADVISE is a function of four arguments: *FN*, *WHEN*, *WHERE*, and *WHAT*. *FN* is the function to be modified by advising, *WHAT* is the modification, or piece of advice. *WHEN* is either **BEFORE**, **AFTER**, or **AROUND**, and indicates whether the advice is to operate **BEFORE**, **AFTER**, or **AROUND** the body of the function

definition. *WHERE* specifies exactly where in the list of advice the new advice is to be placed, e.g., **FIRST**, or **(BEFORE PRINT)** meaning before the advice containing **PRINT**, or **(AFTER 3)** meaning after the third piece of advice, or even **(: TTY:)**. If *WHERE* is specified, **ADVISE** first checks to see if it is one of **LAST**, **BOTTOM**, **END**, **FIRST**, or **TOP**, and operates accordingly. Otherwise, it constructs an appropriate edit command and calls the editor to insert the advice at the corresponding location.

Both *WHEN* and *WHERE* are optional arguments, in the sense that they can be omitted in the call to **ADVISE**. In other words, **ADVISE** can be thought of as a function of two arguments **(ADVISE** *FN WHAT*), or a function of three arguments: **(ADVISE** *FN WHEN WHAT*), or a function of four arguments: **(ADVISE** *FN WHEN WHERE WHAT*). Note that the advice is always the *last* argument. If *WHEN* = **NIL**, **BEFORE** is used. If *WHERE* = **NIL**, **LAST** is used.

(ADVISE *FN WHEN WHERE WHAT*) [Function]

FN is the function to be advised, *WHEN* = **BEFORE**, **AFTER**, or **AROUND**, *WHERE* specifies where in the advice list the advice is to be inserted, and *WHAT* is the piece of advice.

If *FN* is of the form **(***FN1* **IN** *FN2*), *FN1* is changed to *FN1*-**IN**-*FN2* throughout *FN2*, as with break, and then *FN1*-**IN**-*FN2* is used in place of *FN*. If *FN1* and/or *FN2* are lists, they are distributed as with **BREAK0**, page 15.4.

If *FN* is broken, it is unbroken before advising.

If *FN* is not defined, an error is generated, **NOT A FUNCTION**.

If *FN* is being advised for the first time, i.e., if **(GETP** *FN* **'ADVISED) = NIL**, a **GENSYM** is generated and stored on the property list of *FN* under the property **ADVISED**, and the **GENSYM** is defined with the original definition of *FN*. An appropriate expr definition is then created for *FN*, using private versions of **PROG**, **SETQ**, and **RETURN**, so that these functions can also be advised. Finally, *FN* is added to the (front of) **ADVISEDFNS**, so that **(UNADVISE T)** always unadvises the last function advised (see page 15.12).

If *FN* has been advised before, it is moved to the front of **ADVISEDFNS**.

If *WHEN* = **BEFORE** or **AFTER**, the advice is inserted in *FN*'s definition either **BEFORE** or **AFTER** the original body of the function. Within that context, its position is determined by *WHERE*. If *WHERE* = **LAST**, **BOTTOM**, **END**, or **NIL**, the advice is added following all other advice, if any. If *WHERE* = **FIRST** or **TOP**, the advice is inserted as the first piece of advice. Otherwise, *WHERE* is treated as a command for the editor, similar to **BREAKIN**, e.g., **(BEFORE 3)**, **(AFTER PRINT)**.

If *WHEN* = **AROUND**, the body is substituted for * in the advice, and the result becomes the new body, e.g., **(ADVISE 'FOO 'AROUND '(RESETFORM (OUTPUT T) *))**. Note that if several pieces of **AROUND** advice are specified, earlier ones will be embedded inside later ones. The value of *WHERE* is ignored.

Finally **(LIST** *WHEN WHERE WHAT***)** is added (by **ADDPROP**) to the value of property **ADVICE** on the property list of *FN*, so that a record of all the changes is available for subsequent use in readvising. Note that this property value is a list of the advice in order of calls to **ADVISE**, not necessarily in order of appearance of the advice in the definition of *FN*.

The value of **ADVISE** is *FN*.

If *FN* is non-atomic, every function in *FN* is advised with the same values (but copies) for *WHEN*, *WHERE*, and *WHAT*. In this case, **ADVISE** returns a list of individual functions.

Note: advised functions can be broken. However if a function is broken at the time it is advised, it is first unbroken. Similarly, advised functions can be edited, including their advice. **UNADVISE** will still restore the function to its unadvised state, but any changes to the body of the definition will survive. Since the advice stored on the property list is the same structure as the advice inserted in the function, editing of advice can be performed on either the function's definition or its property list.

(UNADVISE *X***)** [NLambda NoSpread Function]

An nlambda nospread like **UNBREAK**. It takes an indefinite number of functions and restores them to their original unadvised state, including removing the properties added by **ADVISE**. **UNADVISE** saves on the list **ADVINFOLST** enough information to allow restoring a function to its advised state using **READVISE**. **ADVINFOLST** and **READVISE** thus correspond to **BRKINFOLST** and **REBREAK**. If a function contains the property **READVICE**, **UNADVISE** moves the current value of the property **ADVICE** to **READVICE**.

(UNADVISE) unadvises all functions on **ADVISEDFNS** in reverse order, so that the most recently advised function is unadvised last. It first sets **ADVINFOLST** to **NIL**.

(UNADVISE T) unadvises the first function of **ADVISEDFNS**, i.e., the most recently advised function.

(READVISE *X***)** [NLambda NoSpread Function]

An nlambda nospread like **REBREAK** for restoring a function to its advised state without having to specify all the advise information. For each function on *X*, **READVISE** retrieves the advise information either from the property **READVICE** for that

function, or from **ADVINFOLST**, and performs the corresponding advise operation(s). In addition it stores this information on the property **READVICE** if not already there. If no information is found for a particular function, value is (*FN* - **NO ADVICE SAVED**).

(READVISE) readvises everything on **ADVINFOLST**.

(READVISE T) readvises the first function on **ADVINFOLST**, i.e., the function most recently unadvised.

A difference between **ADVISE, UNADVISE**, and **READVISE** versus **BREAK, UNBREAK**, and **REBREAK**, is that if a function is not rebroken between successive **(UNBREAK)**'s, its break information is forgotten. However, once **READVISE** is called on a function, that function's advice is permanently saved on its property list (under **READVICE**); subsequent calls to **UNADVISE** will not remove it. In fact, calls to **UNADVISE** update the property **READVICE** with the current value of the property **ADVICE**, so that the sequence **READVISE, ADVISE, UNADVISE** causes the augmented advice to become permanent. Note that the sequence **READVISE, ADVISE, READVISE** removes the "intermediate advice" by restoring the function to its earlier state.

(ADVISEDUMP *X FLG***)** [Function]

Used by **PRETTYDEF** when given a command of the form **(ADVISE ...)** or **(ADVICE ...)**. If *FLG* = **T**, **ADVISEDUMP** writes both a **DEFLIST** and a **READVISE** (this corresponds to **(ADVISE ...)**). If *FLG* = **NIL**, only the **DEFLIST** is written (this corresponds to **(ADVICE ...)**). In either case, **ADVISEDUMP** copies the advise information to the property **READVICE**, thereby making it "permanent" as described above.

[This page intentionally left blank]

TABLE OF CONTENTS

16. LIST STRUCTURE EDITOR

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

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

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

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

16.1 DEdit

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

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

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

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

16.1.1 Calling DEdit

DEdit is normally called using the following functions:

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

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

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

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

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

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

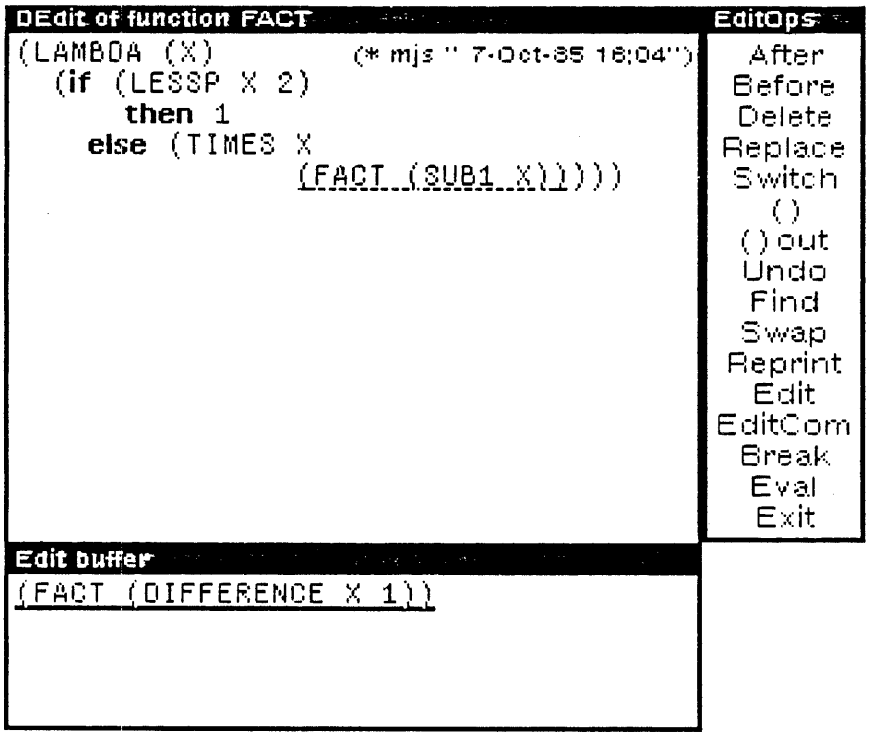
(DP *NAME PROP***)** [NLambda NoSpread Function]

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

(DC *FILE*) [NLambda NoSpread Function]

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

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



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

(RESETDEDIT) [Function]

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

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

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

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

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

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

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

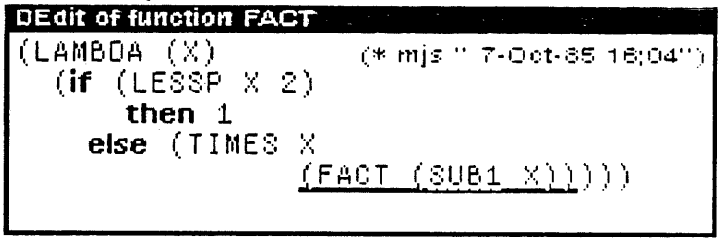
If teletype editor commands are specified in a call to **DEDITL** (e.g., in calls to the editor from Masterscope), **DEDITL** will pass those commands to **EDITL**, after having placed a **TTY:** entry on **EDITMACROS** which will cause DEdit to be invoked if any interaction with the user is called for.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

16.1.2 Selecting Objects and Lists

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

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

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



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

16.1.3 Typing Characters to DEdit

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

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

16.1.4 Copy-Selection

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

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

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

16.1.5 DEdit Commands

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

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

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

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

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

After [DEdit Command]

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

Before [DEdit Command]

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

Delete [DEdit Command]

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

Replace [DEdit Command]

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

Switch [DEdit Command]

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

() [DEdit Command]

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

(in [DEdit Command]

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

) in [DEdit Command]

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

() out [DEdit Command]

Removes parentheses from *TOP*.

(out [DEdit Command]

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

) out [DEdit Command]

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

Undo [DEdit Command]

Undoes last command.

!Undo [DEdit Command]

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

?Undo [DEdit Command]

&Undo [DEdit Command]

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

Find [DEdit Command]

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

Swap [DEdit Command]

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

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

Center [DEdit Command]

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

Clear [DEdit Command]

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

Copy [DEdit Command]

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

Pop [DEdit Command]

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

Reprint [DEdit Command]

Reprints *TOP*.

Edit [DEdit Command]

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

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

EditCom [DEdit Command]

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

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

Break [DEdit Command]

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

Eval [DEdit Command]

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

Exit [DEdit Command]

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

OK [DEdit Command]

Stop [DEdit Command]

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

16.1.6 Multiple Commands

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

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

16.1.7 DEdit Idioms

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

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

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

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

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

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

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

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

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

16.1.8 DEdit Parameters

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

EDITEMBEDTOKEN [Variable]

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

DEditLinger [Variable]

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

DEDITTYPEINCOMS [Variable]

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

DT.EDITMACROS [Variable]

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

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

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

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

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

16.2 Local Attention-Changing Commands

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

UP [Editor Command]

UP modifies the edit chain so that the old current expression (i.e., the one at the time **UP** was called) is the first element in the new current expression. If the current expression is the first element in the next higher expression **UP** simply does a **0**. Otherwise **UP** adds the corresponding tail to the edit chain.

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

For Example:

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

The execution of **UP** is straightforward, except in those cases where the current expression appears more than once in the next higher expression. For example, if the current expression is **(A NIL B NIL C NIL)** and the user performs **4** followed by **UP**, the current expression should then be **... NIL C NIL)**. **UP** can determine which tail is the correct one because the commands that descend save the last tail on an internal editor variable, **LASTAIL**. Thus after the **4** command is executed, **LASTAIL** is **(NIL C NIL)**. When **UP** is called, it first determines if the current expression is a tail of the next higher expression. If it is, **UP** is finished. Otherwise, **UP** computes **(MEMB** *CURRENT-EXPRESSION NEXT-HIGHER-EXPRESSION*) to obtain a tail beginning with the current expression. The current expression should *always* be either a tail or an element of the next higher expression. If it is neither, for example the user has directly (and incorrectly) manipulated the edit chain, **UP** generates an error. If there are no other instances of the current expression in the next higher expression, this tail is the correct one. Otherwise **UP** uses **LASTAIL** to select the correct tail.

Occasionally the user can get the edit chain into a state where **LASTAIL** cannot resolve the ambiguity, for example if there were two non-atomic structures in the same expression that were **EQ**, and the user descended more than one level into one of them and then tried to come back out using **UP**. In this case, **UP** prints **LOCATION UNCERTAIN** and generates an error. Of course, we

could have solved this problem completely in our implementation by saving at each descent *both* elements and tails. However, this would be a costly solution to a situation that arises infrequently, and when it does, has no detrimental effects. The **LASTAIL** solution is cheap and resolves almost all of the ambiguities.

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

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

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

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

0 [Editor Command]

Sets the edit chain to **CDR** of the edit chain, thereby making the next higher expression be the new current expression. Generates an error if there is no higher expression, i.e., **CDR** of edit chain is **NIL**.

Note that **0** usually corresponds to going back to the next higher left parenthesis, but not always. For example:

```
*P
(A B C D E F B)
*3 UP P
... C D E F G)
*3 UP P
... E F G)
*0 P
... C D E F G)
```

If the intention is to go back to the next higher left parenthesis, regardless of any intervening tails, the command **!0** can be used.

!0 [Editor Command]

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

↑ [Editor Command]

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

NX [Editor Command]

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

BK [Editor Command]

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

For example,

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

Both **NX** and **BK** operate by performing a **!0** followed by an appropriate number, i.e., there won't be an extra tail above the new current expression, as there would be if **NX** operated by performing an **UP** followed by a **2**.

(NX *N*) [Editor Command]

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

(BK *N*) [Editor Command]

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

Note: **(NX -*N*)** is equivalent to **(BK *N*)**, and vice versa.

!NX [Editor Command]

Makes the current expression be the next expression at a higher level, i.e., goes through any number of right parentheses to get to the next expression. For example:

```
*PP
(PROG ((L L)
   (UF L))
```

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

NX ?
*!NX P
(ERROR!)
*!NX P
((NULL &) (GO LP))
*!NX P
(EDITCOM (QUOTE NX))
*
```

!NX operates by doing **0**'s until it reaches a stage where the current expression is *not* the last expression in the next higher expression, and then does a **NX**. Thus **!NX** always goes through at least one unmatched right parenthesis, and the new current expression is always on a different level, i.e., **!NX** and **NX** always produce different results. For example using the previous current expression:

```
*F CAR P
(CAR L)
*!NX P
(GO LP)
*\P P
(CAR L)
*NX P
(CADR L)
*
```

(NTH *N*) [Editor Command]

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

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

line-feed [Editor Command]

Moves to the "next" expression and prints it, i.e. performs a **NX** if possible, otherwise performs a **!NX**. (The latter case is indcated by first printing "**>**".)

Control-X [Editor Command]

Control-X moves to the "previous" thing and then prints it, i.e. performs a **BK** if possible, otherwise a **!0** followed by a **BK**.

Control-Z [Editor Command]

Control-Z moves to the last expression and prints it, i.e. does **-1** followed by **P**.

Line-feed, control-X, and control-Z are implemented as *immediate* read macros; as soon as they are read, they abort the current printout. They thus provide a convenient way of moving around in the editor. In order to facilitate using different control characters for those macros, the function **SETTERMCHARS** is provided (see page 16.75).

16.3 Commands That Search

All of the editor commands that search use the same pattern matching routine (the function **EDIT4E**, page 16.72). We will therefore begin our discussion of searching by describing the pattern match mechanism. A pattern *PAT* matches with *X* if any of the following conditions are true:

(1) If *PAT* is **EQ** to *X*.

(2) If *PAT* is **&**.

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

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

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

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

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

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

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

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

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

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

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

(10) If **(CADR** *PAT***)** is the atom .. (two periods), *PAT* matches *X* if **(CAR** *PAT***)** matches **(CAR** *X***)** and **(CDDR** *PAT***)** is contained in *X*, as described on page 16.27.

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

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

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

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

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

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

16.3.1 Search Algorithm

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

If the match is not successful, the search operation is recursive first in the **CAR** direction, and then in the **CDR** direction, i.e., if the element under examination is a list, the search descends into that list before attempting to match with other elements (or tails) at the same level. Note: A find command of the form **(F** *PATTERN* **NIL)** will only attempts matches at the top level of the current expression, i.e., it does not descend into elements, or ascend to higher expressions.

However, at no point is the total recursive depth of the search (sum of number of **CAR**s and **CDR**s descended into) allowed to exceed the value of the variable **MAXLEVEL**. At that point, the search of that element or tail is abandoned, exactly as though the element or tail had been completely searched without finding a match, and the search continues with the element or tail for which the recursive depth is below **MAXLEVEL**. This feature is designed to enable the user to search circular list structures (by setting **MAXLEVEL** small), as well as protecting him from accidentally encountering a circular list structure in the course of normal editing. **MAXLEVEL** can also be set to **NIL**, which is equivalent to infinity. **MAXLEVEL** is initially set to 300.

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

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

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

16.3.2 Search Commands

All of the commands below set **LASTAIL** for use by **UP**, set **UNFIND** for use by \ (page 16.28), and do not change the edit chain or perform any **CONS**es if they are unsuccessful or aborted.

F *PATTERN* [Editor Command]

Actually two commands: the **F** informs the editor that the *next* command is to be interpreted as a pattern. This is the most common and useful form of the find command. If successful, the edit chain always changes, i.e., **F** *PATTERN* means find the next instance of *PATTERN*.

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

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

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

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

F *PATTERN* **T** [Editor Command]

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

For example, if the current expression is **(COND ...)**, **F COND** will look for the next **COND**, but **(F COND T)** will "stay here".

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

(*N* >= 1) Finds the *N*th place that *PATTERN* matches. Equivalent to **(F** *PATTERN* **T)** followed by **(F** *PATTERN* **N)** repeated *N*-1 times. Each time *PATTERN* successfully matches, *N* is decremented by 1, and the search continues, until *N* reaches 0. Note that *PATTERN* does not have to match with *N* identical expressions; it just has to match *N* times. Thus if the current expression is **(FOO1 FOO2 FOO3)**, **(F FOO$ 3)** will find **FOO3**.

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

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

F *PATTERN* **NIL** [Editor Command]

Similar to **F** *PATTERN*, except that it only matches with elements at the top level of the current expression, i.e., the search will not descend into the current expression, nor will it go outside of the current expression. May succeed without changing the edit chain.

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

(FS $PATTERN_1$ ... $PATTERN_N$**)** [Editor Command]

Equivalent to **F** $PATTERN_1$ followed by **F** $PATTERN_2$ ... followed by **F** $PATTERN_N$, so that if **F** $PATTERN_M$ fails, the edit chain is left at the place $PATTERN_{M-1}$ matched.

(F = *EXPRESSION X***)** [Editor Command]

Equivalent to **(F (= = .** *EXPRESSION***)** *X*), i.e., searches for a structure **EQ** to *EXPRESSION* (see page 16.18).

(ORF $PATTERN_1$ ... $PATTERN_N$**)** [Editor Command]

Equivalent to **(F (*ANY***$PATTERN_1$ ... $PATTERN_N$**) N)**, i.e., searches for an expression that is matched by either $PATTERN_1$, $PATTERN_2$, ... or $PATTERN_N$ (see page 16.18).

BF *PATTERN* [Editor Command]

"Backwards Find". Searches in reverse print order, beginning with the expression immediately before the current expression (unless the current expression is the top level expression, in which case **BF** searches the entire expression, in reverse order).

BF uses the same pattern match routine as **F**, and **MAXLEVEL** and **UPFINDFLG** have the same effect, but the searching begins at the *end* of each list, and descends into each element before attempting to match that element. If unsuccessful, the search continues with the next previous element, etc., until the front of the list is reached, at which point **BF** ascends and backs up, etc.

For example, if the current expression is

(PROG NIL (SETQ X (SETQ Y (LIST Z))) (COND ((SETQ W --) --)) --),

the command **F LIST** followed by **BF SETQ** will leave the current expression as **(SETQ Y (LIST Z))**, as will **F COND** followed by **BF SETQ**.

BF *PATTERN* **T** [Editor Command]

Similar to **BF** *PATTERN*, except that the search always includes the current expression, i.e., starts at the end of current expression and works backward, then ascends and backs up, etc.

Thus in the previous example, where **F COND** followed by **BF SETQ** found **(SETQ Y (LIST Z))**, **F COND** followed by **(BF SETQ T)** would find the **(SETQ W --)** expression.

(BF *PATTERN***)** [Editor Command]

BF *PATTERN* **NIL** [Editor Command]

Same as **BF** *PATTERN*.

(GO *LABEL***)** [Editor Command]

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

16.3.3 Location Specification

Many of the more sophisticated commands described later in this chapter use a more general method of specifying position called a "location specification." A location specification is a list of edit commands that are executed in the normal fashion with two exceptions. First, all commands not recognized by the editor are interpreted as though they had been preceded by **F**; normally such commands would cause errors. For example, the location

specification **(COND 2 3)** specifies the 3rd element in the first clause of the next **COND**. Note that the user could always write **F COND** followed by 2 and 3 for **(COND 2 3)** if he were not sure whether or not **COND** was the name of an atomic command.

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

The **IF** command (page 16.60) in conjunction with the *##* function (page 16.59) provide a way of using arbitrary predicates applied to elements in the current expression. **IF** and *##* will be described in detail later in the chapter, along with examples illustrating their use in location specifications.

Throughout this chapter, the meta-symbol @ is used to denote a location specification. Thus @ is a list of commands interpreted as described above. @ can also be atomic, in which case it is interpreted as **(LIST @)**.

(LC . @) [Editor Command]

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

(LCL . @) [Editor Command]

Same as **LC** except the search is confined to the current expression, i.e., the edit chain is rebound during the search so that it looks as though the editor were called on just the current expression. For example, to find a **COND** containing a **RETURN**, one might use the location specification **(COND (LCL RETURN) \)** where the \ would reverse the effects of the **LCL** command, and make the final current expression be the **COND**.

(2ND . @) [Editor Command]

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

(3ND . @) [Editor Command]

Similar to **2ND**.

(← *PATTERN*) [Editor Command]

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

Note: If *PATTERN* is of the form **(IF *EXPRESSION*)**, *EXPRESSION* is evaluated at each link, and if its value is **NIL**, or the evaluation causes an error, the ascent continues. See page 16.60.

For example:

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

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

(BELOW *COM X*) [Editor Command]

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

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

(BELOW *COM*) [Editor Command]

Same as **(BELOW *COM* 1)**.

For example, **(BELOW COND)** will cause the **COND** *clause* containing the current expression to become the new current

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

The **BELOW** command is useful for locating a substructure by specifying something it contains. For example, suppose the user is editing a list of lists, and wants to find a sublist that contains a **FOO** (at any depth). He simply executes **F FOO (BELOW \)**.

(NEX *COM*) [Editor Command]

Same as **(BELOW *COM*)** followed by **NX**.

For example, if the user is deep inside of a **SELECTQ** clause, he can advance to the next clause with **(NEX SELECTQ)**.

NEX [Editor Command]

Same as **(NEX ←)**.

The atomic form of **NEX** is useful if the user will be performing repeated executions of **(NEX *COM*)**. By simply **MARK**ing (see page 16.28) the chain corresponding to *COM*, he can use **NEX** to step through the sublists.

(NTH *COM*) [Editor Command]

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

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

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

In other words, **NTH** locates *COM*, using a search restricted to the current expression, and then backs up to the current level, where the new current expression is the tail whose first element contains, however deeply, the expression that was the terminus of the location operation. For example:

```
*P
(PROG (& &) LP (COND & &) (EDITCOM &) (SETQ UNFIND UF)
(RETURN L))
*(NTH UF)
*P
... (SETQ UNFIND UF) (RETURN L))
*
```

PATTERN .. @ [Editor Command]

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

An infix command, ".." is not a meta-symbol, it *is* the name of the command. @ is **CDDR** of the command. Note that **(***PATTERN* .. **@)** can also be used directly as an edit pattern as described on page 16.18, e.g. **F (***PATTERN* .. **@)**.

For example, if the current expression is

(PROG NIL [COND ((NULL L) (COND (FLG (RETURN L] --),

then **(COND .. RETURN)** will make **(COND (FLG (RETURN L)))** be the current expression. Note that it is the innermost **COND** that is found, because this is the first **COND** encountered when ascending from the **RETURN**. In other words, **(***PATTERN* .. **@)** is not *always* equivalent to **(F** *PATTERN* **N)**, followed by **(LCL . @)** followed by \.

Note that @ is a location specification, not just a pattern. Thus **(RETURN .. COND 2 3)** can be used to find the **RETURN** which contains a **COND** whose first clause contains (at least) three elements. Note also that since @ permits any edit command, the user can write commands of the form **(COND .. (RETURN .. COND))**, which will locate the first **COND** that contains a **RETURN** that contains a **COND**.

16.4 Commands That Save and Restore the Edit Chain

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

MARK [Editor Command]

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

← [Editor Command]

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

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

←← [Editor Command]

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

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

(MARK *LITATOM***)** [Editor Command]

Sets *LITATOM* to the current edit chain,

(*LITATOM***)** [Editor Command]

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

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

**** [Editor Command]

Makes the edit chain be the value of **UNFIND**. Generates an error if **UNFIND = NIL**.

UNFIND is set to the current edit chain by each command that makes a "big jump", i.e., a command that usually performs more than a single ascent or descent, namely ↑, ←, ←←, **!NX**, all commands that involve a search, e.g., **F**, **LC**, .., **BELOW**, et al and \ and **\P** themselves. One exception is that **UNFIND** is not reset when the current edit chain is the top level expression, since this could always be returned to via the ↑ command.

For example, if the user types **F COND**, and then **F CAR**, \ would take him back to the **COND**. Another \ would take him back to the **CAR**, etc.

\P [Editor Command]

Restores the edit chain to its state as of the last print operation, i.e., **P**, **?**, or **PP**. If the edit chain has not changed since the last printing, **\P** restores it to its state as of the printing before that one, i.e., two chains are always saved.

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

Note that if the user had typed **P** followed by **F COND**, he could use *either* \ or **\P** to return to the **P**, i.e., the action of \ and **\P** are independent.

S *LITATOM* **@** [Editor Command]

Sets *LITATOM* (using **SETQ**) to the current expression after performing **(LC . @)**. The edit chain is not changed.

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

16.5 Commands That Modify Structure

The basic structure modification commands in the editor are:

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

Deletes the corresponding element from the current expression.

(*N* E_1 ... E_M) (*N*> = 1) [Editor Command]

Replaces the *N*th element in the current expression with E_1 ... E_M.

(-*N* E_1 ... E_M) (*N*> = 1) [Editor Command]

Inserts E_1 ... E_M before the *N*th element in the current expression.

(N E_1 ... E_M**)** [Editor Command]

Attaches E_1 ... E_M at the end of the current expression.

As mentioned earlier: *all structure modification done by the editor is destructive, i.e., the editor uses* **RPLACA** *and* **RPLACD** *to physically change the structure it was given.* However, all structure modification is undoable, see **UNDO** (page 16.64).

All of the above commands generate errors if the current expression is not a list, or in the case of the first three commands, if the list contains fewer than *N* elements. In addition, the command **(1)**, i.e., delete the first element, will cause an error if there is only one element, since deleting the first element must be done by replacing it with the second element, and then deleting the second element. Or, to look at it another way, deleting the first element when there is only one element would

require changing a list to an atom (i.e., to **NIL**) which cannot be done. However, the command **DELETE** will work even if there is only one element in the current expression, since it will ascend to a point where it *can* do the deletion.

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

CHANGESARRAY is primarily for use by **PRETTYPRINT** (page 26.40). When the value of **CHANGECHAR** is non-**NIL**, **PRETTYPRINT**, when printing to a file or display terminal, prints **CHANGECHAR** in the right margin while printing an expression marked as having been changed. **CHANGECHAR** is initially |.

16.5.1 Implementation

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

For all replacement, insertion, and attaching at the end of a list, unless the command was typed in directly to the editor, *copies* of the corresponding structure are used, because of the possibility that the exact same command, (i.e., same list structure) might be used again. Thus if a program constructs the command **(1 (A B C))** e.g., via **(LIST 1 FOO)**, and gives this command to the editor, the **(A B C)** used for the replacement will *not* be **EQ** to **FOO**. The user can circumvent this by using the I command (page 16.58), which computes the structure to be used. In the above example, the form of the command would be **(I 1 FOO)**, which would replace the first element with the value of **FOO** itself.

Note: Some editor commands take as arguments a list of edit commands, e.g., **(LP F FOO (1 (CAR FOO)))**. In this case, the command **(1 (CAR FOO))** is not considered to have been "typed in" even though the **LP** command itself may have been typed in. Similarly, commands originating from macros, or commands given to the editor as arguments to **EDITF**, **EDITV**, et al, e.g., **EDITF(FOO F COND (N --))** are not considered typed in.

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

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

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

A general solution of the problem isn't possible, as it would require being able to make two lists **EQ** to each other that were originally different. Thus if **FIE** is **CDR** of the current expression, and **FUM** is **CDDR** of the current expression, performing **(2)** would have to make **FIE** be **EQ** to **FUM** if all subsequent operations were to update both **FIE** and **FUM** correctly.

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

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

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

16.5.2 The A, B, and : Commands

In the **(*N*)**, **(*N* E_1 ... E_M)**, and **(-*N* E_1 ... E_M)** commands, the sign of the integer is used to indicate the operation. As a result, there is no direct way to express insertion after a particular element, (hence the necessity for a separate **N** command). Similarly, the user cannot specify deletion or replacement of the *N*th element

from the end of a list without first converting *N* to the corresponding positive integer. Accordingly, we have:

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

Inserts E_1 ... E_M before the current expression. Equivalent to **UP** followed by **(-1** E_1 ... E_M**)**.

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

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

Inserts E_1 ... E_M after the current expression. Equivalent to **UP** followed by **(-2** E_1 ... E_M**)** or **(N** E_1 ... E_M**)**, whichever is appropriate.

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

Replaces the current expression by E_1 ... E_M. Equivalent to **UP** followed by **(1** E_1 ... E_M**)**.

DELETE [Editor Command]

(:) [Editor Command]

Deletes the current expression.

DELETE first tries to delete the current expression by performing an **UP** and then a **(1)**. This works in most cases. However, if after performing **UP**, the new current expression contains only one element, the command **(1)** will not work. Therefore, **DELETE** starts over and performs a **BK**, followed by **UP**, followed by **(2)**. For example, if the current expression is **(COND ((MEMB X Y)) (T Y))**, and the user performs **-1**, and then **DELETE**, the **BK-UP-(2)** method is used, and the new current expression will be ... **((MEMB X Y)))**.

However, if the next higher expression contains only one element, **BK** will not work. So in this case, **DELETE** performs **UP**, followed by **(: NIL)**, i.e., it *replaces* the higher expression by **NIL**. For example, if the current expression is **(COND ((MEMB X Y)) (T Y))** and the user performs **F MEMB** and then **DELETE**, the new current expression will be **... NIL (T Y))** and the original expression would now be **(COND NIL (T Y))**. The rationale behind this is that deleting **(MEMB X Y)** from **((MEMB X Y))** changes a list of one element to a list of no elements, i.e., **()** or **NIL**.

If the current expression is a tail, then **B**, **A**, **:**, and **DELETE** all work exactly the same as though the current expression were the first element in that tail. Thus if the current expression were ...

(PRINT Y) (PRINT Z)), **(B (PRINT X))** would insert **(PRINT X)** before **(PRINT Y)**, leaving the current expression ... **(PRINT X) (PRINT Y) (PRINT Z))**.

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

(INSERT E_1 ... E_M BEFORE . @) [Editor Command]

(@ is (CDR (MEMBER 'BEFORE *COMMAND*))) Similar to **(LC .@)** followed by **(B E_1 ... E_M)**.

Warning: If **@** causes an error, the location process does *not* continue as described on page 16.23. For example if **@ = (COND 3)** and the next **COND** does not have a 3rd element, the search stops and the **INSERT** fails. Note that the user can always write **(LC COND 3)** if he intends the search to continue.

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

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

*
```

Current edit chain is not changed, but **UNFIND** is set to the edit chain after the **B** was performed, i.e., **** will make the edit chain be that chain where the insertion was performed.

(INSERT E_1 ... E_M AFTER . @) [Editor Command]

Similar to **INSERT BEFORE** except uses **A** instead of **B**.

(INSERT E_1 ... E_M FOR . @) [Editor Command]

Similar to **INSERT BEFORE** except uses **:** for **B**.

(REPLACE @ BY E_1 ... E_M) [Editor Command]

(REPLACE @ WITH E_1 ... E_M) [Editor Command]

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

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

(CHANGE @ TO E_1 ... E_M) [Editor Command]

Same as **REPLACE WITH**.

(DELETE . @) [Editor Command]

Does a **(LC . @)** followed by **DELETE** (see warning about **INSERT**, page 16.33). The current edit chain is not changed, but **UNFIND** is set to the edit chain after the **DELETE** was performed.

Note: the edit chain will be changed if the current expression is no longer a part of the expression being edited, e.g., if the current expression is **... C)** and the user performs **(DELETE 1)**, the tail, **(C)**, will have been cut off. Similarly, if the current expression is **(CDR Y)** and the user performs **(REPLACE WITH (CAR X))**.

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

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

For example, **(REPLACE WITH (CAR X))** is equivalent to **(: (CAR X))**. For added readability, **HERE** is also permitted, e.g., **(INSERT (PRINT X) BEFORE HERE)** will insert **(PRINT X)** before the current expression (but not change the edit chain).

Note: @ does not have to specify a location within the current expression, i.e., it is perfectly legal to ascend to **INSERT**, **REPLACE**, *or* **DELETE**

For example, **(INSERT (RETURN) AFTER ↑ PROG -1)** will go to the top, find the first **PROG**, and insert a **(RETURN)** at its end, and not change the current edit chain.

The **A**, **B**, and **:** commands, commands, (and consequently **INSERT**, **REPLACE**, and **CHANGE**), all make special checks in E_1 thru E_M for expressions of the form *(## . COMS)*. In this case, the expression used for inserting or replacing is a *copy* of the current expression after executing *COMS*, a list of edit commands (the execution of *COMS* does not change the current edit chain). For example, **(INSERT (## F COND -1 -1) AFTER 3)** will make a copy of the last form in the last clause of the next **COND**, and insert it after the third element of the current expression. Note that this is not the same as **(INSERT F COND -1 (## -1) AFTER 3)**, which inserts four elements after the third element, namely **F**, **COND**, **-1**, and a copy of the last element in the current expression.

16.5.3 Form Oriented Editing and the Role of UP

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

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

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

Occasionally, however, a user may have a data structure in which no special significance or meaning is attached to the position of an atom in a list, as Interlisp attaches to atoms that appear as **CAR** of a list, versus those appearing elsewhere in a list. In general, the user may not even *know* whether a particular atom is at the head of a list or not. Thus, when he writes **(INSERT** *EXPR* **BEFORE FOO)**, he means before the atom **FOO**, whether or not it is **CAR** of a list. By setting the variable **UPFINDFLG** to **NIL** (initially **T**), the user can suppress the implicit **UP** that follows searches for atoms, and thus achieve the desired effect. With **UPFINDFLG = NIL**, following **F FOO**, for example, the current expression will be the atom **FOO**. In this case, the **A**, **B**, and **:** operations will operate with respect to the atom **FOO**. If the user intends the operation to refer to the list which **FOO** heads, he simply uses instead the pattern **(FOO --)**.

16.5.4 Extract and Embed

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

(XTR . @) [Editor Command]

Replaces the original current expression with the expression that is current after performing **(LCL . @)** (see warning about **INSERT**, page 16.33). If the current expression after **(LCL . @)** is a *tail* of a higher expression, its first element is used.

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

For example, if the current expression is **(COND ((NULL X) (PRINT Y)))**, **(XTR PRINT)**, or **(XTR 2 2)** will replace the **COND** by the **PRINT**. The current expression after the **XTR** would be **(PRINT Y)**.

If the current expression is **(COND ((NULL X) Y) (T Z))**, then **(XTR Y)** will replace the **COND** with **Y**, even though the current expression after performing **(LCL Y)** is **... Y)**. The current expression after the **XTR** would be **... Y** followed by whatever followed the **COND**.

If the current expression *initially* is a tail, extraction works exactly the same as though the current expression were the first element in that tail. Thus if the current expression is **... (COND ((NULL X) (PRINT Y))) (RETURN Z))**, then **(XTR PRINT)** will replace the **COND** by the **PRINT**, leaving **(PRINT Y)** as the current expression.

The extract command can also incorporate a location specification:

(EXTRACT $@_1$ FROM . $@_2$) [Editor Command]

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

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

For example: If the current expression is **(PRINT (COND ((NULL X) Y) (T Z)))** then following **(EXTRACT Y FROM COND)**, the current expression will be **(PRINT Y)**. **(EXTRACT 2 -1 FROM COND)**, **(EXTRACT Y FROM 2)**, and **(EXTRACT 2 -1 FROM 2)** will all produce the same result.

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

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

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

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

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

Examples:

If the current expression is **(PRINT Y)**, **(MBD (COND ((NULL X) &) ((NULL (CAR Y)) & (GO LP))))** would replace **(PRINT Y)** with **(COND ((NULL X) (PRINT Y)) ((NULL (CAR Y)) (PRINT Y) (GO LP)))**.

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

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

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

The embed command can also incorporate a location specification:

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

(**@** is the segment between **EMBED** and **IN**.) Does **(LC . @)** and then **(MBD . *X*)** (see warning about **INSERT**, page 16.33). Edit chain is not changed, but **UNFIND** is set to the edit chain after the **MBD** was performed.

Examples: **(EMBED PRINT IN SETQ X)**, **(EMBED 3 2 IN RETURN)**, **(EMBED COND 3 1 IN (OR & (NULL X)))**.

WITH can be used for **IN**, and **SURROUND** can be used for **EMBED**, e.g., **(SURROUND NUMBERP WITH (AND & (MINUSP X)))**.

EDITEMBEDTOKEN [Variable]

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

16.5.5 The MOVE Command

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

(MOVE $@_1$ TO *COM* . $@_2$) [Editor Command]

($@_1$ is the segment between **MOVE** and **TO**.) *COM* is **BEFORE**, **AFTER**, or the name of a list command, e.g., :, **N**, etc. Performs **(LC . $@_1$)** (see warning about **INSERT**, page 16.33), and obtains the current expression there (or its first element, if it is a tail), which we will call *EXPR*; **MOVE** then goes back to the original edit chain, performs **(LC . $@_2$)** followed by (*COM EXPR*) (setting an internal flag so *EXPR* is not copied), then goes back to $@_1$ and deletes *EXPR*. The edit chain is not changed. **UNFIND** is set to the edit chain after (*COM EXPR*) was performed.

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

For example, if the current expression is **(A B C D)**, **(MOVE 2 TO AFTER 4)** will make the new current expression be **(A C D B)**. Note that **4** was executed as of the original edit chain, and that the second element had not yet been removed.

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

```
*?
(PROG ((L L)) (EDLOC (CDDR C)) (RETURN (CAR L)))
*(MOVE 3 TO : CAR)
*?
(PROG ((L L)) (RETURN (EDLOC (CDDR C))))
*

*P
... (SELECTQ OBJPR & &) (RETURN &) LP2 (COND & &))
*(MOVE 2 TO N 1)
*P
... (SELECTQ OBJPR & & &) LP2 (COND & &))

*

*P
(OR (EQ X LASTAIL) (NOT &) (AND & & &))
*(MOVE 4 TO AFTER (BELOW COND))
*P
(OR (EQ X LASTAIL) (NOT &))
*\P
... (& &) (AND & & &) (T & &))
*

*P
((NULL X) **COMMENT** (COND & &))
*(-3 (GO NXT]
```

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

Note that in the last example, the user could have added the **PROG** label **NXT** and moved the **COND** in one operation by performing **(MOVE 4 TO N (← PROG) (N NXT))**. Similarly, in the next example, in the course of specifying $@_2$, the location where the expression was to be moved to, the user also performs a structure modification, via **(N (T))**, thus creating the structure that will receive the expression being moved.

```
*P
((CDR &) **COMMENT** (SETQ CL &) (EDITSMASH CL & &))
*MOVE 4 TO N 0 (N (T)) -1]
*P
((CDR &) **COMMENT** (SETQ CL &))
*\P
*(T (EDITSMASH CL & &))
*
```

If $@_2$ is **NIL**, or **(HERE)**, the current position specifies where the operation is to take place. In this case, **UNFIND** is set to where the expression that was moved was originally located, i.e., $@_1$. For example:

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

*P
(PROG (& & & ATM IND VAL) (OR & &)  **COMMENT**  (OR & &)
(PRIN1 & T) (
PRIN1 & T) (SETQ IND     user typed control-E

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

*P
(T (PRIN1 C-EXP T))
```

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

Finally, if $@_1$ is **NIL**, the **MOVE** command allows the user to specify where the *current expression* is to be moved to. In this case, the edit chain is changed, and is the chain where the current expression was moved to; **UNFIND** is set to where it was.

```
*P
(SELECTQ OBJPR (&) (PROGN & &))
*(MOVE TO BEFORE LOOP)
*P
... (SELECTQ OBJPR & &) LOOP (FRPLACA DFPRP &) (FRPLACD
DFPRP
&) (SELECTQ     user typed control-E

*
```

16.5.6 Commands That Move Parentheses

The commands presented in this section permit modification of the list structure itself, as opposed to modifying components thereof. Their effect can be described as inserting or removing a single left or right parenthesis, or pair of left and right parentheses. Of course, there will always be the same number of left parentheses as right parentheses in any list structure, since the parentheses are just a notational guide to the structure provided by **PRINT**. Thus, no command can insert or remove just one parenthesis, but this is suggestive of what actually happens.

In all six commands, *N* and *M* are used to specify an element of a list, usually of the current expression. In practice, *N* and *M* are usually positive or negative integers with the obvious interpretation. However, all six commands use the generalized **NTH** command **(NTH** *COM***)** to find their element(s), so that *N*th element means the first element of the tail found by performing **(NTH** *N***)**. In other words, if the current expression is **(LIST (CAR X) (SETQ Y (CONS W Z)))**, then **(BI 2 CONS)**, **(BI X -1)**, and **(BI X Z)** all specify the exact same operation.

All six commands generate an error if the element is not found, i.e., the **NTH** fails. All are undoable.

(BI *N M***)** [Editor Command]

"Both In". Inserts a left parentheses before the *N*th element and after the *M*th element in the current expression. Generates an error if the *M*th element is not contained in the *N*th tail, i.e., the *M*th element must be "to the right" of the *N*th element.

Example: If the current expression is **(A B (C D E) F G)**, then **(BI 2 4)** will modify it to be **(A (B (C D E) F) G)**.

(BI *N*) [Editor Command]

Same as **(BI *N N*)**.

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

(BO *N*) [Editor Command]

"Both Out". Removes both parentheses from the *N*th element. Generates an error if *N*th element is not a list.

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

(LI *N*) [Editor Command]

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

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

(LO *N*) [Editor Command]

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

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

(RI *N M*) [Editor Command]

"Right In". Inserts a right parenthesis after the *M*th element of the *N*th element. The rest of the *N*th element is brought up to the level of the current expression.

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

(RO *N*) [Editor Command]

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

elements following the *N*th element are moved inside of the *N*th element. Generates an error if *N*th element is not a list.

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

16.5.7 TO and THRU

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

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

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

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

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

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

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

THRU and **TO** are not very useful commands by themselves; they are intended to be used in conjunction with **EXTRACT**, **EMBED**, **DELETE**, **REPLACE**, and **MOVE**. After **THRU** and **TO** have operated, they set an internal editor flag informing the above commands that the element they are operating on is actually a segment, and that the extra pair of parentheses should be removed when the operation is complete. Thus:

***P**
(PROG (& & ATM IND VAL WORD) (PRIN1 & T) (PRIN1 & T) (SETQ IND &)
(SETQ VAL &) **COMMENT (SETQQ** *user typed control-E*

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

*

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

*

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

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

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

*

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

($@_1$ TO) [Editor Command]

($@_1$ THRU) [Editor Command]

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

Examples:

```
*P
(VALUE (RPLACA DEPRP &) (RPLACD &) (RPLACA VARSWORD &)
(RETURN))
*(MOVE (2 TO) TO N (← PROG))
*(N (GO VAR))
*P
(VALUE (GO VAR))
*P
(T **COMMENT** (COND &) **COMMENT** (EDITSMASH CL &
&) (COND &))
*(-3 (GO REPLACE))
*(MOVE (COND TO) TO N ↑ PROG (N REPLACE))
*P
(T **COMMENT** (GO REPLACE))
*\P
(PROG (&) **COMMENT** (COND & & &) (COND & & &) DELETE
(COND & &) REPLACE
(COND &) **COMMENT** (EDITSMASH CL & &) (COND &))
*
*PP
[LAMBDA (CLAUSALA X)
  (PROG (A D)
    (SETQ A CLAUSALA)
  LP (COND
       ((NULL A)
         (RETURN)))
    (SERCH X A)
    (RUMARK (CDR A))
    (NOTICECL (CAR A))
    (SETQ A (CDR A))
    (GO LP]
```

```
*(EXTRACT (SERCH THRU NOT$) FROM PROG)
 = NOTICECL
*P
(LAMBDA (CLAUSALA X) (SERCH X A) (RUMARK &) (NOTICECL
&))
*(EMBED (SERCH TO) IN (MAP CLAUSALA (FUNCTION (LAMBDA
(A) *]
*PP
[LAMBDA (CLAUSALA X)
  (MAP CLAUSALA
    (FUNCTION (LAMBDA (A)
          (SERCH X A)
          (RUMARK (CDR A))
          (NOTICECL (CAR A]
*
```

16.5.8 The R Command

(R *X Y*) [Editor Command]

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

The **R** command operates in conjunction with the search mechanism of the editor. The search proceeds as described on page 16.20, and *X* can employ any of the patterns on page 16.18. Each time *X* matches an element of the structure, the element is replaced by (a copy of) *Y*; each time *X* matches a tail of the structure, the tail is replaced by (a copy of) *Y*.

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

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

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

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

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

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

A. If the user wanted to replace every **D** by **A**, he could perform **(LP (R D A))**.

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

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

Note that the **$** (escape) feature can be used to delete or add characters, as well as replace them. For example, **(R $1 $)** will delete the terminating **1**'s from all literal atoms and strings. Similarly, if an **$** in *X* does not have a mate in *Y*, the characters matched by the **$** are effectively deleted. For example, **(R $/$ $)** will change **AND/OR** to **AND**. There is no similar operation for changing **AND/OR** to **OR**, since the first **$** in *Y* always corresponds to the first **$** in *X*, the second **$** in *Y* to the second in *X*, etc. *Y* can also be a list containing **$**s, e.g., **(R $1 (CAR $))** will change **FOO1** to **(CAR FOO)**, **FIE1** to **(CAR FIE)**.

If *X* does not contain **$**s, **$** appearing in *Y* refers to the *entire* expression matched by *X*, e.g., **(R LONGATOM '$)** changes **LONGATOM** to **'LONGATOM**, **(R (SETQ X &) (PRINT $))** changes every **(SETQ X &)** to **(PRINT (SETQ X &))**. If *X* is a pattern containing an **$** pattern somewhere *within* it, the characters matched by the **$**s are not available, and for the purposes of replacement, the effect is the same as though *X* did not contain any **$**s. For example, if the user types **(R (CAR F$) (PRINT $))**, the second **$** will refer to the entire expression matched by **(CAR F$)**.

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

(RC *X Y***)** [Editor Command]

Equivalent to **(R $*X*$ $*Y*$)**

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

(R1 *X Y***)** [Editor Command]

Find the first instance of *X* and replace it by *Y*.

(RC1 *X Y***)** [Editor Command]

(R1 $*X*$ $*Y*$).

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

(SW *N M*) [Editor Command]

Switches the *N*th and *M*th elements of the current expression.

For example, if the current expression is **(LIST (CONS (CAR X) (CAR Y)) (CONS (CDR X) (CDR Y)))**, **(SW 2 3)** will modify it to be **(LIST (CONS (CDR X) (CDR Y)) (CONS (CAR X) (CAR Y)))**. The relative order of *N* and *M* is not important, i.e., **(SW 3 2)** and **(SW 2 3)** are equivalent.

SW uses the generalized **NTH** command **(NTH *COM*)** to find the *N*th and *M*th elements, a la the **BI-BO** commands.

Thus in the previous example, **(SW CAR CDR)** would produce the same result.

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

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

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

16.6 Commands That Print

PP [Editor Command]

Prettyprints the current expression.

P [Editor Command]

Prints the current expression as though **PRINTLEVEL** (page 25.11) were set to 2.

(P *M*) [Editor Command]

Prints the *M*th element of the current expression as though **PRINTLEVEL** were set to 2.

(P 0) [Editor Command]

Same as **P**.

(P *M N*) [Editor Command]

Prints the *M*th element of the current expression as though **PRINTLEVEL** were set to *N*.

(P 0 *N*) [Editor Command]

Prints the current expression as though **PRINTLEVEL** were set to *N*.

? [Editor Command]

Same as **(P 0 100)**.

Both **(P *M*)** and **(P *M N*)** use the generalized **NTH** command **(NTH *COM*)** to obtain the corresponding element, so that *M* does not have to be a number, e.g., **(P COND 3)** will work. **PP** causes all comments to be printed as ****COMMENT**** (see page 26.43). **P** and **?** print as ****COMMENT**** only those comments that are (top level) elements of the current expression. Lower expressions are not really seen by the editor; the printing command simply sets **PRINTLEVEL** and calls **PRINT**.

PP* [Editor Command]

Prettyprints current expression, *including* comments.

PP* is equivalent to **PP** except that it first resets ****COMMENT**FLG** to **NIL** (see page 26.43).

PPV [Editor Command]

Prettyprints the current expression as a variable, i.e., no special treatment for **LAMBDA**, **COND**, **SETQ**, etc., or for CLISP.

PPT [Editor Command]

Prettyprints the current expression, printing CLISP translations, if any.

?= [Editor Command]

Prints the argument names and corresponding values for the current expression. Analagous to the **?=** break command (page 14.7). For example,

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

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

The command **MAKE** (page 16.57) is an imperative form of **? =** . It allows the user to specify a change to the element of the current expression that corresponds to a particular argument name.

All printing functions print to the terminal, regardless of the primary output file. All use the readtable **T**. No printing function ever changes the edit chain. All record the current edit chain for use by **\P** (page 16.28). All can be aborted with control-E.

16.7 Commands for Leaving the Editor

OK [Editor Command]

Exits from the editor.

STOP [Editor Command]

Exits from the editor with an error. Mainly for use in conjunction with **TTY:** commands (page 16.51) that the user wants to abort.

Since all of the commands in the editor are errorset protected, the user must exit from the editor via a command. **STOP** provides a way of distinguishing between a successful and unsuccessful (from the user's standpoint) editing session. For example, if the user is executing **(MOVE 3 TO AFTER COND TTY:)**, and he exits from the lower editor with an **OK**, the **MOVE** command will then complete its operation. If the user wants to abort the **MOVE** command, he must make the **TTY:** command generate an error. He does this by exiting from the lower editor with a **STOP** command. In this case, the higher editor's edit chain will not be changed by the **TTY:** command.

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

SAVE [Editor Command]

Exits from the editor and saves the "state of the edit" on the property list of the function or variable being edited under the property **EDIT-SAVE**. If the editor is called again on the same

structure, the editing is effectively "continued," i.e., the edit chain, mark list, value of **UNFIND** and **UNDOLST** are restored.

For example:

```
*P
(NULL X)
*F COND P
(COND (& &) (T &))
*SAVE
FOO
←.
 .
 .
←EDITF(FOO)
EDIT
*P
(COND (& &) (T &))
*\P
(NULL X)
*
```

SAVE is necessary only if the user is editing many different expressions; an exit from the editor via **OK** always saves the state of the edit of that call to the editor on the property list of the atom **EDIT**, under the property name **LASTVALUE**. **OK** also remprops **EDIT-SAVE** from the property list of the function or variable being edited.

Whenever the editor is entered, it checks to see if it is editing the same expression as the last one edited. In this case, it restores the mark list and **UNDOLST**, and sets **UNFIND** to be the edit chain as of the previous exit from the editor. For example:

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

```
(LAMBDA (X) (PROG & & LP & & & &))
*\P
(COND & &)
*
```

Furthermore, as a result of the history feature, if the editor is called on the same expression within a certain number of **LISPX** inputs (namely, the size of the history list, which can be changed with **CHANGESLICE**, page 13.21) the state of the edit of that expression is restored, regardless of how many other expressions may have been edited in the meantime. For example:

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

Thus the user can always continue editing, including undoing changes from a previous editing session, if (1) No other expressions have been edited since that session (since saving takes place at *exit* time, intervening calls that were aborted via control-D or exited via **STOP** will not affect the editor's memory); or (2) That session was "sufficiently" recent; or (3) It was ended with a **SAVE** command.

16.8 Nested Calls to Editor

TTY: [Editor Command]

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

The **TTY:** command is extremely useful. It enables the user to set up a complex operation, and perform interactive attention-changing commands part way through it. For example the command **(MOVE 3 TO AFTER COND 3 P TTY:)** allows the user to interact, in effect, *within* the **MOVE** command. Thus he can verify for himself that the correct location has been found, or complete the specification "by hand." In effect, **TTY:** says "I'll tell you what you should do when you get there."

The **TTY:** command operates by printing **TTY:** and then calling the editor. The initial edit chain in the lower editor is the one that existed in the higher editor at the time the **TTY:** command was entered. Until the user exits from the lower editor, any attention changing commands he executes only affect the lower editor's edit chain. Of course, if the user performs any structure modification commands while under a **TTY:** command, these will modify the structure in both editors, since it is the same structure. When the **TTY:** command finishes, the lower editor's edit chain becomes the edit chain of the higher editor.

EF [Editor Command]

EV [Editor Command]

EP [Editor Command]

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

16.9 Manipulating the Characters of an Atom or String

RAISE [Editor Command]

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

LOWER [Editor Command]

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

CAP [Editor Command]

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

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

(RAISE *X*) [Editor Command]

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

(LOWER *X*) [Editor Command]

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

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

REPACK [Editor Command]

Permits the "editing" of an atom or string.

REPACK operates by calling the editor recursively on **UNPACK** of the current expression, or if it is a list, on **UNPACK** of its first element. If the lower editor is exited successfully, i.e., via **OK** as opposed to **STOP**, the list of atoms is made into a single atom or string, which replaces the atom or string being "repacked." The new atom or string is always printed.

Example:

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

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

(REPACK @) [Editor Command]

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

16.10 Manipulating Predicates and Conditional Expressions

JOINC [Editor Command]

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

does an **(F COND T)** first so that you don't have to be at the first **COND**.

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

Splits one **COND** into two. *X* specifies the last clause in the first **COND**, e.g. **(SPLITC 3)** splits **(COND** $CLAUSE_1$ $CLAUSE_2$ $CLAUSE_3$ $CLAUSE_4$**)** into **(COND** $CLAUSE_1$ $CLAUSE_2$**) (COND** $CLAUSE_3$ $CLAUSE_4$**)**. Uses the generalized **NTH** command **(NTH** *COM***)**, so that *X* does not have to be a number, e.g., the user can say **(SPLITC RETURN)**, meaning split after the clause containing **RETURN**. **SPLITC** also does an **(F COND T)** first.

NEGATE [Editor Command]

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

NEGATE is implemented via the function **NEGATE** (page 3.20).

SWAPC [Editor Command]

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

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

16.11 History commands in the editor

All of the user's inputs to the editor are stored on the history list **EDITHISTORY** (see page 13.43, the editor's history list, and all of the programmer's assistant commands for manipulating the history list, e.g. **REDO**, **USE**, **FIX**, **NAME**, etc., are available for use on events on **EDITHISTORY**. In addition, the following four history commands are recognized specially by the editor. They always operate on the last, i.e. most recent, event.

DO *COM* [Editor Command]

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

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

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

!F [Editor Command]

Same as **DO F**.

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

!E [Editor Command]

Same as **DO E**.

!N [Editor Command]

Same as **DO N**.

16.12 Miscellaneous Commands

NIL [Editor Command]

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

CL [Editor Command]

Clispifies the current expression (see page 21.22).

DW [Editor Command]

Dwimifies the current expression (see page 21.18).

IFY [Editor Command]

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

GET* [Editor Command]

If the current expression is a comment pointer (see page 26.44), reads in the full text of the comment, and replaces the current expression by it.

(* . *X*) [Editor Command]

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

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

then **(* CALL FACT RECURSIVELY)** would insert **(* CALL FACT RECURSIVELY)** *before* the **ITIMES** expression. If inserted after the **ITIMES**, the comment would then be (incorrectly) returned as the value of the **COND**. However, if the **COND** was itself a **PROG** statement, and hence its value was not being used, the comment could be (and would be) inserted after the **ITIMES** expression.

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

GETD [Editor Command]

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

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

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

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

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

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

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

(MAKE *ARGNAME EXP***)** [Editor Command]

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

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

Q [Editor Command]

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

D [Editor Command]

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

16.13 Commands That Evaluate

E [Editor Command]

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

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

(FIE BROKEN)
```

:

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

(E *X*) [Editor Command]

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

(E *X* T) [Editor Command]

Same as **(E *x*)** but does not print.

The **(E *X*)** and **(E *X* T)** commands are mainly intended for use by macros and subroutine calls to the editor; the user would probably type in a form for evaluation using the more convenient format of the (atomic) **E** command.

(I *C* X_1 ... X_N) [Editor Command]

Executes the *editor command* (*C* Y_1 ... Y_N) where Y_i = (EVAL X_i). If *C* is not an atom, *C* is evaluated also.

Examples:

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

(I N FOO (CAR FIE)) will attach the value of **FOO** and **CAR** of the value of **FIE** to the end of the current expression.

(I F= FOO T) will search for an expression **EQ** to the value of **FOO**.

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

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

EVAL [Editor Command]

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

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

GETVAL [Editor Command]

Replaces the current expression by the result of evaluating it.

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

An nlambda, nospread function (not a command). Its value is what the current expression would be after executing the edit commands COM_1 ... COM_N starting from the present edit chain. Generates an error if any of COM_1 thru COM_N cause errors. The current edit chain is never changed.

Note: The **A**, **B**, **:**, **INSERT**, **REPLACE**, and **CHANGE** commands make special checks for ## forms in the expressions used for inserting or replacing, and use a copy of ## form instead (see page 16.34). Thus, **(INSERT (## 3 2) AFTER 1)** is equivalent to **(I INSERT (COPY (## 3 2)) 'AFTER 1)**.

Example: **(I R 'X (## (CONS .. Z)))** replaces all **X**'s in the current expression by the first **CONS** containing a **Z**.

The **I** command is not very convenient for computing an *entire* edit command for execution, since it computes the command name and its arguments separately. Also, the **I** command cannot be used to compute an atomic command. The following two commands provide more general ways of computing commands.

(COMS X_1 ... X_M) [Editor Command]

Each X_i is evaluated and its value is executed as a command.

For example, **(COMS (COND (X (LIST 1 X))))** will replace the first element of the current expression with the value of **X** if non-**NIL**, otherwise do nothing. The editor command **NIL** is a no-op (page 16.55).

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

Executes COM_1 ... COM_N.

COMSQ is mainly useful in conjunction with the **COMS** command. For example, suppose the user wishes to compute an entire list of commands for evaluation, as opposed to computing each command one at a time as does the **COMS** command. He would then write **(COMS (CONS 'COMSQ *X*))** where *X* computed the list of commands, e.g., **(COMS (CONS 'COMSQ (GETP FOO 'COMMANDS)))**.

16.14 Commands That Test

(IF *X*) [Editor Command]

Generates an error *unless* the value of **(EVAL *X*)** is true. In other words, if **(EVAL *X*)** causes an error or **(EVAL *X*) = NIL**, **IF** will cause an error.

For some editor commands, the occurrence of an error has a well defined meaning, i.e., they use errors to branch on, as **COND** uses **NIL** and non-**NIL**. For example, an error condition in a location specification may simply mean "not this one, try the next." Thus the location specification **(IPLUS (E (OR (NUMBERP (## 3)) (ERROR!)) T))** specifies the first **IPLUS** whose second argument is a number. The **IF** command, by equating **NIL** to error, provides a more natural way of accomplishing the same result. Thus, an equivalent location specification is **(IPLUS (IF (NUMBERP (## 3))))**.

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

(IF *X* $COMS_1$ $COMS_2$) [Editor Command]

If **(EVAL *X*)** is true, execute $COMS_1$; if **(EVAL *X*)** causes an error or is equal to **NIL**, execute $COMS_2$.

Thus **IF** is equivalent to

```
(COMS (CONS 'COMSQ
      (COND
        ((CAR (NLSETQ (EVAL X)))
          COMS1)
        (T COMS2))))
```

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

(IF *X* $COMS_1$) [Editor Command]

If **(EVAL *X*)** is true, execute $COMS_1$; otherwise generate an error.

(LP $COMS_1$ ... $COMS_N$) [Editor Command]

Repeatedly executes $COMS_1$ ... $COMS_N$ until an error occurs.

For example, **(LP F PRINT (N T))** will attach a **T** at the end of every **PRINT** expression. **(LP F PRINT (IF (## 3) NIL ((N T))))** will attach a **T** at the end of each print expression which does not already have a second argument. The form **(## 3)** will cause an error if the edit command **3** causes an error, thereby selecting **((N T))** as

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

When an error occurs, **LP** prints *N* **OCCURRENCES** where *N* is the number of times the commands were successfully executed. The edit chain is left as of the last complete successful execution of $COMS_1$ ... $COMS_N$.

(LPQ $COMS_1$ ... $COMS_N$**)** [Editor Command]

Same as **LP** but does not print the message *N* **OCCURRENCES**.

In order to prevent non-terminating loops, both **LP** and **LPQ** terminate when the number of iterations reaches **MAXLOOP**, initially set to 30. **MAXLOOP** can be set to **NIL**, which is equivalent to setting it to infinity. Since the edit chain is left as of the last successful completion of the loop, the user can simply continue the **LP** command with **REDO** (page 13.8).

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

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

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

Like **SHOW** except calls the editor recursively (via the **TTY:** command, see page 16.51) on each instance of the indicated espression(s) so that the user can examine and/or change them.

(ORR $COMS_1$ ... $COMS_N$**)** [Editor Command]

ORR begins by executing $COMS_1$, a list of commands. If no error occurs, **ORR** is finished. Otherwise, **ORR** restores the edit chain to its original value, and continues by executing $COMS_2$, etc. If none of the command lists execute without errors, i.e., the **ORR** "drops off the end", **ORR** generates an error. Otherwise, the edit chain is left as of the completion of the first command list which executes without an error.

NIL as a command list is perfectly legal, and will always execute successfully. Thus, making the last "argument" to **ORR** be **NIL** will insure that the **ORR** never causes an error. Any other atom is treated as *(ATOM)*, i.e., the above example could be written as **(ORR NX !NX NIL)**.

For example, **(ORR (NX) (!NX) NIL)** will perform a **NX**, if possible, otherwise a **!NX**, if possible, otherwise do nothing. Similarly, **DELETE** could be written as **(ORR (UP (1)) (BK UP (2)) (UP (: NIL)))**.

16.15 Edit Macros

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

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

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

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

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

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

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

C an atom. **M** defines *C* as a list command. Executing **(***C* E_1 ... E_N**)** is then performed by substituting E_1 for ARG_1, ... E_N for ARG_N throughout $COMS_1$ ... $COMS_M$, and then executing $COMS_1$ ... $COMS_M$.

For example, we could define a more general **BP** by **(M (BP) (N) (BK N) UP P)**. Thus, **(BP 3)** would perform **(BK 3)**, followed by an **UP**, followed by a **P**.

A list command can be defined via a macro so as to take a fixed or indefinite number of "arguments", as with spread vs. nospread functions. The form given above specified a macro with a fixed number of arguments, as indicated by its argument list. If the "argument list" is *atomic*, the command takes an indefinite number of arguments.

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

If *C*, *ARG* are both atoms, this defines *C* as a list command. Executing **(***C* E_1 ... E_N**)** is performed by substituting **(***E_1 ... E_N***)**, i.e.,

CDR of the command, for *ARG* throughout $COMS_1$ ... $COMS_M$, and then executing $COMS_1$ ... $COMS_M$.

For example, the command **2ND** (page 16.24), could be defined as a macro by **(M (2ND) X (ORR ((LC . X) (LC . X))))**.

Note that for all editor commands, "built in" commands as well as commands defined by macros as atomic commands and list definitions are *completely* independent. In other words, the existence of an atomic definition for *C* in *no* way affects the treatment of *C* when it appears as **CAR** of a list command, and the existence of a list definition for *C* in *no* way affects the treatment of *C* when it appears as an atom. In particular, *C* can be used as the name of either an atomic command, or a list command, or both. In the latter case, two entirely different definitions can be used.

Note also that once *C* is defined as an atomic command via a macro definition, it will *not* be searched for when used in a location specification, unless it is preceded by an **F**. Thus **(INSERT -- BEFORE BP)** would not search for **BP**, but instead perform a **BK**, and **UP**, and a **P**, and then do the insertion. The corresponding also holds true for list commands.

Occasionally, the user will want to employ the **S** command in a macro to save some temporary result. For example, the **SW** command could be defined as:

```
(M (SW) (N M)
  (NTH N)
  (S FOO 1)
  MARK
  0
  (NTH M)
  (S FIE 1)
  (I 1 FOO)
  ←←
  (I 1 FIE))
```

Since this version of **SW** sets **FOO** and **FIE**, using **SW** may have undesirable side effects, especially when the editor was called from deep in a computation, we would have to be careful to make up unique names for dummy variables used in edit macros, which is bothersome. Furthermore, it would be impossible to define a command that called itself recursively while setting free variables. The **BIND** command solves both problems.

(BIND $COMS_1$ ... $COMS_N$**)** [Editor Command]

Binds three dummy variables **#1**, **#2**, **#3**, (initialized to **NIL**), and then executes the edit commands $COMS_1$ ... $COMS_N$. **BIND** uses a **PROG** to make these bindings, so they are only in effect while the

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

Thus, we can write **SW** safely as:

```
(M (SW) (N M)
  (BIND (NTH N)
     (S #1 1)
     MARK
     0
     (NTH M)
     (S #2 1)
     (I 1 #1)
     ←←
     (I 1 #2)))
```

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

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

User macros are stored on a list **USERMACROS**. The file package command **USERMACROS** (page 17.34), is available for dumping all or selected user macros.

16.16 Undo

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

UNDO [Editor Command]

Undoes the last, i.e., most recent, structure modification command that has not yet been undone, and prints the name of that command, e.g., **MBD undone**. The edit chain is then *exactly* what it was before the "undone" command had been performed. If there are no commands to undo, **UNDO** types **nothing saved**.

!UNDO [Editor Command]

Undoes all modifications performed during this editing session, i.e. this call to the editor. As each command is undone, its name

is printed a la **UNDO**. If there is nothing to be undone, **!UNDO** prints **nothing saved.**

Undoing an event containing an **I**, **E**, or **S** command will also undo the side effects of the evaluation(s), e.g., undoing **(I 3 (/NCONC FOO FIE))** will not only restore the 3rd element but also restore **FOO**. Similarly, undoing an **S** command will undo the set. See the discussion of **UNDO** in page 13.13. (Note that if the **I** command was typed directly to the editor, **/NCONC** would automatically be substituted for **NCONC** as described in page 13.27.)

Since **UNDO** and **!UNDO** cause structure modification, they also add an entry to **UNDOLST**. However, **UNDO** and **!UNDO** entries are skipped by **UNDO**, e.g., if the user performs an **INSERT**, and then an **MBD**, the first **UNDO** will undo the **MBD**, and the second will undo the **INSERT**. However, the user can also specify precisely which commands he wants undone by identifying the corresponding entry. In this case, he can undo an **UNDO** command, e.g., by typing **UNDO UNDO**, or undo a **!UNDO** command, or undo a command other than that most recently performed.

Whenever the user *continues* an editing session, the undo information of the previous session is protected by inserting a special blip, called an undo-block, on the front of **UNDOLST**. This undo-block will terminate the operation of a **!UNDO**, thereby confining its effect to the current session, and will similarly prevent an **UNDO** command from operating on commands executed in the previous session.

Thus, if the user enters the editor continuing a session, and immediately executes an **UNDO** or **!UNDO**, the editor will type **BLOCKED** instead of **NOTHING SAVED**. Similarly, if the user executes several commands and then undoes them all, another **UNDO** or **!UNDO** will also cause **BLOCKED** to be typed.

UNBLOCK [Editor Command]

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

TEST [Editor Command]

Adds an undo-block at the front of **UNDOLST**.

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

(UNDO *EventSpec*) [Editor Command]

EventSpec is an event specification (see page 13.6). Undoes the indicated event on the history list. In this case, the event does not have to be in the current editing session, even if the previous session has not been unblocked as described above. However, the user does have to be editing the same expression as was being edited in the indicated event.

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

16.17 EDITDEFAULT

Whenever a command is not recognized, i.e., is not "built in" or defined as a macro, the editor calls an internal function, **EDITDEFAULT**, to determine what action to take. Since **EDITDEFAULT** is part of the edit block, the user cannot advise or redefine it as a means of augmenting or extending the editor. However, the user can accomplish this via **EDITUSERFN**. If the value of the variable **EDITUSERFN** is **T**, **EDITDEFAULT** calls the function **EDITUSERFN** giving it the command as an argument. If **EDITUSERFN** returns a non-**NIL** value, its value is interpreted as a single command and executed. Otherwise, the error correction procedure described below is performed.

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

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

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

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

(1) If the command is one of the list commands, i.e., a member of **EDITCOMSL**, and there is additional input on the same terminal line, treat the entire line as a single list command. The line is read using **READLINE** (page 13.36), so the line can be terminated by a square bracket, or by a carriage return not preceded by a space. The user may omit parentheses for any list command typed in at the top level (provided the command is not also an atomic command, e.g. **NX, BK**). For example,

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

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

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

MOVE ?
*
```

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

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

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

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

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

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

```
(FOO)
*
```

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

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

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

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

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

(6) Otherwise, generate an error.

16.18 Editor Functions

(EDIT *NAME* **—)** [Function]

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

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

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

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

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

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

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

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

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

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

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

(5) If *NAME* does not have an expr definition, but has an **EXPR** property, **EDITF** prints **prop**, and calls **EDITE** (page 16.71 to edit this saved expr definition. In this case, if the edit completes and no changes have been made, **EDITE** prints "**not changed, so not unsaved.**" If changes were made, but the value of **DFNFLG** (page 10.10) is **PROP**, **EDITE** prints "**changed, but not unsaved.**" Otherwise if changes were made, **EDITE** prints **unsaved** and does an **UNSAVEDEF** (page 17.28).

(6) If *NAME* neither has an expr definition nor an **EXPR** property, and the file package "knows" which file *NAME* is contained in (see **EDITLOADFNS?**, page 16.73), the expr definition of *NAME* is automatically loaded (using **LOADFNS**, page 17.6) onto the **EXPR** property, and **EDITE** proceeds as described above. Because of the existence of file maps (page 17.55), this operation is extremely fast, essentially requiring only the time to perform the **READ** to obtain the actual definition. In addition, if *NAME* is a member of a block, the user will be asked whether he wishes the rest of the functions in the block to be loaded at the same time.

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

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

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

(9) Otherwise, the editor generates an *NAME* **not editable** error.

In all cases, if a function is edited, and changes were made, the function is time-stamped (by **EDITE**), which consists of inserting a comment of the form **(*** *USERS-INITIALS DATE*) (see page 16.76). If the function was already time-stamped, then only the date is changed.

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

An nlambda, nospread function, used to perform the same editing operations on several functions. *NAME* is evaluated to obtain a list of functions. If *NAME* is atomic, and its value is not a list, and it is the name of a file, **(FILEFNSLST '***NAME***)** will be used as the list of functions to be edited.

COM_1, COM_2, ..., COM_N are (optional) edit commands. **EDITFNS** maps down the list of functions, prints the name of each function, and calls the editor (via **EDITF**) on that function. The value of **EDITFNS** is **NIL**.

For example, **(EDITFNS FOOFNS (R FIE FUM))** will change every **FIE** to **FUM** in each of the functions on **FOOFNS**.

The call to the editor is **ERRORSET** protected (page 14.21), so that if the editing of one function causes an error, **EDITFNS** will proceed to the next function. In particular, if an error occurred while editing a function via its **EXPR** property, the function would not be unsaved. Thus in the above example, if one of the functions did not contain a **FIE**, the **R** command would cause an error, it would not be unsaved, and editing would continue with the next function.

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

Similar to **EDITF**, for editing values of variables. *NAME* is the name of the variable, COM_1, COM_2, ..., COM_n are (optional) edit commands.

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

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

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

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

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

Similar to **EDITF** for editing property lists. If the property list of *NAME* is **NIL**, **EDITP** attempts spelling correction using **USERWORDS** (unless **DWIMFLG** = **NIL**). Then **EDITP** calls **EDITE** on the property list of *NAME*, (or the corrected spelling thereof), with *TYPE* = **PROPLST**.

EDITP returns the atom whose property list was edited.

(EDITE *EXPR COMS ATM TYPE IFCHANGEDFN***)** [Function]

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

ATM and *TYPE* are for use in conjunction with the file package. If supplied, *ATM* is the *name* of the object that *EXPR* is associated with, and *TYPE* describes the association (i.e., *TYPE* corresponds to the *TYPE* argument of **MARKASCHANGED**, page 17.17.) For example, if *EXPR* is the definition of **FOO**, *ATM* = **FOO** and *TYPE* = **FNS**. When **EDITE** is called from **EDITP**, *EXPR* is the property list of *ATM*, and *TYPE* = **PROPLST**, etc.

EDITE calls **EDITL** to do the editing (described below). Upon return, if both *ATM* and *TYPE* are non-**NIL**, **ADDSPELL** is called to add *ATM* to the appropriate spelling list. Then, if *EXPR* was changed, and the value of *IFCHANGEDFN* is not **NIL**, the value of *IFCHANGEDFN* is applied to the arguments *ATM*, *EXPR*, *TYPE*, and a flag which is **T** for normal edits from editor, **NIL** for calls that were aborted via control-D or **STOP**. Otherwise, if *EXPR* was changed, and the value of *IFCHANGEDFN* is **NIL**, and *TYPE* is not **NIL**, **MARKASCHANGED** (page 17.17) is called on *ATM* and *TYPE*.

EDITE uses **RESETSAVE** to insure that *IFCHANGEDFN* and **MARKASCHANGED** are called if any change was made even if editing is subsequently aborted via control-D. (In this case, the fourth argument to *IFCHANGEDFN* will be **NIL**.)

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

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

EDITL *is* the editor. Its first argument is the edit chain, and its value is an edit chain, namely the value of *L* at the time **EDITL** is exited. *L* is a **SPECVAR**, and so can be examined or set by edit commands. For example, ↑ is equivalent to **(E (SETQ L (LAST L)) T)**. However, the user should only manipulate or examine *L* directly as a last resort, and then with caution.

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

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

ATM is optional. On calls from **EDITF**, it is the name of the function being edited; on calls from **EDITV**, the name of the variable, and calls from **EDITP**, the atom whose property list is being edited. The property list of *ATM* is used by the **SAVE** command for saving the state of the edit. Thus **SAVE** will not save anything if *ATM* = **NIL**, i.e., when editing arbitrary expressions via **EDITE** or **EDITL** directly.

EDITCHANGES is used for communicating with **EDITE**.

(EDITL0 *L COMS MESS* —**)** [Function]

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

(EDIT4E *PAT X* —**)** [Function]

The editor's pattern match routine. Returns **T**, if *PAT* matches *X*. See page 16.18 for definition of "match".

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

(EDITFPAT *PAT* —**)** [Function]

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

(EDITFINDP *X PAT FLG***)** [Function]

Allows a program to use the edit find command as a pure predicate from outside the editor. *X* is an expression, *PAT* a pattern. The value of **EDITFINDP** is **T** if the command **F** *PAT would* succeed, **NIL** otherwise. **EDITFINDP** calls **EDITFPAT** to convert *PAT* to the form expected by **EDIT4E**, unless *FLG* = **T**. Thus, if the program is applying **EDITFINDP** to several different expressions using the same pattern, it will be more efficient to call **EDITFPAT** once, and then call **EDITFINDP** with the converted pattern and *FLG* = **T**.

(ESUBST *NEW OLD EXPR ERRORFLG CHARFLG***)** [Function]

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

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

ESUBST is always undoable.

(EDITLOADFNS? *FN STR ASKFLG FILES***)** [Function]

FN is the name of a function. **EDITLOADFNS?** returns the name of file *FN* is contained in, or **NIL** if no file is found.

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

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

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

Having decided which file the function is on, if *ASKFLG* = **NIL**, **EDITLOADFNS?** prints the value of *STR* followed by the name of the file, and returns the name of the file. If *ASKFLG* = **NIL** and *STR* = **NIL**, **EDITLOADFNS?** prints "**loading definition of** *FN* **from** *FILENAME*."

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

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

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

(EDITCALLERS *ATOMS FILES COMS***)** [Function]

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

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

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

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

(FINDCALLERS *ATOMS FILES***)** [Function]

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

EDITRACEFN [Variable]

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

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

EDITRACEFN is initially **NIL**.

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

Used to set up the immediate read macros used by the editor, as well as the control-Y read macro (page 25.42). *NEXTCHAR*, *BKCHAR*, *LASTCHAR*, *2CHAR* and *PPCHAR* specify which control character should perform the edit commands **NXP**, **BKP**, **-1P**, **2P** and **PP***, respectively; *UNQUOTECHAR* corresponds to control-Y. For each non-**NIL** argument, **SETTERMCHARS** makes the corresponding control character have the indicated function. The arguments to **SETTERMCHARS** can be character codes, the control characters themselves, or the alphabetic letters corresponding to the control characters.

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

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

16.19 Time Stamps

Whenever a function is edited, and changes were made, the function is time-stamped (by **EDITE**), which consists of inserting a comment of the form **(*** *USERS-INITIALS DATE*). *USERS-INITIALS* is the value of the variable **INITIALS**. After greeting (page 12.1), the function **SETINITIALS** is called. **SETINITIALS** searches **INITIALSLST**, a list of elements of the form (*USERNAME . INITIALS*) or (*USERNAME FIRSTNAME INITIALS*). If the user's name is found, **INITIALS** is set accordingly. If the user's name is *not* found on **INITIALSLST**, **INITIALS** is set to the value of **DEFAULTINITIALS**, initially **edited:**. Thus, the default is to always time stamp. To suppress time stamping, the user must either include an entry of the form (*USERNAME*) on **INITIALSLST**, or set **DEFAULTINITIALS** to **NIL** before greeting, i.e. in his user profile, or else, *after* greeting, explicitly set **INITIALS** to **NIL**.

If the user wishes his functions to be time stamped with his initials when edited, he should include a file package command command of the form **(ADDVARS (INITIALSLST (***USERNAME . INITIALS***)))** in the user's **INIT.LISP** file (see page 12.2).

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

[This page intentionally left blank]

17. FILE PACKAGE

*Warning: The subsystem within the Interlisp-D environment used for managing collections of definitions (of functions, variables, etc.) is known as the "File Package." This terminology is confusing, because the word "file" is also used in the more conventional sense as meaning a collection of data stored some physical media. Unfortunately, it is not possible to change this terminology at this time, because many functions and variables (***MAKEFILE, FILEPKGTYPES****, etc.) incorporate the word "file" in their names. Eventually, the system and the documentation will be revamped to consistantly use the term "module" or "definition group" or "defgroup."*

Most implementations of Lisp treat symbolic files as unstructured text, much as they are treated in most conventional programming environments. Function definitions are edited with a character-oriented text editor, and then the changed definitions (or sometimes the entire file) is read or compiled to install those changes in the running memory image. Interlisp incorporates a different philosophy. A symbolic file is considered as a database of information about a group of data objects---function definitions, variable values, record declarations, etc. The text in a symbolic file is never edited directly. Definitions are edited only after their textual representations on files have been converted to data-structures that reside inside the Lisp address space. The programs for editing definitions inside Interlisp can therefore make use of the full set of data-manipulation capabilities that the environment already provides, and editing operations can be easily intermixed with the processes of evaluation and compilation.

Interlisp is thus a "resident" programming environment, and as such it provides facilities for moving definitions back and forth between memory and the external databases on symbolic files, and for doing the bookkeeping involved when definitions on many symbolic files with compiled counterparts are being manipulated. The file package provides those capabilities. It removes from the user the burden of keeping track of where things are and what things have changed. The file package also keeps track of which files have been modified and need to be updated and recompiled.

The file package is integrated into many other system packages. For example, if only the compiled version of a file is loaded and the user attempts to edit a function, the file package will attempt to load the source of that function from the appropriate symbolic file. In many cases, if a datum is needed by some

program, the file package will automatically retrieve it from a file if it is not already in the user's working environment.

Some of the operations of the file package are rather complex. For example, the same function may appear in several different files, or the symbolic or compiled files may be in different directories, etc. Therefore, this chapter does not document how the file package works in each and every situation, but instead makes the deliberately vague statement that it does the "right" thing with respect to keeping track of what has been changed, and what file operations need to be performed in accordance with those changes.

For a simple illustration of what the file package does, suppose that the symbolic file **FOO** contains the functions **FOO1** and **FOO2**, and that the file **BAR** contains the functions **BAR1** and **BAR2**. These two files could be loaded into the environment with the function **LOAD**:

```
←(LOAD 'FOO)
FILE CREATED 4-MAR-83 09:26:55
FOOCOMS
{DSK}FOO.;1
←(LOAD 'BAR)
FILE CREATED 4-MAR-83 09:27:24
BARCOMS
{DSK}BAR.;1
```

Now, suppose that we change the definition of **FOO2** with the editor, and we define two new functions, **NEW1** and **NEW2**. At that point, the file package knows that the in-memory definition of **FOO2** is no longer consistent with the definition in the file **FOO**, and that the new functions have been defined but have not yet been associated with a symbolic file and saved on permanent storage. The function **FILES?** summarizes this state of affairs and enters into an interactive dialog in which we can specify what files the new functions are to belong to.

```
←(FILES?)
FOO...to be dumped.
  plus the functions: NEW1,NEW2
want to say where the above go ? Yes
(functions)
NEW1  File name: BAR
NEW2  File name: ZAP
 new file ? Yes
NIL
```

The file package knows that the file **FOO** has been changed, and needs to be dumped back to permanent storage. This can be done with **MAKEFILE**.

```
←(MAKEFILE 'FOO)
{DSK}FOO.;2
```

Since we added **NEW1** to the old file **BAR** and established a new file **ZAP** to contain **NEW2**, both **BAR** and **ZAP** now also need to be dumped. This is confirmed by a second call to **FILES?**:

← (FILES?)
BAR, ZAP...to be dumped.
FOO...to be listed.
FOO...to be compiled
NIL

We are also informed that the new version we made of **FOO** needs to be listed (sent to a printer) and that the functions on the file must be compiled.

Rather than doing several **MAKEFILE**s to dump the files **BAR** and **ZAP**, we can simply call **CLEANUP**. Without any further user interaction, this will dump any files whose definitions have been modified. **CLEANUP** will also send any unlisted files to the printer and recompile any files which need to be recompiled. **CLEANUP** is a useful function to use at the end of a debugging session. It will call **FILES?** if any new objects have been defined, so the user does not lose the opportunity to say explicitly where those belong. In effect, the function **CLEANUP** executes all the operations necessary to make the user's permanent files consistent with the definitions in his current core-image.

← (CLEANUP)
FOO...compiling {DSK}FOO.;2
.
.
.
BAR...compiling {DSK}BAR.;2
.
.
.
ZAP...compiling {DSK}ZAP.;1
.
.
.

In addition to the definitions of functions, symbolic files in Interlisp can contain definitions of a variety of other types, e.g. variable values, property lists, record declarations, macro definitions, hash arrays, etc. In order to treat such a diverse assortment of data uniformly from the standpoint of file operations, the file package uses the concept of a *typed definition*, of which a function definition is just one example. A typed definition associates with a name (usually a litatom), a definition of a given type (called the file package type). Note that the same name may have several definitions of different types. For example, a litatom may have both a function

definition and a variable definition. The file package also keeps track of the files that a particular typed definition is stored on, so one can think of a typed definition as a relation between four elements: a name, a definition, a type, and a file.

Symbolic files on permanent storage devices are referred to by names that obey the naming conventions of those devices, usually including host, directory, and version fields. When such definition groups are noticed by the file package, they are assigned simple *root names* and these are used by all file package operations to refer to those groups of definitions. The root name for a group is computed from its full permanent storage name by applying the function **ROOTFILENAME**; this strips off the host, directory, version, etc., and returns just the simple name field of the file. For each file, the file package also has a data structure that describes what definitions it contains. This is known as the commands of the file, or its "filecoms". By convention, the filecoms of a file whose root name is *X* is stored as the value of the litatom *X***COMS**. For example, the value of **FOOCOMS** is the filecoms for the file **FOO**. This variable can be directly manipulated, but the file package contains facilities such as **FILES?** which make constructing and updating filecoms easier, and in some cases automatic. See page 17.48.

The file package is able to maintain its databases of information because it is notified by various other routines in the system when events take place that may change that database. A file is "noticed" when it is loaded, or when a new file is stored (though there are ways to explicitly notice files without completely loading all their definitions). Once a file is noticed, the file package takes it into account when modifying filecoms, dumping files, etc. The file package also needs to know what typed definitions have been changed or what new definitions have been introduced, so it can determine which files need to be updated. This is done by "marking changes". All the system functions that perform file package operations (**LOAD**, **TCOMPL**, **PRETTYDEF**, etc.), as well as those functions that define or change data, (**EDITF**, **EDITV**, **EDITP**, DWIM corrections to user functions) interact with the file package. Also, *typed-in* assignment of variables or property values is noticed by the file package. (Note that modifications to variable or property values during the execution of a function body are not noticed.) In some cases the marking procedure can be subtle, e.g. if the user edits a property list using **EDITP**, only those properties whose values are actually changed (or added) are marked.

All file package operations can be disabled with **FILEPKGFLG**.

FILEPKGFLG [Variable]

The file package can be disabled by setting **FILEPKGFLG** to **NIL**. This will turn off noticing files and marking changes. **FILEPKGFLG** is initially **T**.

The rest of this chapter goes into further detail about the file package. Functions for loading and storing symbolic files are presented first, followed by functions for adding and removing typed definitions from files, moving typed definitions from one file to another, determining which file a particular definition is stored in, and so on.

17.1 Loading Files

The functions below load information from symbolic files into the Interlisp environment. A symbolic file contains a sequence of Interlisp expressions that can be evaluated to establish specified typed definitions. The expressions on symbolic files are read using **FILERDTBL** as the read table.

The loading functions all have an argument *LDFLG*. *LDFLG* affects the operation of **DEFINE**, **DEFINEQ**, **RPAQ**, **RPAQ?**, and **RPAQQ**. While a source file is being loaded, **DFNFLG** (page 10.10) is rebound to *LDFLG*. Thus, if *LDFLG* = **NIL**, and a function is redefined, a message is printed and the old definition saved. If *LDFLG* = **T**, the old definition is simply overwritten. If *LDFLG* = **PROP**, the functions are stored as "saved" definitions on the property lists under the property **EXPR** instead of being installed as the active definitions. If *LDFLG* = **ALLPROP**, not only function definitions but also variables set by **RPAQQ**, **RPAQ**, **RPAQ?** are stored on property lists (except when the variable has the value **NOBIND**, in which case they are set to the indicated value regardless of **DFNFLG**).

Another option is available for users who are loading systems for others to use and who wish to suppress the saving of information used to aid in development and debugging. If *LDFLG* = **SYSLOAD**, **LOAD** will: (1) Rebind **DFNFLG** to **T**, so old definitions are simply overwritten; (2) Rebind **LISPXHIST** to **NIL**, thereby making the **LOAD** not be undoable and eliminating the cost of saving undo information (See page 13.2[illegible]); (3) Rebind **ADDSPELLFLG** to **NIL**, to suppress adding to spelling lists; (4) Rebind **FILEPKGFLG** to **NIL**, to prevent the file from being "noticed" by the file package; (5) Rebind **BUILDMAPFLG** to **NIL**, to prevent a file map from being constructed; (6) After the load has completed, set the filecoms variable and any filevars

variables to **NOBIND**; and (7) Add the file name to **SYSFILES** rather than **FILELST**.

Note: A filevars variable is any variable appearing in a file package command of the form (*FILECOM* * *VARIABLE*) (see page 17.44). Therefore, if the filecoms includes **(FNS * FOOFNS)**, **FOOFNS** is set to **NOBIND**. If the user wants the value of such a variable to be retained, even when the file is loaded with *LDFLG* = **SYSLOAD**, then he should replace the variable with an equivalent, *non-atomic* expression, such as **(FNS * (PROGN FOOFNS))**.

All functions that have *LDFLG* as an argument perform spelling correction using **LOADOPTIONS** as a spelling list when *LDFLG* is not a member of **LOADOPTIONS**. **LOADOPTIONS** is initially **(NIL T PROP ALLPROP SYSLOAD)**.

(LOAD *FILE LDFLG PRINTFLG***)** [Function]

Reads successive expressions from *FILE* (with **FILERDTBL** as read table) and evaluates each as it is read, until it reads either **NIL**, or the single atom **STOP**. Note that **LOAD** can be used to load both symbolic and compiled files. Returns *FILE* (full name).

If *PRINTFLG* = **T**, **LOAD** prints the value of each expression; otherwise it does not.

(LOAD? *FILE LDFLG PRINTFLG***)** [Function]

Similar to **LOAD** except that it does not load *FILE* if it has already been loaded, in which case it returns **NIL**.

Note: **LOAD?** loads *FILE* except when the *same* version of the file has been loaded (either from the same place, or from a copy of it from a different place). Specifically, **LOAD?** considers that *FILE* has already been loaded if the full name of *FILE* is on **LOADEDFILELST** (page 17.20) or the date stored on the **FILEDATES** property of the root file name of *FILE* is the same as the **FILECREATED** expression on *FILE*.

(LOADFNS *FNS FILE LDFLG VARS***)** [Function]

Permits selective loading of definitions. *FNS* is a list of function names, a single function name, or **T**, meaning to load all of the functions on the file. *FILE* can be either a compiled or symbolic file. If a compiled definition is loaded, so are all compiler-generated subfunctions. The interpretation of *LDFLG* is the same as for **LOAD**.

If *FILE* = **NIL**, **LOADFNS** will use **WHEREIS** (page 17.14) to determine where the first function in *FNS* resides, and load from that file. Note that the file must previously have been "noticed" (see page 17.19). If **WHEREIS** returns **NIL**, and the WHEREIS

library package has been loaded, **LOADFNS** will use the WHEREIS data base to find the file containing *FN*.

VARS specifies which non-**DEFINEQ** expressions are to be loaded (i.e., evaluated). It is interpreted as follows:

T Means to load all non-**DEFINEQ** expressions.

NIL Means to load none of the non-**DEFINEQ** expressions.

VARS Means to evaluate all variable assignment expressions (beginning with **RPAQ**, **RPAQQ**, or **RPAQ?**, see page 17.54).

Any other litatom Means the same as specifying a list containing that atom.

A list If *VARS* is a list that is not a valid function definition, each element in *VARS* is "matched" against each non-**DEFINEQ** expression, and if any elements in *VARS* "match" successfully, the expression is evaluated. "Matching" is defined as follows: If an element of *VARS* is an atom, it matches an expression if it is **EQ** to either the **CAR** or the **CADR** of the expression. If an element of *VARS* is a list, it is treated as an edit pattern (page 16.18), and matched with the entire expression (using **EDIT4E**, page 16.72). For example, if *VARS* was **(FOOCOMS DECLARE: (DEFLIST & (QUOTE MACRO)))**, this would cause **(RPAQQ FOOCOMS ...)**, all **DECLARE:**s, and all **DEFLISTs** which set up **MACROs** to be read and evaluated.

A function definition If *VARS* is a list and a valid function definition (**(FNTYP** *VARS*) is true), then **LOADFNS** will invoke that function on every non-**DEFINEQ** expression being considered, applying it to two arguments, the first and second elements in the expression. If the function returns **NIL**, the expression will be skipped; if it returns a non-**NIL** litatom (e.g. **T**), the expression will be evaluated; and if it returns a list, this list is evaluated instead of the expression. Note: The file pointer is set to the very beginning of the expression before calling the *VARS* function definition, so it may read the entire expression if necessary. If the function returns a litatom, the file pointer is reset and the expression is **READ** or **SKREAD**. However, the file pointer is not reset when the function returns a list, so the function must leave it set immediately after the expression that it has presumably read.

LOADFNS returns a list of: (1) The names of the functions that were found; (2) A list of those functions not found (if any) headed by the litatom **NOT-FOUND:**; (3) All of the expressions that were evaluated; (4) A list of those members of *VARS* for which no corresponding expressions were found (if any), again headed by the litatom **NOT-FOUND:**. For example,

← (LOADFNS '(FOO FIE FUM) *FILE* **NIL '(BAZ (DEFLIST &)))**
(FOO FIE (NOT-FOUND: FUM) (RPAQ BAZ ...) (NOT-FOUND: (DEFLIST &)))

(LOADVARS *VARS FILE LDFLG***)** [Function]

Same as **(LOADFNS NIL** *FILE LDFLG VARS***)**.

(LOADFROM *FILE FNS LDFLG***)** [Function]

Same as **(LOADFNS** *FNS FILE LDFLG* **T)**.

Once the file package has noticed a file, the user can edit functions contained in the file without explicitly loading them. Similarly, those functions which have not been modified do not have to be loaded in order to write out an updated version of the file. Files are normally noticed (i.e., their contents become known to the file package; see page 17.19) when either the symbolic or compiled versions of the file are loaded. If the file is *not* going to be loaded completely, the preferred way to notice it is with **LOADFROM**. Note that the user can also load some functions at the same time by giving **LOADFROM** a second argument, but it is normally used simply to inform the file package about the existence and contents of a particular file.

(LOADBLOCK *FN FILE LDFLG***)** [Function]

Calls **LOADFNS** on those functions contained in the block declaration containing *FN* (See page 18.17). **LOADBLOCK** is designed primarily for use with symbolic files, to load the **EXPR**s for a given block. It will not load a function which already has an in-core **EXPR** definition, and it will not load the block name, unless it is also one of the block functions.

(LOADCOMP *FILE LDFLG***)** [Function]

Performs all operations on *FILE* associated with compilation, i.e. evaluates all expressions under a **DECLARE: EVAL@COMPILE** (see page 17.40), and "notices" the function and variable names by adding them to the lists **NOFIXFNSLST** and **NOFIXVARSLST** (see page 21.21).

Thus, if building a system composed of many files with compilation information scattered among them, all that is required to compile one file is to **LOADCOMP** the others.

(LOADCOMP? *FILE LDFLG***)** [Function]

Similar to **LOADCOMP**, except it does not load if file has already been loaded (with **LOADCOMP**), in which case its value is **NIL**.

Note: **LOADCOMP?** will load the file even if it has been loaded with **LOAD**, **LOADFNS**, etc. The only time it will not load the file is if the file has already been loaded with **LOADCOMP**.

FILESLOAD provides an easy way for the user to load a series of files, setting various options:

(FILESLOAD *FILE*$_1$ ... *FILE*$_N$**)** [NLambda NoSpread Function]

Loads the files *FILE*$_1$ ... *FILE*$_N$ (all arguments unevaluated). If any of these arguments are lists, they specify certain loading options for all following files (unless changed by another list). Within these lists, the following commands are recognized:

FROM *DIR* Search the specified directories for the file. *DIR* can either be a single directory, or a list of directories to search in order. For example, **(FILESLOAD (FROM {ERIS}<LISPCORE>SOURCES>) ...)** will search the directory **{ERIS}<LISPCORE>SOURCES>** for the files. If this is not specified, the default is to search the contents of **DIRECTORIES** (page 24.31).

If **FROM** is followed by the key word **VALUEOF**, the following word is evaluated, and the value is used as the list of directories to search. For example, **(FILESLOAD (FROM VALUEOF FOO) ...)** will search the directory list that is the value of the variable **FOO**.

As a special case, if *DIR* is a litatom, and the litatom *DIR***DIRECTORIES** is bound, the value of this variable is used as the directory search list. For example, since the variable **LISPUSERSDIRECTORIES** (page 24.32) is commonly used to contain a list of directories containing "library" packages, **(FILESLOAD (FROM LISPUSERS) ...)** can be used instead of **(FILESLOAD (FROM VALUEOF LISPUSERSDIRECTORIES) ...)**

Note: If a **FILESLOAD** is read and evaluated while loading a file, and it doesn't contain a **FROM** expression, the default is to search the directory containing the **FILESLOAD** expression before the value of **DIRECTORIES**. **FILESLOAD** expressions can be dumped on files using the **FILES** file package command (page 17.39).

SOURCE Load the source version of the file rather than the compiled version.

COMPILED Load the compiled version of the file.

Note: If **COMPILED** is specified, the compiled version will be loaded, if it is found. The source will not be loaded. If neither **SOURCE** or **COMPILED** is specified, the compiled version of the file will be loaded if it is found, otherwise the source will be loaded if it is found.

LOAD Load the file by calling **LOAD**, if it has not already been loaded. This is the default unless **LOADCOMP** or **LOADFROM** is specified.

Note: If **LOAD** is specified, **FILESLOAD** considers that the file has already been loaded if the root name of the file has a non-**NIL FILEDATES** property. This is a somewhat different algorithm than **LOAD?** uses. In particular, **FILESLOAD** will not load a newer version of a file that has already been loaded.

LOADCOMP Load the file with **LOADCOMP?** rather than **LOAD**. Automatically implies **SOURCE**.

LOADFROM Load the file with **LOADFROM** rather than **LOAD**.

NIL
T
PROP
ALLPROP
SYSLOAD The loading function is called with its *LDFLG* argument set to the specified token (see page 17.5). *LDFLG* affects the operation of the loading functions by resetting **DFNFLG** (page 10.10) to *LDFLG* during the loading. If none of these tokens are specified, the value of the variable **LDFLG** is used if it is bound, otherwise **NIL** is used.

NOERROR If **NOERROR** is specified, no error occurs when a file is not found.

Each list determines how all further files in the lists are loaded, unless changed by another list. The tokens above can be joined together in a single list. For example,

(FILESLOAD (LOADCOMP) NET (SYSLOAD FROM VALUEOF NEWDIRECTORIES) CJSYS)

will call **LOADCOMP?** to load the file **NET** searching the value of **DIRECTORIES**, and then call **LOADCOMP?** to load the file **CJSYS** with *LDFLG* set to **SYSLOAD**, searching the directory list that is the value of the variable **NEWDIRECTORIES**.

FILESLOAD expressions can be dumped on files using the **FILES** file package command (page 17.39).

17.2 Storing Files

(MAKEFILE *FILE OPTIONS REPRINTFNS SOURCEFILE***)** [Function]

Makes a new version of the file *FILE*, storing the information specified by *FILE*'s filecoms. Notices *FILE* if not previously noticed (see page 17.19). Then, it adds *FILE* to **NOTLISTEDFILES** and **NOTCOMPILEDFILES**.

OPTIONS is a litatom or list of litatoms which specify options. By specifying certain options, **MAKEFILE** can automatically compile or list *FILE*. Note that if *FILE* does not contain any function definitions, it is not compiled even when *OPTIONS* specifies **C** or **RC**. The options are spelling corrected using the list **MAKEFILEOPTIONS**. If spelling correction fails, **MAKEFILE** generates an error. The options are interpreted as follows:

C
RC After making *FILE*, **MAKEFILE** will compile *FILE* by calling **TCOMPL** (if **C** is specified) or **RECOMPILE** (if **RC** is specified). If there are any block declarations specified in the filecoms for *FILE*, **BCOMPL** or **BRECOMPILE** will be called instead.

If **F, ST, STF,** or **S** is the *next* item on *OPTIONS* following **C** or **RC**, it is given to the compiler as the answer to the compiler's question **LISTING?** (see page 18.1). For example, **(MAKEFILE 'FOO '(C F LIST))** will dump **FOO**, then **TCOMPL** or **BCOMPL** it specifying that functions are not to be redefined, and finally list the file.

LIST After making *FILE*, **MAKEFILE** calls **LISTFILES** to print a hardcopy listing of *FILE*.

CLISPIFY **MAKEFILE** calls **PRETTYDEF** with **CLISPIFYPRETTYFLG** = **T** (see page 21.26). This causes **CLISPIFY** to be called on each function defined as an **EXPR** before it is prettyprinted.

Alternatively, if *FILE* has the property **FILETYPE** with value **CLISP** or a list containing **CLISP**, **PRETTYDEF** is called with **CLISPIFYPRETTYFLG** reset to **CHANGES**, which will cause **CLISPIFY** to be called on all functions marked as having been changed. If *FILE* has property **FILETYPE** with value **CLISP**, the compiler will **DWIMIFY** its functions before compiling them (see page 18.11).

FAST **MAKEFILE** calls **PRETTYDEF** with **PRETTYFLG** = **NIL** (see page 26.48). This causes data objects to be printed rather than prettyprinted, which is much faster.

REMAKE **MAKEFILE** "remakes" *FILE*: The prettyprinted definitions of functions that have not changed are copied from an earlier version of the symbolic file. Only those functions that have changed are prettyprinted. See page 17.15.

NEW **MAKEFILE** does *not* remake *FILE*. If **MAKEFILEREMAKEFLG** = **T** (the initial setting), the default for all calls to **MAKEFILE** is to remake. The **NEW** option can be used to override this default.

REPRINTFNS and *SOURCEFILE* are used when remaking a file, as described on page 17.15.

Note: *FILE* is not added to **NOTLISTEDFILES** if *FILE* has on its property list the property **FILETYPE** with value **DON'TLIST**, or a list containing **DON'TLIST**. *FILE* is not added to **NOTCOMPILEDFILES** if *FILE* has on its property list the property **FILETYPE** with value **DON'TCOMPILE**, or a list containing **DON'TCOMPILE**. Also, if *FILE* does not contain any function definitions, it is not added to **NOTCOMPILEDFILES**, and it is not compiled even when *OPTIONS* specifies **C** or **RC**.

If a remake is *not* being performed, **MAKEFILE** checks the state of *FILE* to make sure that the entire source file was actually **LOAD**ed. If *FILE* was loaded as a compiled file, **MAKEFILE** prints the message **CAN'T DUMP: ONLY THE COMPILED FILE HAS BEEN LOADED**. Similarly, if only some of the symbolic definitions were loaded via **LOADFNS** or **LOADFROM**, **MAKEFILE** prints **CAN'T DUMP: ONLY SOME OF ITS SYMBOLICS HAVE BEEN LOADED**. In both cases, **MAKEFILE** will then ask the user if it should dump

anyway; if the user declines, **MAKEFILE** does not call **PRETTYDEF**, but simply returns (*FILE* **NOT DUMPED**) as its value.

The user can indicate that *FILE* must be block compiled together with other files as a unit by putting a list of those files on the property list of each file under the property **FILEGROUP**. If *FILE* has a **FILEGROUP** property, the compiler will not be called until all files on this property have been dumped that need to be.

MAKEFILE operates by rebinding **PRETTYFLG**, **PRETTYTRANFLG**, and **CLISPIFYPRETTYFLG**, evaluating each expression on **MAKEFILEFORMS** (under errorset protection), and then calling **PRETTYDEF** (page 17.50).

Note: **PRETTYDEF** calls **PRETTYPRINT** with its second argument *PRETTYDEFLG* = **T**, so whenever **PRETTYPRINT** (and hence **MAKEFILE**) start printing a new function, the name of that function is printed if more than 30 seconds (real time) have elapsed since the last time it printed the name of a function.

(MAKEFILES *OPTIONS FILES***)** [Function]

Performs (**MAKEFILE** *FILE OPTIONS*) for each file on *FILES* that needs to be dumped. If *FILES* = **NIL**, **FILELST** is used. For example, **(MAKEFILES 'LIST)** will make and list all files that have been changed. In this case, if any typed definitions for any items have been defined or changed and they are *not* contained in one of the files on **FILELST**, **MAKEFILES** calls **ADDTOFILES?** to allow the user to specify where these go. **MAKEFILES** returns a list of all files that are made.

(CLEANUP $FILE_1$ $FILE_2$ ... $FILE_N$**)** [NLambda NoSpread Function]

Dumps, lists, and recompiles (with **RECOMPILE** or **BRECOMPILE**) any of the specified files (unevaluated) requiring the corresponding operation. If no files are specified, **FILELST** is used. **CLEANUP** returns **NIL**.

CLEANUP uses the value of the variable **CLEANUPOPTIONS** as the *OPTIONS* argument to **MAKEFILE**. **CLEANUPOPTIONS** is initially **(RC)**, to indicate that the files should be recompiled. If **CLEANUPOPTIONS** is set to **(RC F)**, no listing will be performed, and no functions will be redefined as the result of compiling. Alternatively, if $FILE_1$ is a list, it will be interpreted as the list of options regardless of the value of **CLEANUPOPTIONS**.

(FILES?) [Function]

Prints on the terminal the names of those files that have been modified but not dumped, dumped but not listed, dumped but not compiled, plus the names of any functions and other typed definitions (if any) that are not contained in any file. If there are

any, **FILES?** then calls **ADDTOFILES?** to allow the user to specify where these go.

(ADDTOFILES? —) [Function]

Called from **MAKEFILES**, **CLEANUP**, and **FILES?** when there are typed definitions that have been marked as changed which do not belong to any file. **ADDTOFILES?** lists the names of the changed items, and asks the user if he wants to specify where these items should be put. If user answers **N**(o), **ADDTOFILES?** returns **NIL** without taking any action. If the user answers], this is taken to be an answer to each question that would be asked, and all the changed items are marked as dummy items to be ignored. Otherwise, **ADDTOFILES?** prints the name of each changed item, and accepts one of the following responses:

A file name
A filevar — If the user gives a file name or a variable whose value is a list (a filevar), the item is added to the corresponding file or list, using **ADDTOFILE.**

If the user response is not the name of a file on **FILELST** or a variable whose value is a list, the user will be asked whether it is a new file. If he says no, then **ADDTOFILES?** will check whether the item is the name of a list, i.e. whether its value is a list. If not, the user will be asked whether it is a new list.

line-feed — Same as the user's previous response.

space
carriage return — Take no action.

] — The item is marked as a dummy item by adding it to **NILCOMS**. This tells the file package simply to ignore this item.

[— The "definition" of the item in question is prettyprinted to the terminal, and then the user is asked again about its disposition.

(— **ADDTOFILES?** prompts with "**LISTNAME: (**", the user types in the name of a list, i.e. a variable whose value is a list, terminated by a **)**. The item will then only be added to (under) a command in which the named list appears as a filevar. If none are found, a message is printed, and the user is asked again. For example, the user defines a new function **FOO3**, and when asked where it goes, types **(FOOFNS)**. If the command **(FNS * FOOFNS)** is found, **FOO3** will be added to the value of **FOOFNS**. If instead the user types **(FOOCOMS)**, and the command **(COMS * FOOCOMS)** is found, then **FOO3** will be added to a command for dumping functions that is contained in **FOOCOMS**.

Note: If the named list is not also the name of a file, the user can simply type it in without parenthesis as described above.

@ — **ADDTOFILES?** prompts with "**Near: (**", the user types in the name of an object, and the item is then inserted in a command for

dumping objects (of its type) that contains the indicated name. The item is inserted immediately after the indicated name.

(LISTFILES *FILE$_1$ FILE$_2$ ... FILE$_N$***)** [NLambda NoSpread Function]

Lists each of the specified files (unevaluated). If no files are given, **NOTLISTEDFILES** is used. Each file listed is removed from **NOTLISTEDFILES** if the listing is completed. For each file not found, **LISTFILES** prints the message "*FILENAME* **NOT FOUND**" and proceeds to the next file.

LISTFILES calls the function **LISTFILES1** on each file to be listed. Normally, **LISTFILES1** is defined to simply call **SEND.FILE.TO.PRINTER** (page 29.1), but the user can advise or redefine **LISTFILES1** for more specialized applications.

Any lists inside the argument list to LISTFILES are interpreted as property lists that set the various printing options, such as the printer, number of copies, banner page name, etc (see page 29.1). Later properties override earlier ones. For example,

(LISTFILES FOO (HOST JEDI) FUM (#COPIES 3) FIE)

will cause one copy of **FOO** to be printed on the default printer, and 1 copy of **FUM** and 3 copies of **FIE** to be printed on the printer **JEDI**.

(COMPILEFILES *FILE$_1$ FILE$_2$ ... FILE$_N$***)** [NLambda NoSpread Function]

Executes the **RC** and **C** options of **MAKEFILE** for each of the specified files (unevaluated). If no files are given, **NOTCOMPILEDFILES** is used. Each file compiled is removed from **NOTCOMPILEDFILES**. If *FILE$_1$* is a list, it is interpreted as the *OPTIONS* argument to **MAKEFILES**. This feature can be used to supply an answer to the compiler's **LISTING?** question, e.g., **(COMPILEFILES (STF))** will compile each file on **NOTCOMPILEDFILES** so that the functions are redefined without the **EXPR**s definitions being saved.

(WHEREIS *NAME TYPE FILES FN***)** [Function]

TYPE is a file package type. **WHEREIS** sweeps through all the files on the list *FILES* and returns a list of all files containing *NAME* as a *TYPE*. **WHEREIS** knows about and expands all file package commands and file package macros. *TYPE* = **NIL** defaults to **FNS** (to retrieve function definitions). If *FILES* is not a list, the value of **FILELST** is used.

If *FN* is given, it should be a function (with arguments *NAME*, *FILE*, and *TYPE*) which is applied for every file in *FILES* that contains *NAME* as a *TYPE*. In this case, **WHEREIS** returns **NIL**.

If the WHEREIS library package has been loaded, **WHEREIS** is redefined so that *FILES* = **T** means to use the whereis package

data base, so **WHEREIS** will find *NAME* even if the file has not been loaded or noticed. *FILES* = **NIL** always means use **FILELST**.

17.3 Remaking a Symbolic File

Most of the time that a symbolic file is written using **MAKEFILE**, only a few of the functions that it contains have been changed since the last time the file was written. Rather than prettprinting all of the functions, it is often considerably faster to "remake" the file, copying the prettprinted definitions of unchanged functions from an earlier version of the symbolic file, and only prettyprinting those functions that have been changed.

MAKEFILE will remake the symbolic file if the **REMAKE** option is specified. If the **NEW** option is given, the file is not remade, and all of the functions are prettprinted. The default action is specified by the value of **MAKEFILEREMAKEFLG**: if **T** (its initial value), **MAKEFILE** will remake files unless the **NEW** option is given; if **NIL**, **MAKEFILE** will not remake unless the **REMAKE** option is given.

Note: If the file has never been loaded or dumped, for example if the filecoms were simply set up in memory, then **MAKEFILE** will never attempt to remake the file, regardless of the setting of **MAKEFILEREMAKEFLG**, or whether the **REMAKE** option was specified.

When **MAKEFILE** is remaking a symbolic file, the user can explicitly indicate the functions which are to be prettyprinted and the file to be used for copying the rest of the function definitions from via the *REPRINTFNS* and *SOURCEFILE* arguments to **MAKEFILE**. Normally, both of these arguments are defaulted to **NIL**. In this case, *REPRINTFNS* will be set to those functions that have been changed since the last version of the file was written. For *SOURCEFILE*, **MAKEFILE** obtains the full name of the most recent version of the file (that it knows about) from the **FILEDATES** property of the file, and checks to make sure that the file still exists and has the same file date as that stored on the **FILEDATES** property. If it does, **MAKEFILE** uses that file as *SOURCEFILE*. This procedure permits the user to **LOAD** or **LOADFROM** a file in a different directory, and still be able to remake the file with **MAKEFILE**. In the case where the most recent version of the file cannot be found, **MAKEFILE** will attempt to remake using the *original* version of the file (i.e., the one first loaded), specifying as *REPRINTFNS* the union of all changes that have been made since the file was first loaded, which is obtained from the **FILECHANGES** property of the file. If both of these fail, **MAKEFILE** prints the message "**CAN'T FIND**

EITHER THE PREVIOUS VERSION OR THE ORIGINAL VERSION OF *FILE*, SO IT WILL HAVE TO BE WRITTEN ANEW", and does not remake the file, i.e. will prettyprint all of the functions.

When a remake is specified, **MAKEFILE** also checks to see how the file was originally loaded (see page 17.19). If the file was originally loaded as a compiled file, **MAKEFILE** will call **LOADVARS** to obtain those **DECLARE:** expressions that are contained on the symbolic file, but not the compiled file, and hence have not been loaded. If the file was loaded by **LOADFNS** (but not **LOADFROM**), then **LOADVARS** is called to obtain any non-**DEFINEQ** expressions. Before calling **LOADVARS** to re-load definitions, **MAKEFILE** asks the user, e.g. "**Only the compiled version of FOO was loaded, do you want to LOADVARS the (DECLARE: .. DONTCOPY ..) expressions from {DSK}<MYDIR>FOO.;3?**". The user can respond **Yes** to execute the **LOADVARS** and continue the **MAKEFILE**, **No** to proceed with the **MAKEFILE** without performing the **LOADVARS**, or **Abort** to abort the **MAKEFILE**. The use may wish to skip the **LOADVARS** if the user had circumvented the file package in some way, and loading the old definitions would overwrite new ones.

Note: Remaking a symbolic file is considerably faster if the earlier version has a *file map* indicating where the function definitions are located (page 17.55), but it does not depend on this information.

17.4 Loading Files in a Distributed Environment

Each Interlisp source and compiled code file contains the full filename of the file, including the host and directory names, in a **FILECREATED** expression at the beginning of the file. The compiled code file also contains the full file name of the source file it was created from. In earlier versions of Interlisp, the file package used this information to locate the appropriate source file when "remaking" or recompiling a file.

This turned out to be a bad feature in distributed environments, where users frequently move files from one place to another, or where files are stored on removable media. For example, suppose you **MAKEFILE** to a floppy, and then copy the file to a file server. If you loaded and edited the file from a file server, and tried to do **MAKEFILE**, it would try to locate the source file on the floppy, which is probably no longer loaded.

Currently, the file package searches for sources file on the connected directory, and on the directory search path (on the variable **DIRECTORIES**). If it is not found, the host/directory information from the **FILECREATED** expression be used.

Warning: One situation where the new algorithm does the wrong thing is if you explicitly **LOADFROM** a file that is not on your directory search path. Future **MAKEFILE**s and **CLEANUP**s will search the connected directory and **DIRECTORIES** to find the source file, rather than using the file that the **LOADFROM** was done from. Even if the correct file is on the directory search path, you could still create a bad file if there is another version of the file in an earlier directory on the search path. In general, you should either explicitly specify the *SOURCEFILE* argument to **MAKEFILE** to tell it where to get the old source, or connect to the directory where the correct source file is.

17.5 Marking Changes

The file package needs to know what typed definitions have been changed, so it can determine which files need to be updated. This is done by "marking changes". All the system functions that perform file package operations (**LOAD**, **TCOMPL**, **PRETTYDEF**, etc.), as well as those functions that define or change data, (**EDITF**, **EDITV**, **EDITP**, DWIM corrections to user functions) interact with the file package by marking changes. Also, *typed-in* assignment of variables or property values is noticed by the file package. (Note that if a program modifies a variable or property value, this is not noticed.) In some cases the marking procedure can be subtle, e.g. if the user edits a property list using **EDITP**, only those properties whose values are actually changed (or added) are marked.

The various system functions which create or modify objects call **MARKASCHANGED** to mark the object as changed. For example, when a function is defined via **DEFINE** or **DEFINEQ**, or modified via **EDITF**, or a DWIM correction, the function is marked as being a changed object of type **FNS**. Similarly, whenever a new record is declared, or an existing record redeclared or edited, it is marked as being a changed object of type **RECORDS**, and so on for all of the other file package types.

The user can also call **MARKASCHANGED** directly to mark objects of a particular file package type as changed:

(MARKASCHANGED *NAME TYPE REASON***)** [Function]

Marks *NAME* of type *TYPE* as being changed. **MARKASCHANGED** returns *NAME*. **MARKASCHANGED** is undoable.

REASON is a litatom that indicated how *NAME* was changed. **MARKASCHANGED** recognizes the following values for *REASON*:

DEFINED Used to indicate the creation of *NAME*, e.g. from **DEFINEQ** (page 10.9).

CHANGED Used to indicate a change to *NAME*, e.g. from the editor.

DELETED Used to indicate the deletion of *NAME*, e.g. by **DELDEF** (page 17.27).

CLISP Used to indicate the modification of *NAME* by CLISP translation.

For backwards compatibility, **MARKASCHANGED** also accepts a *REASON* of **T** (= **DEFINED**) and **NIL** (= **CHANGED**). New programs should avoid using these values.

Note: The variable **MARKASCHANGEDFNS** is a list of functions that **MARKASCHANGED** calls (with arguments *NAME*, *TYPE*, and *REASON*). Functions can be added to this list to "advise" **MARKASCHANGED** to do additional work for all types of objects. The **WHENCHANGED** file package type property (page 17.31) can be used to specify additional actions when **MARKASCHANGED** gets called on specific types of objects.

(UNMARKASCHANGED *NAME TYPE***)** [Function]

Unmarks *NAME* of type *TYPE* as being changed. Returns *NAME* if *NAME* was marked as changed and is now unmarked, **NIL** otherwise. **UNMARKASCHANGED** is undoable.

(FILEPKGCHANGES *TYPE LST***)** [NoSpread Function]

If *LST* is not specified (as opposed to being **NIL**), returns a list of those objects of type *TYPE* that have been marked as changed but not yet associated with their corresponding files (See page 17.21). If *LST* is specified, **FILEPKGCHANGES** sets the corresponding list. **(FILEPKGCHANGES)** returns a list of *all* objects marked as changed as a list of elements of the form (*TYPENAME* . *CHANGEDOBJECTS*).

Some properties (e.g. **EXPR, ADVICE, MACRO, I.S.OPR**, etc..) are used to implement other file package types. For example, if the user changes the value of the property **I.S.OPR**, he is really changing an object of type **I.S.OPR**, and the effect is the same as though he had redefined the i.s.opr via a direct call to the function **I.S.OPR**. If a property whose value has been changed or added does not correspond to a specific file package type, then it is marked as a changed object of type **PROPS** whose *name* is (*VARIABLENAME PROPNAME*) (except if the property name has a property **PROPTYPE** with value **IGNORE**).

Similarly, if the user changes a variable which implements the file package type **ALISTS** (as indicated by the appearance of the property **VARTYPE** with value **ALIST** on the variable's property list), only those entries that are actually changed are marked as being changed objects of type **ALISTS**, and the "name" of the

object will be (*VARIABLENAME KEY*) where *KEY* is **CAR** of the entry on the alist that is being marked. If the variable corresponds to a specific file package type other than **ALISTS**, e.g. **USERMACROS**, **LISPXMACROS**, etc., then an object of that type is marked. In this case, the name of the changed object will be **CAR** of the corresponding entry on the alist. For example, if the user edits **LISPXMACROS** and changes a definition for **PL**, then the object **PL** of type **LISPXMACROS** is marked as being changed.

17.6 Noticing Files

Already existing files are "noticed" by **LOAD** or **LOADFROM** (or by **LOADFNS** or **LOADVARS** when the *VARS* argument is **T**. New files are noticed when they are constructed by **MAKEFILE**, or when definitions are first associated with them via **FILES?** or **ADDTOFILES?**. Noticing a file updates certain lists and properties so that the file package functions know to include the file in their operations. For example, **CLEANUP** will only dump files that have been noticed.

The user can explicitly tell the file package to notice a newly-created file by defining the filecoms for the file (see page 17.32), and calling **ADDFILE**:

(ADDFILE *FILE* — — —**)** [Function]

Tells the file package that *FILE* should be recognized as a file; it adds *FILE* to **FILELST**, and also sets up the **FILE** property of *FILE* to reflect the current set of changes which are "registered against" *FILE*.

The file package uses information stored on the property list of the root name of noticed files. The following property names are used:

FILE [Property Name]

When a file is noticed, the property **FILE**, value **((***FILECOMS* . *LOADTYPE***))** is added to the property list of its root name. *FILECOMS* is the variable containing the filecoms of the file (see page 17.32). *LOADTYPE* indicates *how* the file was loaded, e.g., completely loaded, only partially loaded as with **LOADFNS**, loaded as a compiled file, etc.

The property **FILE** is used to determine whether or not the corresponding file has been modified since the last time it was loaded or dumped. **CDR** of the **FILE** property records by type

those items that have been changed since the last **MAKEFILE**. Whenever a file is dumped, these items are moved to the property **FILECHANGES**, and **CDR** of the **FILE** property is reset to **NIL**.

FILECHANGES [Property Name]

The property **FILECHANGES** contains a list of all changed items since the file was loaded (there may have been several sequences of editing and rewriting the file). When a file is dumped, the changes in **CDR** of the **FILE** property are added to the **FILECHANGES** property.

FILEDATES [Property Name]

The property **FILEDATES** contains a list of version numbers and corresponding file dates for this file. These version numbers and dates are used for various integrity checks in connection with remaking a file (see page 17.15).

FILEMAP [Property Name]

The property **FILEMAP** is used to store the filemap for the file (see page 17.55). This is used to directly load individual functions from the middle of a file.

To compute the root name, **ROOTFILENAME** is applied to the name of the file as indicated in the **FILECREATED** expression appearing at the front of the file, since this name corresponds to the name the file was originally made under. The file package detects that the file being noticed is a compiled file (regardless of its name), by the appearance of more than one **FILECREATED** expressions. In this case, each of the files mentioned in the following **FILECREATED** expressions are noticed. For example, if the user performs **(BCOMPL '(FOO FIE))**, and subsequently loads **FOO.DCOM**, both **FOO** and **FIE** will be noticed.

When a file is noticed, its root name is added to the list **FILELST**:

FILELST [Variable]

Contains a list of the root names of the files that have been noticed.

LOADEDFILELST [Variable]

Contains a list of the actual names of the files as loaded by **LOAD**, **LOADFNS**, etc. For example, if the user performs **(LOAD '<NEWLISP>EDITA.COM;3)**, **EDITA** will be added to **FILELST**, but **<NEWLISP>EDITA.COM;3** is added to **LOADEDFILELST**. **LOADEDFILELST** is not used by the file package; it is maintained solely for the user's benefit.

17.7 Distributing Change Information

Periodically, the function **UPDATEFILES** is called to find which file(s) contain the elements that have been changed. **UPDATEFILES** is called by **FILES?**, **CLEANUP**, and **MAKEFILES**, i.e., any procedure that requires the **FILE** property to be up to date. This procedure is followed rather than updating the **FILE** property after each change because scanning **FILELST** and examining each file package command can be a time-consuming process; this is not so noticeable when performed in conjunction with a large operation like loading or writing a file.

UPDATEFILES operates by scanning **FILELST** and interrogating the file package commands for each file. When (if) any files are found that contain the corresponding typed definition, the name of the element is added to the value of the property **FILE** for the corresponding file. Thus, after **UPDATEFILES** has completed operating, the files that need to be dumped are simply those files on **FILELST** for which **CDR** of their **FILE** property is non-**NIL**. For example, if the user loads the file **FOO** containing definitions for **FOO1**, **FOO2**, and **FOO3**, edits **FOO2**, and then calls **UPDATEFILES**, **(GETPROP 'FOO 'FILE)** will be **((FOOCOMS . T) (FNS FOO2))**. If any objects marked as changed have not been transferred to the **FILE** property for some file, e.g., the user defines a new function but forgets (or declines) to add it to the file package commands for the corresponding file, then both **FILES?** and **CLEANUP** will print warning messages, and then call **ADDTOFILES?** to permit the user to specify on which files these items belong.

The user can also invoke **UPDATEFILES** directly:

(UPDATEFILES — —) [Function]

(UPDATEFILES) will update the **FILE** properties of the noticed files.

17.8 File Package Types

In addition to the definitions of functions and values of variables, source files in Interlisp can contain a variety of other information, e.g. property lists, record declarations, macro definitions, hash arrays, etc. In order to treat such a diverse assortment of data uniformly from the standpoint of file operations, the file package uses the concept of a *typed definition*, of which a function definition is just one example. A typed definition associates with a name (usually a litatom), a definition of a given type (called the file package type). Note

that the same name may have several definitions of different types. For example, a litatom may have both a function definition and a variable definition. The file package also keeps track of the file that a particular typed definition is stored on, so one can think of a typed definition as a relation between four elements: a name, a definition, a type, and a file.

A file package type is an abstract notion of a class of objects which share the property that every object of the same file package type is stored, retrieved, edited, copied etc., by the file package in the same way. Each file package type is identified by a litatom, which can be given as an argument to the functions that manipulate typed definitions. The user may define new file package types, as described in page 17.32.

FILEPKGTYPES [Variable]

The value of **FILEPKGTYPES** is a list of all file package types, including any that may have been defined by the user.

The file package is initialized with the following built-in file package types:

ADVICE [File Package Type]

Used to access "advice" modifying a function (see page 15.9).

ALISTS [File Package Type]

Used to access objects stored on an association list that is the value of a litatom (see page 3.15).

A variable is declared to have an association list as its value by putting on its property list the property **VARTYPE** with value **ALIST**. In this case, each dotted pair on the list is an object of type **ALISTS**. When the value of such a variable is changed, only those entries in the association list that are actually changed or added are marked as changed objects of type **ALISTS** (with "name" (*LITATOM KEY*)). Objects of type **ALISTS** are dumped via the **ALISTS** or **ADDVARS** file package commands.

Note that some association lists are used to "implement" other file package types. For example, the value of the global variable **USERMACROS** implements the file package type **USERMACROS** and the values of **LISPXMACROS** and **LISPXHISTORYMACROS** implement the file package type **LISPXMACROS**. This is indicated by putting on the property list of the variable the property **VARTYPE** with value a list of the form (**ALIST** *FILEPKGTYPE*). For example, **(GETPROP 'LISPXHISTORYMACROS 'VARTYPE) => (ALIST LISPXMACROS)**.

COURIERPROGRAMS [File Package Type]

Used to access Courier programs (see page 31.15).

EXPRESSIONS [File Package Type]

Used to access lisp expressions that are put on a file by using the **REMEMBER** programmers assistant command (page 13.17), or by explicitly putting the **P** file package command (page 17.40) on the filecoms.

FIELDS [File Package Type]

Used to access fields of records. The "definition" of an object of type **FIELDS** is a list of all the record declarations which contain the name. See page 8.1.

FILEPKGCOMS [File Package Type]

Used to access file package commands and types. A single name can be defined both as a file package type and a file package command. The "definition" of an object of type **FILEPKGCOMS** is a list structure of the form **((COM .** *COMPROPS***) (TYPE .** *TYPEPROPS***))**, where *COMPROPS* is a property list specifying how the name is defined as a file package command by **FILEPKGCOM** (page 17.47), and *TYPEPROPS* is a property list specifying how the name is defined as a file package type by **FILEPKGTYPE** (page 17.32).

FILES [File Package Type]

Used to access files. This file package type is most useful for renaming files. The "definition" of a file is not a useful structure.

FILEVARS [File Package Type]

Used to access Filevars (see page 17.44).

FNS [File Package Type]

Used to access function definitions.

I.S.OPRS [File Package Type]

Used to access the definitions of iterative statement operators (see page 9.9).

LISPXMACROS [File Package Type]

Used to access programmer's assistant commands defined on the variables **LISPXMACROS** and **LISPXHISTORYMACROS** (see page 13.23).

MACROS [File Package Type]

Used to access macro definitions (see page 10.21).

PROPS [File Package Type]

Used to access objects stored on the property list of a litatom (see page 2.5). When a property is changed or added, an object of type **PROPS**, with "name" (*LITATOM PROPNAME*) is marked as being changed.

Note that some litatom properties are used to implement other file package types. For example, the property **MACRO** implements the file package type **MACROS**, the property **ADVICE** implements **ADVICE**, etc. This is indicated by putting the property **PROPTYPE**, with value of the file package type on the property list of the property name. For example, **(GETPROP 'MACRO 'PROPTYPE) => MACROS**. When such a property is changed or added, an object of the corresponding file package type is marked. If **(GETPROP** *PROPNAME* **'PROPTYPE) => IGNORE**, the change is ignored. The **FILE**, **FILEMAP**, **FILEDATES**, etc. properties are all handled this way. (Note that **IGNORE** cannot be the name of a file package type implemented as a property).

RECORDS [File Package Type]

Used to access record declarations (see page 8.1).

RESOURCES [File Package Type]

Used to access resources (see page 12.19).

TEMPLATES [File Package Type]

Used to access Masterscope templates (see page 19.18).

USERMACROS [File Package Type]

Used to access user edit macros (see page 16.62).

VARS [File Package Type]

Used to access top-level variable values.

17.8.1 Functions for Manipulating Typed Definitions

The functions described below can be used to manipulate typed definitions, without needing to know how the manipulations are done. For example, **(GETDEF 'FOO 'FNS)** will return the function definition of **FOO**, **(GETDEF 'FOO 'VARS)** will return the variable value of **FOO**, etc. All of the functions use the following conventions:

(1) All functions which make destructive changes are undoable.

(2) Any argument that expects a list of litatoms will also accept a single litatom, operating as though it were enclosed in a list. For example, if the argument *FILES* should be a list of files, it may also be a single file.

(3) *TYPE* is a file package type. *TYPE* = **NIL** is equivalent to *TYPE* = **FNS**. The singular form of a file package type is also recognized, e.g. *TYPE* = **VAR** is equivalent to *TYPE* = **VARS**.

(4) *FILES* = **NIL** is equivalent to *FILES* = **FILELST**.

(5) *SOURCE* is used to indicate the source of a definition, that is, where the definition should be found. *SOURCE* can be one of:

CURRENT Get the definition currently in effect.

SAVED Get the "saved" definition, as stored by **SAVEDEF** (page 17.27).

FILE Get the definition contained on the (first) file determined by **WHEREIS** (page 17.14).

Note: **WHEREIS** is called with *FILES* = **T**, so that if the WHEREIS library package is loaded, the WHEREIS data base will be used to find the file containing the definition.

? Get the definition currently in effect if there is one, else the saved definition if there is one, otherwise the definition from a file determined by **WHEREIS**. Like specifying **CURRENT**, **SAVED**, and **FILE** in order, and taking the first definition that is found.

a file name
a list of file names — Get the definition from the first of the indicated files that contains one.

NIL In most cases, giving *SOURCE* = **NIL** (or not specifying it at all) is the same as giving **?**, to get either the current, saved, or filed definition. However, with **HASDEF**, *SOURCE* = **NIL** is interpreted as equal to *SOURCE* = **CURRENT**, which only tests if there is a current definition.

The operation of most of the functions described below can be changed or extended by modifying the appropriate properties for the corresponding file package type using the function **FILEPKGTYPE**, described on page 17.32.

(GETDEF *NAME TYPE SOURCE OPTIONS***)** [Function]

Returns the definition of *NAME*, of type *TYPE*, from *SOURCE*. For most types, **GETDEF** returns the expression which would be pretty printed when dumping *NAME* as *TYPE*. For example, for *TYPE* = **FNS**, an **EXPR** definition is returned, for *TYPE* = **VARS**, the value of *NAME* is returned, etc.

OPTIONS is a list which specifies certain options:

NOERROR **GETDEF** causes an error if an appropriate definition cannot be found, unless *OPTIONS* is or contains **NOERROR**. In this case,

GETDEF returns the value of the **NULLDEF** file package type property (page 17.30), usually **NIL**.

a string — If *OPTIONS* is or contains a string, that string will be returned if no definition is found (and **NOERROR** is not among the options). The caller can thus determine whether a definition was found, even for types for which **NIL** or **NOBIND** are acceptable definitions.

NOCOPY — **GETDEF** returns a copy of the definition unless *OPTIONS* is or contains **NOCOPY**.

EDIT — If *OPTIONS* is or contains **EDIT**, **GETDEF** returns a copy of the definition unless it is possible to edit the definition "in place." With some file package types, such as functions, it is meaningful (and efficient) to edit the definition by destructively modifying the list structure, without calling **PUTDEF**. However, some file package types (like records) need to be "installed" with **PUTDEF** after they are edited. The default **EDITDEF** (see page 17.31) calls **GETDEF** with *OPTIONS* of **(EDIT NOCOPY)**, so it doesn't use a copy unless it has to, and only calls **PUTDEF** if the result of editing is not **EQUAL** to the old definition.

NODWIM — A **FNS** definition will be dwimified if it is likely to contain CLISP unless *OPTIONS* is or contains **NODWIM**.

(PUTDEF *NAME TYPE DEFINITION REASON***)** [Function]

Defines *NAME* of type *TYPE* with *DEFINITION*. For *TYPE* = **FNS**, does a **DEFINE**; for *TYPE* = **VARS**, does a **SAVESET**, etc.

For *TYPE* = **FILES**, **PUTDEF** establishes the command list, notices *NAME*, and then calls **MAKEFILE** to actually dump the file *NAME*, copying functions if necessary from the "old" file (supplied as part of *DEFINITION*).

PUTDEF calls **MARKASCHANGED** (page 17.17) to mark *NAME* as changed, giving a reason of *REASON*. If *REASON* is **NIL**, the default is **DEFINED**.

Note: If *TYPE* = **FNS**, **PUTDEF** prints a warning if the user tries to redefine a function on the list **UNSAFE.TO.MODIFY.FNS** (page 10.10).

(HASDEF *NAME TYPE SOURCE SPELLFLG***)** [Function]

Returns **(OR** *NAME* **T)** if *NAME* is the name of something of type *TYPE*. If not, attempts spelling correction if *SPELLFLG* = **T**, and returns the spelling-corrected *NAME*. Otherwise returns **NIL**.

(HASDEF NIL *TYPE***)** returns **T** if **NIL** has a valid definition.

Note: if *SOURCE* = **NIL**, *HASDEF* interprets this as equal to *SOURCE* = **CURRENT**, which only tests if there is a current definition.

(TYPESOF *NAME POSSIBLETYPES IMPOSSIBLETYPES SOURCE***)** [Function]

Returns a list of the types in *POSSIBLETYPES* but not in *IMPOSSIBLETYPES* for which *NAME* has a definition. **FILEPKGTYPES** is used if *POSSIBLETYPES* is **NIL**.

(COPYDEF *OLD NEW TYPE SOURCE OPTIONS***)** [Function]

Defines *NEW* to have a copy of the definition of *OLD* by doing **PUTDEF** on a copy of the definition retrieved by **(GETDEF** *OLD TYPE SOURCE OPTIONS***)**. *NEW* is substituted for *OLD* in the copied definition, in a manner that may depend on the *TYPE*.

For example, **(COPYDEF 'PDQ 'RST 'FILES)** sets up **RSTCOMS** to be a copy of **PDQCOMS**, changes things like **(VARS * PDQVARS)** to be **(VARS * RSTVARS)** in **RSTCOMS**, and performs a **MAKEFILE** on **RST** such that the appropriate definitions get copied from **PDQ**.

COPYDEF disables the **NOCOPY** option of **GETDEF**, so *NEW* will always have a *copy* of the definition of *OLD*.

Note: **COPYDEF** substitutes *NEW* for *OLD* throughout the definition of *OLD*. This is usually the right thing to do, but in some cases, e.g., where the old name appears within a quoted expression but was not used in the same context, the user must re-edit the definition.

(DELDEF *NAME TYPE***)** [Function]

Removes the definition of *NAME* as a *TYPE* that is currently in effect.

(SHOWDEF *NAME TYPE FILE***)** [Function]

Prettyprints the definition of *NAME* as a *TYPE* to *FILE*. This shows the user how *NAME* would be written to a file. Used by **ADDTOFILES?** (page 17.13).

(EDITDEF *NAME TYPE SOURCE EDITCOMS***)** [Function]

Edits the definition of *NAME* as a *TYPE*. Essentially performs

(PUTDEF *NAME TYPE*
  (EDITE (GETDEF *NAME TYPE SOURCE***)**
    *EDITCOMS***))**

(SAVEDEF *NAME TYPE DEFINITION***)** [Function]

Sets the "saved" definition of *NAME* as a *TYPE* to *DEFINITION*. If *DEFINITION* = **NIL**, the current definition of *NAME* is saved.

If *TYPE* = **FNS** (or **NIL**), the function definition is saved on *NAME*'s property list under the property **EXPR**, or **CODE** (depending on the **FNTYP** of the function definition). If **(GETD** *NAME***)** is non-**NIL**, but **(FNTYP** *FN***)** = **NIL**, **SAVEDEF** saves the definition on the property name **LIST**. This can happen if a function was

somehow defined with an illegal expr definition, such as **(LAMMMMDA (X) ...)**.

If *TYPE* = **VARS**, the definition is stored as the value of the **VALUE** property of *NAME*. For other types, the definition is stored in an internal data structure, from where it can be retrieved by **GETDEF** or **UNSAVEDEF**.

(UNSAVEDEF *NAME TYPE* —**)** [Function]

Restores the "saved" definition of *NAME* as a *TYPE*, making it be the current definition. Returns *PROP*.

If *TYPE* = **FNS** (or **NIL**), **UNSAVEDEF** unsaves the function definition from the **EXPR** property if any, else **CODE**, and returns the property name used. **UNSAVEDEF** also recognizes *TYPE* = **EXPR**, **CODE**, or **LIST**, meaning to unsave the definition only from the corresponding property only.

If **DFNFLG** is not **T** (see page 10.10), the current definition of *NAME*, if any, is saved using **SAVEDEF**. Thus one can use **UNSAVEDEF** to switch back and forth between two definitions.

(LOADDEF *NAME TYPE SOURCE***)** [Function]

Equivalent to **(PUTDEF** *NAME TYPE* **(GETDEF** *NAME TYPE SOURCE***))**. **LOADDEF** is essentially a generalization of **LOADFNS**, e.g. it enables loading a single record declaration from a file. Note that **(LOADDEF** *FN***)** will give *FN* an **EXPR** definition, either obtained from its property list or a file, unless it already has one.

(CHANGECALLERS *OLD NEW TYPES FILES METHOD***)** [Function]

Finds all of the places where *OLD* is used as any of the types in *TYPES* and changes those places to use *NEW*. For example, **(CHANGECALLERS 'NLSETQ 'ERSETQ)** will change all calls to **NLSETQ** to be calls to **ERSETQ**. Also changes occurrences of *OLD* to *NEW* inside the filecoms of any file, inside record declarations, properties, etc.

CHANGECALLERS attempts to determine if *OLD* might be used as more than one type; for example, if it is both a function and a record field. If so, rather than performing the transformation *OLD* **->** *NEW* automatically, the user is allowed to edit all of the places where *OLD* occurs. For each occurrence of *OLD*, the user is asked whether he wants to make the replacement. If he responds with anything except **Yes** or **No**, the editor is invoked on the expression containing that occurrence.

There are two different methods for determining which functions are to be examined. If *METHOD* = **EDITCALLERS**, **EDITCALLERS** is used to search *FILES* (see page 16.74). If *METHOD* = **MASTERSCOPE**, then the Masterscope database is used instead. *METHOD* = **NIL** defaults to **MASTERSCOPE** if the

value of the variable **DEFAULTRENAMEMETHOD** is **MASTERSCOPE** and a Masterscope database exists, otherwise it defaults to **EDITCALLERS**.

(RENAME *OLD NEW TYPES FILES METHOD***)** [Function]

First performs **(COPYDEF** *OLD NEW TYPE***)** for all *TYPE* inside *TYPES*. It then calls **CHANGECALLERS** to change all occurrences of *OLD* to *NEW*, and then "deletes" *OLD* with **DELDEF**. For example, if the user has a function **FOO** which he now wishes to call **FIE**, he simply performs **(RENAME 'FOO 'FIE)**, and **FIE** will be given **FOO**'s definition, and all places that **FOO** are called will be changed to call **FIE** instead.

METHOD is interpreted the same as the *METHOD* argument to **CHANGECALLERS**, above.

(COMPARE *NAME1 NAME2 TYPE SOURCE1 SOURCE2***)** [Function]

Compares the definition of *NAME1* with that of *NAME2*, by calling **COMPARELISTS** (page 3.19) on **(GETDEF** *NAME1 TYPE SOURCE1***)** and **(GETDEF** *NAME2 TYPE SOURCE2***)**, which prints their differences on the terminal.

For example, if the current value of the variable **A** is **(A B C (D E F) G)**, and the value of the variable **B** on the file **<lisp>FOO** is **(A B C (D F E) G)**, then:

←(COMPARE 'A 'B 'VARS 'CURRENT '<lisp>FOO)
A from CURRENT and B from <lisp>TEST differ:
(E -> F) (F -> E)
T

(COMPAREDEFS *NAME TYPE SOURCES***)** [Function]

Calls **COMPARELISTS** (page 3.19) on all pairs of definitions of *NAME* as a *TYPE* obtained from the various *SOURCES* (interpreted as a list of source specifications).

17.8.2 Defining New File Package Types

All manipulation of typed definitions in the file package is done using the type-independent functions **GETDEF**, **PUTDEF**, etc. Therefore, to define a new file package type, it is only necessary to specify (via the function **FILEPKGTYPE**) what these functions should do when dealing with a typed definition of the new type. Each file package type has the following properties, whose values are functions or lists of functions:

Note: These functions are defined to take a *TYPE* argument so that the user may have the same function for more than one type.

GETDEF [File Package Type Property]

Value is a function of three arguments, *NAME*, *TYPE*, and *OPTIONS*, which should return the current definition of *NAME* as a type *TYPE*. Used by **GETDEF** (page 17.25), which passes its *OPTIONS* argument.

If there is no **GETDEF** property, a file package command for dumping *NAME* is created (by **MAKENEWCOM**). This command is then used to write the definition of *NAME* as a type *TYPE* onto the file **FILEPKG.SCRATCH** (in Interlisp-D, this file is created on the **{CORE}** device). This expression is then read back in and returned as the current definition.

Note: In some situations, the function **HASDEF** (page 17.26) needs to call **GETDEF** to determine whether a definition exists. In this case, *OPTIONS* will include the litatom **HASDEF**, and it is permissable for a **GETDEF** function to return **T** or **NIL**, rather than creating a complex structure which will not be used.

NULLDEF [File Package Type Property]

The value of the **NULLDEF** property is returned by **GETDEF** (page 17.25) when there is no definition and the **NOERROR** option is supplied. For example, the **NULLDEF** of **VARS** is **NOBIND**.

FILEGETDEF [File Package Type Property]

This enables the user to provide a way of obtaining definitions from a file that is more efficient than the default procedure used by **GETDEF** (page 17.25). Value is a function of four arguments, *NAME*, *TYPE*, *FILE*, and *OPTIONS*. The function is applied by **GETDEF** when it is determined that a typed definition is needed from a particular file. The function must open and search the given file and return any *TYPE* definition for *NAME* that it finds.

CANFILEDEF [File Package Type Property]

If the value of this property is non-**NIL**, this indicates that definitions of this file package type are not loaded when a file is loaded with **LOADFROM** (page 17.8). The default is **NIL**. Initially, only **FNS** has this property set to non-**NIL**.

PUTDEF [File Package Type Property]

Value is a function of three arguments, *NAME*, *TYPE*, and *DEFINITION*, which should store *DEFINITION* as the definition of *NAME* as a type *TYPE*. Used by **PUTDEF** (page 17.26).

HASDEF [File Package Type Property]

Value is a function of three arguments, *NAME*, *TYPE*, and *SOURCE*, which should return **(OR** *NAME* **T)** if *NAME* is the name

of something of type *TYPE*. *SOURCE* is as interpreted by **HASDEF** (page 17.26), which uses this property.

EDITDEF [File Package Type Property]

Value is a function of four arguments, *NAME*, *TYPE*, *SOURCE*, and *EDITCOMS*, which should edit the definition of *NAME* as a type *TYPE* from the source *SOURCE*, interpreting the edit commands *EDITCOMS*. If sucessful, should return *NAME* (or a spelling-corrected *NAME*). If it returns **NIL**, the "default" editor is called. Used by **EDITDEF** (page 17.27).

DELDEF [File Package Type Property]

Value is a function of two arguments, *NAME*, and *TYPE*, which removes the definition of *NAME* as a *TYPE* that is currently in effect. Used by **DELDEF** (page 17.27).

NEWCOM [File Package Type Property]

Value is a function of four arguments, *NAME*, *TYPE*, *LISTNAME*, and *FILE*. Specifies how to make a new (instance of a) file package command to dump *NAME*, an object of type *TYPE*. The function should return the new file package command. Used by **ADDTOFILE** and **SHOWDEF**.

If *LISTNAME* is non-**NIL**, this means that the user specified *LISTNAME* as the filevar in his interaction with **ADDTOFILES?** (see page 17.44).

If no **NEWCOM** is specified, the default is to call **DEFAULTMAKENEWCOM**, which will construct and return a command of the form (*TYPE NAME*). **DEFAULTMAKENEWCOM** can be advised or redefined by the user.

WHENCHANGED [File Package Type Property]

Value is a list of functions to be applied to *NAME*, *TYPE*, and *REASON* when *NAME*, an instance of type *TYPE*, is changed or defined (see **MARKASCHANGED**, page 17.17). Used for various applications, e.g. when an object of type **I.S.OPRS** changes, it is necessary to clear the corresponding translatons from **CLISPARRAY**.

The **WHENCHANGED** functions are called before the object is marked as changed, so that it can, in fact, decide that the object is *not* to be marked as changed, and execute **(RETFROM 'MARKASCHANGED)**.

Note: For backwards compatibility, the *REASON* argument passed to **WHENCHANGED** functions is either **T** (for **DEFINED**) and **NIL** (for **CHANGED**).

WHENFILED [File Package Type Property]

Value is a list of functions to be applied to *NAME*, *TYPE*, and *FILE* when *NAME*, an instance of type *TYPE*, is added to *FILE*.

WHENUNFILED [File Package Type Property]

Value is a list of functions to be applied to *NAME*, *TYPE*, and *FILE* when *NAME*, an instance of type *TYPE*, is removed from *FILE*.

DESCRIPTION [File Package Type Property]

Value is a string which describes instances of this type. For example, for type **RECORDS**, the value of **DESCRIPTION** is the string **"record declarations"**.

The function **FILEPKGTYPE** is used to define new file package types, or to change the properties of existing types. Note that it is possible to redefine the attributes of system file package types, such as **FNS** or **PROPS**.

(FILEPKGTYPE *TYPE* $PROP_1$ VAL_1 ... $PROP_N$ VAL_N**)** [NoSpread Function]

Nospread function for defining new file package types, or changing properties of existing file package types. $PROP_i$ is one of the property names given above; VAL_i is the value to be given to that property. Returns *TYPE*.

(FILEPKGTYPE *TYPE PROP***)** returns the value of the property *PROP*, without changing it.

(FILEPKGTYPE *TYPE***)** returns an alist of all of the defined properties of *TYPE*, using the property names as keys.

Note: Specifying *TYPE* as the litatom **TYPE** can be used to define one file package type as a synonym of another. For example, **(FILEPKGTYPE 'R 'TYPE 'RECORDS)** defines **R** as a synonym for the file package type **RECORDS**.

17.9 File Package Commands

The basic mechanism for creating symbolic files is the function **MAKEFILE** (page 17.10). For each file, the file package has a data structure known as the "filecoms", which specifies what typeu descriptions are contained in the file. A filecoms is a list of file package commands, each of which specifies objects of a certain file package type which should be dumped. For example, the filecoms

```
( (FNS FOO)
  (VARS FOO BAR BAZ)
```

(RECORDS XYZZY))

has a **FNS**, a **VARS**, and a **RECORDS** file package command. This filecoms specifies that the function definition for **FOO**, the variable values of **FOO**, **BAR**, and **BAZ**, and the record declaration for **XYZZY** should be dumped.

By convention, the filecoms of a file *X* is stored as the value of the litatom *X***COMS**. For example, **(MAKEFILE 'FOO.;27)** will use the value of **FOOCOMS** as the filecoms. This variable can be directly manipulated, but the file package contains facilities which make constructing and updating filecoms easier, and in some cases automatic (See page 17.48).

A file package command is an instruction to **MAKEFILE** to perform an explicit, well-defined operation, usually printing an expression. Usually there is a one-to-one correspondence between file package types and file package commands; for each file package type, there is a file package command which is used for writing objects of that type to a file, and each file package command is used to write objects of a particular type. However, in some cases, the same file package type can be dumped by several different file package commands. For example, the file package commands **PROP**, **IFPROP**, and **PROPS** all dump out objects with the file package type **PROPS**. This means if the user changes an object of file package type **PROPS** via **EDITP**, a typed-in call to **PUTPROP**, or via an explicit call to **MARKASCHANGED**, this object can be written out with any of the above three commands. Thus, when the file package attempts to determine whether this typed object is contained on a particular file, it must look at instances of all three file package commands **PROP**, **IFPROP**, and **PROPS**, to see if the corresponding atom and property are specified. It is also permissible for a single file package command to dump several different file package types. For example, the user can define a file package command which dumps both a function definition and its macro. Conversely, some file package comands do not dump any file package types at all, such as the **E** command.

For each file package command, the file package must be able to determine what typed definitions the command will cause to be printed so that the file package can determine on what file (if any) an object of a given type is contained (by searching through the filecoms). Similarly, for each file package type, the file package must be able to construct a command that will print out an object of that type. In other words, the file package must be able to map file package commands into file package types, and vice versa. Information can be provided to the file package about a particular file package command via the function **FILEPKGCOM** (page 17.47), and information about a particular file package type via the function **FILEPKGTYPE** (page 17.32). In the absence of other information, the default is simply that a file

package command of the form (*X NAME*) prints out the definition of *NAME* as a type *X*, and, conversely, if *NAME* is an object of type *X*, then *NAME* can be written out by a command of the form (*X NAME*).

If a file package function is given a command or type that is not defined, it attempts spelling correction using **FILEPKGCOMSPLST** as a spelling list (unless **DWIMFLG** or **NOSPELLFLG** = **NIL**; see page 20.13). If successful, the corrected version of the list of file package commands is written (again) on the output file, since at this point, the uncorrected list of file package commands would already have been printed on the output file. When the file is loaded, this will result in *FILE***COMS** being reset, and may cause a message to be printed, e.g., **(FOOCOMS RESET)**. The value of **FOOCOMS** would then be the corrected version. If the spelling correction is unsuccessful, the file package functions generate an error, **BAD FILE PACKAGE COMMAND**.

File package commands can be used to save on the output file definitions of functions, values of variables, property lists of atoms, advised functions, edit macros, record declarations, etc. The interpretation of each file package command is documented in the following sections.

(USERMACROS *LITATOM*$_1$ ... *LITATOM*$_N$**)** [File Package Command]

Each litatom *LITATOM*$_i$ is the name of a user edit macro. Writes expressions to add the edit macro definitions of *LITATOM*$_i$ to **USERMACROS**, and adds the names of the commands to the appropriate spelling lists.

If *LITATOM*$_i$ is not a user macro, a warning message **"no EDIT MACRO for** *LITATOM*$_i$**"** is printed.

17.9.1 Functions and Macros

(FNS *FN*$_1$ ... *FN*$_N$**)** [File Package Command]

Writes a **DEFINEQ** expression with the function definitions of *FN*$_1$ ... *FN*$_N$.

The user should never print a **DEFINEQ** expression directly onto a file himself (by using the **P** file package command, for example), because **MAKEFILE** generates the filemap of function definitions from the **FNS** file package commands (see page 17.55).

(ADVISE *FN*$_1$ ... *FN*$_N$**)** [File Package Command]

For each function *FN*$_i$, writes expressions to reinstate the function to its advised state when the file is loaded. See page 15.9.

Note: When advice is applied to a function programmatically or by hand, it is additive. That is, if a function already has some advice, further advice is added to the already-existing advice. However, when advice is applied to a function as a result of loading a file with an **ADVISE** file package command, the new advice replaces any earlier advice. **ADVISE** works this way to prevent problems with loading different versions of the same advice. If the user really wants to apply additive advice, a file package command such as **(P (ADVISE ...))** should be used (see page 17.40).

(ADVICE FN_1 ... FN_N**)** [File Package Command]

For each function FN_i, writes a **PUTPROPS** expression which will put the advice back on the property list of the function. The user can then use **READVISE** (page 15.12) to reactivate the advice.

(MACROS $LITATOM_1$ ... $LITATOM_N$**)** [File Package Command]

Each $LITATOM_i$ is a litatom with a **MACRO** definition (and/or a **DMACRO**, **10MACRO**, etc.). Writes out an expression to restore all of the macro properties for each $LITATOM_i$, embedded in a **DECLARE: EVAL@COMPILE** so the macros will be defined when the file is compiled. See page 10.21.

17.9.2 Variables

(VARS VAR_1 ... VAR_N**)** [File Package Command]

For each VAR_i, writes an expression to set its top level value when the file is loaded. If VAR_i is atomic, **VARS** writes out an expression to set VAR_i to the top-level value it had at the time the file was written. If VAR_i is non-atomic, it is interpreted as (*VAR FORM*), and **VARS** write out an expression to set *VAR* to the value of *FORM* (evaluated when the file is loaded).

VARS prints out expressions using **RPAQQ** and **RPAQ**, which are like **SETQQ** and **SETQ** except that they also perform some special operations with respect to the file package (see page 17.54).

Note: **VARS** cannot be used for putting arbitrary variable values on files. For example, if the value of a variable is an array (or many other data types), a litatom which represents the array is dumped in the file instead of the array itself. The **HORRIBLEVARS** file package command (page 17.36) provides a way of saving and reloading variables whose values contain re-entrant or circular list structure, user data types, arrays, or hash arrays.

(INITVARS VAR_1 ... VAR_N) [File Package Command]

INITVARS is used for initializing variables, setting their values only when they are currently **NOBIND**. A variable value defined in an **INITVARS** command will not change an already established value. This means that re-loading files to get some other information will not automatically revert to the initialization values.

The format of an **INITVARS** command is just like **VARS**. The only difference is that if VAR_i is atomic, the current value is not dumped; instead **NIL** is defined as the initialization value. Therefore, **(INITVARS FOO (FUM 2))** is the same as **(VARS (FOO NIL)(FUM 2))**, if **FOO** and **FUM** are both **NOBIND**.

INITVARS writes out an **RPAQ?** expression on the file instead of **RPAQ** or **RPAQQ**.

(ADDVARS (VAR_1 . LST_1) ... (VAR_N . LST_N)) [File Package Command]

For each (VAR_i . LST_i), writes an **ADDTOVAR** (page 17.54) to add each element of LST_i to the list that is the value of VAR_i at the time the file is loaded. The new value of VAR_i will be the union of its old value and LST_i. If the value of VAR_i is **NOBIND**, it is first set to **NIL**.

For example, **(ADDVARS (DIRECTORIES LISP LISPUSERS))** will add **LISP** and **LISPUSERS** to the value of **DIRECTORIES**.

If LST_i is not specified, VAR_i is initialized to **NIL** if its current value is **NOBIND**. In other words, **(ADDVARS (*VAR*))** will initialize *VAR* to **NIL** if *VAR* has not previously been set.

(APPENDVARS (VAR_1 . LST_1) ... (VAR_N . LST_N)) [File Package Command]

The same as **ADDVARS**, except that the values are added to the end of the lists (using **APPENDTOVAR**, page 17.55), rather than at the beginning.

(UGLYVARS VAR_1 ... VAR_N) [File Package Command]

Like **VARS**, except that the value of each VAR_i may contain structures for which **READ** is not an inverse of **PRINT**, e.g. arrays, readtables, user data types, etc. Uses **HPRINT** (page 25.17).

(HORRIBLEVARS VAR_1 .. VAR_N) [File Package Command]

Like **UGLYVARS**, except structures may also contain circular pointers. Uses **HPRINT** (page 25.17). The values of VAR_1 ... VAR_N are printed in the same operation, so that they may contain pointers to common substructures.

UGLYVARS does not do any checking for circularities, which results in a large speed and internal-storage advantage over

HORRIBLEVARS. Thus, if it is known that the data structures do *not* contain circular pointers, **UGLYVARS** should be used instead of **HORRIBLEVARS**.

(ALISTS (*VAR*$_1$ *KEY*$_1$ *KEY*$_2$ ...) ... (*VAR*$_N$ *KEY*$_3$ *KEY*$_4$ ...)) [File Package Command]

VAR$_i$ is a variable whose value is an association list, such as **EDITMACROS**, **BAKTRACELST**, etc. For each *VAR*$_i$, **ALISTS** writes out expressions which will restore the values associated with the specified keys. For example, **(ALISTS (BREAKMACROS BT BTV))** will dump the definition for the **BT** and **BTV** commands on **BREAKMACROS**.

Some association lists (**USERMACROS**, **LISPXMACROS**, etc.) are used to implement other file package types, and they have their own file package commands.

(SPECVARS *VAR*$_1$ ... *VAR*$_N$) [File Package Command]

(LOCALVARS *VAR*$_1$ ... *VAR*$_N$) [File Package Command]

(GLOBALVARS *VAR*$_1$ ... *VAR*$_N$) [File Package Command]

Outputs the corresponding compiler declaration embedded in a **DECLARE: DOEVAL@COMPILE DONTCOPY**. See page 18.5.

(CONSTANTS *VAR*$_1$ ... *VAR*$_N$) [File Package Command]

Like **VARS**, for each *VAR*$_i$ writes an expression to set its top level value when the file is loaded. Also writes a **CONSTANTS** expression to declare these variables as constants (see page 18.8). Both of these expressions are wrapped in a **(DECLARE: EVAL@COMPILE ...)** expression, so they can be used by the compiler.

Like **VARS**, *VAR*$_i$ can be non-atomic, in which case it is interpreted as **(*VAR FORM*)**, and passed to **CONSTANTS** (along with the variable being initialized to *FORM*).

17.9.3 Litatom Properties

(PROP *PROPNAME LITATOM*$_1$ ... *LITATOM*$_N$) [File Package Command]

Writes a **PUTPROPS** expression to restore the value of the *PROPNAME* property of each litatom *LITATOM*$_i$ when the file is loaded.

If *PROPNAME* is a list, expressions will be written for each property on that list. If *PROPNAME* is the litatom **ALL**, the values of all user properties (on the property list of each *LITATOM*$_i$) are

saved. **SYSPROPS** is a list of properties used by system functions. Only properties *not* on that list are dumped when the **ALL** option is used.

If $LITATOM_i$ does not have the property *PROPNAME* (as opposed to having the property with value **NIL**), a warning message **"NO** *PROPNAME* **PROPERTY FOR** $LITATOM_i$**"** is printed. The command **IFPROP** can be used if it is not known whether or not an atom will have the corresponding property.

(IFPROP *PROPNAME* $LITATOM_1$ ... $LITATOM_N$**)** [File Package Command]

Same as the **PROP** file package command, except that it only saves the properties that actually appear on the property list of the corresponding atom. For example, if **FOO1** has property **PROP1** and **PROP2**, **FOO2** has **PROP3**, and **FOO3** has property **PROP1** and **PROP3**, then **(IFPROP (PROP1 PROP2 PROP3) FOO1 FOO2 FOO3)** will save only those five property values.

(PROPS ($LITATOM_1$ $PROPNAME_1$**)** ... **(**$LITATOM_N$ $PROPNAME_N$**))** [File Package Command]

Similar to **PROP** command. Writes a **PUTPROPS** expression to restore the value of $PROPNAME_i$ for each $LITATOM_i$ when the file is loaded.

As with the **PROP** command, if $LITATOM_i$ does not have the property *PROPNAME* (as opposed to having the property with **NIL** value), a warning message **"NO** $PROPNAME_i$ **PROPERTY FOR** $LITATOM_i$**"** is printed.

17.9.4 Miscellaneous File Package Commands

(RECORDS REC_1 ... REC_N**)** [File Package Command]

Each REC_i is the name of a record (see page 8.1). Writes expressions which will redeclare the records when the file is loaded.

(INITRECORDS REC_1 ... REC_N**)** [File Package Command]

Similar to **RECORDS**, **INITRECORDS** writes expressions on a file that will, when loaded, perform whatever initialization/allocation is necessary for the indicated records. However, the record declarations themselves are not written out. This facility is useful for building systems on top of Interlisp, in which the implementor may want to eliminate the record declarations from a production version of the system, but the allocation for these records must still be done.

(LISPXMACROS *LITATOM*$_1$ ... *LITATOM*$_N$**)** [File Package Command]

Each *LITATOM*$_i$ is defined on **LISPXMACROS** or **LISPXHISTORYMACROS** (see page 13.23). Writes expressions which will save and restore the definition for each macro, as well as making the necessary additions to **LISPXCOMS**

(I.S.OPRS *OPR*$_1$ ... *OPR*$_N$**)** [File Package Command]

Each *OPR*$_i$ is the name of a user-defined i.s.opr (see page 9.20). Writes expressions which will redefine the i.s.oprs when the file is loaded.

(RESOURCES *RESOURCE*$_1$ ... *RESOURCE*$_N$**)** [File Package Command]

Each *RESOURCES*$_i$ is the name of a resource (see page 12.19). Writes expressions which will redeclare the resource when the file is loaded.

(INITRESOURCES *RESOURCE*$_1$ ... *RESOURCE*$_N$**)** [File Package Command]

Parallel to **INITRECORDS** (page 17.38), **INITRESOURCES** writes expressions on a file to perform whatever initialization/allocation is necessary for the indicated resources, without writing the resource declaration itself.

(COURIERPROGRAMS *NAME*$_1$ ... *NAME*$_N$**)** [File Package Command]

Each *NAME*$_i$ is the name of a Courier program (see page 31.15). Writes expressions which will redeclare the Courier program when the file is loaded.

(TEMPLATES *LITATOM*$_1$ ... *LITATOM*$_N$**)** [File Package Command]

Each *LITATOM*$_i$ is a litatom which has a Masterscope template (see page 19.21). Writes expressions which will restore the templates when the file is loaded.

(FILES *FILE*$_1$ ... *FILE*$_N$**)** [File Package Command]

Used to specify auxiliary files to be loaded in when the file is loaded. Dumps an expression calling **FILESLOAD** (page 17.9), with *FILE*$_1$ ... *FILE*$_N$ as the arguments. **FILESLOAD** interprets *FILE*$_1$ ... *FILE*$_N$ as files to load, possibly interspersed with lists used to specify certain loading options.

(FILEPKGCOMS *LITATOM*$_1$ ... *LITATOM*$_N$**)** [File Package Command]

Each litatom *LITATOM*$_i$ is either the name of a user-defined file package command or a user-defined file package type (or both). Writes expressions which will restore each command/type.

If $LITATOM_i$ is not a file package command or type, a warning message **"no FILE PACKAGE COMMAND for** $LITATOM_i$**"** is printed.

(* . *TEXT*) [File Package Command]

Used for inserting comments in a file. The file package command is simply written on the output file; it will be ignored when the file is loaded.

If the first element of *TEXT* is another *, a form-feed is printed on the file before the comment.

(P EXP_1 ... EXP_N**)** [File Package Command]

Writes each of the expressions EXP_1 ... EXP_N on the output file, where they will be evaluated when the file is loaded.

(E $FORM_1$ ... $FORM_N$**)** [File Package Command]

Each of the forms $FORM_1$ ... $FORM_N$ is evaluated at *output* time, when **MAKEFILE** interpretes this file package command.

(COMS COM_1 ... COM_N**)** [File Package Command]

Each of the commands COM_1 ... COM_N is interpreted as a file package command.

(ORIGINAL COM_1 ... COM_N**)** [File Package Command]

Each of the commands COM_i will be interpreted as a file package command without regard to any file package macros (as defined by the **MACRO** property of the **FILEPKGCOM** function, page 17.47). Useful for redefining a built-in file package command in terms of itself.

Note that some of the "built-in" file package commands are defined by file package macros, so interpreting them (or new user-defined file package commands) with **ORIGINAL** will fail. **ORIGINAL** was never intended to be used outside of a file package command macro.

17.9.5 DECLARE:

(DECLARE: . *FILEPKGCOMS/FLAGS*) [File Package Command]

Normally expressions written onto a symbolic file are (1) evaluated when loaded; (2) copied to the compiled file when the symbolic file is compiled (see page 18.1); and (3) not evaluated at compile time. **DECLARE:** allows the user to override these defaults.

FILEPKGCOMS/FLAGS is a list of file package commands, possibly interspersed with "tags". The output of those file package commands within *FILEPKGCOMS/FLAGS* is embedded in a **DECLARE:** expression, along with any tags that are specified. For example, **(DECLARE: EVAL@COMPILE DONTCOPY (FNS ...) (PROP ...))** would produce **(DECLARE: EVAL@COMPILE DONTCOPY (DEFINEQ ...) (PUTPROPS ...))**. **DECLARE:** is *defined* as an nlambda nospread function, which processes its arguments by evaluating or not evaluating each expression depending on the setting of internal state variables. The initial setting is to evaluate, but this can be overridden by specifying the **DONTEVAL@LOAD** tag.

DECLARE: expressions are specially processed by the compiler. For the purposes of compilation, **DECLARE:** has two principal applications: (1) to specify forms that are to be evaluated at compile time, presumably to affect the compilation, e.g., to set up macros; and/or (2) to indicate which expressions appearing in the symbolic file are *not* to be copied to the output file. (Normally, expressions are *not* evaluated and *are* copied.) Each expression in **CDR** of a **DECLARE:** form is either evaluated/not-evaluated and copied/not-copied depending on the settings of two internal state variables, initially set for copy and not-evaluate. These state variables can be reset for the remainder of the expressions in the **DECLARE:** by means of the tags **DONTCOPY**, **EVAL@COMPILE**, etc.

The tags are:

EVAL@LOAD
DOEVAL@LOAD Evaluate the following forms when the file is loaded (unless overridden by **DONTEVAL@LOAD**).

DONTEVAL@LOAD Do not evaluate the following forms when the file is loaded.

EVAL@LOADWHEN This tag can be used to provide conditional evaluation. The value of the expression immediately following the tag determines whether or not to evaluate subsequent expressions when loading. **... EVAL@LOADWHEN T ...** is equivalent to **... EVAL@LOAD ...**

COPY
DOCOPY When compiling, copy the following forms into the compiled file.

DONTCOPY When compiling, do not copy the following forms into the compiled file.

Note: If the file package commands following **DONTCOPY** include record declarations for datatypes, or records with initialization forms, it is necessary to include a **INITRECORDS** file package command (page 17.38) outside of the **DONTCOPY** form so that the initialization information is copied. For example, if **FOO** was defined as a datatype,

(DECLARE: DONTCOPY (RECORDS FOO))
(INITRECORDS FOO)

would copy the data type declaration for **FOO**, but would not copy the whole record declaration.

COPYWHEN When compiling, if the next form evaluates to non-**NIL**, copy the following forms into the compiled file.

EVAL@COMPILE
DOEVAL@COMPILE When compiling, evaluate the following forms.

DONTEVAL@COMPILE When compiling, do not evaluate the following forms.

EVAL@COMPILEWHEN When compiling, if the next form evaluates to non-**NIL**, evaluate the following forms.

FIRST For expressions that are to be copied to the compiled file, the tag **FIRST** can be used to specify that the following expressions in the **DECLARE:** are to appear at the front of the compiled file, before anything else except the **FILECREATED** expressions (see page 17.51). For example, **(DECLARE: COPY FIRST (P (PRINT** *MESS1* **T)) NOTFIRST (P (PRINT** *MESS2* **T)))** will cause **(PRINT** *MESS1* **T)** to appear first in the compiled file, followed by any functions, then **(PRINT** *MESS2* **T)**.

NOTFIRST Reverses the effect of **FIRST**.

The value of **DECLARETAGSLST** is a list of all the tags used in **DECLARE:** expressions. If a tag not on this list appears in a **DECLARE:** file package command, spelling correction is performed using **DECLARETAGSLST** as a spelling list.

Note that the function **LOADCOMP** (page 17.8) provides a convenient way of obtaining information from the **DECLARE:** expressions in a file, without reading in the entire file. This information may be used for compiling other files.

(BLOCKS $BLOCK_1$ ... $BLOCK_N$**)** [File Package Command]

For each $BLOCK_i$, writes a **DECLARE:** expression which the block compile functions interpret as a block declaration. See page 18.17.

17.9.6 Exporting Definitions

When building a large system in Interlisp, it is often the case that there are record definitions, macros and the like that are needed by several different system files when running, analyzing and compiling the source code of the system, but which are not needed for running the compiled code. By using the **DECLARE:** file package command with tag **DONTCOPY** (page 17.40), these definitions can be kept out of the compiled files, and hence out of the system constructed by loading the compiled files files into

Interlisp. This saves loading time, space in the resulting system, and whatever other overhead might be incurred by keeping those definitions around, e.g., burden on the record package to consider more possibilities in translating record accesses, or conflicts between system record fields and user record fields.

However, if the implementor wants to debug or compile code in the resulting system, the definitions are needed. And even if the definitions *had* been copied to the compiled files, a similar problem arises if one wants to work on system code in a regular Interlisp environment where none of the system files had been loaded. One could mandate that any definition needed by more than one file in the system should reside on a distinguished file of definitions, to be loaded into any environment where the system files are worked on. Unfortunately, this would keep the definitions away from where they logically belong. The **EXPORT** mechanism is designed to solve this problem.

To use the mechanism, the implementor identifies any definitions needed by files other than the one in which the definitions reside, and wraps the corresponding file package commands in the **EXPORT** file package command. Thereafter, **GATHEREXPORTS** can be used to make a single file containing all the exports.

(EXPORT COM_1 ... COM_N**)** [File Package Command]

This command is used for "exporting" definitions. Like **COM**, each of the commands COM_1 ... COM_N is interpreted as a file package command. The commands are also flagged in the file as being "exported" commands, for use with **GATHEREXPORTS** (see page 17.43).

(GATHEREXPORTS *FROMFILES TOFILE FLG***)** [Function]

FROMFILES is a list of files containing **EXPORT** commands. **GATHEREXPORTS** extracts all the exported commands from those files and produces a loadable file *TOFILE* containing them. If *FLG* = **EVAL**, the expressions are evaluated as they are gathered; i.e., the exports are effectively loaded into the current environment as well as being written to *TOFILE*.

(IMPORTFILE *FILE RETURNFLG***)** [Function]

If *RETURNFLG* is **NIL**, this loads any exported definitions from *FILE* into the current environment. If *RETURNFLG* is **T**, this returns a list of the exported definitions (evaluable expressions) without actually evaluating them.

(CHECKIMPORTS *FILES NOASKFLG***)** [Function]

Checks each of the files in *FILES* to see if any exists in a version newer than the one from which the exports in memory were

taken (**GATHEREXPORTS** and **IMPORTFILE** note the creation dates of the files involved), or if any file in the list has not had its exports loaded at all. If there are any such files, the user is asked for permission to **IMPORTFILE** each such file. If *NOASKFLG* is non-**NIL**, **IMPORTFILE** is performed without asking.

For example, suppose file **FOO** contains records **R1**, **R2**, and **R3**, macros **BAR** and **BAZ**, and constants **CON1** and **CON2**. If the definitions of **R1**, **R2**, **BAR**, and **BAZ** are needed by files other than **FOO**, then the file commands for **FOO** might contain the command

```
(DECLARE: EVAL@COMPILE DONTCOPY
  (EXPORT (RECORDS R1 R2)
      (MACROS BAR BAZ))
  (RECORDS R3)
  (CONSTANTS BAZ))
```

None of the commands inside this **DECLARE:** would appear on **FOO**'s compiled file, but **(GATHEREXPORTS '(FOO) 'MYEXPORTS)** would copy the record definitions for **R1** and **R2** and the macro definitions for **BAR** and **BAZ** to the file **MYEXPORTS**.

17.9.7 FileVars

In each of the file package commands described above, if the litatom * follows the command type, the form following the *, i.e., **CADDR** of the command, is evaluated and its value used in executing the command, e.g., **(FNS * (APPEND FNS1 FNS2))**. When this form is a litatom, e.g. **(FNS * FOOFNS)**, we say that the variable is a "filevar". Note that **(COMS *** *FORM***)** provides a way of *computing* what should be done by **MAKEFILE**.

Example:

```
←(SETQ FOOFNS '(FOO1 FOO2 FOO3))
(FOO1 FOO2 FOO3)
←(SETQ FOOCOMS
   '( (FNS * FOOFNS)
     (VARS FIE)
     (PROP MACRO FOO1 FOO2)
     (P (MOVD 'FOO1 'FIE1))]
←(MAKEFILE 'FOO)
```

would create a file **FOO** containing:

```
(FILECREATED "time and date the file was made" . "other
information")
(PRETTYCOMPRINT FOOCOMS)
(RPAQQ FOOCOMS ((FNS * FOOFNS) ...)
(RPAQQ FOOFNS (FOO1 FOO3 FOO3))
```

(DEFINEQ "*definitions of* FOO1, FOO2, *and* FOO3")
(RPAQQ FIE "*value of* FIE")
(PUTPROPS FOO1 MACRO *PROPVALUE*)
(PUTPROPS FOO2 MACRO *PROPVALUE*)
(MOVD (QUOTE FOO1) (QUOTE FIE1))
STOP

Note: For the **PROP** and **IFPROP** commands (page 17.37), the * follows the property name instead of the command, e.g., **(PROP MACRO * FOOMACROS)**. Also, in the form **(* * comment ...)**, the word **comment** is not treated as a filevar.

17.9.8 Defining New File Package Commands

A file package command is defined by specifying the values of certain properties. The user can specify the various attributes of a file package command for a new command, or respecify them for an existing command. The following properties are used:

MACRO [File Package Command Property]

Defines how to dump the file package command. Used by **MAKEFILE**. Value is a pair (*ARGS* . *COMS*). The "arguments" to the file package command are substituted for *ARGS* throughout *COMS*, and the result treated as a list of file package commands. For example, following **(FILEPKGCOM 'FOO 'MACRO '((X Y) . *COMS*))**, the file package command **(FOO A B)** will cause **A** to be substituted for **X** and **B** for **Y** throughout *COMS*, and then *COMS* treated as a list of commands.

The substitution is carried out by **SUBPAIR** (page 3.14), so that the "argument list" for the macro can also be atomic. For example, if **(X . *COMS*)** was used instead of **((X Y) . *COMS*)**, then the command **(FOO A B)** would cause **(A B)** to be substituted for **X** throughout *COMS*.

Note: Filevars are evaluated *before* substitution. For example, if the litatom * follows *NAME* in the command, **CADDR** of the command is evaluated substituting in *COMS*.

ADD [File Package Command Property]

Specifies how (if possible) to add an instance of an object of a particular type to a given file package command. Used by **ADDTOFILE**. Value is *FN*, a function of three arguments, *COM*, a file package command **CAR** of which is **EQ** to *COMMANDNAME*, *NAME*, a typed object, and *TYPE*, its type. *FN* should return **T** if it (undoably) adds *NAME* to *COM*, **NIL** if not. If no **ADD** property is specified, then the default is (1) if **(CAR *COM*) = *TYPE*** and **(CADR *COM*) = ***, and **(CADDR *COM*)** is a filevar (i.e. a literal atom), add

NAME to the value of the filevar, or (2) if (**CAR** *COM*) = *TYPE* and (**CADR** *COM*) is not *, add *NAME* to (**CDR** *COM*).

Actually, the function is given a fourth argument, *NEAR*, which if non-**NIL**, means the function should try to add the item after *NEAR*. See discussion of **ADDTOFILES?**, page 17.13.

DELETE [File Package Command Property]

Specifies how (if possible) to delete an instance of an object of a particular type from a given file package command. Used by **DELFROMFILES**. Value is *FN*, a function of three arguments, *COM*, *NAME*, and *TYPE*, same as for **ADD**. *FN* should return **T** if it (undoably) deletes *NAME* from *COM*, **NIL** if not. If no **DELETE** property is specified, then the default is (1) (**CAR** *COM*) = *TYPE* and (**CADR** *COM*) = *, and (**CADDR** *COM*) is a filevar (i.e. a literal atom), and *NAME* is contained in the value of the filevar, then remove *NAME* from the filevar, or (2) if (**CAR** *COM*) = *TYPE* and (**CADR** *COM*) is not *, and *NAME* is contained in (**CDR** *COM*), then remove *NAME* from (**CDR** *COM*).

If *FN* returns the value of **ALL**, it means that the command is now "empty", and can be deleted entirely from the command list.

CONTENTS [File Package Command Property]

CONTAIN [File Package Command Property]

Determines whether an instance of an object of a given type is contained in a given file package command. Used by **WHEREIS** and **INFILECOMS?**. Value is *FN*, a function of three arguments, *COM*, a file package command **CAR** of which is **EQ** to *COMMANDNAME*, *NAME*, and *TYPE*. The interpretation of *NAME* is as follows: if *NAME* is **NIL**, *FN* should return a list of elements of type *TYPE* contained in *COM*. If *NAME* is **T**, *FN* should return **T** if there are *any* elements of type *TYPE* in *COM*. If *NAME* is an atom other than **T** or **NIL**, return **T** if *NAME* of type *TYPE* is contained in *COM*. Finally, if *NAME* is a list, return a list of those elements of type *TYPE* contained in *COM* that are also contained in *NAME*.

Note that it is sufficient for the **CONTENTS** function to simply return the list of items of type *TYPE* in command *COM*, i.e. it can in fact ignore the *NAME* argument. The *NAME* argument is supplied mainly for those situations where producing the entire list of items involves significantly more computation or creates more storage than simply determining whether a particular item (or any item) of type *TYPE* is contained in the command.

If a **CONTENTS** property is specified and the corresponding function application returns **NIL** and (**CAR** *COM*) = *TYPE*, then the operation indicated by *NAME* is performed (1) on the value

of **(CADDR** *COM*), if **(CADR** *COM*) = *, otherwise (2) on **(CDR** *COM*). In other words, by specifying a **CONTENTS** property that returns **NIL**, e.g. the function **NILL**, the user specifies that a file package command of name **FOO** produces objects of file package type **FOO** and only objects of type **FOO**.

If the **CONTENTS** property is not provided, the command is simply expanded according to its **MACRO** definition, and each command on the resulting command list is then interrogated.

Note that if *COMMANDNAME* is a file package command that is used frequently, its expansion by the various parts of the system that need to interrogate files can result in a large number of **CONS**es and garbage collections. By informing the file package as to what this command actually does and does not produce via the **CONTENTS** property, this expansion is avoided. For example, suppose the user has a file package command called **GRAMMARS** which dumps various property lists but no functions. The file package could ignore this command when seeking information about **FNS**.

The function **FILEPKGCOM** is used to define new file package commands, or to change the properties of existing commands. Note that it is possible to redefine the attributes of system file package commands, such as **FNS** or **PROPS**, and to cause unpredictable results.

(FILEPKGCOM *COMMANDNAME* $PROP_1$ VAL_1 ... $PROP_N$ VAL_N**)** [NoSpread Function]

Nospread function for defining new file package commands, or changing properties of existing file package commands. $PROP_i$ is one of of the property names described above; VAL_i is the value to be given that property of the file package command *COMMANDNAME*. Returns *COMMANDNAME*.

(FILEPKGCOM *COMMANDNAME PROP***)** returns the value of the property *PROP*, without changing it.

(FILEPKGCOM *COMMANDNAME***)** returns an alist of all of the defined properties of *COMMANDNAME*, using the property names as keys.

Note: Specifying *TYPE* as the litatom **COM** can be used to define one file package command as a synonym of another. For example, **(FILEPKGCOM 'INITVARIABLES 'COM 'INITVARS)** defines **INITVARIABLES** as a synonym for the file package command **INITVARS**.

17.10 Functions for Manipulating File Command Lists

The following functions may be used to manipulate filecoms. The argument *COMS* does *not* have to correspond to the filecoms for some file. For example, *COMS* can be the list of commands generated as a result of expanding a user defined file package command.

Note: The following functions will accept a file package command as a valid value for their *TYPE* argument, even if it does not have a corresponding file package type. User-defined file package commands are expanded as necessary.

(INFILECOMS? *NAME TYPE COMS* —**)** [Function]

COMS is a list of file package commands, or a variable whose value is a list of file package commands. *TYPE* is a file package type. **INFILECOMS?** returns **T** if *NAME* of type *TYPE* is "contained" in *COMS*.

If *NAME* = **NIL**, **INFILECOMS?** returns a list of all elements of type *TYPE*.

If *NAME* = **T**, **INFILECOMS?** returns **T** if there are *any* elements of type *TYPE* in *COMS*.

(ADDTOFILE *NAME TYPE FILE NEAR LISTNAME***)** [Function]

Adds *NAME* of type *TYPE* to the file package commands for *FILE*. If *NEAR* is given and it is the name of an item of type *TYPE* already on *FILE*, then *NAME* is added to the command that dumps *NEAR*. If *LISTNAME* is given and is the name of a list of items of *TYPE* items on *FILE*, then *NAME* is added to that list. Uses **ADDTOCOMS** and **MAKENEWCOM**. Returns *FILE*. **ADDTOFILE** is undoable.

(DELFROMFILES *NAME TYPE FILES***)** [Function]

Deletes all instances of *NAME* of type *TYPE* from the filecoms for each of the files on *FILES*. If *FILES* is a non-**NIL** litatom, **(LIST** *FILES***)** is used. *FILES* = **NIL** defaults to **FILELST**. Returns a list of files from which *NAME* was actually removed. Uses **DELFROMCOMS**. **DELFROMFILES** is undoable.

Note: Deleting a function will also remove the function from any **BLOCKS** declarations in the filecoms.

(ADDTOCOMS *COMS NAME TYPE NEAR LISTNAME***)** [Function]

Adds *NAME* as a *TYPE* to *COMS*, a list of file package commands or a variable whose value is a list of file package commands. Returns **NIL** if **ADDTOCOMS** was unable to find a command appropriate for adding *NAME* to *COMS*. *NEAR* and *LISTNAME*

are described in the discussion of **ADDTOFILE**. **ADDTOCOMS** is undoable.

Note that the exact algorithm for adding commands depends the particular command itself. See discussion of the **ADD** property, in the description of **FILEPKGCOM**, page 17.47.

Note: **ADDTOCOMS** will not attempt to add an item to any command which is inside of a **DECLARE:** unless the user specified a specific name via the **LISTNAME** or **NEAR** option of **ADDTOFILES?**.

(DELFROMCOMS *COMS NAME TYPE***)** [Function]

Deletes *NAME* as a *TYPE* from *COMS*. Returns **NIL** if **DELFROMCOMS** was unable to modify *COMS* to delete *NAME*. **DELFROMCOMS** is undoable.

(MAKENEWCOM *NAME TYPE — —***)** [Function]

Returns a file package command for dumping *NAME* of type *TYPE*. Uses the procedure described in the discussion of **NEWCOM**, page 17.32.

(MOVETOFILE *TOFILE NAME TYPE FROMFILE***)** [Function]

Moves the definition of *NAME* as a *TYPE* from *FROMFILE* to *TOFILE* by modifying the file commands in the appropriate way (with **DELFROMFILES** and **ADDTOFILE**).

Note that if *FROMFILE* is specified, the definition will be retrieved from that file, even if there is another definition currently in the user's environment.

(FILECOMSLST *FILE TYPE —***)** [Function]

Returns a list of all objects of type *TYPE* in *FILE*.

(FILEFNSLST *FILE***)** [Function]

Same as **(FILECOMSLST** *FILE* **'FNS)**.

(FILECOMS *FILE TYPE***)** [Function]

Returns **(PACK*** *FILE* **(OR** *TYPE* **'COMS))**. Note that **(FILECOMS 'FOO)** returns the litatom **FOOCOMS**, not the value of **FOOCOMS**.

(SMASHFILECOMS *FILE***)** [Function]

Maps down **(FILECOMSLST** *FILE* **'FILEVARS)** and sets to **NOBIND** all filevars (see page 17.44), i.e. any variable used in a command of the form **(***COMMAND * VARIABLE***)**. Also sets **(FILECOMS** *FILE***)** to **NOBIND**. Returns *FILE*.

17.11 Symbolic File Format

The file package manipulates symbolic files in a particular format. This format is defined so that the information in the file is easily readable when the file is listed, as well as being easily manipulated by the file package functions. In general, there is no reason for the user to manually change the contents of a symbolic file. However, in order to allow users to extend the file package, this section describes some of the functions used to write symbolic files, and other matters related to their format.

(PRETTYDEF *PRTTYFNS PRTTYFILE PRTTYCOMS REPRINTFNS SOURCEFILE CHANGES*)
[Function]

Writes a symbolic file in **PRETTYPRINT** format for loading, using **FILERDTBL** as its read table. **PRETTYDEF** returns the name of the symbolic file that was created.

PRETTYDEF operates under a **RESETLST** (see page 14.24), so if an error occurs, or a control-D is typed, all files that **PRETTYDEF** has opened will be closed, the (partially complete) file being written will be deleted, and any undoable operations executed will be undone. The **RESETLST** also means that any **RESETSAVE**s executed in the file package commands will also be protected.

PRTTYFNS is an optional list of function names. It is equivalent to including **(FNS *** *PRTTYFNS*) in the file package commands in *PRTTYCOMS*. *PRTTYFNS* is an anachronism from when **PRETTYDEF** did not use a list of file package commands, and should be specified as **NIL**.

PRTTYFILE is the name of the file on which the output is to be written. If *PRTTYFILE* = **NIL**, the primary output file is used. If *PRTTYFILE* is atomic the file is opened if not already open, and it becomes the primary output file. *PRTTYFILE* is closed at end of **PRETTYDEF**, and the primary output file is restored. Finally, if *PRTTYFILE* is a list, **CAR** of *PRTTYFILE* is assumed to be the file name, and is opened if not already open. In this case, the file is left open at end of **PRETTYDEF**.

PRTTYCOMS is a list of file package commands interpreted as described on page 17.32. If *PRTTYCOMS* is atomic, its top level value is used and an **RPAQQ** is written which will set that atom to the list of commands when the file is subsequently loaded. A **PRETTYCOMPRINT** expression (see below) will also be written which informs the user of the named atom or list of commands when the file is subsequently loaded. In addition, if any of the functions in the file are nlambda functions, **PRETTYDEF** will automatically print a **DECLARE:** expression suitable for informing the compiler about these functions, in case the user recompiles the file without having first loaded the nlambda functions (see page 18.8).

REPRINTFNS and *SOURCEFILE* are for use in conjunction with remaking a file (see page 17.15). *REPRINTFNS* can be a list of functions to be prettyprinted, or **EXPRS**, meaning prettyprint all functions with **EXPR** definitions, or **ALL** meaning prettyprint all functions either defined as **EXPR**s, or with **EXPR** properties. Note that doing a remake with *REPRINTFNS* = **NIL** makes sense if there have been changes in the file, but not to any of the functions, e.g., changes to variables or property lists. *SOURCEFILE* is the name of the file from which to copy the definitions for those functions that are *not* going to be prettyprinted, i.e., those not specified by *REPRINTFNS*. *SOURCEFILE* = **T** means to use most recent version (i.e., highest number) of *PRTTYFILE*, the second argument to **PRETTYDEF**. If *SOURCEFILE* cannot be found, **PRETTYDEF** prints the message "*FILE* **NOT FOUND, SO IT WILL BE WRITTEN ANEW**", and proceeds as it does when *REPRINTFNS* and *SOURCEFILE* are both **NIL**.

PRETTYDEF calls **PRETTYPRINT** with its second argument *PRETTYDEFLG* = **T**, so whenever **PRETTYPRINT** starts a new function, it prints (on the terminal) the name of that function if more than 30 seconds (real time) have elapsed since the last time it printed the name of a function.

Note that normally if **PRETTYPRINT** is given a litatom which is not defined as a function but is known to be on one of the files noticed by the file package, **PRETTYPRINT** will load in the definition (using **LOADFNS**) and print it. This is not done when **PRETTYPRINT** is called from **PRETTYDEF**.

(PRINTFNS *X* **—)** [Function]

X is a list of functions. **PRINTFNS** prettyprints a **DEFINEQ** epression that defines the functions to the primary output stream using the primary read table. Used by **PRETTYDEF** to implement the **FNS** file package command.

(PRINTDATE *FILE CHANGES***)** [Function]

Prints the **FILECREATED** expression at beginning of **PRETTYDEF** files. *CHANGES* used by the file package.

(FILECREATED *X***)** [NLambda NoSpread Function]

Prints a message (using **LISPXPRINT**) followed by the time and date the file was made, which is **(CAR** *X***)**. The message is the value of **PRETTYHEADER**, initially **"FILE CREATED"**. If **PRETTYHEADER** = **NIL**, nothing is printed. **(CDR** *X***)** contains information about the file, e.g., full name, address of file map, list of changed items, etc. **FILECREATED** also stores the time and date the file was made on the property list of the file under the property **FILEDATES** and performs other initialization for the file package.

(PRETTYCOMPRINT *X*) [NLambda Function]

Prints *X* (unevaluated) using **LISPXPRINT**, unless **PRETTYHEADER = NIL**.

PRETTYHEADER [Variable]

Value is the message printed by **FILECREATED**. **PRETTYHEADER** is initially **"FILE CREATED"**. If **PRETTYHEADER = NIL**, neither **FILECREATED** nor **PRETTYCOMPRINT** will print anything. Thus, setting **PRETTYHEADER** to **NIL** will result in "silent loads". **PRETTYHEADER** is reset to **NIL** during greeting (page 12.1).

(FILECHANGES *FILE TYPE*) [Function]

Returns a list of the changed objects of file package type *TYPE* from the **FILECREATED** expression of *FILE*. If *TYPE* = **NIL**, returns an alist of all of the changes, with the file package types as the **CAR**s of the elements..

(FILEDATE *FILE* —) [Function]

Returns the file date contained in the **FILECREATED** expression of *FILE*.

(LISPSOURCEFILEP *FILE*) [Function]

Returns a non-**NIL** value if *FILE* is an Interlisp source file, **NIL** otherwise.

17.11.1 Copyright Notices

The system has a facility for automatically printing a copyright notice near the front of files, right after the **FILECREATED** expression, specifying the years it was edited and the copyright owner. The format of the copyright notice is:

(* Copyright (c) 1981 by Foo Bars Corporation)

Once a file has a copyright notice then every version will have a new copyright notice inserted into the file without user intervention. (The copyright information necessary to keep the copyright up to date is stored at the end of the file.).

Any year the file has been edited is considered a "copyright year" and therefore kept with the copyright information. For example, if a file has been edited in 1981, 1982, and 1984, then the copyright notice would look like:

(* Copyright (c) 1981,1982,1984 by Foo Bars Corporation)

When a file is made, if it has no copyright information, the system will ask the user to specify the copyright owner (if **COPYRIGHTFLG = T**). The user may specify one of the names

from **COPYRIGHTOWNERS**, or give one of the following responses:

(1) Type a left-square-bracket. The system will then prompt for an arbitrary string which will be used as the owner-string

(2) Type a right-square-bracket, which specifies that the user really does not want a copyright notice.

(3) Type "**NONE**" which specifies that this file should never have a copyright notice.

For example, if COPYRIGHTOWNERS has the value

((BBN "Bolt Beranek and Newman Inc.")
(XEROX "Xerox Corporation"))

then for a new file **FOO** the following interaction will take place:

Do you want to Copyright FOO? Yes
Copyright owner: *(user typed ?)*
one of:
BBN - Bolt Beranek and Newman Inc.
XEROX - Xerox Corporation
NONE - no copyright ever for this file
[- new copyright owner -- type one line of text
] - no copyright notice for this file now

Copyright owner: BBN

Then "Foo Bars Corporation" in the above copyright notice example would have been "Bolt Beranek and Newman Inc."

The following variables control the operation of the copyright facility:

COPYRIGHTFLG [Variable]

The value of **COPYRIGHTFLG** determines whether copyright information is maintained in files. Its value is interpreted as follows:

NIL The system will preserve old copyright information, but will not ask the user about copyrighting new files. This is the default value of **COPYRIGHTFLG**.

T When a file is made, if it has no copyright information, the system will ask the user to specify the copyright owner.

NEVER The system will neither prompt for new copyright information nor preserve old copyright information.

DEFAULT The value of **DEFAULTCOPYRIGHTOWNER** (below) is used for putting copyright information in files that don't have any other copyright. The prompt "**Copyright owner for file xx:**" will still be printed, but the default will be filled in immediately.

COPYRIGHTOWNERS [Variable]

COPYRIGHTOWNERS is a list of entries of the form (*KEY OWNERSTRING*), where *KEY* is used as a response to **ASKUSER** and *OWNERSTRING* is a string which is the full identification of the owner.

DEFAULTCOPYRIGHTOWNER [Variable]

If the user does not respond in **DWIMWAIT** seconds to the copyright query, the value of **DEFAULTCOPYRIGHTOWNER** is used.

17.11.2 Functions Used Within Source Files

The following functions are normally only used within symbolic files, to set variable values, property values, etc. Most of these have special behavior depending on file package variables.

(RPAQ *VAR VALUE*) [NLambda Function]

An nlambda function like **SETQ** that sets the top level binding of *VAR* (unevaluated) to *VALUE*.

(RPAQQ *VAR VALUE*) [NLambda Function]

An nlambda function like **SETQQ** that sets the top level binding of *VAR* (unevaluated) to *VALUE* (unevaluated).

(RPAQ? *VAR VALUE*) [NLambda Function]

Similar to **RPAQ**, except that it does nothing if *VAR* already has a top level value other than **NOBIND**. Returns *VALUE* if *VAR* is reset, otherwise **NIL**.

RPAQ, **RPAQQ**, and **RPAQ?** generate errors if *X* is not a litatom. All are affected by the value of **DFNFLG** (page 10.10). If **DFNFLG = ALLPROP** (and the value of *VAR* is other than **NOBIND**), instead of setting *X*, the corresponding value is stored on the property list of *VAR* under the property **VALUE**. All are undoable.

(ADDTOVAR *VAR* X_1 X_2 ... X_N) [NLambda NoSpread Function]

Each X_i that is not a member of the value of *VAR* is added to it, i.e. after **ADDTOVAR** completes, the value of *VAR* will be **(UNION (LIST** X_1 X_2 ... X_N) *VAR*). **ADDTOVAR** is used by **PRETTYDEF** for implementing the **ADDVARS** command. It performs some file package related operations, i.e. "notices" that *VAR* has been changed. Returns the atom *VAR* (not the value of *VAR*).

(APPENDTOVAR *VAR* X_1 X_2 ... X_N**)** [NLambda NoSpread Function]

Similar to **ADDTOVAR**, except that the values are added to the end tof the list, rather than at the beginning.

(PUTPROPS *ATM* $PROP_1$ VAL_1 ... $PROP_N$ VAL_N**)** [NLambda NoSpread Function]

Nlambda nospread version of **PUTPROP** (none of the arguments are evaluated). For $i = 1...N$, puts property $PROP_i$, value VAL_i, on the property list of *ATM*. Performs some file package related operations, i.e., "notices" that the corresponding properties have been changed.

(SAVEPUT *ATM PROP VAL*) [Function]

Same as **PUTPROP**, but marks the corresponding property value as having been changed (used by the file package).

17.11.3 File Maps

A file map is a data structure which contains a symbolic 'map' of the contents of a file. Currently, this consists of the begin and end byte address (see **GETFILEPTR**, page 25.19) for each **DEFINEQ** expression in the file, the begin and end address for each function definition within the **DEFINEQ**, and the begin and end address for each compiled function.

MAKEFILE, **PRETTYDEF**, **LOADFNS**, **RECOMPILE**, and numerous other system functions depend heavily on the file map for efficient operation. For example, the file map enables **LOADFNS** to load selected function definitions simply by setting the file pointer to the corresponding address using **SETFILEPTR**, and then performing a single **READ**. Similarly, the file map is heavily used by the "remake" option of **MAKEFILE** (page 17.15): those function definitions that have been changed since the previous version are prettyprinted; the rest are simply copied from the old file to the new one, resulting in a considerable speedup.

Whenever a file is written by **MAKEFILE**, a file map for the new file is built. Building the map in this case essentially comes for free, since it requires only reading the current file pointer before and after each definition is written or copied. However, building the map does require that **PRETTYPRINT** *know* that it is printing a **DEFINEQ** expression. For this reason, the user should never print a **DEFINEQ** expression onto a file himself, but should instead always use the **FNS** file package command (page 17.34).

The file map is stored on the property list of the root name of the file, under the property **FILEMAP**. In addition, **MAKEFILE** writes the file map on the file itself. For cosmetic reasons, the file map is written as the last expression in the file. However, the *address* of the file map in the file is (over)written into the **FILECREATED**

expression that appears at the beginning of the file so that the file map can be rapidly accessed without having to scan the entire file. In most cases, **LOAD** and **LOADFNS** do not have to build the file map at all, since a file map will usually appear in the corresponding file, unless the file was written with **BUILDMAPFLG = NIL**, or was written outside of Interlisp.

Currently, file maps for *compiled* files are not written onto the files themselves. However, **LOAD** and **LOADFNS** will build maps for a compiled file when it is loaded, and store it on the property **FILEMAP**. Similary, **LOADFNS** will obtain and use the file map for a compiled file, when available.

The use and creation of file maps is controlled by the following variables:

BUILDMAPFLG [Variable]

Whenever a file is read by **LOAD** or **LOADFNS**, or written by **MAKEFILE**, a file map is automatically built unless **BUILDMAPFLG = NIL**. (**BUILDMAPFLG** is initially **T**.)

While building the map will not help the first reference to a file, it will help in future references. For example, if the user performs **(LOADFROM 'FOO)** where **FOO** does not contain a file map, the **LOADFROM** will be (slightly) slower than if **FOO** did contain a file map, but subsequent calls to **LOADFNS** for this version of **FOO** will be able to use the map that was built as the result of the **LOADFROM**, since it will be stored on **FOO**'s **FILEMAP** property.

USEMAPFLG [Variable]

If **USEMAPFLG = T** (the initial setting), the functions that use file maps will first check the **FILEMAP** property to see if a file map for this file was previously obtained or built. If not, the first expression on the file is checked to see if it is a **FILECREATED** expression that also contains the address of a file map. If the file map is not on the **FILEMAP** property or in the file, a file map will be built (unless **BUILDMAPFLG = NIL**).

If **USEMAPFLG = NIL**, the **FILEMAP** property and the file will not be checked for the file map. This allows the user to recover in those cases where the file and its map for some reason do not agree. For example, if the user uses a text editor to change a symbolic file that contains a map (not recommended), inserting or deleting just one character will throw that map off. The functions which use file maps contain various integrity checks to enable them to detect that something is wrong, and to generate the error **FILEMAP DOES NOT AGREE WITH CONTENTS OF** *FILE*. In such cases, the user can set **USEMAPFLG** to **NIL**, causing the map contained in the file to be ignored, and then reexecute the operation.

[This page intentionally left blank]

18. COMPILER

The compiler is contained in the standard Interlisp system. It may be used to compile functions defined in the user's Interlisp system, or to compile definitions stored in a file. The resulting compiled code may be stored as it is compiled, so as to be available for immediate use, or it may be written onto a file for subsequent loading.

The most common way to use the compiler is to use one of the file package functions, such as **MAKEFILE** (page 17.10), which automatically updates source files, and produces compiled versions. However, it is also possible to compile individual functions defined in the user's Interlisp system, by directly calling the compiler using functions such as **COMPILE** (page 18.14). No matter how the compiler is called, the function **COMPSET** is called which asks the user certain questions concerning the compilation. (**COMPSET** sets the free variables **LAPFLG**, **STRF**, **SVFLG**, **LCFIL** and **LSTFIL** which determine various modes of operation.) Those that can be answered "yes" or "no" can be answered with **YES**, **Y**, or **T** for "yes"; and **NO**, **N**, or **NIL** for "no". The questions are:

LISTING? This asks whether to generate a listing of the compiled code. The LAP and machine code are usually not of interest but can be helpful in debugging macros. Possible answers are:

1 Prints output of pass 1, the **LAP** macro code.

2 Prints output of pass 2, the machine code.

YES Prints output of both passes.

NO Prints no listings.

The variable **LAPFLG** is set to the answer.

FILE: This question (which only appears if the answer to **LISTING?** is affirmative) ask where the compiled code listing(s) should be written. Answering **T** will print the listings at the terminal. The variable **LSTFIL** is set to the answer.

REDEFINE? This question asks whether the functions compiled should be redefined to their compiled definitions. If this is answered **YES**, the compiled code is stored and the function definition changed, otherwise the function definition remains unchanged.

Note: The compiler does NOT respect the value of **DFNFLG** (page 10.10) when it redefines functions to their compiled definitions. Therefore, if you set **DFNFLG** to **PROP** to completely avoid

inadvertantly redefining something in your running system, you MUST not answer **YES** to this question.

The variable **STRF** is set to **T** (if this is answered **YES**) or **NIL**.

SAVE EXPRS? This question asks whether the original defining **EXPR**s of functions should be saved. If answered **YES**, then before redefining a function to its compiled definition, the **EXPR** definition is saved on the property list of the function name. Otherwise they are discarded.

It is very useful to save the **EXPR** definitions, just in case the compiled function needs to be changed. The editing functions will retrieve this saved definition if it exists, rather than reading from a source file.

The variable **SVFLG** is set to **T** (if this is answered **YES**) or **NIL**.

OUTPUT FILE? This question asks whether (and where) the compiled definitions should be written into a file for later loading. If you answer with the name of a file, that file will be used. If you answer **Y** or **YES**, you will be asked the name of the file. If the file named is already open, it will continue to be used. If you answer **T** or **TTY:**, the output will be typed on the teletype (not particularly useful). If you answer **N**, **NO**, or **NIL**, output will *not* be done.

The variable **LCFIL** is set to the name of the file.

In order to make answering these questions easier, there are four other possible answers to the **LISTING?** question, which specify common compiling modes:

S Same as last setting. Uses the same answers to compiler questions as given for the last compilation.

F Compile to File, without redefining functions.

ST STore new definitions, saving **EXPR** definitions.

STF STore new definitions; Forget **EXPR** definitions.

Implicit in these answers are the answers to the questions on disposition of compiled code and **EXPR** definitions, so the questions **REDEFINE?** and **SAVE EXPRS?** would not be asked if these answers were given. **OUTPUT FILE?** would still be asked, however. For example:

```
←COMPILE((FACT FACT1 FACT2))
LISTING? ST
OUTPUT FILE? FACT.DCOM
(FACT COMPILING)
.
.
(FACT REDEFINED)
.
.
(FACT2 REDEFINED)
```

(FACT FACT1 FACT2)
←

This process caused the functions **FACT**, **FACT1**, and **FACT2** to be compiled, redefined, and the compiled definitions also written on the file **FACT.DCOM** for subsequent loading.

18.1 Compiler Printout

In Interlisp-D, for each function *FN* compiled, whether by **TCOMPL**, **RECOMPILE**, or **COMPILE**, the compiler prints:

(*FN* **(***ARG_1 ... ARG_N***) (uses:** *VAR_1 ... VAR_N***) (calls:** *FN_1 ... FN_N***))**

The message is printed at the beginning of the second pass of the compilation of *FN*. (*ARG_1 ... ARG_N*) is the list of arguments to *FN*; following "**uses:**" are the free variables referenced or set in *FN* (not including global variables); following "**calls:**" are the undefined functions called within *FN*.

If the compilation of *FN* causes the generation of one or more auxilary functions (see page 18.10), a compiler message will be printed for these functions before the message for *FN*, e.g.,

(FOOA0027 (X) (uses: XX))
(FOO (A B))

When compiling a block, the compiler first prints (*BLKNAME* *$BLKFN_1$* *$BLKFN_2$* ...). Then the normal message is printed for the entire block. The names of the arguments to the block are generated by suffixing "#" and a number to the block name, e.g., **(FOOBLOCK (FOOBLOCK#0 FOOBLOCK#1)** *FREE-VARIABLES*). Then a message is printed for each *entry* to the block.

In addition to the above output, both **RECOMPILE** and **BRECOMPILE** print the name of each function that is being copied from the old compiled file to the new compiled file. The normal compiler message is printed for each function that is actually compiled.

The compiler prints out error messages when it encounters problems compiling a function. For example:

----- In BAZ:
******* (BAZ - illegal RETURN)**

The above error message indicates that an "**illegal RETURN**" compiler error occurred while trying to compile the function **BAZ**. Some compiler errors cause the compilation to terminate, producing nothing; however, there are other compiler errors

which do not stop compilation. The compiler error messages are described on page 18.22.

Compiler printout and error messages go to the file **COUTFILE**, initially **T**. **COUTFILE** can also be set to the name of a file opened for output, in which case all compiler printout will go to **COUTFILE**, i.e. the compiler will compile "silently." However, any error messages will be printed to both **COUTFILE** as well as **T**.

18.2 Global Variables

Variables that appear on the list **GLOBALVARS**, or have the property **GLOBALVAR** with value **T**, or are declared with the **GLOBALVARS** file package command (page 17.37), are called global variables. Such variables are always accessed through their top level value when they are used freely in a compiled function. In other words, a reference to the value of a global variable is equivalent to calling **GETTOPVAL** (page 2.4) on the variable, regardless of whether or not it is bound in the current access chain. Similarly, **(SETQ** *VARIABLE VALUE***)** will compile as **(SETTOPVAL (QUOTE** *VARIABLE***)** *VALUE***)**.

All system parameters, unless otherwise specified, are declared as global variables. Thus, *rebinding* these variables in a deep bound system (like Interlisp-D) will not affect the behavior of the system: instead, the variables must be *reset* to their new values, and if they are to be restored to their original values, reset again. For example, the user might write

(SETQ *GLOBALVARIABLE NEWVALUE***)**
FORM
(SETQ *GLOBALVARIABLE OLDVALUE***)**

Note that in this case, if an error occurred during the evaluation of *FORM*, or a control-D was typed, the global variable would not be restored to its original value. The function **RESETVAR** (page 14.25) provides a convenient way of resetting global variables in such a way that their values are restored even if an error occurred or control-D is typed.

Note: The variables that a given function accesses as global variables can be determined by using the function **CALLS** (page 19.22).

18.3 Local Variables and Special Variables

In normal compiled and interpreted code, all variable bindings are accessible by lower level functions because the variable's name is associated with its value. We call such variables *special* variables, or specvars. As mentioned earlier, the block compiler normally does *not* associate names with variable values. Such unnamed variables are not accessible from outside the function which binds them and are therefore *local* to that function. We call such unnamed variables local variables, or localvars.

The time economies of local variables can be achieved without block compiling by use of declarations. Using local variables will increase the speed of compiled code; the price is the work of writing the necessary specvar declarations for those variables which need to be accessed from outside the block.

LOCALVARS and **SPECVARS** are variables that affect compilation. During regular compilation, **SPECVARS** is normally **T**, and **LOCALVARS** is **NIL** or a list. This configuration causes all variables bound in the functions being compiled to be treated as special *except* those that appear on **LOCALVARS**. During block compilation, **LOCALVARS** is normally **T** and **SPECVARS** is **NIL** or a list. All variables are then treated as local *except* those that appear on **SPECVARS**.

Declarations to set **LOCALVARS** and **SPECVARS** to other values, and therefore affect how variables are treated, may be used at several levels in the compilation process with varying scope.

(1) The declarations may be included in the filecoms of a file, by using the **LOCALVARS** and **SPECVARS** file package commands (page 17.37). The scope of the declaration is then the entire file:

... (LOCALVARS . T) (SPECVARS X Y) ...

(2) The declarations may be included in block declarations; the scope is then the block, e.g.,

(BLOCKS ((FOOBLOCK FOO FIE (SPECVARS . T) (LOCALVARS X)))

(3) The declarations may also appear in individual functions, or in **PROG**'s or **LAMBDA**'s within a function, using the **DECLARE** function. In this case, the scope of the declaration is the function or the **PROG** or **LAMBDA** in which it appears. **LOCALVARS** and **SPECVARS** declarations must appear immediately after the variable list in the function, **PROG**, or **LAMBDA**, but intervening comments are permitted. For example:

```
(DEFINEQ ((FOO
    (LAMBDA (X Y)
      (DECLARE (LOCALVARS Y))
      (PROG (X Y Z)
        (DECLARE (LOCALVARS X))
        ... ]
```

If the above function is compiled (non-block), the outer **X** will be special, the **X** bound in the **PROG** will be local, and both bindings of **Y** will be local.

Declarations for **LOCALVARS** and **SPECVARS** can be used in two ways: either to cause variables to be treated the same whether the function(s) are block compiled or compiled normally, or to affect one compilation mode while not affecting the default in the other mode. For example:

```
(LAMBDA (X Y)
    (DECLARE (SPECVARS . T))
    (PROG (Z) ... ]
```

will cause **X**, **Y**, and **Z** to be specvars for both block and normal compilation while

```
(LAMBDA (X Y)
    (DECLARE (SPECVARS X))
    ... ]
```

will make **X** a specvar when block compiling, but when regular compiling the declaration will have no effect, because the default value of specvars would be **T**, and therefore *both* **X** and **Y** will be specvars by default.

Although **LOCALVARS** and **SPECVARS** declarations have the same form as other components of block declarations such as **(LINKFNS . T)**, their operation is somewhat different because the two variables are not independent. **(SPECVARS . T)** will cause **SPECVARS** to be set to **T**, and **LOCALVARS** to be set to **NIL**. **(SPECVARS V1 V2 ...)** will have *no* effect if the value of **SPECVARS** is **T**, but if it is a list (or **NIL**), **SPECVARS** will be set to the union of its prior value and **(V1 V2 ...)**. The operation of **LOCALVARS** is analogous. Thus, to affect both modes of compilation one of the two (**LOCALVARS** or **SPECVARS**) must be declared **T** before specifying a list for the other.

Note: The variables that a given function binds as local variables or accesses as special variables can be determined by using the function **CALLS** (page 19.22).

Note: **LOCALVARS** and **SPECVARS** declarations affect the compilation of local variables within a function, but the arguments to functions are always accessable as specvars. This can be changed by re-defining the following function:

(DASSEM.SAVELOCALVARS *FN***)** [Function]

This function is called by the compiler to determine whether argument information for *FN* should be written on the compiled file for *FN*. If it returns **NIL**, the argument information is *not* saved, and the function is stored with arguments **U**, **V**, **W**, etc instead of the originals.

Initially, **DASSEM.SAVELOCALVARS** is defined to return **T**. **(MOVD 'NILL 'DASSEM.SAVELOCALVARS)** causes the compiler to retain no local variable or argument names. Alternatively, **DASSEM.SAVELOCALVARS** could be redefined as a more complex predicate, to allow finer discrimination.

18.4 Constants

Interlisp allows the expression of constructions which are intended to be description of their constant values. The following functions are used to define constant values. The function **SELECTC** (page 9.7) provides a mechanism for comparing a value to a number of constants.

(CONSTANT *X*) [Function]

This function enables the user to define that the expression *X* should be treated as a "constant" value. When **CONSTANT** is interpreted, *X* is evaluted each time it is encountered. If the **CONSTANT** form is compiled, however, the expression will be evaluated only once.

If the value of *X* has a readable print name, then it will be evaluated at compile-time, and the value will be saved as a literal in the compiled function's definition, as if **(QUOTE** *VALUE-OF-EXPRESSION***)** had appeared instead of **(CONSTANT** *EXPRESSION***)**.

If the value of *X* does not have a readable print name, then the expression *X* itself will be saved with the function, and it will be evaluated when the function is first loaded. The value will then be stored in the function's literals, and will be retrieved on future references.

If a user program needed a list of 30 **NIL**s, the user could specify **(CONSTANT (to 30 collect NIL))** instead of **(QUOTE (NIL NIL ...))**. The former is more concise and displays the important parameter much more directly than the latter.

CONSTANT can also be used to denote values that cannot be quoted directly, such as **(CONSTANT (PACK NIL))**, **(CONSTANT (ARRAY 10))**. It is also useful to parameterize quantities that are constant at run time but may differ at compile time, e.g., **(CONSTANT BITSPERWORD)** in a program is exactly equivalent to 36, if the variable **BITSPERWORD** is bound to 36 when the **CONSTANT** expression is evaluated at compile time.

Whereas the function **CONSTANT** attempts to evaluate the expression as soon as possible (compile-time, load-time, or

first-run-time), other options are available, using the folowing two function:

(LOADTIMECONSTANT *X*) [Function]

Similar to **CONSTANT**, except that the evaluation of *X* is deferred until the compiled code for the containing function is loaded in. For example, **(LOADTIMECONSTANT (DATE))** will return the date the code was loaded. If **LOADTIMECONSTANT** is interpreted, it merely returns the value of *X*.

(DEFERREDCONSTANT *X*) [Function]

Similar to **CONSTANT**, except that the evaluation of *X* is always deferred until the compiled function is first run. This is useful when the storage for the constant is excessive so that it shouldn't be allocated until (unless) the function is actually invoked. If **DEFERREDCONSTANT** is interpreted, it merely returns the value of *X*.

(CONSTANTS VAR_1 VAR_2 ... VAR_N) [NLambda NoSpread Function]

Defines VAR_1, ... VAR_N (unevaluated) to be compile-time constants. Whenever the compiler encounters a (free) reference to one of these constants, it will compile the form **(CONSTANT** VAR_i**)** instead.

If VAR_i is a list of the form (*VAR FORM*), a free reference to the variable will compile as **(CONSTANT** *FORM***)**.

The compiler prints a warning if user code attempts to bind a variable previously declared as a constant.

Constants can be saved using the **CONSTANTS** file package command (page 17.37).

18.5 Compiling Function Calls

When compiling the call to a function, the compiler must know the type of the function, to determine how the arguments should be prepared (evaluated/unevaluated, spread/nospread). There are three seperate cases: lambda, nlambda spread, and nlambda nospread functions.

To determine which of these three cases is appropriate, the compiler will first look for a definition among the functions in the file that is being compiled. The function can be defined anywhere in any of the files given as arguments to **BCOMPL**, **TCOMPL**, **BRECOMPILE** or **RECOMPILE**. If the function is not contained in the file, the compiler will look for other

information in the variables **NLAMA**, **NLAML**, and **LAMS**, which can be set by the user:

NLAMA [Variable]

(for **NLAM**bda **A**toms) A list of functions to be treated as nlambda nospread functions by the compiler.

NLAML [Variable]

(for **NLAM**bda **L**ist) A list of functions to be treated as nlambda spread functions by the compiler.

LAMS [Variable]

A list of functions to be treated as lambda functions by the compiler. Note that including functions on **LAMS** is only necessary to override in-core nlambda definitions, since in the absence of other information, the compiler assumes the function is a lambda.

If the function is not contained in a file, or on the lists **NLAMA**, **NLAML**, or **LAMS**, the compiler will look for a current definition in the Interlisp system, and use its type. If there is no current definition, next **COMPILEUSERFN** is called:

COMPILEUSERFN [Variable]

When compiling a function call, if the function type cannot be found by looking in files, the variables **NLAMA**, **NLAML**, or **LAMS**, or at a current definition, then if the value of **COMPILEUSERFN** is not **NIL**, the compiler calls (the value of) **COMPILEUSERFN** giving it as arguments **CDR** of the form and the form itself, i.e., the compiler does **(APPLY* COMPILEUSERFN (CDR** *FORM***)** *FORM***)**. If a non-**NIL** value is returned, it is compiled instead of *FORM*. If **NIL** is returned, the compiler compiles the original expression as a call to a lambda spread that is not yet defined.

Note that **COMPILEUSERFN** is only called when the compiler encounters a *list* **CAR** of which is not the name of a defined function. The user can instruct the compiler about how to compile other data types via **COMPILETYPELST**, page 18.11.

CLISP uses **COMPILEUSERFN** to tell the compiler how to compile iterative statements, **IF-THEN-ELSE** statements, and pattern match constructs (See page 18.11).

If the compiler cannot determine the function type by any of the means above, it assumes that the function is a lambda function, and its arguments are to be evaluated.

If there are nlambda functions called from the functions being compiled, and they are only defined in a separate file, they must

be included on **NLAMA** or **NLAML**, or the compiler will incorrectly assume that their arguments are to be evaluated, and compile the calling function correspondingly. Note that this is only necessary if the compiler does not "know" about the function. If the function is defined at compile time, or is handled via a macro, or is contained in the same group of files as the functions that call it, the compiler will automatically handle calls to that function correctly.

18.6 FUNCTION and Functional Arguments

Compiling the function **FUNCTION** (page 10.18) may involve creating and compiling a seperate "auxiliary function", which will be called at run time. An auxiliary function is named by attaching a **GENSYM** (page 2.10) to the end of the name of the function in which they appear, e.g., **FOOA0003**. For example, suppose **FOO** is defined as **(LAMBDA (X) ... (FOO1 X (FUNCTION ...)) ...)** and compiled. When **FOO** is run, **FOO1** will be called with two arguments, **X**, and **FOOA000***N* and **FOO1** will call **FOOA000***N* each time it uses its functional argument.

Compiling **FUNCTION** will *not* create an auxiliary function if it is a functional argument to a function that compiles open, such as most of the mapping functions (**MAPCAR**, **MAPLIST**, etc.). Note that a considerable savings in time could be achieved by making **FOO1** compile open via a computed macro (page 10.21), e.g.

```
(PUTPROP 'FOO1 'MACRO
  '(Z (LIST (SUBST (CADADR Z)
                   (QUOTE FN)
                   DEF)
            (CAR Z)))
```

DEF is the definition of **FOO1** as a function of just its first argument, and **FN** is the name used for its functional argument in its definition. In this case, **(FOO1 X (FUNCTION ...))** would compile as an expression, containing the argument to **FUNCTION** as an open **LAMBDA** expression. Thus you save not only the function call to **FOO1**, but also each of the function calls to its functional argument. For example, if **FOO1** operates on a list of length ten, eleven function calls will be saved. Of course, this savings in time costs space, and the user must decide which is more important.

18.7 Open Functions

When a function is called from a compiled function, a system routine is invoked that sets up the parameter and control push lists as necessary for variable bindings and return information. If the amount of time spent *inside* the function is small, this function calling time will be a significant percentage of the total time required to use the function. Therefore, many "small" functions, e.g., **CAR**, **CDR**, **EQ**, **NOT**, **CONS** are always compiled "open", i.e., they do not result in a function call. Other larger functions such as **PROG**, **SELECTQ**, **MAPC**, etc. are compiled open because they are frequently used. The user can make other functions compile open via **MACRO** definitions (see page 10.21). The user can also affect the compiled code via **COMPILEUSERFN** (page 18.9) and **COMPILETYPELST** (page 18.11).

18.8 COMPILETYPELST

Most of the compiler's mechanism deals with how to handle forms (lists) and variables (literal atoms). The user can affect the compiler's behaviour with respect to lists and literal atoms in a number of ways, e.g. macros, declarations, **COMPILEUSERFN** (page 18.9), etc. **COMPILETYPELST** allows the user to tell the compiler what to do when it encounters a data type *other* than a list or an atom. It is the facility in the compiler that corresponds to **DEFEVAL** (page 10.13) for the interpreter.

COMPILETYPELST [Variable]

A list of elements of the form (*TYPENAME* . *FUNCTION*). Whenever the compiler encounters a datum that is not a list and not an atom (or a number) in a context where the datum is being evaluated, the type name of the datum is looked up on **COMPILETYPELST**. If an entry appears **CAR** of which is equal to the type name, **CDR** of that entry is applied to the datum. If the value returned by this application is *not* **EQ** to the datum, then that value is compiled instead. If the value *is* **EQ** to the datum, or if there is no entry on **COMPILETYPELST** for this type name, the compiler simply compiles the datum as (**QUOTE** *DATUM*).

18.9 Compiling CLISP

Since the compiler does not know about CLISP, in order to compile functions containing CLISP constructs, the definitions

must first be **DWIMIFY**ed (page 21.18). The user can automate this process in several ways:

(1) If the variable **DWIMIFYCOMPFLG** is **T**, the compiler will always **DWIMIFY** expressions before compiling them. **DWIMIFYCOMPFLG** is initially **NIL**.

(2) If a file has the property **FILETYPE** with value **CLISP** on its property list, **TCOMPL**, **BCOMPL**, **RECOMPILE**, and **BRECOMPILE** will operate as though **DWIMIFYCOMPFLG** is **T** and **DWIMIFY** all expressions before compiling.

(3) If the function definition has a local **CLISP** declaration (see page 21.13), including a null declaration, i.e., just **(CLISP:)**, the definition will be automatically **DWIMIFY**ed before compiling.

Note: **COMPILEUSERFN** (page 18.9) is defined to call **DWIMIFY** on iterative statements, **IF-THEN** statements, and **fetch**, **replace**, and **match** expressions, i.e., any CLISP construct which can be recognized by its **CAR** of form. Thus, if the only CLISP constructs in a function appear inside of iterative statements, **IF** statements, etc., the function does not have to be dwimified before compiling.

If **DWIMIFY** is ever unsuccessful in processing a **CLISP** expression, it will print the error message **UNABLE TO DWIMIFY** followed by the expression, and go into a break unless **DWIMESSGAG = T**. In this case, the expression is just compiled as is, i.e. as though CLISP had not been enabled. The user can exit the break in several different ways: (1) type **OK** to the break, which will cause the compiler to try again, e.g. the user could define some missing records while in the break, and then continue; or (2) type ↑, which will cause the compiler to simply compile the expression as is, i.e. as though CLISP had not been enabled in the first place; or (3) return an expression to be compiled in its place by using the **RETURN** break command (page 14.6).

Note: **TCOMPL**, **BCOMPL**, **RECOMPILE**, and **BRECOMPILE** all scan the entire file before doing any compiling, and take note of the names of all functions that are defined in the file as well as the names of all variables that are set by adding them to **NOFIXFNSLST** and **NOFIXVARSLST**, respectively. Thus, if a function is not currently defined, but *is* defined in the file being compiled, when **DWIMIFY** is called before compiling, it will not attempt to interpret the function name as CLISP when it appears as **CAR** of a form. **DWIMIFY** also takes into account variables that have been declared to be **LOCALVARS**, or **SPECVARS**, either via block declarations or **DECLARE** expressions in the function being compiled, and does not attempt spelling correction on these variables. The declaration **USEDFREE** may also be used to declare variables simply used freely in a function. These variables will also be left alone by **DWIMIFY**. Finally, **NOSPELLFLG** (page 20.13) is reset to **T** when compiling functions from a file (as opposed to

from their in-core definition) so as to suppress spelling correction.

18.10 Compiler Functions

Normally, the compiler is envoked through file package commands that keep track of the state of functions, and manage a set of files, such as **MAKEFILE** (page 17.10). However, it is also possible to explicitly call the compiler using one of a number of functions. Functions may be compiled from in-core definitions (via **COMPILE**), or from definitions in files (**TCOMPL**), or from a combination of in-core and file definitions (**RECOMPILE**).

TCOMPL and **RECOMPILE** produce "compiled" files. Compiled files usually have the same name as the symbolic file they were made from, suffixed with **DCOM** (the compiled file extension is stored as the value of the variable **COMPILE.EXT**). The file name is constructed from the name field only, e.g., **(TCOMPL '<BOBROW>FOO.TEM;3)** produces **FOO.DCOM** on the connected directory. The version number will be the standard default.

A "compiled file" contains the same expressions as the original symbolic file, except that (1) a special **FILECREATED** expression appears at the front of the file which contains information used by the file package, and which causes the message **COMPILED ON** *DATE* to be printed when the file is loaded (the actual string printed is the value of **COMPILEHEADER**); (2) every **DEFINEQ** in the symbolic file is replaced by the corresponding compiled definitions in the compiled file; and (3) expressions following a **DONTCOPY** tag inside of a **DECLARE:** (page 17.40) that appears in the symbolic file are not copied to the compiled file. The compiled definitions appear at the front of the compiled file, i.e., before the other expressions in the symbolic file, *regardless of where they appear in the symbolic file.* The only exceptions are expressions that follow a **FIRST** tag inside of a **DECLARE:** (page 17.40). This "compiled" file can be loaded into any Interlisp system with **LOAD** (page 17.6).

Note: When a function is compiled from its in-core definition (as opposed to being compiled from a definition in a file), and the function has been modified by **BREAK**, **TRACE**, **BREAKIN**, or **ADVISE**, it is first restored to its original state, and a message is printed out, e.g., **FOO UNBROKEN**. If the function is not defined by an expr definition, the value of the function's **EXPR** property is used for the compilation, if there is one. If there is no **EXPR** property, and the compilation is being performed by **RECOMPILE**, the definition of the function is obtained from the

file (using **LOADFNS**). Otherwise, the compiler prints (*FN* **NOT COMPILEABLE**), and goes on to the next function.

(COMPILE *X FLG***)** [Function]

X is a list of functions (if atomic, **(LIST** *X***)** is used). **COMPILE** first asks the standard compiler questions (page 18.1), and then compiles each function on *X*, using its in-core definition. Returns *X*.

If compiled definitions are being written to a file, the file is closed unless *FLG* = **T**.

(COMPILE1 *FN DEF* —**)** [Function]

Compiles *DEF*, redefining *FN* if **STRF** = **T** (**STRF** is one of the variables set by **COMPSET**, page 18.1). **COMPILE1** is used by **COMPILE**, **TCOMPL**, and **RECOMPILE**.

If **DWIMIFYCOMPFLG** is **T**, or *DEF* contains a CLISP declaration, *DEF* is dwimified before compiling. See page 18.11.

(TCOMPL *FILES***)** [Function]

TCOMPL is used to "compile files"; given a symbolic **LOAD** file (e.g., one created by **MAKEFILE**), it produces a "compiled file". *FILES* is a list of symbolic files to be compiled (if atomic, **(LIST** *FILES***)** is used). **TCOMPL** asks the standard compiler questions (page 18.1), except for "**OUTPUT FILE:**". The output from the compilation of each symbolic file is written on a file of the same name suffixed with **DCOM**, e.g., **(TCOMPL '(SYM1 SYM2))** produces two files, **SYM1.DCOM** and **SYM2.DCOM**.

TCOMPL processes the files one at a time, reading in the entire file. For each **FILECREATED** expression, the list of functions that were marked as changed by the file package is noted, and the **FILECREATED** expression is written onto the output file. For each **DEFINEQ** expression, **TCOMPL** adds any nlambda functions defined in the **DEFINEQ** to **NLAMA** or **NLAML**, and adds lambda functions to **LAMS**, so that calls to these functions will be compiled correctly (see page 18.9). **NLAMA**, **NLAML**, and **LAMS** are rebound to their top level values (using **RESETVAR**) by all of the compiling functions, so that any additions to these lists while inside of these functions will not propagate outside. Expressions beginning with **DECLARE:** are processed specially (see page 17.40). All other expressions are collected to be subsequently written onto the output file.

After processing the file in this fashion, **TCOMPL** compiles each function, except for those functions which appear on the list **DONTCOMPILEFNS** (initially **NIL**), and writes the compiled definition onto the output file. **TCOMPL** then writes onto the output file the other expressions found in the symbolic file.

DONTCOMPILEFNS might be used for functions that compile open, since their definitions would be superfluous when operating with the compiled file. Note that **DONTCOMPILEFNS** can be set via block declarations (see page 18.17).

Note: If the rootname of a file has the property **FILETYPE** with value **CLISP**, or value a list containing **CLISP**, **TCOMPL** rebinds **DWIMIFYCOMPFLG** to **T** while compiling the functions on *FILE*, so the compiler will **DWIMIFY** all expressions before compiling them. See page 18.11.

TCOMPL returns a list of the names of the output files. All files are properly terminated and closed. If the compilation of any file is aborted via an error or control-D, all files are properly closed, and the (partially complete) compiled file is deleted.

(RECOMPILE *PFILE CFILE FNS***)** [Function]

The purpose of **RECOMPILE** is to allow the user to update a compiled file without recompiling every function in the file. **RECOMPILE** does this by using the results of a previous compilation. It produces a compiled file similar to one that would have been produced by **TCOMPL**, but at a considerable savings in time by only compiling selected functions, and copying the compiled definitions for the remainder of the functions in the file from an earlier **TCOMPL** or **RECOMPILE** file.

PFILE is the name of the **P**retty file (source file) to be compiled; *CFILE* is the name of the **C**ompiled file containing compiled definitions that may be copied. *FNS* indicates which functions in *PFILE* are to be recompiled, e.g., have been changed or defined for the first time since *CFILE* was made. Note that *PFILE*, not *FNS*, drives **RECOMPILE**.

RECOMPILE asks the standard compiler questions (page 18.1), except for "**OUTPUT FILE:**". As with **TCOMPL**, the output automatically goes to *PFILE*.**DCOM**. **RECOMPILE** processes *PFILE* the same as does **TCOMPL** except that **DEFINEQ** expressions are not actually read into core. Instead, **RECOMPILE** uses the filemap (page 17.55) to obtain a list of the functions contained in *PFILE*. The filemap enables **RECOMPILE** to skip over the **DEFINEQ**s in the file by simply resetting the file pointer, so that in most cases the scan of the symbolic file is very fast (the only processing required is the reading of the non-**DEFINEQ**s and the processing of the **DECLARE:** expressions as with **TCOMPL**). A map is built if the symbolic file does not already contain one, for example if it was written in an earlier system, or with **BUILDMAPFLG** = **NIL** (page 17.56).

After this initial scan of *PFILE*, **RECOMPILE** then processes the functions defined in the file. For each function in *PFILE*, **RECOMPILE** determines whether or not the function is to be (re)compiled. Functions that are members of **DONTCOMPILEFNS**

are simply ignored. Otherwise, a function is recompiled if (1) *FNS* is a list and the function is a member of that list; or (2) *FNS* = **T** or **EXPRS** and the function is defined by an expr definition; or (3) *FNS* = **CHANGES** and the function is marked as having been changed in the **FILECREATED** expression in *PFILE*; or (4) *FNS* = **ALL**. If a function is not to be recompiled, **RECOMPILE** obtains its compiled definition from *CFILE*, and copies it (and all generated subfunctions) to the output file, *PFILE*.**DCOM**. If the function does not appear on *CFILE*, **RECOMPILE** simply recompiles it. Finally, after processing all functions, **RECOMPILE** writes out all other expressions that were collected in the prescan of *PFILE*.

Note: If *FNS* = **ALL**, *CFILE* is superfluous, and does not have to be specified. This option may be used to compile a symbolic file that has never been compiled before, but which has already been loaded (since using **TCOMPL** would require reading the file in a second time).

If *CFILE* = **NIL**, *PFILE*.**DCOM** (the old version of the output file) is used for copying *from*. If both *FNS* and *CFILE* are **NIL**, *FNS* is set to the value of **RECOMPILEDEFAULT**, which is initially **CHANGES**. Thus the user can perform his edits, dump the file, and then simply (**RECOMPILE** '*FILE*) to update the compiled file.

The value of **RECOMPILE** is the file name of the new compiled file, *PFILE*.**DCOM**. If **RECOMPILE** is aborted due to an error or control-D, the new (partially complete) compiled file will be closed and deleted.

RECOMPILE is designed to allow the user to conveniently and *efficiently* update a compiled file, even when the corresponding symbolic file has not been (completely) loaded. For example, the user can perform a **LOADFROM** (page 17.8) to "notice" a symbolic file, edit the functions he wants to change (the editor will automatically load those functions not already loaded), call **MAKEFILE** (page 17.10) to update the symbolic file (**MAKEFILE** will copy the unchanged functions from the old symbolic file), and then perform (**RECOMPILE** *PFILE*).

Note: Since **PRETTYDEF** automatically outputs a suitable **DECLARE:** expression to indicate which functions in the file (if any) are defined as **NLAMBDA**s, calls to these functions will be handled correctly, even though the **NLAMBDA** functions themselves may never be loaded, or even looked at, by **RECOMPILE**.

18.11 Block Compiling

In Interlisp-10, block compiling provides a way of compiling several functions into a single block. Function calls between the component functions of the block are very fast. Thus, compiling a block consisting of just a single recursive function may be yield great savings if the function calls itself many times. The output of a block compilation is a single, usually large, function. Calls from within the block to functions outside of the block look like regular function calls. A block can be entered via several different functions, called entries. These must be specified when the block is compiled.

In Interlisp-D, block compiling is handled somewhat differently; block compiling provides a mechanism for hiding function names internal to a block, but it does not provide a performance improvement. Block compiling in Interlisp-D works by automatically renaming the block functions with special names, and calling these functions with the normal function-calling mechanisms. Specifically, a function *FN* is renamed to *\BLOCK-NAME/FN*. For example, function **FOO** in block **BAR** is renamed to "**\BAR/FOO**". Note that it is possible with this scheme to break functions internal to a block.

18.11.1 Block Declarations

Block compiling a file frequently involves giving the compiler a lot of information about the nature and structure of the compilation, e.g., block functions, entries, specvars, etc. To help with this, there is the **BLOCKS** file package command (page 17.42), which has the form:

(BLOCKS $BLOCK_1$ $BLOCK_2$ ... $BLOCK_N$**)**

where each $BLOCK_i$ is a block declaration. The **BLOCKS** command outputs a **DECLARE:** expression, which is noticed by **BCOMPL** and **BRECOMPILE**. **BCOMPL** and **BRECOMPILE** are sensitive to these declarations and take the appropriate action.

Note: Masterscope (page 19.1) includes a facility for checking the block declarations of a file or files for various anomalous conditions, e.g. functions in block declarations which aren't on the file(s), functions in **ENTRIES** not in the block, variables that may not need to be **SPECVARS** because they are not used freely below the places they are bound, etc.

A block declaration is a list of the form:

(*BLKNAME* $BLKFN_1$ ... $BLKFN_M$
(VAR_1 . $VALUE_1$) ... (VAR_N . $VALUE_N$))

BLKNAME is the name of a block. $BLKFN_1$ ... $BLKFN_M$ are the functions in the block and correspond to *BLKFNS* in the call to

BLOCKCOMPILE. The (VAR_i . $VALUE_i$) expressions indicate the settings for variables affecting the compilation of that block. If $VALUE_i$ is atomic, then VAR_i is set to $VALUE_i$, otherwise VAR_i is set to the **UNION** (page 3.11) of $VALUE_i$ and the current value of the variable VAR_i. Also, expressions of the form (*VAR* * *FORM*) will cause *FORM* to be evaluated and the resulting list used as described above (e.g. **(GLOBALVARS * MYGLOBALVARS)**).

For example, consider the block declaration below. The block name is **EDITBLOCK**, it includes a number of functions (**EDITL0**, **EDITL1**, ... **EDITH**), and it sets the variables **ENTRIES**, **SPECVARS**, **RETFNS**, and **GLOBALVARS**.

```
(EDITBLOCK
  EDITL0 EDITL1 UNDOEDITL EDITCOM EDITCOMA
  EDITMAC EDITCOMS EDIT]UNDO UNDOEDITCOM EDITH
  (ENTRIES EDITL0 ## UNDOEDITL)
  (SPECVARS L COM LCFLG #1 #2 #3 LISPXBUFS)
  (RETFNS EDITL0)
  (GLOBALVARS EDITCOMSA EDITCOMSL EDITOPS))
```

Whenever **BCOMPL** or **BRECOMPILE** encounter a block declaration, they rebind **RETFNS**, **SPECVARS**, **GLOBALVARS**, **BLKLIBRARY**, and **DONTCOMPILEFNS** to their top level values, bind **BLKAPPLYFNS** and **ENTRIES** to **NIL**, and bind **BLKNAME** to the first element of the declaration. They then scan the rest of the declaration, setting these variables as described above. When the declaration is exhausted, the block compiler is called and given **BLKNAME**, the list of block functions, and **ENTRIES**.

If a function appears in a block declaration, but is not defined in one of the files, then if it has an in-core definition, this definition is used and a message printed **NOT ON FILE, COMPILING IN CORE DEFINITION**. Otherwise, the message **NOT COMPILEABLE**, is printed and the block declaration processed as though the function were not on it, i.e. calls to the function will be compiled as external function calls.

Note that since all compiler variables are rebound for each block declaration, the declaration only has to set those variables it wants *changed*. Furthermore, setting a variable in one declaration has no effect on the variable's value for another declaration.

After finishing all blocks, **BCOMPL** and **BRECOMPILE** treat any functions in the file that did not appear in a block declaration in the same way as do **TCOMPL** and **RECOMPILE**. If the user wishes a function compiled separately as well as in a block, or if he wishes to compile some functions (not blockcompile), with some compiler variables changed, he can use a special pseudo-block declaration of the form

(NIL $BLKFN_1$ ... $BLKFN_M$ (VAR_1 . $VALUE_1$) ... (VAR_N . $VALUE_N$))

which means that $BLKFN_1$ ... $BLKFN_M$ should be compiled after first setting VAR_1 ... VAR_N as described above.

The following variables control other aspects of compiling a block:

RETFNS [Variable]

Value is a list of internal block functions whose names must appear on the stack, e.g., if the function is to be returned from **RETFROM**, **RETTO**, **RETEVAL**, etc. Usually, internal calls between functions in a block are not put on the stack.

BLKAPPLYFNS [Variable]

Value is a list of internal block functions called by other functions in the same block using **BLKAPPLY** or **BLKAPPLY*** for efficiency reasons.

Normally, a call to **APPLY** from inside a block would be the same as a call to any other function outside of the block. If the first argument to **APPLY** turned out to be one of the entries to the block, the block would have to be reentered. **BLKAPPLYFNS** enables a program to compute the name of a function in the block to be called next, without the overhead of leaving the block and reentering it. This is done by including on the list **BLKAPPLYFNS** those functions which will be called in this fashion, and by using **BLKAPPLY** in place of **APPLY**, and **BLKAPPLY*** in place of **APPLY***. If **BLKAPPLY** or **BLKAPPLY*** is given a function not on **BLKAPPLYFNS**, the effect is the same as a call to **APPLY** or **APPLY*** and no error is generated. Note however, that **BLKAPPLYFNS** must be set at *compile* time, not run time, and furthermore, that all functions on **BLKAPPLYFNS** must be in the block, or an error is generated (at compile time), **NOT ON BLKFNS**.

BLKAPPLYFNS [Variable]

Value is a list of functions that are considered to be in the "block library" of functions that should automatically be included in the block if they are called within the block.

Compiling a function open via a macro provides a way of eliminating a function call. For block compiling, the same effect can be achieved by including the function in the block. A further advantage is that the code for this function will appear only once in the block, whereas when a function is compiled open, its code appears at each place where it is called.

The block library feature provides a convenient way of including functions in a block. It is just a convenience since the user can always achieve the same effect by specifying the function(s) in question as one of the block functions, provided it has an expr

definition at compile time. The block library feature simply eliminates the burden of supplying this definition.

To use the block library feature, place the names of the functions of interest on the list **BLKLIBRARY**, and their expr definitions on the property list of the functions under the property **BLKLIBRARYDEF**. When the block compiler compiles a form, it first checks to see if the function being called is one of the block functions. If not, and the function is on **BLKLIBRARY**, its definition is obtained from the property value of **BLKLIBRARYDEF**, and it is automatically included as part of the block.

18.11.2 Block Compiling Functions

There are three user level functions for block compiling, **BLOCKCOMPILE**, **BCOMPL**, and **BRECOMPILE**, corresponding to **COMPILE**, **TCOMPL**, and **RECOMPILE**. Note that all of the remarks on macros, globalvars, compiler messages, etc., all apply equally for block compiling. Using block declarations, the user can intermix in a single file functions compiled normally and block compiled functions.

(BLOCKCOMPILE *BLKNAME BLKFNS ENTRIES FLG***)** [Function]

BLKNAME is the name of a block, *BLKFNS* is a list of the functions comprising the block, and *ENTRIES* a list of entries to the block.

Each of the entries must also be on *BLKFNS* or an error is generated, **NOT ON BLKFNS**. If only one entry is specified, the block name can also be one of the *BLKFNS*, e.g., **(BLOCKCOMPILE 'FOO '(FOO FIE FUM) '(FOO))**. However, if more than one entry is specified, an error will be generated, **CAN'T BE BOTH AN ENTRY AND THE BLOCK NAME**.

If *ENTRIES* is **NIL**, **(LIST** *BLKNAME***)** is used, e.g., **(BLOCKCOMPILE 'COUNT '(COUNT COUNT1))**

If *BLKFNS* is **NIL**, **(LIST** *BLKNAME***)** is used, e.g., **(BLOCKCOMPILE 'EQUAL)**

BLOCKCOMPILE asks the standard compiler questions (page 18.1), and then begins compiling. As with **COMPILE**, if the compiled code is being written to a file, the file is closed unless *FLG* = **T**. The value of **BLOCKCOMPILE** is a list of the entries, or if *ENTRIES* = **NIL**, the value is *BLKNAME*.

The output of a call to **BLOCKCOMPILE** is one function definition for *BLKNAME*, plus definitions for each of the functions on *ENTRIES* if any. These entry functions are very short functions which immediately call *BLKNAME*.

(BCOMPL *FILES CFILE* — —**)** [Function]

FILES is a list of symbolic files (if atomic, **(LIST** *FILES***)** is used). **BCOMPL** differs from **TCOMPL** in that it compiles all of the files at once, instead of one at a time, in order to permit one block to contain functions in several files. (If you have several files to be **BCOMPL**ed *separately*, you must make several calls to **BCOMPL**.) Output is to *CFILE* if given, otherwise to a file whose name is **(CAR** *FILES***)** suffixed with **DCOM**. For example, **(BCOMPL '(EDIT WEDIT))** produces one file, **EDIT.DCOM**.

BCOMPL asks the standard compiler questions (page 18.1), except for "**OUTPUT FILE:**", then processes each file exactly the same as **TCOMPL** (page 18.14). **BCOMPL** next processes the block declarations as described above. Finally, it compiles those functions not mentioned in one of the block declarations, and then writes out all other expressions.

If *any* of the files have property **FILETYPE** with value **CLISP**, or a list containing **CLISP**, then **DWIMIFYCOMPFLG** is rebound to **T** for *all* of the files. See page 18.11.

The value of **BCOMPL** is the output file (the new compiled file). If the compilation is aborted due to an error or control-D, all files are closed and the (partially complete) output file is deleted.

Note that it is permissible to **TCOMPL** files set up for **BCOMPL**; the block declarations will simply have no effect. Similarly, you can **BCOMPL** a file that does not contain any block declarations and the result will be the same as having **TCOMPL**ed it.

(BRECOMPILE *FILES CFILE FNS* —**)** [Function]

BRECOMPILE plays the same role for **BCOMPL** that **RECOMPILE** plays for **TCOMPL**. Its purpose is to allow the user to update a compiled file without requiring an entire **BCOMPL**.

FILES is a list of symbolic files (if atomic, **(LIST** *FILES***)** is used). *CFILE* is the compiled file produced by **BCOMPL** or a previous **BRECOMPILE** that contains compiled definitions that may be copied. The interpretation of *FNS* is the same as with **RECOMPILE**.

BRECOMPILE asks the standard compiler questions (page 18.1), except for "**OUTPUT FILE:**". As with **BCOMPL**, output automatically goes to *FILE*.**DCOM**, where *FILE* is the first file in *FILES*.

BRECOMPILE processes each file the same as **RECOMPILE** (page 18.15), then processes each block declaration. If *any* of the functions in the block are to be recompiled, the entire block must be (is) recompiled. Otherwise, the block is copied from *CFILE* as with **RECOMPILE**. For pseudo-block declarations of the form **(NIL** *FN1* **...)**, all variable assignments are made, but only those functions indicated by *FNS* are recompiled.

After completing the block declarations, **BRECOMPILE** processes all functions that do not appear in a block declaration, recompiling those dictated by *FNS*, and copying the compiled definitions of the remaining from *CFILE*.

Finally, **BRECOMPILE** writes onto the output file the "other expressions" collected in the initial scan of *FILES*.

The value of **BRECOMPILE** is the output file (the new compiled file). If the compilation is aborted due to an error or control-D, all files are closed and the (partially complete) output file is deleted.

If *CFILE* = **NIL**, the old version of *FILE*.**DCOM** is used, as with **RECOMPILE**. In addition, if *FNS* and *CFILE* are both **NIL**, *FNS* is set to the value of **RECOMPILEDEFAULT**, initially **CHANGES**.

18.12 Compiler Error Messages

Messages describing errors in the function being compiled are also printed on the terminal. These messages are always preceded by *****. Unless otherwise indicated below, the compilation will continue.

(*FN* NOT ON FILE, COMPILING IN CORE DEFINITION)

From calls to **BCOMPL** and **BRECOMPILE**.

(*FN* NOT COMPILEABLE) An **EXPR** definition for *FN* could not be found. In this case, no code is produced for *FN*, and the compiler proceeds to the next function to be compiled, if any.

(*FN* NOT FOUND) Occurs when **RECOMPILE** or **BRECOMPILE** try to copy the compiled definition of *FN* from *CFILE*, and cannot find it. In this case, no code is copied and the compiler proceeds to the next function to be compiled, if any.

(*FN* NOT ON BLKFNS) *FN* was specified as an entry to a block, or else was on **BLKAPPLYFNS**, but did not appear on the *BLKFNS*. In this case, no code is produced for the entire block and the compiler proceeds to the next function to be compiled, if any.

(*FN* CAN'T BE BOTH AN ENTRY AND THE BLOCK NAME)

In this case, no code is produced for the entire block and the compiler proceeds to the next function to be compiled, if any.

(*BLKNAME* - USED BLKAPPLY WHEN NOT APPLICABLE)

BLKAPPLY is used in the block *BLKNAME*, but there are no **BLKAPPLYFNS** or **ENTRIES** declared for the block.

(*VAR* SHOULD BE A SPECVAR - USED FREELY BY *FN*)

While compiling a block, the compiler has already generated code to bind *VAR* as a **LOCALVAR**, but now discovers that *FN* uses

VAR freely. *VAR* should be declared a **SPECVAR** and the block recompiled.

((* --) COMMENT USED FOR VALUE) A comment appears in a context where its value is being used, e.g. **(LIST X (* --) Y)**. The compiled function will run, but the value at the point where the comment was used is undefined.

((*FORM*) - NON-ATOMIC CAR OF FORM) If user intended to treat the value of *FORM* as a function, he should use **APPLY*** (page 10.12). *FORM* is compiled as if **APPLY*** had been used.

((SETQ *VAR EXPR* --) BAD SETQ) **SETQ** of more than two arguments.

(*FN* - USED AS ARG TO NUMBER FN?) The value of a predicate, such as **GREATERP** or **EQ**, is used as an argument to a function that expects numbers, such as **IPLUS**.

(*FN* - NO LONGER INTERPRETED AS FUNCTIONAL ARGUMENT)
The compiler has assumed *FN* is the name of a function. If the user intended to treat the *value* of *FN* as a function, **APPLY*** (page 10.12) should be used. This message is printed when *FN* is not defined, and is also a local variable of the function being compiled.

(*FN* - ILLEGAL RETURN) **RETURN** encountered when not in **PROG**.

(*TG* - ILLEGAL GO) **GO** encountered when not in a **PROG**.

(*TG* - MULTIPLY DEFINED TAG) *TG* is a **PROG** label that is defined more than once in a single **PROG**. The second definition is ignored.

(*TG* - UNDEFINED TAG) *TG* is a **PROG** label that is referenced but not defined in a **PROG**.

(*VAR* - NOT A BINDABLE VARIABLE) *VAR* is **NIL**, **T**, or else not a literal atom.

(*VAR VAL* -- BAD PROG BINDING) Occurs when there is a prog binding of the form (*VAR* VAL_1 ... VAL_N).

(*TG* - MULTIPLY DEFINED TAG, LAP) *TG* is a label that was encountered twice during the second pass of the compilation. If this error occurs with no indication of a multiply defined tag during pass one, the tag is in a LAP macro.

(*TG* - UNDEFINED TAG, LAP) *TG* is a label that is referenced during the second pass of compilation and is not defined. LAP treats *TG* as though it were a **COREVAL**, and continues the compilation.

(*TG* - MULTIPLY DEFINED TAG, ASSEMBLE) *TG* is a label that is defined more than once in an assemble form.

(*TG* - UNDEFINED TAG, ASSEMBLE) *TG* is a label that is referenced but not defined in an assemble form.

(*OP* - OPCODE? - ASSEMBLE) *OP* appears as **CAR** of an assemble statement, and is illegal.

(NO BINARY CODE GENERATED OR LOADED FOR *FN*)
A previous error condition was sufficiently serious that binary code for *FN* cannot be loaded without causing an error.

[This page intentionally left blank]

[This page intentionally left blank]

19. MASTERSCOPE

Masterscope is an interactive program for analyzing and cross referencing user programs. It contains facilities for analyzing user functions to determine what other functions are called, how and where variables are bound, set, or referenced, as well as which functions use particular record declarations. Masterscope is able to analyze definitions directly from a file as well as in-core definitions.

Masterscope maintains a database of the results of the analyses it performs. Via a simple command language, the user may interrogate the database, call the editor on those expressions in functions that were analyzed which use variables or functions in a particular way, or display the tree structure of function calls among any set of functions.

Masterscope is interfaced with the editor and file package so that when a function is edited or a new definition loaded in, Masterscope knows that it must re-analyze that function.

The following sample session illustrates some of these facilities.

```
50←. ANALYZE FUNCTIONS ON RECORD
.....
NIL
51←. WHO CALLS RECFIELDLOOK
(RECFIELDLOOK ACCESSDEF ACCESSDEF2 EDITREC)
52←. EDIT WHERE ANY CALL RECFIELDLOOK
RECFIELDLOOK :
(RECFIELDLOOK (CDR Y) FIELD)
tty:
5*OK
ACCESSDEF :
(RECFIELDLOOK DECLST FIELD VAR1)
6*OK
(RECFIELDLOOK USERRECLST FIELD)
7*N VAR1
8*OK
ACCESSDEF2 :
(RECFIELDLOOK (RECORD.SUBDECS TRAN) FIELD)
tty:
(RECFIELDLOOK (RECORD.SUBDECS TRAN) FIELD)
9*N (CAR TAIL]
10*OK
EDITREC :
(RECFIELDLOOK USERRECLST (CAR EDITRECX))
```

```
11*OK
NIL
53←. WHO CALLS ERROR
..
(EDITREC)
54←. SHOW PATHS TO RECFIELDLOOK FROM ACCESSDEF
(inverted tree)

1. RECFIELDLOOK RECFIELDLOOK
2.          ACCESSDEF
3.          ACCESSDEF2 ACCESSDEF2
4.                  ACCESSDEF
5.                  RECORDCHAIN ACCESSDEF
NIL
55←. WHO CALLS WHO IN /FNS
RECORDSTATEMENT -- /RPLNODE
RECORDECL1 --    /NCONC, /RPLACD, /RPLNODE
RECREDECLARE1 -- /PUTHASH
UNCLISPTRAN --   /PUTHASH, /RPLNODE2
RECORDWORD --    /RPLACA
RECORD1 --       /RPLACA, /SETTOPVAL
EDITREC --       /SETTOPVAL
```

Statement 50 The user directs that the functions on file **RECORD** be analyzed. The leading period and space specify that this line is a Masterscope command. The user may also call Masterscope directly by typing **(MASTERSCOPE)**. Masterscope prints a greeting and prompts with "←. ". Within the top-level executive of Masterscope, the user may issue Masterscope commands, programmer's assistant commands, (e.g., **REDO**, **FIX**), or run programs. The user can exit from the Masterscope executive by typing **OK**. The function . is defined as a nlambda nospread function which interprets its argument as a Masterscope command, executes the command and returns.

Masterscope prints a . whenever it (re)analyzes a function, to let the user know what it is happening. The feedback when Masterscope analyzes a function is controlled by the flag **MSPRINTFLG**: if **MSPRINTFLG** is the atom ".", Masterscope will print out a period. (If an error in the function is detected, "?" is printed instead.) If **MSPRINTFLG** is a number *N*, Masterscope will print the name of the function it is analyzing every *N*th function. If **MSPRINTFLG** is **NIL**, Masterscope won't print anything. Initial setting is ".". Note that the function name is printed when Masterscope starts analyzing, and the comma is printed when it finishes.

Statement 51 The user asks which functions call **RECFIELDLOOK**. Masterscope responds with the list.

Statement 52 The user asks to edit the expressions where the function **RECFIELDLOOK** is called. Masterscope calls **EDITF** on the functions it had analyzed that call **RECFIELDLOOK**, directing the editor to the appropriate expressions. The user then edits some of those expressions. In this example, the teletype editor is used. If Dedit is enabled as the primary editor, it would be called to edit the appropriate functions (see page 16.1).

Statement 53 Next the user asks which functions call **ERROR**. Since some of the functions in the database have been changed, Masterscope re-analyzes the changed definitions (and prints out .'s for each function it analyzes). Masterscope responds that **EDITREC** is the only analyzed function that calls **ERROR**.

Statement 54 The user asks to see a map of the ways in which **RECFIELDLOOK** is called from **ACCESSDEF**. A tree structure of the calls is displayed.

Statement 55 The user then asks to see which functions call which functions in the list **/FNS**. Masterscope responds with a structured printout of these relations.

19.1 Command Language

The user communicates with Masterscope using an English-like command language, e.g., **WHO CALLS PRINT**. With these commands, the user can direct that functions be analyzed, interrogate Masterscope's database, and perform other operations. The commands deal with sets of functions, variables, etc., and relations between them (e.g., call, bind). Sets correspond to English nouns, relations to verbs.

A set of atoms can be specified in a variety of ways, either *explicitly*, e.g., **FUNCTIONS ON FIE** specifies the atoms in **(FILEFNSLST 'FIE)**, or *implicitly*, e.g., **NOT CALLING Y**, where the meaning must be determined in the context of the rest of the command. Such sets of atoms are the basic building blocks which the command language deals with.

Masterscope also deals with relations *between* sets. For example, the relation **CALL** relates functions and other functions; the relations **BIND** and **USE FREELY** relate functions and variables. These relations are what get stored in the Masterscope database when functions are analyzed. In addition, Masterscope "knows" about file package conventions; **CONTAIN** relates files and various types of objects (functions, variables).

Sets and relations are used (along with a few additional words) to form sentence-like *commands*. For example, the command **WHO ON 'FOO USE 'X FREELY** will print out the list of functions contained in the file **FOO** which use the variable **X** freely. The

command **EDIT WHERE ANY CALLS 'ERROR** will call **EDITF** on those functions which have previously been analyzed that directly call **ERROR**, pointing at each successive expression where the call to **ERROR** actually occurs.

19.1.1 Commands

The normal mode of communication with Masterscope is via "commands". These are sentences in the Masterscope command language which direct Masterscope to answer questions or perform various operations.

Note: any command may be followed by **OUTPUT** *FILENAME* to send output to the given file rather than the terminal, e.g. **WHO CALLS WHO OUTPUT CROSSREF**.

ANALYZE *SET* [Masterscope Command]

Analyze the functions in *SET* (and any functions called by them) and include the information gathered in the database. Masterscope will not re-analyzing a function if it thinks it already has valid information about that function in its database. The user may use the command **REANALYZE** (below) to force re-analysis.

Note that whenever a function is referred to in a command as a "subject" of one of the relations, it is automatically analyzed; the user need not give an explicit **ANALYZE** command. Thus, **WHO IN MYFNS CALLS FIE** will automatically analyze the functions in **MYFNS** if they have not already been analyzed.

Note also that only expr definitions will be analyzed; that is, Masterscope will not analyze compiled code. If necessary, the definition will be **DWIMIFY**ed before analysis. If there is no in-core definition for a function (either in the function definition cell or an **EXPR** property), Masterscope will attempt to read in the definition from a file. Files which have been explicitly mentioned previously in some command are searched first. If the definition cannot be found on any of those files, Masterscope looks among the files on **FILELST** for a definition. If a function is found in this manner, Masterscope will print a message "**(reading from** *FILENAME*)". If no definition can be found at all, Masterscope will print a message "*FN* **can't be analyzed**". If the function previously was known, the message "*FN* **disappeared!**" is printed.

REANALYZE *SET* [Masterscope Command]

Causes Masterscope to reanalyze the functions in *SET* (and any functions called by them) even if it thinks it already has valid information in its database. For example, this would be

necessary if the user had disabled or subverted the file package, e.g. performed **PUTD**'s to change the definition of functions.

ERASE *SET* [Masterscope Command]

Erase all information about the functions in *SET* from the database. **ERASE** by itself clears the entire database.

SHOW PATHS *PATHOPTIONS* [Masterscope Command]

Displays a tree of function calls. This is described on page 19.15.

SET RELATION SET [Masterscope Command]

SET **IS** *SET* [Masterscope Command]

SET **ARE** *SET* [Masterscope Command]

This command has the same format as an English sentence with a subject (the first *SET*), a verb (the *RELATION* or **IS** or **ARE**), and an object (the second *SET*). Any of the *SET*s within the command may be preceded by the question determiners **WHICH** or **WHO** (or just **WHO** alone). For example, **WHICH FUNCTIONS CALL X** prints the list of functions that call the function **X**. *RELATION* may be one of the relation words in present tense (**CALL**, **BIND**, **TEST**, **SMASH**, etc.) or used as a passive (e.g., **WHO IS CALLED BY WHO**). Other variants are allowed, e.g. **WHO DOES X CALL**, **IS FOO CALLED BY FIE**, etc.

The interpretation of the command depends on the number of question elements present:

(1) If there is *no* question element, the command is treated as an assertion and Masterscope returns either **T** or **NIL**, depending on whether that assertion is true. Thus, **ANY IN MYFNS CALL HELP** will print **T** if any function in **MYFNS** call the function **HELP**, and **NIL** otherwise.

(2) If there is *one* question element, Masterscope returns the list of items for which the assertion would be true. For example **MYFN BINDS WHO USED FREELY BY YOURFN** prints the list of variables bound by **MYFN** which are also used freely by **YOURFN**.

(3) If there are two question elements, Masterscope will print a doubly indexed list:

←. WHO CALLS WHO IN /FNS^{cr}
RECORDSTATEMENT -- /RPLNODE
RECORDECL1 -- /NCONC, /RPLACD, /RPLNODE
RECREDECLARE1 -- /PUTHASH
UNCLISPTRAN -- /PUTHASH, /RPLNODE2
RECORDWORD -- /RPLACA
RECORD1 -- /RPLACA, /SETTOPVAL

EDITREC -- /SETTOPVAL

EDIT WHERE *SET RELATION SET [- EDITCOMS]* [Masterscope Command]

(**WHERE** may be omitted.) The first *SET* refers to a set of functions. The **EDIT** command calls the editor on each expression where the *RELATION* actually occurs. For example, **EDIT WHERE ANY CALL ERROR** will call **EDITF** on each (analyzed) function which calls **ERROR** stopping within a **TTY:** at each call to **ERROR**. Currently one cannot **EDIT WHERE** a file which **CONTAINS** a datum, nor where one function **CALLS** another **SOMEHOW**.

EDITCOMS, if given, are a list of commands passed to **EDITF** to be performed at each expression. For example, **EDIT WHERE ANY CALLS MYFN DIRECTLY - (SW 2 3) P** will switch the first and second arguments to **MYFN** in every call to **MYFN** and print the result. **EDIT WHERE ANY ON MYFILE CALL ANY NOT @ GETD** will call the editor on any expression involving a call to an undefined function. Note that **EDIT WHERE X SETS Y** will point only at those expressions where **Y** is actually set, and will skip over places where **Y** is otherwise mentioned.

SHOW WHERE *SET RELATION SET* [Masterscope Command]

Like the **EDIT** command except merely prints out the expressions without calling the editor.

EDIT *SET [- EDITCOMS]* [Masterscope Command]

Calls **EDITF** on each function in *SET*. *EDITCOMS*, if given, will be passed as a list of editor commands to be executed. For example **EDIT ANY CALLING FN1 - (R FN1 FN2)** will replace **FN1** by **FN2** in those functions that call **FN1**.

DESCRIBE *SET* [Masterscope Command]

Prints out the **BIND, USE FREELY** and **CALL** information about the functions in *SET*. For example, the command **DESCRIBE PRINTARGS** might print out:

```
PRINTARGS[N,FLG]
  binds:     TEM,LST,X
  calls:     MSRECORDFILE,SPACES,PRIN1
  called by: PRINTSENTENCE,MSHELP,CHECKER
```

This shows that **PRINTARGS** has two arguments, **N** and **FLG**, binds internally the variables **TEM, LST** and **X**, calls **MSRECORDFILE, SPACES** and **PRIN1** and is called by **PRINTSENTENCE, MSHELP,** and **CHECKER**.

The user can specify additional information to be included in the description. **DESCRIBELST** is a list each of whose elements is a list containing a descriptive string and a form. The form is evaluated (it can refer to the name of the funtion being described by the

free variable **FN**); if it returns a non-**NIL** value, the description string is printed followed by the value. If the value is a list, its elements are printed with commas between them. For example, the entry **("types: " (GETRELATION FN '(USE TYPE) T)** would include a listing of the types used by each function.

CHECK *SET* [Masterscope Command]

Checks for various anomolous conditions (mainly in the compiler declarations) for the files in *SET* (if *SET* is not given, **FILELST** is used). For example, this command will warn about variables which are bound but never referenced, functions in **BLOCKS** delarations which aren't on the file containing the declaration, functions declared as **ENTRIES** but not in the block, variables which may not need to be declared **SPECVARS** because they are not used freely below the places where they are bound, etc.

FOR *VARIABLE SET I.S.TAIL* [Masterscope Command]

This command provides a way of combining **CLISP** iterative statements with Masterscope. An iterative statement will be constructed in which *VARIABLE* is iteratively assigned to each element of *SET*, and then the iterative statement tail *I.S.TAIL* is executed. For example,

FOR X CALLED BY FOO WHEN CCODEP DO (PRINTOUT T X ,,, (ARGLIST X) T)

will print out the name and argument list of all of the compiled functions which are called by **FOO**.

19.1.2 Relations

A relation is specified by one of the keywords below. Some of these "verbs" accept modifiers. For example, **USE**, **SET**, **SMASH** and **REFERENCE** all may be modified by **FREELY**. The modifier may occur anywhere within the command. If there is more than one verb, any modifier *between* two verbs is assumed to modify the first one. For example, in **USING ANY FREELY OR SETTING X**, the **FREELY** modifies **USING** but not **SETTING**; the entire phrase is interpreted as the set of all functions which either use any variable freely or set the variable **X**, whether or not **X** is set freely. Verbs can occur in the present tense (e.g., **USE**, **CALLS**, **BINDS**, **USES**) or as present or past participles (e.g., **CALLING**, **BOUND**, **TESTED**). The relations (with their modifiers) recognized by Masterscope are:

CALL [Masterscope Relation]

Function **F1** calls **F2** if the definition of **F1** contains a form **(F2 --)**. The **CALL** relation also includes any instance where a function

uses a name as a function, as in **(APPLY (QUOTE F2) --)**, **(FUNCTION F2)**, etc.

CALL SOMEHOW [Masterscope Relation]

One function calls another **SOMEHOW** if there is some path from the first to the other. That is, if **F1** calls **F2**, and **F2** calls **F3**, then **F1 CALLS F3 SOMEHOW**.

This information is not stored directly in the database; instead, Masterscope stores only information about direct function calls, and (re)computes the **CALL SOMEHOW** relation as necessary.

USE [Masterscope Relation]

If unmodified, the relation **USE** denotes variable usage in any way; it is the union of the relations **SET**, **SMASH**, **TEST**, and **REFERENCE**.

SET [Masterscope Relation]

A function **SET**s a variable if the function contains a form **(SETQ var --)**, **(SETQQ var --)**, etc.

SMASH [Masterscope Relation]

A function **SMASH**es a variable if the function calls a destructive list operation (**RPLACA**, **RPLACD**, **DREMOVE**, **SORT**, etc.) on the value of that variable. Masterscope will also find instances where the operation is performed on a "part" of the value of the variable; for example, if a function contains a form **(RPLACA (NTH X 3) T)** it will be noted as **SMASHING X**.

Note that if the function contains a sequence **(SETQ Y X)**, **(RPLACA Y T)** then **Y** is noted as being smashed, but not **X**.

TEST [Masterscope Relation]

A variable is **TEST**ed by a function if its value is only distinguished between **NIL** and non-**NIL**. For example, the form **(COND ((AND X --) --))** *tests* the value of **X**.

REFERENCE [Masterscope Relation]

This relation includes all variable usage *except* for **SET**.

The verbs **USE**, **SET**, **SMASH**, **TEST** and **REFERENCE** may be modified by the words **FREELY** or **LOCALLY**. A variable is used **FREELY** if it is not bound in the function at the place of its use; alternatively, it is used **LOCALLY** if the use occurs within a **PROG** or **LAMBDA** that binds the variable.

Masterscope also distinguishes between **CALL DIRECTLY** and **CALL INDIRECTLY**. A function is called **DIRECTLY** if it occurs as

CAR-of-form in a normal evaluation context. A function is called **INDIRECTLY** if its name appears in a context which does not imply its *immediate* evaluation, for example **(SETQ Y (LIST (FUNCTION FOO) 3))**. The distinction is whether or not the compiled code of the caller would contain a direct call to the callee. Note that an occurrence of **(FUNCTION FOO)** as the functional argument to one of the built-in mapping functions which compile open is considered to be a direct call.

In addition, **CALL FOR EFFECT** (where the value of the function is not used) is distinguished from **CALL FOR VALUE**.

BIND [Masterscope Relation]

The **BIND** relation between functions and variables includes both variables bound as function arguments and those bound in an internal **PROG** or **LAMBDA** expression.

USE AS A FIELD [Masterscope Relation]

Masterscope notes all uses of record field names within **FETCH**, **REPLACE** or **CREATE** expressions.

FETCH [Masterscope Relation]

Use of a field within a **FETCH** expression.

REPLACE [Masterscope Relation]

Use of a record field name within a **REPLACE** or **CREATE** expression.

USE AS A RECORD [Masterscope Relation]

Masterscope notes all uses of record names within **CREATE** or **TYPE?** expressions. Additionally, in **(fetch (FOO FIE) of X)**, **FOO** is used as a record name.

CREATE [Masterscope Relation]

Use of a record name within a **CREATE** expression.

USE AS A PROPERTY NAME [Masterscope Relation]

Masterscope notes the property names used in **GETPROP**, **PUTPROP**, **GETLIS**, etc. expressions if the name is quoted. E.g. if a function contains a form **(GETPROP X (QUOTE INTERP))**, then that function **USEs INTERP** as a property name.

USE AS A CLISP WORD [Masterscope Relation]

Masterscope notes all iterative statement operators and user defined CLISP words as being used as a CLISP word.

CONTAIN [Masterscope Relation]

Files *contain* functions, records, and variables. This relation is not stored in the database but is computed using the file package.

DECLARE AS LOCALVAR [Masterscope Relation]

DECLARE AS SPECVAR [Masterscope Relation]

Masterscope notes internal "calls" to **DECLARE** from within functions.

The following abbreviations are recognized: **FREE** = **FREELY**, **LOCAL** = **LOCALLY**, **PROP** = **PROPERTY**, **REF** = **REFERENCE**. Also, the words **A**, **AN** and **NAME** (after **AS**) are "noise" words and may be omitted.

Note: Masterscope uses "templates" (page 19.18) to decide which relations hold between functions and their arguments. For example, the information that **SORT SMASH**es its first argument is contained in the template for **SORT**. Masterscope initially contains templates for most system functions which set variables, test their arguments, or perform destructive operations. The user may change existing templates or insert new ones in Masterscope's tables via the **SETTEMPLATE** function (page 19.21).

19.1.3 Set Specifications

A "set" is a collection of things (functions, variables, etc.). A set is specified by a set phrase, consisting of a *determiner* (e.g., **ANY**, **WHICH**, **WHO**) followed by a *type* (e.g., **FUNCTIONS**, **VARIABLES**) followed by a *specification* (e.g., **IN MYFNS**, **@ SUBRP**). The determiner, type and specification may be used alone or in combination. For example, **ANY FUNCTIONS IN MYFNS**, **ANY @ SUBRP**, **VARIABLES IN GLOBALVARS**, and **WHO** are all acceptable set phrases. Set specifications are explained below:

'ATOM [Masterscope Set Specification]

The simplest way to specify a set consisting of a single thing is by the name of that thing. For example, in the command **WHO CALLS 'ERROR**, the function **ERROR** is referred to by its name. Although the ' can be left out, to resolve possible ambiguities names should usually be quoted; e.g., **WHO CALLS 'CALLS** will return the list of functions which call the function **CALLS**.

'LIST [Masterscope Set Specification]

Sets consisting of several atoms may be specified by naming the atoms. For example, the command **WHO USES '(A B)** returns the list of functions that use the variables **A** or **B**.

IN *EXPRESSION* [Masterscope Set Specification]

The form *EXPRESSION* is evaluated, and its value is treated as a list of the elements of a set. For example, **IN GLOBALVARS** specifies the list of variables in the value of the variable **GLOBALVARS**.

@ PREDICATE [Masterscope Set Specification]

A set may also be specified by giving a predicate which the elements of that set must satisfy. *PREDICATE* is either a function name, a **LAMBDA** expression, or an expression in terms of the variable **X**. The specification *@ PREDICATE* represents all atom for which the value of *PREDICATE* is non-**NIL**. For example, **@ EXPRP** specifies all those atoms which have expr definitions; **@ (STRPOSL X CLISPCHARRAY)** specifies those atoms which contain CLISP characters. The universe to be searched is either determined by the context within the command (e.g., in **WHO IN FOOFNS CALLS ANY NOT @ GETD**, the predicate is only applied to functions which are called by any functions in the list **FOOFNS**), or in the extreme case, the universe defaults to the entire set of things which have been noticed by Masterscope, as in the command **WHO IS @ EXPRP**.

LIKE *ATOM* [Masterscope Set Specification]

ATOM may contain **ESC**s; it is used as a pattern to be matched (as in the editor). For example, **WHO LIKE /R$ IS CALLED BY ANY** would find both **/RPLACA** and **/RPLNODE**.

A set may also be specified by giving a relation its members must have with the members of another set:

RELATIONING SET [Masterscope Set Specification]

RELATIONING is used here generically to mean any of the relation words in the present participle form (possibly with a modifier), e.g., **USING**, **SETTING**, **CALLING**, **BINDING**. *RELATIONING SET* specifies the set of all objects which have that relation with some element of *SET*. For example, **CALLING X** specifies the set of functions which call the function **X**; **USING ANY IN FOOVARS FREELY** specifies the set of functions which uses freely any variable in the value of **FOOVARS**.

RELATIONED **BY** *SET* [Masterscope Set Specification]

RELATIONED **IN** *SET* [Masterscope Set Specification]

This is similar to the *RELATIONING* construction. For example, **CALLED BY ANY IN FOOFNS** represents the set of functions which are called by any element of **FOOFNS**; **USED FREELY BY ANY CALLING ERROR** is the set of variables which are used freely by any function which also calls the function **ERROR**.

BLOCKTYPE **OF** *FUNCTIONS* [Masterscope Set Specification]

BLOCKTYPE **ON** *FILES* [Masterscope Set Specification]

These phrases allow the user to ask about **BLOCKS** declarations on files (see page 18.17). *BLOCKTYPE* is one of **LOCALVARS**, **SPECVARS**, **GLOBALVARS**, **ENTRIES**, **BLKFNS**, **BLKAPPLYFNS**, or **RETFNS**.

BLOCKTYPE **OF** *FUNCTIONS* specifies the names which are declared to be *BLOCKTYPE* in any blocks declaration which contain any of *FUNCTIONS* (a "set" of functions). The "functions" in *FUNCTIONS* can either be block names or just functions in a block. For example, **WHICH ENTRIES OF ANY CALLING 'Y BIND ANY GLOBALVARS ON 'FOO**.

BLOCKTYPE **ON** *FILES* specifies all names which are declared to be *BLOCKTYPE* on any of the given *FILES* (a "set" of files).

FIELDS OF *SET* [Masterscope Set Specification]

SET is a set of records. This denotes the field names of those records. For example, the command **WHO USES ANY FIELDS OF BRECORD** returns the list of all functions which do a **fetch** or **replace** with any of the field names declared in the record declaration of **BRECORD**.

KNOWN [Masterscope Set Specification]

The set of all functions which have been analyzed. For example, the command **WHO IS KNOWN** will print out the list of functions which have been analyzed.

THOSE [Masterscope Set Specification]

The set of things printed out by the last Masterscope question. For example, following the command **WHO IS USED FREELY BY PARSE**, the user could ask **WHO BINDS THOSE** to find out where those variables are bound.

ON PATH *PATHOPTIONS* [Masterscope Set Specification]

Refers to the set of functions which *would be* printed by the command **SHOW PATHS** *PATHOPTIONS*. For example, **IS FOO BOUND BY ANY ON PATH TO 'PARSE** tests if FOO might be bound "above" the function **PARSE**. **SHOW PATHS** is explained in detail on page 19.15.

Note: sets may also be specified with "relative clauses" introduced by the word **THAT**, e.g. **THE FUNCTIONS THAT BIND 'X.**

19.1.4 Set Determiners

Set phrases may be preceded by a *determiner*. A determiner is one of the words **THE, ANY, WHO** or **WHICH**. The "question" determiners (**WHO** and **WHICH**) are only meaningful in some of the commands, namely those that take the form of questions. **ANY** and **WHO** (or **WHOM**) can be used alone; they are "wild-card" elements, e.g., the command **WHO USES ANY FREELY**, will print out the names of all (known) functions which use any variable freely. If the determiner is omitted, **ANY** is assumed; e.g. the command **WHO CALLS '(PRINT PRIN1 PRIN2)** will print the list of functions which call *any* of **PRINT, PRIN1, PRIN2. THE** is also allowed, e.g. **WHO USES THE RECORD FIELD FIELDX.**

19.1.5 Set Types

Any set phrase has a *type*; that is, a set may specify either functions, variables, files, record names, record field names or property names. The type may be determined by the context within the command (e.g., in **CALLED BY ANY ON FOO**, the set **ANY ON FOO** is interpreted as meaning the *functions* on **FOO** since only functions can be **CALLED**), or the type may be given explicitly by the user (e.g., **FUNCTIONS ON FIE**). The following types are recognized: **FUNCTIONS, VARIABLES, FILES, PROPERTY NAMES, RECORDS, FIELDS, I.S.OPRS**. Also, the abbreviations **FNS, VARS, PROPNAMES** or the singular forms **FUNCTION, FN, VARIABLE, VAR, FILE, PROPNAME, RECORD, FIELD** are recognized. Note that most of these types correspond to built-in "file package types" (see page 17.21).

The type is used by Masterscope in a variety of ways when interpreting the set phrase:

(1) Set types are used to disambiguate possible parsings. For example, both commands **WHO SETS ANY BOUND IN X OR USED BY Y** and **WHO SETS ANY BOUND IN X OR CALLED BY Y** have the

same general form. However, the first case is parsed as **WHO SETS ANY (BOUND BY X OR USED BY Y)** since both **BOUND BY X** and **USED BY Y** refer to variables; while the second case as **WHO SETS ANY BOUND IN (X OR CALLED BY Y)**, since **CALLED BY Y** and **X** must refer to functions. Note that parentheses may be used to group phrases.

(2) The type is used to determine the modifier for **USE: FOO USES WHICH RECORDS** is equivalent to **FOO USES WHO AS A RECORD FIELD**.

(3) The interpretation of **CONTAIN** depends on the type of its object: the command **WHAT FUNCTIONS ARE CONTAINED IN MYFILE** prints the list of functions in **MYFILE**; **WHAT RECORDS ARE ON MYFILE** prints the list of records.

(4) The implicit "universe" in which a set expression is interpreted depends on the type: **ANY VARIABLES @ GETD** is interpreted as the set of all variables which have been noticed by Masterscope (i.e., bound or used in any function which has been analyzed) that also have a definition. **ANY FUNCTIONS @ (NEQ (GETTOPVAL X) 'NOBIND)** is interpreted as the set of all functions which have been noticed (either analyzed or called by a function which has been analyzed) that also have a top-level value.

19.1.6 Conjunctions of Sets

Sets may be joined by the conjunctions **AND** and **OR** or preceded by **NOT** to form new sets. **AND** is always interpreted as meaning "intersection"; **OR** as "union", while **NOT** means "complement". For example, the set **CALLING X AND NOT CALLED BY Y** specifies the set of all functions which call the function **X** but are not called by **Y**.

Masterscope's interpretation of **AND** and **OR** follow LISP conventions rather than the conventional English interpretation. For example "calling **X** and **Y**" would, in English, be interpreted as the intersection of **(CALLING X)** and **(CALLING Y)**; but Masterscope interprets **CALLING X AND Y** as **CALLING ('X AND 'Y)**; which is the null set. Only sets may be joined with conjunctions: joining modifiers, as in **USING X AS A RECORD FIELD OR PROPERTY NAME**, is not allowed; in this case, the user must say **USING X AS A RECORD FIELD OR USING X AS A PROPERTY NAME**.

As described above, the type of sets is used to disambiguate parsings. The algorithm used is to first try to match the type of the phrases being joined and then try to join with the longest preceding phrase. In any case, the user may group phrases with parentheses to specify the manner in which conjunctions should be parsed.

19.2 SHOW PATHS

In trying to work with large programs, the user can lose track of the hierarchy of functions. The Masterscope **SHOW PATHS** command aids the user by providing a map showing the calling structure of a set of functions. **SHOW PATHS** prints out a tree structure showing which functions call which other functions. For example, the command **SHOW PATHS FROM MSPARSE** will print out the structure of Masterscope's parser:

```
1.MSPARSE  MSINIT MSMARKINVALID
2.     |   MSINITH MSINITH
3.     MSINTERPRET MSRECORDFILE
4.     |       MSPRINTWORDS
5.     |       PARSECOMMAND GETNEXTWORD CHECKADV
6.     |       |       PARSERELATION {a}
7.     |       |       PARSESET {b}
8.     |       |       PARSEOPTIONS {c}
9.     |       |       MERGECONJ GETNEXTWORD {5}
10.    |       GETNEXTWORD {5}
11.    |       FIXUPTYPES SUBJTYPE
12.    |       |       OBJTYPE
13.    |       FIXUPCONJUNCTIONS MERGECONJ {9}
14.    |                   MATCHSCORE
15.    MSPRINTSENTENCE
---------------------------------------------------------- overflow - a
16.PARSERELATION GETNEXTWORD {5}
17.         CHECKADV
---------------------------------------------------------- overflow - b
19.PARSESET PARSESET
20.      GETNEXTWORD {5}
21.      PARSERELATION {6}
22.      SUBPARSE GETNEXTWORD {5}
---------------------------------------------------------- overflow - c
23.PARSEOPTIONS GETNEXTWORD {5}
24.         PARSESET {19}
```

The above printout displays that the function **MSPARSE** calls **MSINIT**, **MSINTERPRET**, and **MSPRINTSENTENCE**. **MSINTERPRET** in turn calls **MSRECORDFILE**, **MSPRINTWORDS**, **PARSECOMMAND**, **GETNEXTWORD**, **FIXUPTYPES**, and **FIXUPCONJUNCTIONS**. The numbers in braces {} after a function name are backward references: they indicate that the tree for that function was expanded on a previous line. The lowercase letters in braces are *forward* references: they indicate that the tree for that function will be expanded below, since there is no more room on the line. The vertical bar is used to keep the output aligned.

Note: Loading the Browser library package modifies the **SHOW PATHS** command so the command's output is displayed as an undirected graph.

The **SHOW PATHS** command takes the form: **SHOW PATHS** followed by some combination of the following *path options:*

FROM *SET* [Masterscope Path Option]

Display the function calls from the elements of *SET*.

TO *SET* [Masterscope Path Option]

Display the function calls leading *to* elements of *SET*. If **TO** is given before **FROM** (or no **FROM** is given), the tree is "inverted" and a message, **(inverted tree)** is printed to warn the user that if **FN1** appears after **FN2** it is because **FN1** is *called by* **FN2**.

When both **FROM** and **TO** are given, the first one indicates a set of functions which are to be displayed while the second restricts the paths that will be traced; i.e., the command **SHOW PATHS FROM X TO Y** will trace the elements of the set **CALLED SOMEHOW BY X AND CALLING Y SOMEHOW**.

If **TO** is not given, **TO KNOWN OR NOT @ GETD** is assumed; that is, only functions which have been analyzed or which are undefined will be included. Note that Masterscope will analyze a function while printing out the tree if that function has not previously been seen and it currently has an expr definition; thus, any function which *can be* analyzed will be displayed.

AVOIDING *SET* [Masterscope Path Option]

Do not display any function in *SET*. **AMONG** is recognized as a synonym for **AVOIDING NOT**. For example, **SHOW PATHS TO ERROR AVOIDING ON FILE2** will not display (or trace) any function on **FILE2**.

NOTRACE *SET* [Masterscope Path Option]

Do not trace from any element of *SET*. **NOTRACE** differs from **AVOIDING** in that a function which is marked **NOTRACE** will be printed, but the tree beyond it will not be expanded; the functions in an **AVOIDING** set will not be printed at all. For example, **SHOW PATHS FROM ANY ON FILE1 NOTRACE ON FILE2** will display the tree of calls eminating from **FILE1**, but will not expand any function on **FILE2**.

SEPARATE *SET* [Masterscope Path Option]

Give each element of *SET* a separate tree. Note that **FROM** and **TO** only insure that the designated functions will be displayed. **SEPARATE** can be used to guarantee that certain functions will

begin new tree structures. **SEPARATE** functions are displayed in the same manner as overflow lines; i.e., when one of the functions indicated by **SEPARATE** is found, it is printed followed by a forward reference (a lower-case letter in braces) and the tree for that function is then expanded below.

LINELENGTH *N* [Masterscope Path Option]

Resets **LINELENGTH** to *N* before displaying the tree. The linelength is used to determine when a part of the tree should "overflow" and be expanded lower.

19.3 Error Messages

When the user gives Masterscope a command, the command is first parsed, i.e. translated to an internal representation, and then the internal representation is interpreted. If a command cannot be parsed, e.g. if the user typed **SHOW WHERE CALLED BY X**, the message "**Sorry, I can't parse that!**" is printed and an error is generated. If the command is of the correct form but cannot be interpreted (e.g., the command **EDIT WHERE ANY CONTAINS ANY**) Masterscope will print the message "**Sorry, that isn't implemented!**" and generate an error. If the command requires that some functions having been analyzed (e.g., the command **WHO CALLS X**) and the database is empty, Masterscope will print the message "**Sorry, no functions have been analyzed!**" and generate an error.

19.4 Macro Expansion

As part of analysis, Masterscope will expand the macro definition of called functions, if they are not otherwise defined (see page 10.21). Masterscope macro expansion is controlled by the variable **MSMACROPROPS**:

MSMACROPROPS [Variable]

Value is an ordered list of macro-property names that Masterscope will search to find a macro definition. Only the kinds of macros that appear on **MSMACROPROPS** will be expanded. All others will be treated as function calls and left unexpanded. Initially **(MACRO)**.

Note: **MSMACROPROPS** initially contains only **MACRO** (and not **10MACRO**, **DMACRO**, etc.) in the theory that the

machine-dependent macro definitions are more likely "optimizers".

Note that if you edit a macro, Masterscope will know to reanalyze the functions which call that macro. However, if your macro is of the "computed-macro" style, and it calls functions which you edit, Masterscope will not notice. You must be careful to tell masterscope to **REANALYZE** the appropriate functions (e.g., if you edit **FOOEXPANDER** which is used to expand **FOO** macros, you have to . **REANALYZE ANY CALLING FOO**.

19.5 Affecting Masterscope Analysis

Masterscope analyzes the expr definitions of functions and notes in its database the relations that function has with other functions and with variables. To perform this analysis, Masterscope uses *templates* which describe the behavior of functions. For example, the information that **SORT** destructively modifies its first argument is contained in the template for **SORT**. Masterscope initially contains templates for most system functions which set variables, test their arguments, or perform destructive operations.

A template is a list structure containing any of the following atoms:

PPE [in Masterscope template]

If an expression appears in this location, there is most likely a parenthesis error.

Masterscope notes this as a "call" to the function "ppe" (lowercase). Therefore, **SHOW WHERE ANY CALLS ppe** will print out all possible parenthesis errors. When Masterscope finds a possible parenthesis error in the course of analyzing a function definition, rather than printing the usual ".", it prints out a "?" instead.

NIL [in Masterscope template]

The expression occuring at this location is not evaluated.

SET [in Masterscope template]

A variable appearing at this place is set.

SMASH [in Masterscope template]

The value of this expression is smashed.

TEST [in Masterscope template]

This expression is used as a predicate (that is, the only use of the value of the expression is whether it is **NIL** or non-**NIL**).

PROP [in Masterscope template]

The value of this expression is used as a property name. If the expression is of the form **(QUOTE** *ATOM***)**, Masterscope will note that *ATOM* is **USED AS A PROPERTY NAME**. For example, the template for **GETPROP** is **(EVAL PROP . PPE)**.

FUNCTION [in Masterscope template]

The expression at this point is used as a functional argument. For example, the template for **MAPC** is **(SMASH FUNCTION FUNCTION . PPE)**.

FUNCTIONAL [in Masterscope template]

The expression at this point is used as a functional argument. This is like **FUNCTION**, except that Masterscope distinguishes between functional arguments to functions which "compile open" from those that do not. For the latter (e.g. **SORT** and **APPLY**), **FUNCTIONAL** should be used rather than **FUNCTION**.

EVAL [in Masterscope template]

The expression at this location is evaluated (but not set, smashed, tested, used as a functional argument, etc.).

RETURN [in Masterscope template]

The value of the function (of which this is the template) is the value of this expression.

TESTRETURN [in Masterscope template]

A combination of **TEST** and **RETURN**: If the value of the function is non-**NIL**, then it is returned. For instance, a one-element **COND** clause is this way.

EFFECT [in Masterscope template]

The expression at this location is evaluated, but the value is not used.

FETCH [in Masterscope template]

An atom at this location is a field which is fetched.

REPLACE [in Masterscope template]

An atom at this location is a field which is replaced.

RECORD [in Masterscope template]

An atom at this location is used as a record name.

CREATE [in Masterscope template]

An atom at this location is a record which is created.

BIND [in Masterscope template]

An atom at this location is a variable which is bound.

CALL [in Masterscope template]

An atom at this location is a function which is called.

CLISP [in Masterscope template]

An atom at this location is used as a CLISP word.

! [in Masterscope template]

This atom, which can only occur as the first element of a template, allows one to specify a template for the **CAR** of the function form. If **!** doesn't appear, the **CAR** of the form is treated as if it had a **CALL** specified for it. In other words, the templates **(.. EVAL)** and **(! CALL .. EVAL)** are equivalent.

If the next atom after a **!** is **NIL**, this specifies that the function name should not be remembered. For example, the template for **AND** is **(! NIL .. TEST RETURN)**, which means that if you see an **"AND"**, don't remember it as being called. This keeps the Masterscope database from being cluttered by too many uninteresting relations; Masterscope also throws away relations for **COND**, **CAR**, **CDR**, and a couple of others.

In addition to the above atoms which occur in templates, there are some "special forms" which are lists keyed by their **CAR**.

.. TEMPLATE [in Masterscope template]

Any part of a template may be preceded by the atom .. (two periods) which specifies that the template should be repeated an indefinite number ($N >= 0$) of times to fill out the expression. For example, the template for **COND** might be **(.. (TEST .. EFFECT RETURN))** while the template for **SELECTQ** is **(EVAL .. (NIL .. EFFECT RETURN) RETURN)**.

(BOTH $TEMPLATE_1$ $TEMPLATE_2$**)** [in Masterscope template]

Analyze the current expression twice, using the each of the templates in turn.

(IF *EXPRESSION TEMPLATE$_1$ TEMPLATE$_2$***)** [in Masterscope template]

Evaluate *EXPRESSION* at analysis time (the variable **EXPR** will be bound to the expression which corresponds to the **IF**), and if the result is non-**NIL**, use *TEMPLATE$_1$*, otherwise *TEMPLATE$_2$*. If *EXPRESSION* is a literal atom, it is **APPLY**'d to **EXPR**. For example, **(IF LISTP (RECORD FETCH) FETCH)** specifies that if the current expression is a list, then the first element is a record name and the second element a field name, otherwise it is a field name.

(@ *EXPRFORM TEMPLATEFORM***)** [in Masterscope template]

Evaluate *EXPRFORM* giving *EXPR*, evaluate *TEMPLATEFORM* giving *TEMPLATE*. Then analyze *EXPR* with *TEMPLATE*. @ lets the user compute on the fly both a template and an expression to analyze with it. The forms can use the variable **EXPR**, which is bound to the current expression.

(MACRO . *MACRO***)** [in Masterscope template]

MACRO is interpreted in the same way as a macro (see page 10.21) and the resulting form is analyzed. If the template is the atom **MACRO** alone, Masterscope will use the **MACRO** property of the function itself. This is useful when analyzing code which contains calls to user-defined macros. If the user changes a macro property (e.g. by editing it) of an atom which has template of **MACRO**, Masterscope will mark any function which used that macro as needing to be reanalyzed.

Some examples of templates:

function:	template:
DREVERSE	**(SMASH . PPE)**
AND	**(! NIL TEST .. RETURN)**
MAPCAR	**(EVAL FUNCTION FUNCTION)**
COND	**(! NIL .. (IF CDR (TEST .. EFFECT RETURN) (TESTRETURN . PPE)))**

Templates may be changed and new templates defined using the functions:

(GETTEMPLATE *FN***)** [Function]

Returns the current template of *FN*.

(SETTEMPLATE *FN TEMPLATE***)** [Function]

Changes the template for the function *FN* and returns the old value. If any functions in the database are marked as calling *FN*, they will be marked as needing re-analysis.

19.6 Data Base Updating

Masterscope is interfaced to the editor and file package so that it notes whenever a function has been changed, either through editing or loading in a new definition. Whenever a command is given which requires knowing the information about a specific function, if that function has been noted as being changed, the function is automatically re-analyzed before the command is interpreted. If the command requires that all the information in the database be consistent (e.g., the user asks **WHO CALLS X**) then *all* functions which have been marked as changed are re-analyzed.

19.7 Masterscope Entries

(CALLS *FN USEDATABASE* **—)** [Function]

FN can be a function name, a definition, or a form. Note: **CALLS** will also work on compiled code. **CALLS** returns a list of four elements: a list of all the functions called by *FN*, a list of all the variables bound in *FN*, a list of all the variables used freely in *FN*, and a list of the variables used globally in *FN*. For the purpose of **CALLS**, variables used freely which are on **GLOBALVARS** or have a property **GLOBALVAR** value **T** are considered to be used globally. If *USEDATABASE* is **NIL** (or *FN* is not a litatom), **CALLS** will perform a one-time analysis of *FN*. Otherwise (i.e. if *USEDATABASE* is non-**NIL** and *FN* a function name), **CALLS** will use the information in Masterscope's database (*FN* will be analyzed first if necessary).

(CALLSCCODE *FN* **— —)** [Function]

The sub-function of **CALLS** which analyzes compiled code. **CALLSCCODE** returns a list of *five* elements: a list of all the functions called via "linked" function calls (not implemented in Interlisp-D), a list of all functions called regularly, a list of variables bound in *FN*, a list of variables used freely, and a list of variables used globally.

(FREEVARS *FN USEDATABASE***)** [Function]

Equivalent to **(CADDR (CALLS** *FN USEDATABASE***))**. Returns the list of variables used freely within *FN*.

(MASTERSCOPE *COMMAND* **—)** [Function]

Top level entry to Masterscope. If *COMMAND* is **NIL**, will enter into an executive in which the user may enter commands. If

COMMAND is not **NIL**, the command is interpreted and **MASTERSCOPE** will return the value that would be printed by the command. Note that only the question commands return meaningful values.

(SETSYNONYM *PHRASE MEANING* —**)** [Function]

Defines a new synonym for Masterscope's parser. Both *OLDPHRASE* and *NEWPHRASE* are words or lists of words; anywhere *OLDPHRASE* is seen in a command, *NEWPHRASE* will be substituted. For example, **(SETSYNONYM 'GLOBALS '(VARS IN GLOBALVARS OR @(GETPROP X 'GLOBALVAR)))** would allow the user to refer with the single word **GLOBALS** to the set of variables which are either in **GLOBALVARS** or have a **GLOBALVAR** property.

The following functions are provided for users who wish to write their own routines using Masterscope's database:

(PARSERELATION *RELATION***)** [Function]

RELATION is a relation phrase; e.g., **(PARSERELATION '(USE FREELY))**. **PARSERELATION** returns an internal representation for *RELATION*. For use in conjunction with **GETRELATION**.

(GETRELATION *ITEM RELATION INVERTED***)** [Function]

RELATION is an internal representation as returned by **PARSERELATION** (if not, **GETRELATION** will first perform **(PARSERELATION** *RELATION***)**); *ITEM* is an atom. **GETRELATION** returns the list of all atoms which have the given relation to *ITEM*. For example, **(GETRELATION 'X '(USE FREELY))** returns the list of variables that **X** uses freely. If *INVERTED* is **T**, the inverse relation is used; e.g. **(GETRELATION 'X '(USE FREELY) T)** returns the list of functions which use **X** freely.

If *ITEM* is **NIL**, **GETRELATION** will return the list of atoms which have *RELATION* with *any* other item; i.e., answers the question **WHO** *RELATION***S ANY**. Note that **GETRELATION** does *not* check to see if *ITEM* has been analyzed, or that other functions that have been changed have been re-analyzed.

(TESTRELATION *ITEM RELATION ITEM2 INVERTED***)** [Function]

Equivalent to **(MEMB** *ITEM2* **(GETRELATION** *ITEM RELATION INVERTED***))**, that is, tests if *ITEM* and *ITEM2* are related via *RELATION*. If *ITEM2* is **NIL**, the call is equivalent to **(NOT (NULL (GETRELATION** *ITEM RELATION INVERTED***)))**, i.e., **TESTRELATION** tests if *ITEM* has the given *RELATION* with *any* other item.

(MAPRELATION *RELATION MAPFN***)** [Function]

Calls the function *MAPFN* on every pair of items related via *RELATION*. If **(NARGS** *MAPFN***)** is 1, then *MAPFN* is called on every item which has the given *RELATION* to *any* other item.

(MSNEEDUNSAVE *FNS MSG MARKCHANGEFLG***)** [Function]

Used to mark functions which depend on a changed record declaration (or macro, etc.), and which must be LOADed or UNSAVEd (see below). *FNS* is a list of functions to be marked, and *MSG* is a string describing the records, macros, etc. on which they depend. If *MARKCHANGEFLG* is non-NIL, each function in the list is marked as needing re-analysis.

(UPDATEFN *FN EVENIFVALID* —**)** [Function]

Equivalent to the command **ANALYZE** '*FN*; that is, **UPDATEFN** will analyze *FN* if *FN* has not been analyzed before or if it has been changed since the time it was analyzed. If *EVENIFVALID* is non-**NIL**, **UPDATEFN** will re-analyze *FN* even if Masterscope thinks it has a valid analysis in the database.

(UPDATECHANGED) [Function]

Performs **(UPDATEFN** *FN***)** on every function which has been marked as changed.

(MSMARKCHANGED *NAME TYPE REASON***)** [Function]

Mark that *NAME* has been changed and needs to be reanalyzed. See **MARKASCHANGED**, page 17.17.

(DUMPDATABASE *FNLST***)** [Function]

Dumps the current Masterscope database on the current output file in a **LOAD**able form. If *FNLST* is not **NIL**, **DUMPDATABASE** will only dump the information for the list of functions in *FNLST*. The variable **DATABASECOMS** is initialized to **((E (DUMPDATABASE)))**; thus, the user may merely perform **(MAKEFILE 'DATABASE.***EXTENSION***)** to save the current Masterscope database. If a Masterscope database already exists when a **DATABASE** file is loaded, the database on the file will be merged with the one in core. Note that functions whose definitions are different from their definition when the database was made must be **REANALYZE**d if their new definitions are to be noticed.

The Databasefns library package provides a more convenient way of saving data bases along with the source files which they correspond to.

19.8 Noticing Changes that Require Recompiling

When a record declaration, iterative statement operator or macro is changed, and Masterscope has "noticed" a use of that declaration or macro (i.e. it is used by some function known about in the data base), Masterscope will alert the user about those functions which might need to be re-compiled (e.g. they do not currently have expr definitions). Extra functions may be noticed; for example if **FOO** contains **(fetch (REC X) --)**, and some declaration other than **REC** which contains **X** is changed, Masterscope will still think that **FOO** needs to be loaded/unsaved. The functions which need recompiling are added to the list **MSNEEDUNSAVE** and a message is printed out:

The functions *FN1, FN2,...* use macros which have changed.
Call UNSAVEFNS() to load and/or unsave them.

In this situation, the following function is useful:

(UNSAVEFNS —) [Function]

Uses **LOADFNS** or **UNSAVEDEF** to make sure that all functions in the list **MSNEEDUNSAVE** have expr definitions, and then sets **MSNEEDUNSAVE** to **NIL**.

Note: If **RECOMPILEDEFAULT** (page 18.16) is set to **CHANGES**, **UNSAVEFNS** prints out "**WARNING: you must set RECOMPILEDEFAULT to EXPRS in order to have these functions recompiled automatically**".

19.9 Implementation Notes

Masterscope keeps a database of the relations noticed when functions are analyzed. The relations are intersected to form "primitive relationships" such that there is little or no overlap of any of the primitives. For example, the relation **SET** is stored as the union of **SET LOCAL** and **SET FREE**. The **BIND** relation is divided into **BIND AS ARG, BIND AND NOT USE**, and **SET LOCAL, SMASH LOCAL**, etc. Splitting the relations in this manner reduces the size of the database considerably, to the point where it is reasonable to maintain a Masterscope database for a large system of functions during a normal debugging session.

Each primitive relationship is stored in a pair of hash-tables, one for the "forward" direction and one for the "reverse". For example, there are two hash tables, **USE AS PROPERTY** and **USED AS PROPERTY**. To retrieve the information from the database, Masterscope performs unions of the hash-values. For example, to answer **FOO BINDS WHO** Masterscope will look in all of the tables which make up the **BIND** relation. The "internal

representation" returned by **PARSERELATION** is just a list of dotted pairs of hash-tables. To perform **GETRELATION** requires only mapping down that list, doing **GETHASH**'s on the appropriate hash-tables and **UNION**ing the result.

Hash tables are used for a variety of reasons: storage space is smaller; it is not necessary to maintain separate lists of which functions have been analyzed (a special table, **DOESN'T DO ANYTHING** is maintained for functions which neither call other functions nor bind or use any variables); and accessing is relatively fast. Within any of the tables, if the hash-value would be a list of one atom, then the atom itself, rather than the list, is stored as the hash-value. This also reduces the size of the database significantly.

TABLE OF CONTENTS

[This page intentionally left blank]

20. DWIM

A surprisingly large percentage of the errors made by Interlisp users are of the type that could be corrected by another Lisp programmer without any information about the purpose of the program or expression in question, e.g., misspellings, certain kinds of parentheses errors, etc. To correct these types of errors we have implemented in Interlisp a DWIM facility, short for Do-What-I-Mean. DWIM is called automatically whenever an error occurs in the evaluation of an Interlisp expression. (Currently, DWIM only operates on unbound atoms and undefined function errors.) DWIM then proceeds to try to correct the mistake using the current context of computation plus information about what the user had previously been doing, (and what mistakes he had been making) as guides to the remedy of the error. If DWIM is able to make the correction, the computation continues as though no error had occurred. Otherwise, the procedure is the same as though DWIM had not intervened: a break occurs, or an unwind to the last **ERRORSET** (page 14.21). The following protocol illustrates the operation of DWIM.

For example, suppose the user defines the factorial function **(FACT N)** as follows:

```
←DEFINEQ((FACT (LAMBDA (N) (COND
((ZEROP N0 1) ((T (ITIMS N (FACCT 9SUB1 N]
(FACT)
←
```

Note that the definition of **FACT** contains several mistakes: **ITIMES** and **FACT** have been misspelled; the **0** in **N0** was intended to be a right parenthesis, but the shift key was not depressed; similarly, the **9** in **9SUB1** was intended to be a left parenthesis; and finally, there is an extra left parenthesis in front of the **T** that begins the final clause in the conditional.

```
←PRETTYPRNT((FACCT]
= PRETTYPRINT
= FACT

(FACT
 [LAMBDA (N)
  (COND
   ((ZEROP N0 1)
    ((T (ITIMS N (FACCT 9SUB1 N])
(FACT)
```

```
←
```

After defining **FACT**, the user wishes to look at its definition using **PRETTYPRINT**, which he unfortunately misspells. Since there is no function **PRETTYPRNT** in the system, an undefined function error occurs, and DWIM is called. DWIM invokes its spelling corrector, which searches a list of functions frequently used (by *this* user) for the best possible match. Finding one that is extremely close, DWIM proceeds on the assumption that **PRETTYPRNT** meant **PRETTYPRINT**, notifies the user of this, and calls **PRETTYPRINT.**

At this point, **PRETTYPRINT** would normally print **(FACCT NOT PRINTABLE)** and exit, since **FACCT** has no definition. Note that this is *not* an Interlisp error condition, so that DWIM would not be called as described above. However, it is obviously not what the user *meant.*

This sort of mistake is corrected by having **PRETTYPRINT** itself explicitly invoke the spelling corrector portion of DWIM whenever given a function with no **EXPR** definition. Thus, with the aid of DWIM **PRETTYPRINT** is able to determine that the user wants to see the definition of the function **FACT**, and proceeds accordingly.

```
←FACT(3]
N0 [IN FACT] -> N ) ? YES
[IN FACT] (COND -- ((T --))) ->
       (COND -- (T --))
ITIMS [IN FACT] -> ITIMES
FACCT [IN FACT] -> FACT
9SUB1 [IN FACT] -> ( SUB1 ? YES
6
←PP FACT
(FACT
 [LAMBDA (N)
  (COND
   ((ZEROP N)
    1)
   (T (ITIMES N (FACT (SUB1 N])
FACT
←
```

The user now calls **FACT**. During its execution, five errors occur, and DWIM is called five times. At each point, the error is corrected, a message is printed describing the action taken, and the computation is allowed to continue as if no error had occurred. Following the last correction, 6 is printed, the value of **(FACT 3)**. Finally, the user prettyprints the new, now correct, definition of **FACT**.

In this particular example, the user was shown operating in **TRUSTING** mode, which gives DWIM carte blanche for most

corrections. The user can also operate in **CAUTIOUS** mode, in which case DWIM will inform him of intended corrections before they are made, and allow the user to approve or disapprove of them. If DWIM was operating in **CAUTIOUS** mode in the example above, it would proceed as follows:

```
←FACT(3)
NO [IN FACT] -> N ) ? YES
U.D.F. T [IN FACT]  FIX?  YES
[IN FACT] (COND -- ((T --))) ->
        (COND -- (T --))
ITIMS [IN FACT] -> ITIMES ?  ...YES
FACCT [IN FACT] -> FACT ?  ...YES
9SUB1 [IN FACT] -> ( SUB1 ? NO
U.B.A.
(9SUB1 BROKEN)
:
```

For most corrections, if the user does not respond in a specified interval of time, DWIM automatically proceeds with the correction, so that the user need intervene only when he does not approve. Note that the user responded to the first, second, and fifth questions; DWIM responded for him on the third and fourth.

Note: DWIM uses **ASKUSER** for its interactions with the user (page 26.12). Whenever an interaction is about to take place and the user has typed ahead, **ASKUSER** types several bells to warn the user to stop typing, then clears and saves the input buffers, restoring them after the interaction is complete. Thus if the user has typed ahead before a DWIM interaction, DWIM will not confuse his type ahead with the answer to its question, nor will his typeahead be lost. The bells are printed by the function **PRINTBELLS**, which can be advised or redefined for specialized applications, e.g. to flash the screen for a display terminal.

A great deal of effort has gone into making DWIM "smart", and experience with a large number of users indicates that DWIM works very well; DWIM seldom fails to correct an error the user feels it should have, and almost never mistakenly corrects an error. However, it is important to note that even when DWIM *is* wrong, no harm is done: since an error had occurred, the user would have had to intervene anyway if DWIM took no action. Thus, if DWIM mistakenly corrects an error, the user simply interrupts or aborts the computation, **UNDO**es the DWIM change using **UNDO** (page 13.13), and makes the correction he would have had to make without DWIM. An exception is if DWIM's correction mistakenly caused a destructive computation to be initiated, and information was lost before the user could interrupt. We have not yet had such an incident occur.

(DWIM *X*) [Function]

Used to enable/disable DWIM. If *X* is the litatom **C**, DWIM is enabled in **CAUTIOUS** mode, so that DWIM will ask the user before making corrections. If *X* is **T**, DWIM is enabled in **TRUSTING** mode, so DWIM will make most corrections automatically. If *X* is **NIL**, DWIM is disabled. Interlisp initially has DWIM enabled in **CAUTIOUS** mode.

DWIM returns **CAUTIOUS**, **TRUSTING** or **NIL**, depending to what mode it has just been put into.

For corrections to expressions typed in by the user for immediate execution (typed into **LISPX**, page 13.35), DWIM always acts as though it were in **TRUSTING** mode, i.e., no approval necessary. For certain types of corrections, e.g., run-on spelling corrections, 9-0 errors, etc., DWIM always acts like it was in **CAUTIOUS** mode, and asks for approval. In either case, DWIM always informs the user of its action as described below.

20.1 Spelling Correction Protocol

One type of error that DWIM can correct is the misspelling of a function or a variable name. When an unbound litatom or undefined function error occurs, DWIM tries to correct the spelling of the bad litatom. If a litatom is found whose spelling is "close" to the offender, DWIM proceeds as follows:

If the correction occurs in the typed-in expression, DWIM prints = *CORRECT-SPELLING*^{cr} and continues evaluating the expression. For example:

```
←(SETQ FOO (IPLUSS 1 2))
= IPLUS
3
```

If the correction does not occur in type-in, DWIM prints

BAD-SPELLING [**IN** *FUNCTION-NAME*] **->** *CORRECT-SPELLING*

The appearance of **->** is to call attention to the fact that the user's function will be or has been changed.

Then, if DWIM is in **TRUSTING** mode, it prints a carriage return, makes the correction, and continues the computation. If DWIM is in **CAUTIOUS** mode, it prints a few spaces and **?** and then wait for approval. The user then has six options:

(1) Type **Y**. DWIM types **es**, and proceeds with the correction.

(2) Type **N**. DWIM types **o**, and does not make the correction.

(3) Type ↑. DWIM does not make the correction, and furthermore guarantees that the error will not cause a break.

(4) Type control-E. For error correction, this has the same effect as typing **N**.

(5) Do nothing. In this case DWIM waits for **DWIMWAIT** seconds, and if the user has not responded, DWIM will type ... followed by the default answer.

The default on spelling corrections is determined by the value of the variable **FIXSPELLDEFAULT**, whose top level value is initially **Y**.

(6) Type space or carriage-return. In this case DWIM will wait indefinitely. This option is intended for those cases where the user wants to think about his answer, and wants to insure that DWIM does not get "impatient" and answer for him.

The procedure for spelling correction on other than Interlisp errors is analogous. If the correction is being handled as type-in, DWIM prints = followed by the correct spelling, and returns it to the function that called DWIM. Otherwise, DWIM prints the incorrect spelling, followed by the correct spelling. Then, if DWIM if in **TRUSTING** mode, DWIM prints a carriage-return and returns the correct spelling. Otherwise, DWIM prints a few spaces and a **?** and waits for approval. The user can then respond with **Y**, **N**, control-E, space, carriage return, or do nothing as described above.

Note that the spelling corrector itself is not **ERRORSET** protected like the DWIM error correction routines. Therefore, typing **N** and typing control-E may have different effects when the spelling corrector is called directly. The former simply instructs the spelling corrector to return **NIL**, and lets the calling function decide what to do next; the latter causes an error which unwinds to the last **ERRORSET**, however far back that may be.

20.2 Parentheses Errors Protocol

When an unbound litatom or undefined error occurs, and the offending litatom contains **9** or **0**, DWIM tries to correct errors caused by typing **9** for left parenthesis and **0** for right parenthesis. In these cases, the interaction with the user is similar to that for spelling correction. If the error occurs in type-in, DWIM types = *CORRECTION*^{cr}, and continues evaluating the expression. For example:

```
←(SETQ FOO 9IPLUS 1 2]
= (IPLUS
3
```

If the correction does not occur in type-in, DWIM prints

BAD-ATOM [**IN** *FUNCTION-NAME*] **->** *CORRECTION* ?

and then waits for approval. The user then has the same six options as for spelling correction, except the waiting time is 3***DWIMWAIT** seconds. If the user types **Y**, DWIM then operates as if it were in **TRUSTING** mode, i.e., it makes the correction and prints its message.

Note: Actually, DWIM uses the value of the variables **LPARKEY** and **RPARKEY** to determine the corresponding lower case character for left and right parentheses. **LPARKEY** and **RPARKEY** are initially **9** and **0** respectively, but they can be reset for other keyboard layouts, e.g., on some terminals left parenthesis is over **8**, and right parenthesis is over **9**.

20.3 Undefined Function T Errors

When an undefined function error occurs, and the offending function is **T**, DWIM tries to correct certain types of parentheses errors involving a **T** clause in a conditional. DWIM recognizes errors of the following forms:

(COND --) (T --) The **T** clause appears outside and immediately following the **COND**.

(COND -- (-- & (T --))) The **T** clause appears inside a previous clause.

(COND -- ((T --))) The **T** clause has an extra pair of parentheses around it.

For undefined function errors that are not one of these three types, DWIM takes no corrective action at all, and the error will occur.

If the error occurs in type-in, DWIM simply types **T FIXED** and makes the correction. Otherwise if DWIM is in **TRUSTING** mode, DWIM makes the correction and prints the message:

[IN *FUNCTION-NAME*] **{BAD-COND} ->**
{CORRECTED-COND}

If DWIM is in **CAUTIOUS** mode, DWIM prints

UNDEFINED FUNCTION T
[IN *FUNCTION-NAME*] **FIX?**

and waits for approval. The user then has the same options as for spelling corrections and parenthesis errors. If the user types **Y** or defaults, DWIM makes the correction and prints its message.

Having made the correction, DWIM must then decide how to proceed with the computation. In the first case, **(COND --) (T --)**, DWIM cannot know whether the **T** clause would have been executed if it had been inside of the **COND**. Therefore DWIM

asks the user **CONTINUE WITH T CLAUSE** (with a default of **YES**). If the user types **N**, DWIM continues with the form after the **COND**, i.e., the form that originally followed the **T** clause.

In the second case, **(COND -- (-- & (T --)))**, DWIM has a different problem. After moving the **T** clause to its proper place, DWIM must return as the value of **&** as the value of the **COND**. Since this value is no longer around, DWIM asks the user, **OK TO REEVALUATE** and then prints the expression corresponding to **&**. If the user types **Y**, or defaults, DWIM continues by reevaluating **&**, otherwise DWIM aborts, and a **U.D.F. T** error will then occur (even though the **COND** has in fact been fixed). If DWIM can determine for itself that the form can safely be reevaluated, it does not consult the user before reevaluating. DWIM can do this if the form is atomic, or **CAR** of the form is a member of the list **OKREEVALST**, and each of the arguments can safely be reevaluated. For example, **(SETQ X (CONS (IPLUS Y Z) W))** is safe to reevaluate because **SETQ**, **CONS**, and **IPLUS** are all on **OKREEVALST**.

In the third case, **(COND -- ((T --)))**, there is no problem with continuation, so no further interaction is necessary.

20.4 DWIM Operation

Whenever the interpreter encounters an atomic form with no binding, or a non-atomic form **CAR** of which is not a function or function object, it calls the function **FAULTEVAL**. Similarly, when **APPLY** is given an undefined function, **FAULTAPPLY** is called. When DWIM is enabled, **FAULTEVAL** and **FAULTAPPLY** are redefined to first call the DWIM package, which tries to correct the error. If DWIM cannot decide how to fix the error, or the user disapproves of DWIM's correction (by typing **N**), or the user types control-E, then **FAULTEVAL** and **FAULTAPPLY** cause an error or break. If the user types ↑ to DWIM, DWIM exits by performing **(RETEVAL 'FAULTEVAL '(ERROR!))**, so that an error will be generated at the position of the call to **FAULTEVAL**.

If DWIM can (and is allowed to) correct the error, it exits by performing **RETEVAL** of the corrected form, as of the position of the call to **FAULTEVAL** or **FAULTAPPLY**. Thus in the example at the beginning of the chapter, when DWIM determined that **ITIMS** was **ITIMES** misspelled, DWIM called **RETEVAL** with **(ITIMES N (FACCT 9SUB1 N))**. Since the interpreter uses the value returned by **FAULTEVAL** exactly as though it were the value of the erroneous form, the computation will thus proceed exactly as though no error had occurred.

In addition to continuing the computation, DWIM also repairs the cause of the error whenever possible; in the above example, DWIM also changed (with **RPLACA**) the expression **(ITIMS N (FACCT 9SUB1 N))** that caused the error. Note that if the user's program had *computed* the form and called **EVAL**, it would not be possible to repair the cause of the error, although DWIM could correct the misspelling each time it occurred.

Error correction in DWIM is divided into three categories: unbound atoms, undefined **CAR** of form, and undefined function in **APPLY**. Assuming that the user approves DWIM's corrections, the action taken by DWIM for the various types of errors in each of these categories is summarized below.

20.4.1 DWIM Correction: Unbound Atoms

If DWIM is called as the result of an unbound atom error, it proceeds as follows:

(1) If the first character of the unbound atom is ', DWIM assumes that the user (intentionally) typed '*ATOM* for **(QUOTE** *ATOM***)** and makes the appropriate change. No message is typed, and no approval is requested.

If the unbound atom is just ' itself, DWIM assumes the user wants the *next* expression quoted, e.g., **(CONS X '(A B C))** will be changed to **(CONS X (QUOTE (A B C)))**. Again no message will be printed or approval asked. If no expression follows the ', DWIM gives up.

Note: ' is normally defined as a read-macro character which converts **'FOO** to **(QUOTE FOO)** on input, so DWIM will not see the ' in the case of expressions that are typed-in.

(2) If CLISP (page 21.1) is enabled, and the atom is part of a CLISP construct, the CLISP transformation is performed and the result returned. For example, **N-1** is transformed to **(SUB1 N)**, and **(... FOO←3 ...)** is transformed into **(... (SETQ FOO 3) ...)**.

(3) If the atom contains an **9** (actually **LPARKEY**, see page 20.14), DWIM assumes the **9** was intended to be a left parenthesis, and calls the editor to make appropriate repairs on the expression containing the atom. DWIM assumes that the user did not notice the mistake, i.e., that the entire expression was affected by the missing left parenthesis. For example, if the user types **(SETQ X (LIST (CONS 9CAR Y) (CDR Z)) Y)**, the expression will be changed to **(SETQ X (LIST (CONS (CAR Y) (CDR Z)) Y))**. Note that the **9** does not have to be the first character of the atom: DWIM will handle **(CONS X9CAR Y)** correctly.

(4) If the atom contains a **0** (actually **RPARKEY**, see page 20.14), DWIM assumes the **0** was intended to be a right parenthesis and operates as in the case above.

(5) If the atom begins with a **7**, the **7** is treated as a '. For example, **7FOO** becomes **'FOO**, and then **(QUOTE FOO)**.

(6) The expressions on **DWIMUSERFORMS** (see page 20.11) are evaluated in the order that they appear. If any of these expressions returns a non-**NIL** value, this value is treated as the form to be used to continue the computation, it is evaluated and its value is returned by **DWIM**.

(7) If the unbound atom occurs in a function, DWIM attempts spelling correction using the **LAMBDA** and **PROG** variables of the function as the spelling list.

(8) If the unbound atom occurred in a type-in to a break, DWIM attempts spelling correction using the **LAMBDA** and **PROG** variables of the broken function as the spelling list.

(9) Otherwise, DWIM attempts spelling correction using **SPELLINGS3** (see page 20.17).

(10) If all of the above fail, DWIM gives up.

20.4.2 Undefined CAR of Form

If DWIM is called as the result of an undefined **CAR** of form error, it proceeds as follows:

(1) If **CAR** of the form is **T**, DWIM assumes a misplaced **T** clause and operates as described on page 20.6.

(2) If **CAR** of the form is **F/L**, DWIM changes the "**F/L**" to "**FUNCTION(LAMBDA**". For example, **(F/L (Y) (PRINT (CAR Y)))** is changed to **(FUNCTION (LAMBDA (Y) (PRINT (CAR Y)))**. No message is printed and no approval requested. If the user omits the variable list, DWIM supplies **(X)**, e.g., **(F/L (PRINT (CAR X)))** is changed to **(FUNCTION (LAMBDA (X) (PRINT (CAR X))))**. DWIM determines that the user has supplied the variable list when more than one expression follows **F/L**, **CAR** of the first expression is not the name of a function, and every element in the first expression is atomic. For example, DWIM will supply **(X)** when correcting **(F/L (PRINT (CDR X)) (PRINT (CAR X)))**.

(3) If **CAR** of the form is a CLISP word (**IF**, **FOR**, **DO**, **FETCH**, etc.), the indicated CLISP transformation is performed, and the result is returned as the corrected form. See page 21.1.

(4) If **CAR** of the form has a function definition, DWIM attempts spelling correction on **CAR** of the definition using as spelling list the value of **LAMBDASPLST**, initially **(LAMBDA NLAMBDA)**.

(5) If **CAR** of the form has an **EXPR** or **CODE** property, DWIM prints *CAR-OF-FORM* **UNSAVED**, performs an **UNSAVEDEF**, and continues. No approval is requested.

(6) If **CAR** of the form has a **FILEDEF** property, the definition is loaded from a file (except when **DWIMIFY**ing). If the value of the property is atomic, the entire file is to be loaded. If the value is a list, **CAR** is the name of the file and **CDR** the relevant functions, and **LOADFNS** will be used. For both cases, *LDFLG* will be **SYSLOAD** (see page 17.5). DWIM uses **FINDFILE** (page 24.32), so that the file can be on any of the directories on **DIRECTORIES**, initially **(NIL NEWLISP LISP LISPUSERS)**. If the file is found, DWIM types **SHALL I LOAD** followed by the file name or list of functions. If the user approves, DWIM loads the function(s) or file, and continues the computation.

(7) If CLISP is enabled, and **CAR** of the form is part of a CLISP construct, the indicated transformation is performed, e.g., **(N←N-1)** becomes **(SETQ N (SUB1 N))**.

(8) If **CAR** of the form contains an **9**, DWIM assumes a left parenthesis was intended e.g., **(CONS9CAR X)**.

(9) If **CAR** of the form contains a **0**, DWIM assumes a right parenthesis was intended.

(10) If **CAR** of the form is a list, DWIM attempts spelling correction on **CAAR** of the form using **LAMBDASPLST** as spelling list. If successful, DWIM returns the corrected expression itself.

(11) The expressions on **DWIMUSERFORMS** are evaluated in the order they appear. If any returns a non-**NIL** value, this value is treated as the corrected form, it is evaluated, and **DWIM** returns its value.

(12) Otherwise, DWIM attempts spelling correction using **SPELLINGS2** as the spelling list (see page 20.17). When **DWIMIFY**ing, **DWIM** also attemps spelling correction on function names not defined but previously encountered, using **NOFIXFNSLST** as a spelling list (see page 21.21).

(13) If all of the above fail, DWIM gives up.

20.4.3 Undefined Function in APPLY

If DWIM is called as the result of an undefined function in **APPLY** error, it proceeds as follows:

(1) If the function has a definition, DWIM attempts spelling correction on **CAR** of the definition using **LAMBDASPLST** as spelling list.

(2) If the function has an **EXPR** or **CODE** property, DWIM prints *FN* **UNSAVED**, performs an **UNSAVEDEF** and continues. No approval is requested.

(3) If the function has a property **FILEDEF**, DWIM proceeds as in case 6 of undefined **CAR** of form.

(4) If the error resulted from type-in, and CLISP is enabled, and the function name contains a CLISP operator, DWIM performs the indicated transformation, e.g., the user types **FOO←(APPEND FIE FUM)**.

(5) If the function name contains an **9**, DWIM assumes a left parenthesis was intended, e.g., **EDIT9FOO]**.

(6) If the "function" is a list, DWIM attempts spelling correction on **CAR** of the list using **LAMBDASPLST** as spelling list.

(7) The expressions on **DWIMUSERFORMS** are evaluated in the order they appear, and if any returns a non-**NIL** value, this value is treated as the function used to continue the computation, i.e., it will be applied to its arguments.

(8) DWIM attempts spelling correction using **SPELLINGS1** as the spelling list.

(9) DWIM attempts spelling correction using **SPELLINGS2** as the spelling list.

(10) If all fail, DWIM gives up.

20.5 DWIMUSERFORMS

The variable **DWIMUSERFORMS** provides a convenient way of adding to the transformations that DWIM performs. For example, the user might want to change atoms of the form **$X** to **(QA4LOOKUP X)**. Before attempting spelling correction, but after performing other transformations (**F/L**, **9**, **0**, CLISP, etc.), DWIM evaluates the expressions on **DWIMUSERFORMS** in the order they appear. If any expression returns a non-**NIL** value, this value is treated as the transformed form to be used. If DWIM was called from **FAULTEVAL**, this form is evaluated and the resulting value is returned as the value of **FAULTEVAL**. If DWIM is called from **FAULTAPPLY**, this form is treated as a function to be applied to **FAULTARGS**, and the resulting value is returned as the value of **FAULTAPPLY**. If all of the expressions on **DWIMUSERFORMS** return **NIL**, DWIM proceeds as though **DWIMUSERFORMS = NIL**, and attempts spelling correction. Note that DWIM simply takes the value and returns it; the expressions on **DWIMUSERFORMS** are responsible for making any modifications to the original expression. The expressions on **DWIMUSERFORMS** should make the transformation permanent, either by associating it with **FAULTX** via **CLISPTRAN**, or by destructively changing **FAULTX**.

In order for an expression on **DWIMUSERFORMS** to be able to be effective, it needs to know various things about the context of the error. Therefore, several of DWIM's internal variables have

been made **SPECVARS** (see page 18.5) and are therefore "visible" to **DWIMUSERFORMS**. Below are a list of those variables that may be useful.

FAULTX [Variable]

For unbound atom and undefined car of form errors, **FAULTX** is the atom or form. For undefined function in **APPLY** errors, **FAULTX** is the name of the function.

FAULTARGS [Variable]

For undefined function in **APPLY** errors, **FAULTARGS** is the list of arguments. **FAULTARGS** may be modified or reset by expressions on **DWIMUSERFORMS**.

FAULTAPPLYFLG [Variable]

Value is **T** for undefined function in **APPLY** errors; **NIL** otherwise. The value of **FAULTAPPLYFLG** *after* an expression on **DWIMUSERFORMS** returns a non-**NIL** value determines how the latter value is to be treated. Following an undefined function in **APPLY** error, if an expression on **DWIMUSERFORMS** sets **FAULTAPPLYFLG** to **NIL**, the value returned is treated as a form to be evaluated, rather than a function to be applied.

FAULTAPPLYFLG is necessary to distinguish between unbound atom and undefined function in **APPLY** errors, since **FAULTARGS** may be **NIL** and **FAULTX** atomic in both cases.

TAIL [Variable]

For unbound atom errors, **TAIL** is the tail of the expression **CAR** of which is the unbound atom. **DWIMUSERFORMS** expression can replace the atom by another expression by performing **(/RPLACA TAIL** *EXPR***)**

PARENT [Variable]

For unbound atom errors, **PARENT** is the form in which the unbound atom appears. **TAIL** is a tail of **PARENT**.

TYPE-IN? [Variable]

True if the error occurred in type-in.

FAULTFN [Variable]

Name of the function in which error occurred. **FAULTFN** is **TYPE-IN** when the error occurred in type-in, and **EVAL** or **APPLY** when the error occurred under an explicit call to **EVAL** or **APPLY**.

DWIMIFYFLG [Variable]

True if the error was encountered while **DWIMIFY**ing (as opposed to happening while running a program).

EXPR [Variable]

Definition of **FAULTFN**, or argument to **EVAL**, i.e., the superform in which the error occurs.

The initial value of **DWIMUSERFORMS** is **((DWIMLOADFNS?))**. **DWIMLOADFNS?** is a function for automatically loading functions from files. If **DWIMLOADFNSFLG** is **T** (its initial value), and **CAR** of the form is the name of a function, and the function is contained on a file that has been noticed by the file package, the function is loaded, and the computation continues.

20.6 DWIM Functions and Variables

DWIMWAIT [Variable]

Value is the number of seconds that DWIM will wait before it assumes that the user is not going to respond to a question and uses the default response **FIXSPELLDEFAULT**.

DWIM operates by dismissing for 250 milliseconds, then checking to see if anything has been typed. If not, it dismisses again, etc. until **DWIMWAIT** seconds have elapsed. Thus, there will be a delay of at most 1/4 second before DWIM responds to the user's answer.

FIXSPELLDEFAULT [Variable]

If approval is requested for a spelling correction, and user does not respond, defaults to value of **FIXSPELLDEFAULT**, initially **Y**. **FIXSPELLDEFAULT** is rebound to **N** when **DWIMIFY**ing.

ADDSPELLFLG [Variable]

If **NIL**, suppresses calls to **ADDSPELL**. Initially **T**.

NOSPELLFLG [Variable]

If **T**, suppresses *all* spelling correction. If some other non-**NIL** value, suppresses spelling correction in programs but not type-in. **NOSPELLFLG** is initially **NIL**. It is rebound to **T** when compiling from a file.

RUNONFLG [Variable]

If **NIL**, suppresses run-on spelling corrections. Initially **NIL**.

DWIMLOADFNSFLG [Variable]

If **T**, tells DWIM that when it encounters a call to an undefined function contained on a file that has been noticed by the file package, to simply load the function. **DWIMLOADFNSFLG** is initially **T**. See page 20.13.

LPARKEY [Variable]

RPARKEY [Variable]

DWIM uses the value of the variables **LPARKEY** and **RPARKEY** (initially **9** and **0** respectively) to determine the corresponding lower case character for left and right parentheses. **LPARKEY** and **RPARKEY** can be reset for other keyboard layouts. For example, on some terminals left parenthesis is over **8**, and right parenthesis is over **9**.

OKREEVALST [Variable]

The value of **OKREEVALST** is a list of functions that DWIM can safely reevaluate. If a form is atomic, or **CAR** of the form is a member of **OKREEVALST**, and each of the arguments can safely be reevaluated, then the form can be safely reevaluated. For example, **(SETQ X (CONS (IPLUS Y Z) W))** is safe to reevaluate because **SETQ**, **CONS**, and **IPLUS** are all on **OKREEVALST**.

DWIMFLG [Variable]

DWIMFLG = NIL, all DWIM operations are disabled. **(DWIM 'C)** and **(DWIM T)** set **DWIMFLG** to **T**; **(DWIM NIL)** sets **DWIMFLG** to **NIL**.

APPROVEFLG [Variable]

APPROVEFLG = T if DWIM should ask the user for approval before making a correction that will modify the definition of one of his functions; **NIL** otherwise.

When DWIM is put into **CAUTIOUS** mode with **(DWIM 'C)**, **APPROVEFLG** is set to **T**; for **TRUSTING** mode, **APPROVEFLG** is set to **NIL**.

LAMBDASPLST [Variable]

DWIM uses the value of **LAMBDASPLST** as the spelling list when correcting "bad" function definitions. Initially **(LAMBDA NLAMBDA)**. The user may wish to add to **LAMBDASPLST** if he elects to define new "function types" via an appropriate

DWIMUSERFORMS entry. For example, the **QLAMBDA**s of SRI's QLISP are handled in this way.

20.7 Spelling Correction

The spelling corrector is given as arguments a misspelled word (word means literal atom), a spelling list (a list of words), and a number: *XWORD*, *SPLST*, and *REL* respectively. Its task is to find that word on *SPLST* which is closest to *XWORD*, in the sense described below. This word is called a *respelling* of *XWORD*. *REL* specifies the minimum "closeness" between *XWORD* and a respelling. If the spelling corrector cannot find a word on *SPLST* closer to *XWORD* than *REL*, or if it finds two or more words equally close, its value is **NIL**, otherwise its value is the respelling. The spelling corrector can also be given an optional functional argument, *FN*, to be used for selecting out a subset of *SPLST*, i.e., only those members of *SPLST* that satisfy *FN* will be considered as possible respellings.

The exact algorithm for computing the spelling metric is described later, but briefly "closeness" is inversely proportional to the number of disagreements between the two words, and directly proportional to the length of the longer word. For example, **PRTTYPRNT** is "closer" to **PRETTYPRINT** than **CS** is to **CONS** even though both pairs of words have the same number of disagreements. The spelling corrector operates by proceeding down *SPLST*, and computing the closeness between each word and *XWORD*, and keeping a list of those that are closest. Certain differences between words are not counted as disagreements, for example a single transposition, e.g., **CONS** to **CNOS**, or a doubled letter, e.g., **CONS** to **CONSS**, etc. In the event that the spelling corrector finds a word on *SPLST* with *no* disagreements, it will stop searching and return this word as the respelling. Otherwise, the spelling corrector continues through the entire spelling list. Then if it has found one and only one "closest" word, it returns this word as the respelling. For example, if *XWORD* is **VONS**, the spelling corrector will probably return **CONS** as the respelling. However, if *XWORD* is **CONZ**, the spelling corrector will not be able to return a respelling, since **CONZ** is equally close to both **CONS** and **COND**. If the spelling corrector finds an acceptable respelling, it interacts with the user as described earlier.

In the special case that the misspelled word contains one or more **$**s (escape), the spelling corrector searches for those words on *SPLST* that match *XWORD*, where a **$** can match any number of characters (including 0), e.g., **FOO$** matches **FOO1** and **FOO**, but not **NEWFOO**. **FOO** matches all three. Both completion and

correction may be involved, e.g. **RPETTY$** will match **PRETTYPRINT**, with one mistake. The entire spelling list is always searched, and if more than one respelling is found, the spelling corrector prints **AMBIGUOUS**, and returns **NIL**. For example, **CON$** would be ambiguous if both **CONS** and **COND** were on the spelling list. If the spelling corrector finds one and only one respelling, it interacts with the user as described earlier.

For both spelling correction and spelling completion, regardless of whether or not the user approves of the spelling corrector's choice, the respelling is moved to the front of *SPLST*. Since many respellings are of the type with no disagreements, this procedure has the effect of considerably reducing the time required to correct the spelling of frequently misspelled words.

20.7.1 Synonyms

Spelling lists also provide a way of defining synonyms for a particular context. If a dotted pair appears on a spelling list (instead of just an atom), **CAR** is interpreted as the correct spelling of the misspelled word, and **CDR** as the antecedent for that word. If **CAR** is *identical* with the misspelled word, the antecedent is returned without any interaction or approval being necessary. If the misspelled word *corrects* to **CAR** of the dotted pair, the usual interaction and approval will take place, and then the antecedent, i.e., **CDR** of the dotted pair, is returned. For example, the user could make **IFLG** synonymous with **CLISPIFTRANFLG** by adding **(IFLG . CLISPIFTRANFLG)** to **SPELLINGS3**, the spelling list for unbound atoms. Similarly, the user could make **OTHERWISE** mean the same as **ELSEIF** by adding **(OTHERWISE . ELSEIF)** to **CLISPIFWORDSPLST**, or make **L** be synonymous with **LAMBDA** by adding **(L . LAMBDA)** to **LAMBDASPLST**. Note that **L** could also be used as a variable without confusion, since the association of **L** with **LAMBDA** occurs only in the appropriate context.

20.7.2 Spelling Lists

Any list of atoms can be used as a spelling list, e.g., **BROKENFNS**, **FILELST**, etc. Various system packages have their own spellings lists, e.g., **LISPXCOMS**, **CLISPFORWORDSPLST**, **EDITCOMSA**, etc. These are documented under their corresponding sections, and are also indexed under "spelling lists." In addition to these spelling lists, the system maintains, i.e., automatically adds to, and occasionally prunes, four lists used solely for spelling correction: **SPELLINGS1**, **SPELLINGS2**, **SPELLINGS3**, and **USERWORDS**. These spelling lists are maintained *only* when **ADDSPELLFLG** is non-**NIL**. **ADDSPELLFLG** is initially **T**.

SPELLINGS1 [Variable]

SPELLINGS1 is a list of functions used for spelling correction when an input is typed in apply format, and the function is undefined, e.g., **EDTIF(FOO)**. **SPELLINGS1** is initialized to contain **DEFINEQ**, **BREAK**, **MAKEFILE**, **EDITF**, **TCOMPL**, **LOAD**, etc. Whenever **LISPX** is given an input in apply format, i.e., a function and arguments, the name of the function is added to **SPELLINGS1** if the function has a definition.

For example, typing **CALLS(EDITF)** will cause **CALLS** to be added to **SPELLINGS1**. Thus if the user typed **CALLS(EDITF)** and later typed **CALLLS(EDITV)**, since **SPELLINGS1** would then contain **CALLS**, DWIM would be successful in correcting **CALLLS** to **CALLS**.

SPELLINGS2 [Variable]

SPELLINGS2 is a list of functions used for spelling correction for all other undefined functions. It is initialized to contain functions such as **ADD1**, **APPEND**, **COND**, **CONS**, **GO**, **LIST**, **NCONC**, **PRINT**, **PROG**, **RETURN**, **SETQ**, etc. Whenever **LISPX** is given a non-atomic form, the name of the function is added to **SPELLINGS2**. For example, typing **(RETFROM (STKPOS (QUOTE FOO) 2))** to a break would add **RETFROM** to **SPELLINGS2**. Function names are also added to **SPELLINGS2** by **DEFINE**, **DEFINEQ**, **LOAD** (when loading compiled code), **UNSAVEDEF**, **EDITF**, and **PRETTYPRINT**.

SPELLINGS3 [Variable]

SPELLINGS3 is a list of words used for spelling correction on all unbound atoms. **SPELLINGS3** is initialized to **EDITMACROS**, **BREAKMACROS**, **BROKENFNS**, and **ADVISEDFNS**. Whenever **LISPX** is given an atom to evaluate, the name of the atom is added to **SPELLINGS3** if the atom has a value. Atoms are also added to **SPELLINGS3** whenever they are edited by **EDITV**, and whenever they are set via **RPAQ** or **RPAQQ**. For example, when a file is loaded, all of the variables set in the file are added to **SPELLINGS3**. Atoms are also added to **SPELLINGS3** when they are set by a **LISPX** input, e.g., typing **(SETQ FOO (REVERSE (SETQ FIE ...)))** will add both **FOO** and **FIE** to **SPELLINGS3**.

USERWORDS [Variable]

USERWORDS is a list containing both functions and variables that the user has *referred* to, e.g., by breaking or editing. **USERWORDS** is used for spelling correction by **ARGLIST**, **UNSAVEDEF**, **PRETTYPRINT**, **BREAK**, **EDITF**, **ADVISE**, etc. **USERWORDS** is initially **NIL**. Function names are added to it by **DEFINE**, **DEFINEQ**, **LOAD**, (when loading compiled code, or loading exprs to property lists) **UNSAVEDEF**, **EDITF**, **EDITV**, **EDITP**, **PRETTYPRINT**, etc. Variable names are added to **USERWORDS** at the same time as they are added to **SPELLINGS3**. In addition, the

variable **LASTWORD** is always set to the last word added to **USERWORDS**, i.e., the last function or variable referred to by the user, and the respelling of **NIL** is defined to be the value of **LASTWORD**. Thus, if the user has just defined a function, he can then prettyprint it by typing **PP()**.

Each of the above four spelling lists are divided into two sections separated by a special marker (the value of the variable **SPELLSTR1**). The first section contains the "permanent" words; the second section contains the temporary words. New words are added to the corresponding spelling list at the front of its temporary section (except that functions added to **SPELLINGS1** or **SPELLINGS2** by **LISPX** are always added to the end of the permanent section. If the word is already in the temporary section, it is moved to the front of that section; if the word is in the permanent section, no action is taken. If the length of the temporary section then exceeds a specified number, the last (oldest) word in the temporary section is forgotten, i.e., deleted. This procedure prevents the spelling lists from becoming cluttered with unimportant words that are no longer being used, and thereby slowing down spelling correction time. Since the spelling corrector usually moves each word selected as a respelling to the front of its spelling list, the word is thereby moved into the permanent section. Thus once a word is misspelled and corrected, it is considered important and will never be forgotten.

Note: The spelling correction algorithm will not alter a spelling list unless it contains the special marker (the value of **SPELLSTR1**). This provides a way to ensure that a spelling list will not be altered.

#SPELLINGS1 [Variable]

#SPELLINGS2 [Variable]

#SPELLINGS3 [Variable]

#USERWORDS [Variable]

The maximum length of the temporary section for **SPELLINGS1**, **SPELLINGS2**, **SPELLINGS3** and **USERWORDS** is given by the value of **#SPELLINGS1**, **#SPELLINGS2**, **#SPELLINGS3**, and **#USERWORDS**, initialized to 30, 30, 30, and 60 respectively.

Users can alter these values to modify the performance behavior of spelling correction.

20.7.3 Generators for Spelling Correction

For some applications, it is more convenient to *generate* candidates for a respelling one by one, rather than construct a complete list of all possible candidates, e.g., spelling correction involving a large directory of files, or a natural language data base. For these purposes, *SPLST* can be an array (of any size). The first element of this array is the generator function, which is called with the array itself as its argument. Thus the function can use the remainder of the array to store "state" information, e.g., the last position on a file, a pointer into a data structure, etc. The value returned by the function is the next candidate for respelling. If **NIL** is returned, the spelling "list" is considered to be exhausted, and the closest match is returned. If a candidate is found with no disagreements, it is returned immediately without waiting for the "list" to exhaust.

SPLST can also be a generator, i.e. the value of the function **GENERATOR** (page 11.17). The generator *SPLST* will be started up whenever the spelling corrector needs the next candidate, and it should return candidates via the function **PRODUCE**. For example, the following could be used as a "spelling list" which effectively contains all functions in the system:

```
[GENERATOR
  (MAPATOMS (FUNCTION (LAMBDA (X) (if (GETD X) then
(PRODUCE X]
```

20.7.4 Spelling Corrector Algorithm

The basic philosophy of DWIM spelling correction is to count the number of disagreements between two words, and use this number divided by the length of the longer of the two words as a measure of their relative disagreement. One minus this number is then the relative agreement or closeness. For example, **CONS** and **CONX** differ only in their last character. Such substitution errors count as one disagreement, so that the two words are in 75% agreement. Most calls to the spelling corrector specify a relative agreement of 70, so that a single substitution error is permitted in words of four characters or longer. However, spelling correction on shorter words is possible since certain types of differences such as single transpositions are not counted as disagreements. For example, **AND** and **NAD** have a relative agreement of 100. Calls to the spelling corrector from **DWIM** use the value of **FIXSPELLREL**, which is initially 70. Note that by setting **FIXSPELLREL** to 100, only spelling corrections with "zero" mistakes, will be considered, e.g., transpositions, double characters, etc.

The central function of the spelling corrector is **CHOOZ**. **CHOOZ** takes as arguments: a word, a minimum relative agreement, a

spelling list, and an optional functional argument, *XWORD*, *REL*, *SPLST*, and *FN* respectively.

CHOOZ proceeds down *SPLST* examining each word. Words not satisfying *FN* (if *FN* is non-**NIL**), or those obviously too long or too short to be sufficiently close to *XWORD* are immediately rejected. For example, if *REL* = 70, and *XWORD* is 5 characters long, words longer than 7 characters will be rejected.

Special treatment is necessary for words shorter than *XWORD*, since doubled letters are not counted as disagreements. For example, **CONNSSS** and **CONS** have a relative agreement of 100. **CHOOZ** handles this by counting the number of doubled characters in *XWORD* before it begins scanning *SPLST*, and taking this into account when deciding whether to reject shorter words.

If *TWORD*, the current word on *SPLST*, is not rejected, **CHOOZ** computes the number of disagreements between it and *XWORD* by calling a subfunction, **SKOR**.

SKOR operates by scanning both words from left to right one character at a time. **SKOR** operates on the list of character codes for each word. This list is computed by **CHOOZ** before calling **SKOR**. Characters are considered to agree if they are the same characters or appear on the same key (i.e., a shift mistake). The variable **SPELLCASEARRAY** is a CASEARRAY which is used to determine equivalence classes for this purpose. It is initialized to equivalence lowercase and upper case letters, as well as the standard key transitions: for example, **1** with **!**, **3** with **#**, etc.

If the first character in *XWORD* and *TWORD* do *not* agree, **SKOR** checks to see if either character is the same as one previously encountered, and not accounted-for at that time. (In other words, transpositions are not handled by lookahead, but by *lookback*.) A displacement of two or fewer positions is counted as a tranposition; a displacement by more than two positions is counted as a disagreement.In either case, both characters are now considered as accounted for and are discarded, and **SKOR**ing continues.

If the first character in *XWORD* and *TWORD* do not agree, and neither agree with previously unaccounted-for characters, and *TWORD* has more characters remaining than *XWORD*, **SKOR** removes and saves the first character of *TWORD*, and continues by comparing the rest of *TWORD* with *XWORD* as described above. If *TWORD* has the same or fewer characters remaining than *XWORD*, the procedure is the same except that the character is removed from *XWORD*. In this case, a special check is first made to see if that character is equal to the *previous* character in *XWORD*, or to the *next* character in *XWORD*, i.e., a double character typo, and if so, the character is considered accounted-for, and not counted as a disagreement. In this case,

the "length" of *XWORD* is also decremented. Otherwise making *XWORD* sufficiently long by adding double characters would make it be arbitrarily close to *TWORD*, e.g., **XXXXXX** would correct to **PP**.

When **SKOR** has finished processing both *XWORD* and *TWORD* in this fashion, the value of **SKOR** is the number of unaccounted-for characters, plus the number of disagreements, plus the number of tranpositions, with two qualifications: (1) if both *XWORD* and *TWORD* have a character unaccounted-for in the same position, the two characters are counted only once, i.e., substitution errors count as only one disagreement, not two; and (2) if there are no unaccounted-for characters and no disagreements, transpositions are not counted. This permits spelling correction on very short words, such as edit commands, e.g., **XRT->XTR**. Transpositions are also not counted when **FASTYPEFLG = T**, for example, **IPULX** and **IPLUS** will be in 80% agreement with **FASTYPEFLG = T**, only 60% with **FASTYPEFLG = NIL**. The rationale behind this is that transpositions are much more common for fast typists, and should not be counted as disagreements, whereas more deliberate typists are not as likely to combine tranpositions and other mistakes in a single word, and therefore can use more conservative metric. **FASTYPEFLG** is initially **NIL**.

20.7.5 Spelling Corrector Functions and Variables

(ADDSPELL *X SPLST N***)** [Function]

Adds *X* to one of the spelling lists as determined by the value of *SPLST*:

NIL Adds *X* to **USERWORDS** and to **SPELLINGS2**. Used by **DEFINEQ**.

0 Adds *X* to **USERWORDS**. Used by **LOAD** when loading **EXPR**s to property lists.

1 Adds *X* to **SPELLINGS1** (at end of permanent section). Used by **LISPX**.

2 Adds *X* to **SPELLINGS2** (at end of permanent section). Used by **LISPX**.

3 Adds *X* to **USERWORDS** and **SPELLINGS3**.

a spelling list If *SPLST* is a spelling list, *X* is added to it. In this case, *N* is the (optional) length of the temporary section.

If *X* is already on the spelling list, and in its temporary section, **ADDSPELL** moves *X* to the front of that section.

ADDSPELL sets **LASTWORD** to *X* when *SPLST* = **NIL**, **0** or **3**.

If *X* is not a literal atom, **ADDSPELL** takes no action.

Note that the various systems calls to **ADDSPELL**, e.g. from **DEFINE**, **EDITF**, **LOAD**, etc., can all be suppressed by setting or binding **ADDSPELLFLG** to **NIL** (page 20.13).

(MISSPELLED? *XWORD REL SPLST FLG TAIL FN*) [Function]

If *XWORD* = **NIL** or **$** (<esc>), **MISSPELLED?** prints = followed by the value of **LASTWORD**, and returns this as the respelling, without asking for approval. Otherwise, **MISSPELLED?** checks to see if *XWORD* is really misspelled, i.e., if *FN* applied to *XWORD* is true, or *XWORD* is already contained on *SPLST*. In this case, **MISSPELLED?** simply returns *XWORD*. Otherwise **MISSPELLED?** computes and returns **(FIXSPELL** *XWORD REL SPLST FLG TAIL FN*).

(FIXSPELL *XWORD REL SPLST FLG TAIL FN TIEFLG DONTMOVETOPFLG* — —) [Function]

The value of **FIXSPELL** is either the respelling of *XWORD* or **NIL**. If for some reason *XWORD* itself is on *SPLST*, then **FIXSPELL** aborts and calls **ERROR!**. If there is a possibility that *XWORD* is spelled correctly, **MISSPELLED?** should be used instead of **FIXSPELL**. **FIXSPELL** performs all of the interactions described earlier, including requesting user approval if necessary.

If *XWORD* = **NIL** or **$** (escape), the respelling is the value of **LASTWORD**, and no approval is requested.

If *XWORD* contains lowercase characters, and the corresponding uppercase word is correct, i.e. on *SPLST* or satisfies *FN*, the uppercase word is returned and no interaction is performed. If **FIXSPELL.UPPERCASE.QUIET** is **NIL** (the default), a warning "=XX" is printed when coercing from "xx" to "XX". If **FIXSPELL.UPPERCASE.QUIET** is non-**NIL**, no warning is given.

If *REL* = **NIL**, defaults to the value of **FIXSPELLREL** (initially 70).

If *FLG* = **NIL**, the correction is handled in type-in mode, i.e., approval is never requested, and *XWORD* is not typed. If *FLG* = **T**, *XWORD* is typed (before the =) and approval is requested if **APPROVEFLG** = **T**. If *FLG* = **NO-MESSAGE**, the correction is returned with no further processing. In this case, a run-on correction will be returned as a dotted pair of the two parts of the word, and a synonym correction as a list of the form (*WORD1 WORD2*), where *WORD1* is (the corrected version of) *XWORD*, and *WORD2* is the synonym. Note that the effect of the function **CHOOZ** can be obtained by calling **FIXSPELL** with *FLG* = **NO-MESSAGE**.

If *TAIL* is not **NIL**, and the correction is successful, **CAR** of *TAIL* is replaced by the respelling (using /**RPLACA**).

FIXSPELL will attempt to correct misspellings caused by running two words together, if the global variable **RUNONFLG** is non-**NIL** (default is **NIL**). In this case, approval is always requested. When a run-on error is corrected, **CAR** of *TAIL* is replaced by the two

words, and the value of **FIXSPELL** is the first one. For example, if **FIXSPELL** is called to correct the edit command **(MOVE TO AFTERCOND 3 2)** with *TAIL* = **(AFTERCOND 3 2)**, *TAIL* would be changed to **(AFTER COND 2 3)**, and **FIXSPELL** would return **AFTER** (subject to user approval where necessary). If *TAIL* = **T**, **FIXSPELL** will also perform run-on corrections, returning a dotted pair of the two words in the event the correction is of this type.

If *TIEFLG* = **NIL** and a tie occurs, i.e., more than one word on *SPLST* is found with the same degree of "closeness", **FIXSPELL** returns **NIL**, i.e., no correction. If *TIEFLG* = **PICKONE** and a tie occurs, the first word is taken as the correct spelling. If *TIEFLG* = **LIST**, the value of **FIXSPELL** is a list of the respellings (even if there is only one), and **FIXSPELL** will not perform any interaction with the user, nor modify *TAIL*, the idea being that the calling program will handle those tasks. Similarly, if *TIEFLG* = **EVERYTHING**, a list of all candidates whose degree of closeness is above *REL* will be returned, regardless of whether some are better than others. No interaction will be performed.

If *DONTMOVETOPFLG* = **T** and a correction occurs, it will *not* be moved to the front of the spelling list. Also, the spelling list will not be altered unless it contains the special marker used to separate the temporary and perminant parts of the system spelling lists (the value of **SPELLSTR1**).

(FNCHECK *FN NOERRORFLG SPELLFLG PROPFLG TAIL***)** [Function]

The task of **FNCHECK** is to check whether *FN* is the name of a function and if not, to correct its spelling. If *FN* is the name of a function or spelling correction is successful, **FNCHECK** adds the (corrected) name of the function to **USERWORDS** using **ADDSPELL**, and returns it as its value.

Since **FNCHECK** is called by many low level functions such as **ARGLIST**, **UNSAVEDEF**, etc., spelling correction only takes place when **DWIMFLG** = **T**, so that these functions can operate in a small Interlisp system which does not contain DWIM.

NOERRORFLG informs **FNCHECK** whether or not the calling function wants to handle the unsuccessful case: if *NOERRORFLG* is **T**, **FNCHECK** simply returns **NIL**, otherwise it prints **fn NOT A FUNCTION** and generates a non-breaking error.

If *FN* does not have a definition, but does have an **EXPR** property, then spelling correction is not attempted. Instead, if *PROPFLG* = **T**, *FN* is considered to be the name of a function, and is returned. If *PROPFLG* = **NIL**, *FN* is *not* considered to be the name of a function, and **NIL** is returned or an error generated, depending on the value of *NOERRORFLG*.

FNCHECK calls **MISSPELLED?** to perform spelling correction, so that if *FN* = **NIL**, the value of **LASTWORD** will be returned.

SPELLFLG corresponds to **MISSPELLED?**'s fourth argument, *FLG*. If *SPELLFLG* = **T**, approval will be asked if DWIM was enabled in **CAUTIOUS** mode, i.e., if **APPROVEFLG** = **T**. *TAIL* corresponds to the fifth argument to **MISSPELLED?**.

FNCHECK is currently used by **ARGLIST**, **UNSAVEDEF**, **PRETTYPRINT**, **BREAK0**, **BREAKIN**, **ADVISE**, and **CALLS**. For example, **BREAK0** calls **FNCHECK** with *NOERRORFLG* = **T** since if **FNCHECK** cannot produce a function, **BREAK0** wants to define a dummy one. **CALLS** however calls **FNCHECK** with *NOERRORFLG* = **NIL**, since it cannot operate without a function.

Many other system functions call **MISSPELLED?** or **FIXSPELL** directly. For example, **BREAK1** calls **FIXSPELL** on unrecognized atomic inputs before attempting to evaluate them, using as a spelling list a list of all break commands. Similarly, **LISPX** calls **FIXSPELL** on atomic inputs using a list of all **LISPX** commands. When **UNBREAK** is given the name of a function that is not broken, it calls **FIXSPELL** with two different spelling lists, first with **BROKENFNS**, and if that fails, with **USERWORDS**. **MAKEFILE** calls **MISSPELLED?** using **FILELST** as a spelling list. Finally, **LOAD**, **BCOMPL**, **BRECOMPILE**, **TCOMPL**, and **RECOMPILE** all call **MISSPELLED?** if their input file(s) won't open.

TABLE OF CONTENTS

[This page intentionally left blank]

The syntax of Lisp is very simple, in the sense that it can be described concisely, but not in the sense that Lisp programs are easy to read or write! This simplicity of syntax is achieved by, and at the expense of, extensive use of explicit structuring, namely grouping through parenthesization. Unlike many languages, there are no reserved words in Lisp such as **IF**, **THEN**, **FOR**, **DO**, etc., nor reserved characters like +, -, =, ←, etc. The only special characters are left and right parentheses and period, which are used for indicating structure, and space and end-of-line, which are used for delimiting identifiers. This eliminates entirely the need for parsers and precedence rules in the Lisp interpreter and compiler, and thereby makes program manipulation of Lisp programs straightforward. In other words, a program that "looks at" other Lisp programs does not need to incorporate a lot of syntactic information. For example, a Lisp interpreter can be written in one or two pages of Lisp code. It is for this reason that Lisp is by far the most suitable, and frequently used, programming language for writing programs that deal with other programs as data, e.g., programs that analyze, modify, or construct other programs.

However, it is precisely this same simplicity of syntax that makes Lisp programs difficult to read and write (especially for beginners). 'Pushing down' is something programs do very well, and people do poorly. As an example, consider the following two "equivalent" sentences:

"The rat that the cat that the dog that I owned chased caught ate the cheese."

versus

"I own the dog that chased the cat that caught the rat that ate the cheese."

Natural language contains many linguistic devices such as that illustrated in the second sentence above for minimizing embedding, because embedded sentences are more difficult to grasp and understand than equivalent non-embedded ones (even if the latter sentences are somewhat longer). Similarly, most high level programming languages offer syntactic devices for reducing apparent depth and complexity of a program: the reserved words and infix operators used in ALGOL-like languages simultaneously delimit operands and operations, and also convey meaning to the programmer. They are far more intuitive than parentheses. In fact, since Lisp uses parentheses (i.e., lists)

for almost all syntactic forms, there is very little information contained in the parentheses for the person reading a Lisp program, and so the parentheses tend mostly to be ignored: the meaning of a particular Lisp expression for people is found almost entirely in the *words*, not in the structure. For example, the expression

(COND (EQ N 0) 1) (T TIMES N FACTORIAL ((SUB1 N)))

is recognizable as factorial even though there are five misplaced or missing parentheses. Grouping words together in parentheses is done more for Lisp's benefit, than for the programmer's.

CLISP is designed to make Interlisp programs easier to read and write by permitting the user to employ various infix operators, **IF** statements (page 9.5), and iterative statements (page 9.9), which are automatically converted to equivalent Interlisp expressions when they are first interpreted. For example, factorial could be written in CLISP:

(IF N = 0 THEN 1 ELSE N*(FACTORIAL N-1))

Note that this expression would become an equivalent **COND** after it had been interpreted once, so that programs that might have to analyze or otherwise process this expression could take advantage of the simple syntax.

There have been similar efforts in other Lisp systems. CLISP differs from these in that it does not attempt to *replace* the Lisp syntax so much as to *augment* it. In fact, one of the principal criteria in the design of CLISP was that users be able to freely intermix Lisp and CLISP without having to identify which is which. Users can write programs, or type in expressions for evaluation, in Lisp, CLISP, or a mixture of both. In this way, users do not have to learn a whole new language and syntax in order to be able to use selected facilities of CLISP when and where they find them useful.

CLISP is implemented via the error correction machinery in Interlisp (see page 20.1). Thus, any expression that is well-formed from Interlisp's standpoint will never be seen by CLISP (i.e., if the user defined a function IF, he would effectively turn off that part of CLISP). This means that interpreted programs that do not use CLISP constructs do not pay for its availability by slower execution time. In fact, the Interlisp interpreter does not "know" about CLISP at all. It operates as before, and when an erroneous form is encountered, the interpreter calls an error routine which in turn invokes the Do-What-I-Mean (DWIM) analyzer which contains CLISP. If the expression in question turns out to be a CLISP construct, the equivalent Interlisp form is returned to the interpreter. In addition, the original CLISP expression, is modified so that it

becomes the correctly translated Interlisp form. In this way, the analysis and translation are done only once.

Integrating CLISP into the Interlisp system (instead of, for example, implementing it as a separate preprocessor) makes possible Do-What-I-Mean features for CLISP constructs as well as for pure Lisp expressions. For example, if the user has defined a function named **GET-PARENT**, CLISP would know not to attempt to interpret the form **(GET-PARENT)** as an arithmetic infix operation. (Actually, CLISP would never get to see this form, since it does not contain any errors.) If the user mistakenly writes **(GET-PRAENT)**, CLISP would know he meant **(GET-PARENT)**, and not **(DIFFERENCE GET PRAENT)**, by using the information that **PRAENT** is not the name of a variable, and that **GET-PARENT** is the name of a user function whose spelling is "very close" to that of **GET-PRAENT**. Similarly, by using information about the program's environment not readily available to a preprocessor, CLISP can successfully resolve the following sorts of ambiguities:

(1) **(LIST X*FACT N)**, where **FACT** is the name of a variable, means **(LIST (X*FACT) N)**.

(2) **(LIST X*FACT N)**, where **FACT** is *not* the name of a variable but instead is the name of a function, means **(LIST X*(FACT N))**, i.e., **N** is **FACT**'s argument.

(3) **(LIST X*FACT(N))**, **FACT** the name of a function (and not the name of a variable), means **(LIST X*(FACT N))**.

(4) cases (1), (2) and (3) with **FACT** misspelled!

The first expression is correct both from the standpoint of CLISP syntax and semantics and the change would be made without the user being notified. In the other cases, the user would be informed or consulted about what was taking place. For example, to take an extreme case, suppose the expression **(LIST X*FCCT N)** were encountered, where there was both a function named **FACT** and a variable named **FCT**. The user would first be asked if **FCCT** were a misspelling of **FCT**. If he said YES, the expression would be interpreted as **(LIST (X*FCT) N)**. If he said NO, the user would be asked if **FCCT** were a misspelling of **FACT**, i.e., if he intended **X*FCCT N** to mean **X*(FACT N)**. If he said YES to this question, the indicated transformation would be performed. If he said NO, the system would then ask if **X*FCCT** should be treated as CLISP, since **FCCT** is not the name of a (bound) variable. If he said YES, the expression would be transformed, if NO, it would be left alone, i.e., as **(LIST X*FCCT N)**. Note that we have not even considered the case where **X*FCCT** is itself a misspelling of a variable name, e.g., a variable named **XFCT** (as with **GET-PRAENT**). This sort of transformation would be considered after the user said NO to **X*FCCT N -> X*(FACT N)**.

The question of whether **X*FCCT** should be treated as CLISP is important because Interlisp users may have programs that employ identifiers containing CLISP operators. Thus, if CLISP encounters the expression **A/B** in a context where either **A** or **B** are not the names of variables, it will ask the user if **A/B** is intended to be CLISP, in case the user really does have a free variable named **A/B**.

Note: Through the discussion above, we speak of CLISP or DWIM asking the user. Actually, if the expression in question was typed in by the user for immediate execution, the user is simply informed of the transformation, on the grounds that the user would prefer an occasional misinterpretation rather than being continuously bothered, especially since he can always retype what he intended if a mistake occurs, and ask the programmer's assistant to **UNDO** the effects of the mistaken operations if necessary. For transformations on expressions in user programs, the user can inform CLISP whether he wishes to operate in **CAUTIOUS** or **TRUSTING** mode. In the former case (most typical) the user will be asked to approve transformations, in the latter, CLISP will operate as it does on type-in, i.e., perform the transformation after informing the user.

CLISP can also handle parentheses errors caused by typing **8** or **9** for "(" or ")". (On most terminals, **8** and **9** are the lower case characters for "(" and ")", i.e., "(" and **8** appear on the same key, as do ")" and **9**.) For example, if the user writes **N*8FACTORIAL N-1**, the parentheses error can be detected and fixed before the infix operator * is converted to the Interlisp function **TIMES**. CLISP is able to distinguish this situation from cases like **N*8*X** meaning **(TIMES N 8 X)**, or **N*8X**, where **8X** is the name of a variable, again by using information about the programming environment. In fact, by integrating CLISP with DWIM, CLISP has been made sufficiently tolerant of errors that almost everything can be misspelled! For example, CLISP can successfully translate the definition of **FACTORIAL**:

(IFF N = 0 THENN1 ESLE N*8FACTTORIALNN-1)

to the corresponding **COND**, while making 5 spelling corrections and fixing the parenthesis error. CLISP also contains a facility for converting from Interlisp back to CLISP, so that after running the above incorrect definition of **FACTORIAL**, the user could "clispify" the now correct version to obtain **(IF N = 0 THEN 1 ELSE N*(FACTORIAL N-1))**.

This sort of robustness prevails throughout CLISP. For example, the iterative statement permits the user to say things like:

(FOR OLD X FROM M TO N DO (PRINT X) WHILE (PRIMEP X))

However, the user can also write **OLD (X←M)**, **(OLD X←M)**, **(OLD (X←M))**, permute the order of the operators, e.g., **(DO PRINT X TO N FOR OLD X←M WHILE PRIMEP X)**, omit either or both sets

of parentheses, misspell any or all of the operators **FOR**, **OLD**, **FROM**, **TO**, **DO**, or **WHILE**, or leave out the word **DO** entirely! And, of course, he can also misspell **PRINT**, **PRIMEP**, **M** or **N**! In this example, the only thing the user could not misspell is the first **X**, since it specifies the *name* of the variable of iteration. The other two instances of **X** could be misspelled.

CLISP is well integrated into the Interlisp system. For example, the above iterative statement translates into an following equivalent Interlisp form using **PROG**, **COND**, **GO**, etc. When the interpreter subsequently encounters this CLISP expression, it automatically obtains and evaluates the translation. Similarly, the compiler "knows" to compile the translated form. However, if the user **PRETTYPRINT**s his program, **PRETTYPRINT** "knows" to print the original CLISP at the corresponding point in his function. Similarly, when the user edits his program, the editor keeps the translation invisible to the user. If the user modifies the CLISP, the translation is automatically discarded and recomputed the next time the expression is evaluated.

In short, CLISP is not a language at all, but rather a system. It plays a role analagous to that of the programmer's assistant (page 13.1). Whereas the programmer's assistant is an invisible intermediary agent between the user's console requests and the Interlisp executive, CLISP sits between the user's programs and the Interlisp interpreter.

Only a small effort has been devoted to defining the core syntax of CLISP. Instead, most of the effort has been concentrated on providing a facility which "makes sense" out of the input expressions using context information as well as built-in and acquired information about user and system programs. It has been said that communication is based on the intention of the speaker to produce an effect in the recipient. CLISP operates under the assumption that what the user said was *intended* to represent a meaningful operation, and therefore tries very hard to make sense out of it. The motivation behind CLISP is not to provide the user with many different ways of saying the same thing, but to enable him to worry less about the *syntactic* aspects of his communication with the system. In other words, it gives the user a new degree of freedom by permitting him to concentrate more on the problem at hand, rather than on translation into a formal and unambiguous language.

DWIM and CLISP are invoked on iterative statements because **CAR** of the iterative statement is not the name of a function, and hence generates an error. If the user defines a function by the same name as an i.s. operator, e.g., **WHILE**, **TO**, etc., the operator will no longer have the CLISP interpretation when it appears as **CAR** of a form, although it will continue to be treated as an i.s. operator if it appears in the interior of an i.s. To alert the user, a

warning message is printed, e.g., **(WHILE DEFINED, THEREFORE DISABLED IN CLISP)**.

21.1 CLISP Interaction with User

Syntactically and semantically well formed CLISP transformations are always performed without informing the user. Other CLISP transformations described in the previous section, e.g., misspellings of operands, infix operators, parentheses errors, unary minus - binary minus errors, all follow the same protocol as other DWIM transformations (page 20.1). That is, if DWIM has been enabled in **TRUSTING** mode, or the transformation is in an expression typed in by the user for immediate execution, user approval is not requested, but the user is informed. However, if the transformation involves a user program, and DWIM was enabled in **CAUTIOUS** mode, the user will be asked to approve. If he says NO, the transformation is not performed. Thus, in the previous section, phrases such as "one of these (transformations) succeeds" and "the transformation **LAST-ELL -> LAST-EL** would be found" etc., all mean if the user is in **CAUTIOUS** mode and the error is in a program, the corresponding transformation will be performed only if the user approves (or defaults by not responding). If the user says NO, the procedure followed is the same as though the transformation had not been found. For example, if **A*B** appears in the function **FOO**, and **B** is not bound (and no other transformations are found) the user would be asked **A*B [IN FOO] TREAT AS CLISP ?** (The waiting time on such interactions is three times as long as for simple corrections, i.e., **3*DWIMWAIT**).

In certain situations, DWIM will ask for approval even if DWIM is enabled in **TRUSTING** mode. For example, the user will always be asked to approve a spelling correction that might also be interpreted as a CLISP transformation, as in **LAST-ELL -> LAST-EL**.

If the user approved, **A*B** would be transformed to **(ITIMES A B)**, which would then cause a **U.B.A. B** error in the event that the program was being run (remember the entire discussion also applies to **DWIMIFY**ing). If the user said **NO**, **A*B** would be left alone.

If the value of **CLISPHELPFLG = NIL** (initally **T**), the user will not be asked to approve any CLISP transformation. Instead, in those situations where approval would be required, the effect is the same as though the user had been asked and said **NO**.

21.2 CLISP Character Operators

CLISP recognizes a number of special characters operators, both prefix and infix, which are translated into common expressions. For example, the character + is recognized to represent addition, so CLISP translates the litatom **A + B** to the form **(IPLUS A B)**. Note that CLISP is envoked, and this translation is made, only if an error occurs, such as an unbound atom error or an undefined function error for the perfectly legitamate litatom **A + B**. Therefore the user may choose not to use these facilities with no penalty, similar to other CLISP facilities.

The user has a lot of flexability in using CLISP character operators. A list, can always be substituted for a litatom, and vice versa, without changing the interpretation of a phrase. For example, if the value of **(FOO X)** is **A**, and the value of **(FIE Y)** is **B**, then **(LIST (FOO X) + (FIE Y))** has the same value as **(LIST A + B)**. Note that the first expression is a list of *four* elements: the atom **"LIST"**, the list **"(FOO X)"**, the atom " + ", and the list **"(FIE X)"**, whereas the second expression, **(LIST A + B)**, is a list of only *two* elements: the litatom **"LIST"** and the litatom **"A + B"**. Since **(LIST (FOO X) + (FIE Y))** is indistinguishable from **(LIST (FOO X) + (FIE Y))** because spaces before or after parentheses have no effect on the Interlisp READ program, to be consistent, extra spaces have no effect on atomic operands either. In other words, CLISP will treat **(LIST A + B)**, **(LIST A + B)**, and **(LIST A + B)** the same as **(LIST A + B)**.

Note: CLISP does not use its own special READ program because this would require the user to explicitly identify CLISP expressions, instead of being able to intermix Interlisp and CLISP.

+ [CLISP Operator]

- [CLISP Operator]

***** [CLISP Operator]

/ [CLISP Operator]

↑ [CLISP Operator]

CLISP recognizes **+**, **-**, *****, **/**, and **↑** as the normal arithmetic infix operators. **-** is also recognized as the prefix operator, unary minus. These are converted to **PLUS**, **DIFFERENCE** (or in the case of unary minus, **MINUS**), **TIMES**, **QUOTIENT**, and **EXPT**.

Normally, CLISP uses the "generic" arithmetic functions **PLUS**, **TIMES**, etc. CLISP contains a facility for declaring which type of arithmetic is to be used, either by making a global declaration, or

by separate declarations about individual functions or variables (see page 21.12).

The usual precedence rules apply (although these can be easily changed by the user), i.e., * has higher precedence than + so that **A + B*C** is the same as **A + (B*C)**, and both * and / are lower than ↑ so that **2*X ↑ 2** is the same as **2*(X ↑ 2)**. Operators of the same precedence group from left to right, e.g., **A/B/C** is equivalent to **(A/B)/C**. Minus is binary whenever possible, i.e., except when it is the first operator in a list, as in **(-A)** or **(-A)**, or when it immediately follows another operator, as in **A*-B**. Note that grouping with parentheses can always be used to override the normal precedence grouping, or when the user is not sure how a particular expression will parse. The complete order of precedence for CLISP operators is given below.

Note that + in front of a number will disappear when the number is read, e.g., **(FOO X +2)** is indistinguishable from **(FOO X 2)**. This means that **(FOO X +2)** will not be interpreted as CLISP, or be converted to **(FOO (IPLUS X 2))**. Similarly, **(FOO X -2)** will not be interpreted the same as **(FOO X-2)**. To circumvent this, always type a space between the + or - and a number if an infix operator is intended, e.g., write **(FOO X + 2)**.

= [CLISP Operator]

GT [CLISP Operator]

LT [CLISP Operator]

GE [CLISP Operator]

LE [CLISP Operator]

These are infix operators for "Equal", "Greater Than", "Less Than", "Greater Than or Equal", and "Less Than or Equal".

GT, **LT**, **GE**, and **LE** are all affected by the same declarations as + and *, with the initial default to use **GREATERP** and **LESSP**.

Note that only single character operators, e.g., +, ←, =, etc., can appear in the *interior* of an atom. All other operators must be set off from identifiers with spaces. For example, **XLTY** will not be recognized as CLISP. In some cases, DWIM will be able to diagnose this situation as a run-on spelling error, in which case after the atom is split apart, CLISP will be able to perform the indicated transformation.

A number of lisp functions, such as **EQUAL**, **MEMBER**, **AND**, **OR**, etc., can also be treated as CLISP infix operators. New infix

operators can be easily added (see page 21.27). Spelling correction on misspelled infix operators is peformed using **CLISPINFIXSPLST** as a spelling list.

AND is higher than **OR**, and both **AND** and **OR** are lower than the other infix operators, so **(X OR Y AND Z)** is the same as **(X OR (Y AND Z))**, and **(X AND Y EQUAL Z)** is the same as **(X AND (Y EQUAL Z))**. All of the infix predicates have lower precedence than Interlisp forms, since it is far more common to apply a predicate to two forms, than to use a Boolean as an argument to a function. Therefore, **(FOO X GT FIE Y)** is translated as **((FOO X) GT (FIE Y))**, rather than as **(FOO (X GT (FIE Y)))**. However, the user can easily change this.

: [CLISP Operator]

X:*N* extracts the *N*th element of the list *X*. **FOO:3** specifies the third element of **FOO**, or **(CADDR FOO)**. If *N* is less than zero, this indicates elements counting from the end of the list; i.e. **FOO:-1** is the last element of **FOO**. **:** operators can be nested, so **FOO:1:2** means the second element of the first element of **FOO**, or **(CADAR FOO)**.

The **:** operator can also be used for extracting substructures of records (see page 8.1). Record operations are implemented by replacing expressions of the form **X:FOO** by **(fetch FOO of X)**. Both lower and upper case are acceptable.

: is also used to indicate operations in the pattern match facility (page 12.24). **X:(& 'A -- 'B)** translates to **(match X with (& 'A -- 'B))**

. [CLISP Operator]

In combination with **:**, a period can be used to specify the "data path" for record operations. For example, if **FOO** is a field of the **BAR** record, **X:BAR.FOO** is translated into **(fetch (BAR FOO) of X)**. Subrecord fields can be specified with multiple periods: **X:BAR.FOO.BAZ** translates into **(fetch (BAR FOO BAZ) of X)**.

Note: If a record contains fields with periods in them, **CLISPIFY** will not translate a record operation into a form using periods to specify the data path. For example, **CLISPIFY** will NOT translate **(fetch A.B of X)** into **X:A.B**.

:: [CLISP Operator]

X:*N*, returns the *N*th *tail* of the list *X*. For example, **FOO::3** is **(CDDDR FOO)**, and **FOO::-1** is **(LAST FOO)**.

← [CLISP Operator]

← is used to indicate assignment. For example, **X←Y** translates to **(SETQ X Y)**. If **X** does not have a value, and is not the name of one of the bound variables of the function in which it appears,

spelling correction is attempted. However, since this may simply be a case of assigning an initial value to a new free variable, DWIM will always ask for approval before making the correction.

In conjunction with : and ::, ← can also be used to perform a more general type of assignment, involving structure modification. For example, **X:2←Y** means "make the second element of **X** *be* **Y**", in Interlisp terms **(RPLACA (CDR X) Y)**. Note that the *value* of this operation is the value of **RPLACA**, which is **(CDR X)**, rather than **Y**. Negative numbers can also be used, e.g., **X:-2←Y**, which translates to **(RPLACA (NLEFT X 2) Y)**.

The user can indicate he wants /**RPLACA** and /**RPLACD** used (undoable version of **RPLACA** and **RPLACD**, see page 13.26), or **FRPLACA** and **FRPLACD** (fast versions of **RPLACA** and **RPLACD**, see page 3.3), by means of CLISP declarations (page 21.12). The initial default is to use **RPLACA** and **RPLACD**.

← is also used to indicate assignment in record operations (**X:FOO←Y** translates to **(replace FOO of X with Y)**.), and pattern match operations (page 12.24).

← has different precedence on the left from on the right. On the left, ← is a "tight" operator, i.e., high precedence, so that **A + B←C** is the same as **A + (B←C)**. On the right, ← has broader scope so that **A←B + C** is the same as **A←(B + C)**.

On typein, **$←***FORM* (where **$** is the escape key) is equivalent to set the "last thing mentioned", i.e., is equivalent to **(SET LASTWORD** *FORM***)** (see page 20.18). For example, immediately after examining the value of **LONGVARIABLENAME**, the user could set it by typing **$←** followed by a form.

Note that an atom of the form **X←Y**, appearing at the top level of a **PROG**, will *not* be recognized as an assignment statement because it will be interpreted as a **PROG** label by the Interlisp interpreter, and therefore will not cause an error, so DWIM and CLISP will never get to see it. Instead, one must write **(X←Y)**.

< [CLISP Operator]

> [CLISP Operator]

Angle brackets are used in CLISP to indicate list construction. The appearance of a "**<**" corresponds to a "**(**" and indicates that a list is to be constructed containing all the elements up to the corresponding "**>**". For example, **<A B <C>>** translates to **(LIST A B (LIST C))**. **!** can be used to indicate that the next expression is to be inserted in the list as a *segment*, e.g., **<A B ! C>** translates to **(CONS A (CONS B C))** and **<! A ! B C>** to **(APPEND A B (LIST C))**. **!!** is used to indicate that the next expression is to be inserted as a segment, and furthermore, all list

structure to its right in the angle brackets is to be physically attached to it, e.g., **<!! A B>** translates to **(NCONC1 A B)**, and **<!!A !B !C>** to **(NCONC A (APPEND B C))**. Not **(NCONC (APPEND A B) C)**, which would have the same value, but would attach **C** to **B**, and not attach either to **A**. Note that **<**, **!**, **!!**, and **>** need not be separate atoms, for example, **<A B ! C>** may be written equally well as **< A B !C >**. Also, arbitrary Interlisp or CLISP forms may be used within angle brackets. For example, one can write **<FOO←(FIE X) ! Y>** which translates to **(CONS (SETQ FOO (FIE X)) Y)**. **CLISPIFY** converts expressions in **CONS**, **LIST**, **APPEND**, **NCONC**, **NCONC1**, **/NCONC**, and **/NCONC1** into equivalent CLISP expressions using **<**, **>**, **!**, and **!!**.

Note: brackets differ from other CLISP operators. For example, **<A B 'C>** translates to **(LIST A B (QUOTE C))** even though following **'**, all *operators* are ignored for the rest of the identifier. (This is true only if a previous unmatched **<** has been seen, e.g., **(PRINT 'A>B)** will print the atom **A>B**.) Note however that **<A B ' C> D>** is equivalent to **(LIST A B (QUOTE C>) D)**.

' [CLISP Operator]

CLISP recognizes **'** as a prefix operator. **'** means **QUOTE** when it is the first character in an identifier, and is ignored when it is used in the interior of an identifier. Thus, **X = 'Y** means **(EQ X (QUOTE Y))**, but **X = CAN'T** means **(EQ X CAN'T)**, *not* **(EQ X CAN)** followed by **(QUOTE T)**. This enables users to have variable and function names with **'** in them (so long as the **'** is not the first character).

Following **'**, all operators are ignored for the rest of the identifier, e.g., **'*A** means **(QUOTE *A)**, and **'X = Y** means **(QUOTE X = Y)**, not **(EQ (QUOTE X) Y)**. To write **(EQ (QUOTE X) Y)**, one writes **Y = 'X**, or **'X = Y**. This is one place where an extra space does make a difference.

On typein, **'$** (escape) is equivalent to **(QUOTE *VALUE-OF-LASTWORD*)** (see page 20.18). For example, after calling **PRETTYPRINT** on **LONGFUNCTION**, the user could move its definition to **FOO** by typing **(MOVD '$ 'FOO)**.

Note that this is not **(MOVD $ 'FOO)**, which would be equivalent to **(MOVD LONGFUNCTION 'FOO)**, and would (probably) cause a **U.B.A. LONGFUNCTION** error, nor **MOVD($ FOO)**, which would actually move the definition of **$** to **FOO**, since DWIM and the spelling corrector would never be invoked.

~ [CLISP Operator]

CLISP recognizes **~** as a prefix operator meaning **NOT**. **~** can negate a form, as in **~(ASSOC X Y)**, or **~X**, or negate an infix operator, e.g., **(A ~GT B)** is the same as **(A LEQ B)**. Note that **~A = B** means **(EQ (NOT A) B)**.

When ~ negates an operator, e.g., ~ = , ~LT, the two operators are treated as a single operator whose precedence is that of the second operator. When ~ negates a function, e.g., (~FOO X Y), it negates the whole form, i.e., (~(FOO X Y)).

Order of Prededence of CLISP Operators:

.

:

← (left precedence)

- (unary), ~

↑

*, /

+, - (binary)

← (right precedence)

=

Interlisp forms

LT, GT, EQUAL, MEMBER, etc.

AND

OR

IF, THEN, ELSEIF, ELSE

iterative statement operators

21.3 Declarations

CLISP declarations are used to affect the choice of Interlisp function used as the translation of a particular operator. For example, **A + B** can be translated as either **(PLUS A B)**, **(FPLUS A B)**, or **(IPLUS A B)**, depending on the declaration in effect. Similarly **X:1←Y** can mean **(RPLACA X Y)**, **(FRPLACA X Y)**, or **(/RPLACA X Y)**, and **<!! A B>** either **(NCONC1 A B)** or **(/NCONC1 A B)**. Note that the choice of function on all CLISP transformations are affected by the CLISP declaration in effect, i.e., iterative statements, pattern matches, record operations, as well as infix and prefix operators.

(CLISPDEC *DECLST***)** [Function]

Puts into effect the declarations in *DECLST*. **CLISPDEC** performs spelling corrections on words not recognized as declarations. **CLISPDEC** is undoable.

The user can makes (changes) a global declaration by calling **CLISPDEC** with *DECLST* a list of declarations, e.g., **(CLISPDEC '(FLOATING UNDOABLE))**. Changing a global declaration does not affect the speed of subsequent CLISP transformations, since

all CLISP transformation are table driven (i.e., property list), and global declarations are accomplished by making the appropriate internal changes to CLISP at the time of the declaration. If a function employs *local* declarations (described below), there will be a slight loss in efficiency owing to the fact that for each CLISP transformation, the declaration list must be searched for possibly relevant declarations.

Declarations are implemented in the order that they are given, so that later declarations override earlier ones. For example, the declaration **FAST** specifies that **FRPLACA, FRPLACD, FMEMB,** and **FLAST** be used in place of **RPLACA, RPLACD, MEMB,** and **LAST**; the declaration **RPLACA** specifies that **RPLACA** be used. Therefore, the declarations **(FAST RPLACA RPLACD)** will cause **FMEMB, FLAST, RPLACA,** and **RPLACD** to be used.

The initial global declaration is **MIXED** and **STANDARD**.

The table below gives the declarations available in CLISP, and the Interlisp functions they indicate:

Declaration:	Interlisp Functions to be used:
MIXED	**PLUS, MINUS, DIFFERENCE, TIMES, QUOTIENT, LESSP, GREATERP**
INTEGER or **FIXED**	**IPLUS, IMINUS, IDIFFERENCE, ITIMES, IQUOTIENT, ILESSP, IGREATERP**
FLOATING	**FPLUS, FMINUS, FDIFFERENCE, FTIMES, FQUOTIENT, LESSP, FGREATERP**
FAST	**FRPLACA, FRPLACD, FMEMB, FLAST, FASSOC**
UNDOABLE	**/RPLACA, /RPLACD, /NCONC, /NCONC1, /MAPCONC, /MAPCON**
STANDARD	**RPLACA, RPLACD, MEMB, LAST, ASSOC, NCONC, NCONC1, MAPCONC, MAPCON**
RPLACA, RPLACD, /RPLACA, etc.	corresponding function

The user can also make local declarations affecting a selected function or functions by inserting an expression of the form **(CLISP: .** *DECLARATIONS*) immediately following the argument list, i.e., as **CADDR** of the definition. Such local declarations take precedence over global declarations. Declarations affecting selected variables can be indicated by lists, where the first element is the name of a variable, and the rest of the list the declarations for that variable. For example, **(CLISP: FLOATING (X INTEGER))** specifies that in this function integer arithmetic be used for computations involving **X**, and floating arithmetic for all other computations, where "involving" means where the variable itself is an operand. For example, with the declaration **(FLOATING (X INTEGER))** in effect, **(FOO X) + (FIE X)** would translate to **FPLUS**, i.e., use floating arithmetic, even though **X** appears somewhere inside of the operands, whereas **X + (FIE X)**

would translate to **IPLUS**. If there are declarations involving *both* operands, e.g., **X + Y**, with **(X FLOATING) (Y INTEGER)**, whichever appears first in the declaration list will be used.

The user can also make local record declarations by inserting a record declaration, e.g., **(RECORD --)**, **(ARRAYRECORD --)**, etc., in the local declaration list. In addition, a local declaration of the form **(RECORDS A B C)** is equivalent to having copies of the global declarations **A**, **B**, and **C** in the local declaration. Local record declarations override global record declarations for the function in which they appear. Local declarations can also be used to override the global setting of certain DWIM/CLISP parameters effective only for transformations within that function, by including in the local declaration an expression of the form (*VARIABLE* = *VALUE*), e.g., **(PATVARDEFAULT = QUOTE)**.

The **CLISP:** expression is converted to a comment of a special form recognized by CLISP. Whenever a CLISP transformation that is affected by declarations is about to be performed in a function, this comment will be searched for a relevant declaration, and if one is found, the corresponding function will be used. Otherwise, if none are found, the global declaration(s) currently in effect will be used.

Local declarations are effective in the order that they are given, so that later declarations can be used to override earlier ones, e.g., **(CLISP: FAST RPLACA RPLACD)** specifies that **FMEMB**, **FLAST**, **RPLACA**, and **RPLACD** be used. An exception to this is that declarations for specific variables take precedence of general, function-wide declarations, regardless of the order of appearance, as in **(CLISP: (X INTEGER) FLOATING)**.

CLISPIFY also checks the declarations in effect before selecting an infix operator to ensure that the corresponding CLISP construct would in fact translate back to this form. For example, if a **FLOATING** declaration is in effect, **CLISPIFY** will convert **(FPLUS X Y)** to **X + Y**, but leave **(IPLUS X Y)** as is. Note that if **(FPLUS X Y)** is **CLISPIFY**ed while a **FLOATING** declaration is under effect, and then the declaration is changed to **INTEGER**, when **X + Y** is translated back to Interlisp, it will become **(IPLUS X Y)**.

21.4 CLISP Operation

CLISP is a part of the basic Interlisp system. Without any special preparations, the user can include CLISP constructs in programs, or type them in directly for evaluation (in **EVAL** or **APPLY** format), then, when the "error" occurrs, and DWIM is called, it will destructively transform the CLISP to the equivalent Interlisp

expression and evaluate the Interlisp expression. CLISP transformations, like all DWIM corrections, are undoable. User approval is not requested, and no message is printed. Note that this entire discussion also applies to CLISP transformation initiated by calls to DWIM from **DWIMIFY**.

However, if a CLISP construct contains an error, an appropriate diagnostic is generated, and the form is left unchanged. For example, if the user writes **(LIST X + Y*)**, the error diagnostic **MISSING OPERAND AT X + Y* IN (LIST X + Y*)** would be generated. Similarly, if the user writes **(LAST + EL X)**, CLISP knows that **((IPLUS LAST EL) X)** is not a valid Interlisp expression, so the error diagnostic **MISSING OPERATOR IN (LAST + EL X)** is generated. (For example, the user might have meant to say **(LAST + EL*X)**.) Note that if **LAST + EL** were the name of a defined function, CLISP would never see this form.

Since the bad CLISP transformation might not be CLISP at all, for example, it might be a misspelling of a user function or variable, DWIM holds all CLISP error messages until after trying other corrections. If one of these succeeds, the CLISP message is discarded. Otherwise, if all fail, the message is printed (but no change is made). For example, suppose the user types **(R/PLACA X Y)**. CLISP generates a diagnostic, since **((IQUOTIENT R PLACA) X Y)** is obviously not right. However, since **R/PLACA** spelling corrects to **/RPLACA**, this diagnostic is never printed.

Note: CLISP error messages are not printed on type-in. For example, typing **X + *Y** will just produce a **U.B.A. X + *Y** message.

If a CLISP infix construct is well formed from a syntactic standpoint, but one or both of its operands are atomic and not bound, it is possible that either the operand is misspelled, e.g., the user wrote **X + YY** for **X + Y**, or that a CLISP transformation operation was not intended at all, but that the entire expression is a misspelling. For the purpose of **DWIMIFY**ing, "not bound" means no top level value, not on list of bound variables built up by **DWIMIFY** during its analysis of the expression, and not on **NOFIXVARSLST**, i.e., not previously seen.

For example, if the user has a variable named **LAST-EL**, and writes **(LIST LAST-ELL)**. Therefore, CLISP computes, but does not actually perform, the indicated infix transformation. DWIM then continues, and if it is able to make another correction, does so, and ignores the CLISP interpretation. For example, with **LAST-ELL**, the transformation **LAST-ELL -> LAST-EL** would be found.

If no other transformation is found, and DWIM is about to interpret a construct as CLISP for which one of the operands is not bound, DWIM will ask the user whether CLISP was intended, in this case by printing **LAST-ELL TREAT AS CLISP ?**.

Note: If more than one infix operator was involved in the CLISP construct, e.g., **X + Y + Z**, or the operation was an assignment to a variable already noticed, or **TREATASCLISPFLG** is **T** (initially **NIL**), the user will simply be informed of the correction, e.g., **X + Y + Z TREATED AS CLISP**. Otherwise, even if DWIM was enabled in **TRUSTING** mode, the user will be asked to approve the correction.

The same sort of procedure is followed with 8 and 9 errors. For example, suppose the user writes **FOO8*X** where **FOO8** is not bound. The CLISP transformation is noted, and DWIM proceeds. It next asks the user to approve **FOO8*X -> FOO (*X**. For example, this would make sense if the user has (or plans to define) a function named ***X**. If he refuses, the user is asked whether **FOO8*X** is to be treated as CLISP. Similarly, if **FOO8** were the name of a variable, and the user writes **FOOO8*X**, he will first be asked to approve **FOOO8*X -> FOOO (XX**, and if he refuses, then be offered the **FOOO8 -> FOO8** correction. The 8-9 transformation is tried before spelling correction since it is empirically more likely that an unbound atom or undefined function containing an 8 or a 9 is a parenthesis error, rather than a spelling error.

CLISP also contains provision for correcting misspellings of infix operators (other than single characters), **IF** words, and i.s. operators. This is implemented in such a way that the user who does not misspell them is not penalized. For example, if the user writes **IF N = 0 THEN 1 ELSSE N*(FACT N-1)** CLISP does *not* operate by checking each word to see if it is a misspelling of **IF**, **THEN**, **ELSE**, or **ELSEIF**, since this would seriously degrade CLISP's performance on *all* **IF** statements. Instead, CLISP assumes that all of the **IF** words are spelled correctly, and transforms the expression to **(COND ((ZEROP N) 1 ELSSE N*(FACT N-1)))**. Later, after DWIM cannot find any other interpretation for **ELSSE**, and using the fact that this atom originally appeared in an IF statement, DWIM attempts spelling correction, using **(IF THEN ELSE ELSEIF)** for a spelling list. When this is successful, DWIM "fails" all the way back to the original **IF** statement, changes **ELSSE** to **ELSE**, and starts over. Misspellings of **AND**, **OR**, **LT**, **GT**, etc. are handled similarly.

CLISP also contains many Do-What-I-Mean features besides spelling corrections. For example, the form **(LIST + X Y)** would generate a **MISSING OPERATOR** error. However, **(LIST -X Y)** makes sense, if the minus is unary, so DWIM offers this interpretation to the user. Another common error, especially for new users, is to write **(LIST X*FOO(Y))** or **(LIST X*FOO Y)**, where **FOO** is the name of a function, instead of **(LIST X*(FOO Y))**. Therefore, whenever an operand that is not bound is also the name of a function (or corrects to one), the above interpretations are offered.

21.5 CLISP Translations

The translation of CLISP character operators and the CLISP word **IF** are handled by *replacing* the CLISP expression with the corresponding Interlisp expression, and discarding the original CLISP. This is done because (1) the CLISP expression is easily recomputable (by **CLISPIFY**) and (2) the Interlisp expressions are simple and straightforward. Another reason for discarding the original CLISP is that it may contain errors that were corrected in the course of translation (e.g., **FOO←FOOO:1**, **N*8FOO X**), etc.). If the original CLISP were retained, either the user would have to go back and fix these errors by hand, thereby negating the advantage of having DWIM perform these corrections, or else DWIM would have to keep correcting these errors over and over.

Note that **CLISPIFY** is sufficiently fast that it is practical for the user to configure his Interlisp system so that all expressions are automatically **CLISPIFY**ed immediately before they are presented to him. For example, he can define an edit macro to use in place of **P** which calls **CLISPIFY** on the current expression before printing it. Similarly, he can inform **PRETTYPRINT** to call **CLISPIFY** on each expression before printing it, etc.

Where (1) or (2) are not the case, e.g., with iterative statements, pattern matches, record expressions, etc. the original CLISP *is* retained (or a slightly modified version thereof), and the translation is stored elsewhere (by the function **CLISPTRAN**, page 21.25), usually in the hash array **CLISPARRAY**. The interpreter automatically checks this array when given a form **CAR** of which is not a function. Similarly, the compiler performs a **GETHASH** when given a form it does not recognize to see if it has a translation, which is then compiled instead of the form. Whenever the user *changes* a CLISP expresson by editing it, the editor automatically deletes its translation (if one exists), so that the next time it is evaluated or dwimified, the expression will be retranslated (if the value of **CLISPRETRANFLG** is **T**, **DWIMIFY** will also (re)translate any expressions which have translations stored remotely, see page 21.22). The function **PPT** and the edit commands **PPT** and **CLISP:** are available for examining translations (page 21.26).

The user can also indicate that he wants the original CLISP retained by embedding it in an expression of the form **(CLISP .** *CLISP-EXPRESSION*), e.g., **(CLISP X:5:3)** or **(CLISP <A B C ! D>)**. In such cases, the translation will be stored remotely as described above. Furthermore, such expressions will be treated as CLISP even if infix and prefix transformations have been disabled by setting **CLISPFLG** to **NIL** (page 21.25). In other words, the user can instruct the system to interpret as CLISP infix or prefix constructs only those expressions that are specifically flagged as such. The user can also include CLISP declarations by writing

(**CLISP** *DECLARATIONS . FORM*), e.g., (**CLISP** (**CLISP: FLOATING**) ...). These declarations will be used in place of any CLISP declarations in the function definition. This feature provides a way of including CLISP declarations in macro definitions.

Note: CLISP translations can also be used to supply an interpretation for function objects, as well as forms, either for function objects that are used openly, i.e., appearing as **CAR** of form, function objects that are explicitly **APPLY**ed, as with arguments to mapping functions, or function objects contained in function definition cells. In all cases, if **CAR** of the object is not **LAMBDA** or **NLAMBDA**, the interpreter and compiler will check **CLISPARRAY**.

21.6 DWIMIFY

DWIMIFY is effectively a preprocessor for CLISP. **DWIMIFY** operates by scanning an expression as though it were being interpreted, and for each form that would generate an error, calling DWIM to "fix" it. **DWIMIFY** performs *all* DWIM transformations, not just CLISP transformations, so it does spelling correction, fixes 8-9 errors, handles **F/L**, etc. Thus the user will see the same messages, and be asked for approval in the same situations, as he would if the expression were actually run. If DWIM is unable to make a correction, no message is printed, the form is left as it was, and the analysis proceeds.

DWIMIFY knows exactly how the interpreter works. It knows the syntax of **PROG**s, **SELECTQ**s, **LAMBDA** expressions, **SETQ**s, et al. It knows how variables are bound, and that the argument of **NLAMBDA**s are not evaluated (the user can inform **DWIMIFY** of a function or macro's nonstandard binding or evaluation by giving it a suitable **INFO** property, see page 21.21). In the course of its analysis of a particular expression, **DWIMIFY** builds a list of the bound variables from the **LAMBDA** expressions and **PROG**s that it encounters. It uses this list for spelling corrections. **DWIMIFY** also knows not to try to "correct" variables that are on this list since they would be bound if the expression were actually being run. However, note that **DWIMIFY** cannot, a priori, know about variables that are used freely but would be bound in a higher function if the expression were evaluated in its normal context. Therefore, **DWIMIFY** will try to "correct" these variables. Similarly, **DWIMIFY** will attempt to correct forms for which **CAR** is undefined, even when the form is not in error from the user's standpoint, but the corresponding function has simply not yet been defined.

Note: **DWIMIFY** rebinds **FIXSPELLDEFAULT** to **N**, so that if the user is not at the terminal when dwimifying (or compiling), spelling corrections will not be performed.

DWIMIFY will also inform the user when it encounters an expression with too *many* arguments (unless **DWIMCHECK#ARGSFLG** = **NIL**), because such an occurrence, although does not cause an error in the Interlisp interpreter, nevertheless is frequently symptomatic of a parenthesis error. For example, if the user wrote **(CONS (QUOTE FOO X))** instead of **(CONS (QUOTE FOO) X)**, **DWIMIFY** will print:

POSSIBLE PARENTHESIS ERROR IN
(QUOTE FOO X)
TOO MANY ARGUMENTS (MORE THAN 1)

DWIMIFY will also check to see if a **PROG** label contains a clisp character (unless **DWIMCHECKPROGLABELSFLG** = **NIL**, or the label is a member of **NOFIXVARSLST**), and if so, will alert the user by printing the message **SUSPICIOUS PROG LABEL**, followed by the label. The **PROG** label will *not* be treated as CLISP.

Note that in most cases, an attempt to transform a form that is already as the user intended will have no effect (because there will be nothing to which that form could reasonably be transformed). However, in order to avoid needless calls to DWIM or to avoid possible confusion, the user can inform **DWIMIFY** *not* to attempt corrections or transformations on certain functions or variables by adding them to the list **NOFIXFNSLST** or **NOFIXVARSLST** respectively. Note that the user could achieve the same effect by simply setting the corresponding variables, and giving the functions dummy definitions.

DWIMIFY will never attempt corrections on global variables, i.e., variables that are a member of the list **GLOBALVARS**, or have the property **GLOBALVAR** with value **T**, on their property list. Similarly, **DWIMIFY** will not attempt to correct variables declared to be **SPECVARS** in block declarations or via **DECLARE** expressions in the function body. The user can also declare variables that are simply used freely in a function by using the **USEDFREE** declaration.

DWIMIFY and **DWIMIFYFNS** (used to **DWIMIFY** several functions) maintain two internal lists of those functions and variables for which corrections were unsuccessfully attempted. These lists are initialized to the values of **NOFIXFNSLST** and **NOFIXVARSLST**. Once an attempt is made to fix a particular function or variable, and the attempt fails, the function or variable is added to the corresponding list, so that on subsequent occurrences (within this call to **DWIMIFY** or **DWIMIFYFNS**), no attempt at correction is made. For example, if **FOO** calls **FIE** several times, and **FIE** is undefined at the time **FOO** is dwimified, **DWIMIFY** will not bother with **FIE** after the first occurrence. In other words, once

DWIMIFY "notices" a function or variable, it no longer attempts to correct it. **DWIMIFY** and **DWIMIFYFNS** also "notice" free variables that are set in the expression being processed. Moreover, once **DWIMIFY** "notices" such functions or variables, it subsequently treats them the same as though they were actually defined or set.

Note that these internal lists are local to each call to **DWIMIFY** and **DWIMIFYFNS**, so that if a function containing **FOOO**, a misspelled call to **FOO**, is **DWIMIFY**ed before **FOO** is defined or mentioned, if the function is **DWIMIFY**ed again after **FOO** has been defined, the correction will be made.

The user can undo selected transformations performed by **DWIMIFY**, as described on page 13.14.

(DWIMIFY *X QUIETFLG L***)** [Function]

Performs all DWIM and CLISP corrections and transformations on *X* that would be performed if *X* were run, and prints the result unless *QUIETFLG* = **T**.

If *X* is an atom and *L* is **NIL**, *X* is treated as the name of a function, and its entire definition is dwimified. If *X* is a list or *L* is not **NIL**, *X* is the expression to be dwimified. If *L* is not **NIL**, it is the edit push-down list leading to *X*, and is used for determining context, i.e., what bound variables would be in effect when *X* was evaluated, whether *X* is a form or sequence of forms, e.g., a **COND** clause, etc.

If *X* is an iterative statement and *L* is **NIL**, **DWIMIFY** will also print the translation, i.e., what is stored in the hash array.

(DWIMIFYFNS FN_1 ... FN_N**)** [NLambda NoSpread Function]

Dwimifies each of the functions given. If only one argument is given, it is evalued. If its value is a list, the functions on this list are dwimified. If only one argument is given, it is atomic, its value is not a list, and it is the name of a known file, **DWIMIFYFNS** will operate on **(FILEFNSLST** FN_1**)**, e.g. **(DWIMIFYFNS FOO.LSP)** will dwimify every function in the file **FOO.LSP**.

Every 30 seconds, **DWIMIFYFNS** prints the name of the function it is processing, a la **PRETTYPRINT**.

Value is a list of the functions dwimified.

DWIMINMACROSFLG [Variable]

Controls how **DWIMIFY** treats the arguments in a "call" to a macro, i.e., where the **CAR** of the form is undefined, but has a macro definition. If **DWIMINMACROSFLG** is **T**, then macros are treated as **LAMBDA** functions, i.e., the arguments are assumed to

be evaluated, which means that **DWIMIFY** will descend into the argument list. If **DWIMINMACROSFLG** is **NIL**, macros are treated as **NLAMBDA** functions. **DWIMINMACROSFLG** is initially **T**.

INFO [Property Name]

Used to inform **DWIMIFY** of nonstandard behavior of particular forms with respect to evaluation, binding of arguments, etc. The **INFO** property of a litatom is a single atom or list of atoms chosen from among the following:

EVAL Informs **DWIMIFY** (and CLISP and Masterscope) that an nlambda function *does* evaluate its arguments. Can also be placed on a macro name to override the behavior of **DWIMINMACROSFLG** = **NIL**.

NOEVAL Informs **DWIMIFY** that a macro does *not* evaluate all of its arguments, even when **DWIMINMACROSFLG** = **T**.

BINDS Placed on the **INFO** property of a function or the **CAR** of a special form to inform **DWIMIFY** that the function or form binds variables. In this case, **DWIMIFY** assumes that **CADR** of the form is the variable list, i.e., a list of litatoms, or lists of the form (*VAL VALUE*). **LAMBDA**, **NLAMBDA**, **PROG**, and **RESETVARS** are handled in this fashion.

LABELS Informs **CLISPIFY** that the form interprets top-level litatoms as labels, so that **CLISPIFY** will never introduce an atom (by packing) at the top level of the expression. **PROG** is handled in this fashion.

NOFIXFNSLST [Variable]

List of functions that **DWIMIFY** will not try to correct.

NOFIXVARSLST [Variable]

List of variables that **DWIMIFY** will not try to correct.

NOSPELLFLG [Variable]

If **T**, **DWIMIFY** will not perform any spelling corrections. Initially **NIL**. **NOSPELLFLG** is reset to **T** when compiling functions whose definitions are obtained from a file, as opposed to being in core.

CLISPHELPFLG [Variable]

If **NIL**, **DWIMIFY** will not ask the user for approval of any CLISP transformations. Instead, in those situations where approval would be required, the effect is the same as though the user had been asked and said **NO**. Initially **T**.

DWIMIFYCOMPFLG [Variable]

If **T**, **DWIMIFY** is called before compiling an expression. Initially **NIL**.

DWIMCHECK#ARGSFLG [Variable]

If **T**, causes **DWIMIFY** to check for too many arguments in a form. Initially **T**.

DWIMCHECKPROGLABELSFLG [Variable]

If **T**, causes **DWIMIFY** to check whether a **PROG** label contains a CLISP character. Initially **T**.

DWIMESSGAG [Variable]

If **T**, suppresses all **DWIMIFY** error messages. Initially **NIL**.

CLISPRETRANFLG [Variable]

If **T**, informs **DWIMIFY** to (re)translate all expressions which have remote translations in the CLISP hash array. Initially **NIL**.

21.7 CLISPIFY

CLISPIFY converts Interlisp expressions to CLISP. Note that the expression given to **CLISPIFY** need *not* have originally been input as CLISP, i.e., **CLISPIFY** can be used on functions that were written before CLISP was even implemented. **CLISPIFY** is cognizant of declaration rules as well as all of the precedence rules. For example, **CLISPIFY** will convert **(IPLUS A (ITIMES B C))** into **A+B*C**, but **(ITIMES A (IPLUS B C))** into **A*(B+C)**. **CLISPIFY** handles such cases by first **DWIMIFY**ing the expression. **CLISPIFY** also knows how to handle expressions consisting of a mixture of Interlisp and CLISP, e.g., **(IPLUS A B*C)** is converted to **A+B*C**, but **(ITIMES A B+C)** to **(A*(B+C))**. **CLISPIFY** converts calls to the six basic mapping functions, **MAP**, **MAPC**, **MAPCAR**, **MAPLIST**, **MAPCONC**, and **MAPCON**, into equivalent iterative statements. It also converts certain easily recognizable internal **PROG** loops to the corresponding iterative statements. **CLISPIFY** can convert all iterative statements input in CLISP back to CLISP, regardless of how complicated the translation was, because the original CLISP is saved.

CLISPIFY is not destructive to the original Interlisp expression, i.e., **CLISPIFY** produces a new expression without changing the original. The new expression may however contain some "pieces" of the original, since **CLISPIFY** attempts to minimize the number of **CONS**es by not copying structure whenever possible.

CLISPIFY will not convert expressions appearing as arguments to **NLAMBDA** functions, except for those functions whose **INFO** property is or contains the atom **EVAL**. **CLISPIFY** also contains built in information enabling it to process special forms such as **PROG, SELECTQ**, etc. If the **INFO** property is or contains the atom **LABELS, CLISPIFY** will never create an atom (by packing) at the top level of the expression. **PROG** is handled in this fashion.

Note: Disabling a CLISP operator with **CLDISABLE** (page 21.26) will also disable the corresponding **CLISPIFY** transformation. Thus, if ← is "turned off", **A←B** will not transform to **(SETQ A B)**, nor vice versa.

(CLISPIFY *X EDITCHAIN***)** [Function]

Clispifies *X*. If *X* is an atom and *EDITCHAIN* is **NIL**, *X* is treated as the name of a function, and its definition (or **EXPR** property) is clispified. After **CLISPIFY** has finished, *X* is redefined (using **/PUTD**) with its new CLISP definition. The value of **CLISPIFY** is *X*. If *X* is atomic and not the name of a function, spelling correction is attempted. If this fails, an error is generated.

If *X* is a list, or *EDITCHAIN* is not **NIL**, *X* itself is the expression to be clispified. If *EDITCHAIN* is not **NIL**, it is the edit push-down list leading to *X* and is used to determine context as with **DWIMIFY**, as well as to obtain the local declarations, if any. The value of **CLISPIFY** is the clispified version of *X*.

(CLISPIFYFNS $FN_1 \ldots FN_N$**)** [NLambda NoSpread Function]

Like **DWIMIFYFNS** (page 21.20) except calls **CLISPIFY** instead of **DWIMIFY**.

CL:FLG [Variable]

Affects **CLISPIFY**'s handling of forms beginning with **CAR, CDR**, ... **CDDDDR**, as well as pattern match and record expressions. If **CL:FLG** is **NIL**, these are not transformed into the equivalent **:** expressions. This will prevent **CLISPIFY** from constructing any expression employing a **:** infix operator, e.g., **(CADR X)** will not be transformed to **X:2**. If **CL:FLG** is **T**, **CLISPIFY** will convert to **:** notation only when the argument is atomic or a simple list (a function name and one atomic argument). If **CL:FLG** is **ALL**, **CLISPIFY** will convert to **:** expressions whenever possible.

CL:FLG is initially **T**.

CLREMPARSFLG [Variable]

If **T**, **CLISPIFY** will remove parentheses in certain cases from simple forms, where "simple" means a function name and one or two atomic arguments. For example, **(COND ((ATOM X) --))** will **CLISPIFY** to **(IF ATOM X THEN --)**. However, if **CLREMPARSFLG** is

set to **NIL**, **CLISPIFY** will produce **(IF (ATOM X) THEN --)**. Note that regardless of the setting of this flag, the expression can be input in either form.

CLREMPARSFLG is initially **NIL**.

CLISPIFYPACKFLG [Variable]

CLISPIFYPACKFLG affects the treatment of infix operators with atomic operands. If **CLISPIFYPACKFLG** is **T**, **CLISPIFY** will pack these into single atoms, e.g., **(IPLUS A (ITIMES B C))** becomes **A + B*C**. If **CLISPIFYPACKFLG** is **NIL**, no packing is done, e.g., the above becomes **A + B * C**.

CLISPIFYPACKFLG is initially **T**.

CLISPIFYUSERFN [Variable]

If **T**, causes the function **CLISPIFYUSERFN**, which should be a function of one argument, to be called on each form (list) not otherwise recognized by **CLISPIFY**. If a non-**NIL** value is returned, it is treated as the clispified form. Initially **NIL**

Note that **CLISPIFYUSERFN** must be both set and defined to use this feature.

FUNNYATOMLST [Variable]

Suppose the user has variables named **A**, **B**, and **A*B**. If **CLISPIFY** were to convert **(ITIMES A B)** to **A*B**, **A*B** would not translate back correctly to **(ITIMES A B)**, since it would be the name of a variable, and therefore would not cause an error. The user can prevent this from happening by adding **A*B** to the list **FUNNYATOMLST**. Then, **(ITIMES A B)** would **CLISPIFY** to **A * B**.

Note that **A*B**'s appearance on **FUNNYATOMLST** would *not* enable DWIM and CLISP to decode **A*B + C** as **(IPLUS A*B C)**; **FUNNYATOMLST** is used only by **CLISPIFY**. Thus, if an identifier contains a CLISP character, it should always be separated (with spaces) from other operators. For example, if **X*** is a variable, the user should write **(SETQ X*** *FORM***)** in CLISP as **X*** ←*FORM*, not **X***←*FORM*. In general, it is best to avoid use of identifiers containing CLISP character operators as much as possible.

21.8 Miscellaneous Functions and Variables

CLISPFLG [Variable]

If **CLISPFLG** = **NIL**, disables all CLISP infix or prefix transformations (but does not affect **IF/THEN/ELSE** statements, or iterative statements).

If **CLISPFLG** = **TYPE-IN**, CLISP transformations are performed only on expressions that are typed in for evaluation, i.e., not on user programs.

If **CLISPFLG** = **T**, CLISP transformations are performed on all expressions.

The initial value for **CLISPFLG** is **T**. **CLISPIFY**ing anything will cause **CLISPFLG** to be set to **T**.

CLISPCHARS [Variable]

A list of the operators that can appear in the interior of an atom. Currently (+ - * / ↑ ~ ' = ← : < > + - ~ = @ !).

CLISPCHARRAY [Variable]

A bit table of the characters on **CLISPCHARS** used for calls to **STRPOSL** (page 4.6). **CLISPCHARRAY** is initialized by performing **(SETQ CLISPCHARRAY (MAKEBITTABLE CLISPCHARS))**.

CLISPINFIXSPLST [Variable]

A list of infix operators used for spelling correction.

CLISPARRAY [Variable]

Hash array used for storing CLISP translations. **CLISPARRAY** is checked by **FAULTEVAL** and **FAULTAPPLY** on erroneous forms before calling DWIM, and by the compiler.

(CLISPTRAN *X TRAN*) [Function]

Gives *X* the translation *TRAN* by storing (key *X*, value *TRAN*) in the hash array **CLISPARRAY**. **CLISPTRAN** is called for all CLISP translations, via a non-linked, external function call, so it can be advised.

(CLISPDEC *DECLST*) [Function]

Puts into effect the declarations in *DECLST* (see page 21.12). **CLISPDEC** performs spelling corrections on words not recognized as declarations. **CLISPDEC** is undoable.

(CLDISABLE *OP*) [Function]

Disables the CLISP operator *OP*. For example, **(CLDISABLE '-)** makes **-** be just another character. **CLDISABLE** can be used on all CLISP operators, e.g., infix operators, prefix operators, iterative statement operators, etc. **CLDISABLE** is undoable.

Note: Simply removing a character operator from **CLISPCHARS** will prevent it from being treated as a CLISP operator when it appears as part of an atom, but it will continue to be an operator when it appears as a separate atom, e.g. **(FOO + X)** vs **FOO + X**.

CLISPIFTRANFLG [Variable]

Affects handling of translations of **IF-THEN-ELSE** statements (see page 9.5). If **T**, the translations are stored elsewhere, and the (modified) CLISP retained. If **NIL**, the corresponding **COND** expression replaces the CLISP. Initially **T**.

CLISPIFYPRETTYFLG [Variable]

If non-**NIL**, causes **PRETTYPRINT** (and therefore **PP** and **MAKEFILE**) to **CLISPIFY** selected function definitions before printing them according to the following interpretations of **CLISPIFYPRETTYFLG**:

ALL	Clispify all functions.
T or **EXPRS**	Clispify all functions currently defined as **EXPR**s.
CHANGES	Clispify all functions marked as having been changed.
a list	Clispify all functions in that list.

CLISPIFYPRETTYFLG is (temporarily) reset to **T** when **MAKEFILE** is called with the option **CLISPIFY**, and reset to **CHANGES** when the file being dumped has the property **FILETYPE** value **CLISP**. **CLISPIFYPRETTYFLG** is initially **NIL**.

Note: If **CLISPIFYPRETTYFLG** is non-**NIL**, and the only transformation performed by **DWIM** are well formed **CLISP** transformations, i.e., no spelling corrections, the function will *not* be marked as changed, since it would only have to be re-clispified and re-prettyprinted when the file was written out.

(PPT *X*) [NLambda NoSpread Function]

Both a function and an edit macro for prettyprinting translations. It performs a **PP** after first resetting **PRETTYTRANFLG** to **T**, thereby causing any translations to be printed instead of the corresponding CLISP.

CLISP: [Editor Command]

Edit macro that obtains the translation of the correct expression, if any, from **CLISPARRAY**, and calls **EDITE** on it.

CL [Editor Command]

Edit macro. Replaces current expression with **CLISPIFY**ed current expression. Current expression can be an element or tail.

DW [Editor Command]

Edit macro. **DWIMIFY**s current expression, which can be an element (atom or list) or tail.

Both **CL** and **DW** can be called when the current expression is either an element or a tail and will work properly. Both consult the declarations in the function being edited, if any, and both are undoable.

(LOWERCASE *FLG***)** [Function]

If *FLG* = **T**, **LOWERCASE** makes the necessary internal modifications so that **CLISPIFY** will use lower case versions of **AND**, **OR**, **IF**, **THEN**, **ELSE**, **ELSEIF**, and all i.s. operators. This produces more readable output. Note that the user can always type in *either* upper or lower case (or a combination), regardless of the action of **LOWERCASE**. If *FLG* = **NIL**, **CLISPIFY** will use uppercase versions of **AND**, **OR**, et al. The value of **LOWERCASE** is its previous "setting". **LOWERCASE** is undoable. The initial setting for **LOWERCASE** is **T**.

21.9 CLISP Internal Conventions

CLISP is almost entirely table driven by the property lists of the corresponding infix or prefix operators. For example, much of the information used for translating the + infix operator is stored on the property list of the litatom "+". Thus it is relatively easy to add new infix or prefix operators or change old ones, simply by adding or changing selected property values. (There *is* some built in information for handling minus, :, ', and ~, i.e., the user could not himself add such "special" operators, although he can disable or redefine them.)

Global declarations operate by changing the **LISPFN** and **CLISPINFIX** properties of the appropriate operators.

CLISPTYPE [Property Name]

The property value of the property **CLISPTYPE** is the precedence number of the operator: higher values have higher precedence, i.e., are tighter. Note that the actual value is unimportant, only the value relative to other operators. For example, **CLISPTYPE** for :, ↑, and * are 14, 6, and 4 respectively. Operators with the

same precedence group left to right, e.g., / also has precedence 4, so **A/B*C** is **(A/B)*C**.

An operator can have a different left and right precedence by making the value of **CLISPTYPE** be a dotted pair of two numbers, e.g., **CLISPTYPE** of ← is **(8 . -12)**. In this case, **CAR** is the left precedence, and **CDR** the right, i.e., **CAR** is used when comparing with operators on the *left*, and **CDR** with operators on the *right*. For example, **A*B←C + D** is parsed as **A*(B←(C + D))** because the left precedence of ← is 8, which is higher than that of *, which is 4. The right precedence of ← is -12, which is lower than that of +, which is 2.

If the **CLISPTYPE** property for any operator is removed, the corresponding CLISP transformation is disabled, as well as the inverse **CLISPIFY** transformation.

UNARYOP [Property Name]

The value of property **UNARYOP** must be **T** for unary operators or brackets. The operand is always on the right, i.e., unary operators or brackets are always prefix operators.

BROADSCOPE [Property Name]

The value of property **BROADSCOPE** is **T** if the operator has lower precedence than Interlisp forms, e.g., **LT**, **EQUAL**, **AND**, etc. For example, **(FOO X AND Y)** parses as **((FOO X) AND Y)**. If the **BROADSCOPE** property were removed from the property list of **AND**, **(FOO X AND Y)** would parse as **(FOO (X AND Y))**.

LISPFN [Property Name]

The value of the property **LISPFN** is the name of the function to which the infix operator translates. For example, the value of **LISPFN** for ↑ is **EXPT**, for ' **QUOTE**, etc. If the value of the property **LISPFN** is **NIL**, the infix operator itself is also the function, e.g., **AND**, **OR**, **EQUAL**.

SETFN [Property Name]

If **FOO** has a **SETFN** property **FIE**, then **(FOO --)←X** translates to **(FIE -- X)**. For example, if the user makes **ELT** be an infix operator, e.g. #, by putting appropriate **CLISPTYPE** and **LISPFN** properties on the property list of # then he can also make # followed by ← translate to **SETA**, e.g., **X#N←Y** to **(SETA X N Y)**, by putting **SETA** on the property list of **ELT** under the property **SETFN**. Putting the list **(ELT)** on the property list of **SETA** under property **SETFN** will enable **SETA** forms to **CLISPIFY** back to **ELT**'s.

CLISPINFIX [Property Name]

The value of this property is the CLISP infix to be used in **CLISPIFY**ing. This property is stored on the property list of the corresponding Interlisp function, e.g., the value of property **CLISPINFIX** for **EXPT** is ↑, for **QUOTE** is ' etc.

CLISPWORD [Property Name]

Appears on the property list of clisp operators which can appear as **CAR** of a form, such as **FETCH**, **REPLACE**, **IF**, iterative statement operators, etc. Value of property is of the form (*KEYWORD* . *NAME*), where *NAME* is the lowercase version of the operator, and *KEYWORD* is its type, e.g. **FORWORD**, **IFWORD**, **RECORDWORD**, etc.

KEYWORD can also be the name of a function. When the atom appears as **CAR** of a form, the function is applied to the form and the result taken as the correct form. In this case, the function should either physically change the form, or call **CLISPTRAN** (page 21.25) to store the translation.

As an example, to make & be an infix character operator meaning **OR**, the user could do the following:

```
←(PUTPROP '& 'CLISPTYPE (GETPROP 'OR 'CLISPTYPE))
←(PUTPROP '& 'LISPFN 'OR)
←(PUTPROP '& 'BROADSCOPE T)
←(PUTPROP 'OR 'CLISPINFIX '&)
←(SETQ CLISPCHARS (CONS '& CLISPCHARS))
←(SETQ CLISPCHARRAY (MAKEBITTABLE CLISPCHARS))
```

[This page intentionally left blank]

[This page intentionally left blank]

22. PERFORMANCE ISSUES

This chapter describes a number of areas that often contribute to performance problems in Interlisp-D programs. Many performance problems can be improved by optimizing the use of storage, since allocating and reclaiming large amounts of storage is expensive. Another tactic that can sometimes yield performance improvements is to change the use of variable bindings on the stack to reduce variable lookup time. There are a number of tools that can be used to determine which parts of a computation cause performance bottlenecks.

22.1 Storage Allocation and Garbage Collection

As an Interlisp-D applications program runs, it creates data structures (allocated out of free storage space), manipulates them, and then discards them. If there were no way of reclaiming this space, over time the Interlisp-D memory (both the physical memory in the machine and the virtual memory stored on the disk) would fill up, and the computation would come to a halt. Actually, long before this could happen the system would probably become intolerably slow, due to "data fragmentation," which occurs when the data currently in use are spread over many virtual memory pages, so that most of the computer time must be spent swapping disk pages into physical memory. The problem of fragmentation will occur in any situation where the virtual memory is significantly larger than the real physical memory. To reduce swapping, it is desirable to keep the "working set" (the set of pages containing actively referenced data) as small as possible.

It is possible to write programs that don't generate much "garbage" data, or which recycle data, but such programs tend to be overly complicated and difficult to debug. Spending effort writing such programs defeats the whole point of using a system with automatic storage allocation. An important part of any Lisp implementation is the "garbage collector" which identifies discarded data and reclaims its space. There are several well-known approaches to garbage collection. One method is the traditional mark-and-sweep garbage collection algorithm, which identifies "garbage" data by marking all accessible data structures, and then sweeping through the data spaces to find all

unmarked objects (i.e., not referenced by any other object). Although this method is guaranteed to reclaim all garbage, it takes time proportional to the number of allocated objects, which may be very large. (Some allocated objects will have been marked during the "mark" phase, and the remainder will be collected during the "sweep" phase; so all will have to be touched in some way.) Also, the time that a mark-and-sweep garbage collection takes is independent of the amount of garbage collected; it is possible to sweep through the whole virtual memory, and only recover a small amount of garbage.

For interactive applications, it is not acceptable to have long interruptions in a computation for the purpose of garbage collection. Interlisp-D solves this problem by using a reference-counting garbage collector. With this scheme, there is a table containing counts of how many times each object is referenced. This table is incrementally updated as pointers are created and discarded, incurring a small overhead distributed over the computation as a whole. (Note: References from the stack are not counted, but are handled separately at "sweep" time; thus the vast majority of data manipulations do not cause updates to this table.) At opportune moments, the garbage collector scans this table, and reclaims all objects that are no longer accessible (have a reference count of zero). The pause while objects are reclaimed is only the time for scanning the reference count tables (small) plus time proportional to the amount of garbage that has to be collected (typically less than a second). "Opportune" times occur when a certain number of cells have been allocated or when the system has been waiting for the user to type something for long enough. The frequency of garbage collection is controlled by the functions and variables described below. For the best system performance, it is desirable to adjust these parameters for frequent, short garbage collections, which will not interrupt interactive applications for very long, and which will have the added benefit of reducing data fragmentation, keeping the working set small.

One problem with the Interlisp-D garbage collector is that not all garbage is guaranteed to be collected. Circular data structures, which point to themselves directly or indirectly, are never reclaimed, since their reference counts are always at least one. With time, this unreclaimable garbage may increase the working set to unacceptable levels. Some users have worked with the same Interlisp-D virtual memory for a very long time, but it is a good idea to occasionally save all of your functions in files, reinitialize Interlisp-D, and rebuild your system. Many users end their working day by issuing a command to rebuild their system and then leaving the machine to perform this task in their absence. If the system seems to be spending too much time swapping (an indication of fragmented working set), this procedure is definitely recommended.

Garbage collection in Interlisp-D is controlled by the following functions and variables:

(RECLAIM) [Function]

Initiates a garbage collection. Returns 0.

(RECLAIMMIN *N*) [Function]

Sets the frequency of garbage collection. Interlisp keeps track of the number of cells of any type that have been allocated; when it reaches a given number, a garbage collection occurs. If *N* is non-**NIL**, this number is set to *N*. Returns the current setting of the number.

RECLAIMWAIT [Variable]

Interlisp-D will invoke a **RECLAIM** if the system is idle and waiting for user input for **RECLAIMWAIT** seconds (currently set for 4 seconds).

(GCGAG *MESSAGE*) [Function]

Sets the behavior that occurs while a garbage collection is taking place. If *MESSAGE* is non-**NIL**, the cursor is complemented during a **RECLAIM**; if *MESSAGE* = **NIL**, nothing happens. The value of **GCGAG** is its previous setting.

(GCTRP) [Function]

Returns the number of cells until the next garbage collection, according to the **RECLAIMMIN** number.

The amount of storage allocated to different data types, how much of that storage is in use, and the amount of data fragmentation can be determined using the following function:

(STORAGE *TYPES PAGETHRESHOLD*) [Function]

STORAGE prints out a summary, for each data type, of the amount of space allocated to the data type, and how much of that space is currently in use. If *TYPES* is non-**NIL**, **STORAGE** only lists statistics for the specified types. *TYPES* can be a litatom or a list of types. If *PAGETHRESHOLD* is non-**NIL**, then **STORAGE** only lists statistics for types that have at least *PAGETHRESHOLD* pages allocated to them.

STORAGE prints out a table with the column headings **Type**, **Assigned**, **Free Items**, **In use**, and **Total alloc**. **Type** is the name of the data type. **Assigned** is how much of your virtual memory is set aside for items of this type. Currently, memory is allocated in quanta of two pages (1024 bytes). The numbers under **Assigned** show the number of pages and the total number of items that fit

on those pages. **Free Items** shows how many items are available to be allocated (using the **create** construct, page 8.3); these constitute the "free list" for that data type. **In use** shows how many items of this type are currently in use, i.e., have pointers to them and hence have not been garbage collected. If this number is higher than your program seems to warrant, you may want to look for storage leaks. The sum of **Free Items** and **In use** is always the same as the total **Assigned** items. **Total alloc** is the total number of items of this type that have ever been allocated (see **BOXCOUNT**, page 22.8).

Note: The information about the number of items of type **LISTP** is only approximate, because list cells are allocated in a special way that precludes easy computation of the number of items per page.

Note: When a data type is redeclared, the data type name is reassigned. Pages which were assigned to instances of the old data type are labeled ****DEALLOC****.

At the end of the table printout, **STORAGE** prints a "Data Spaces Summary" listing the number of pages allocated to the major data areas in the virtual address space: the space for fixed-length items (including datatypes), the space for variable-length items, and the space for litatoms. Variable-length data types such as arrays have fixed-length "headers," which is why they also appear in the printout of fixed-length data types. Thus, the line printed for the **BITMAP** data type says how many bitmaps have been allocated, but the "assigned pages" column counts only the headers, not the space used by the variable-length part of the bitmap. This summary also lists "Remaining Pages" in relation to the largest possible virtual memory, not the size of the virtual memory backing file in use. This file may fill up, causing a **STORAGE FULL** error, long before the "Remaining Pages" numbers reach zero.

STORAGE also prints out information about the sizes of the entries on the variable-length data free list. The block sizes are broken down by the value of the variable **STORAGE.ARRAYSIZES**, initially **(4 16 64 256 1024 4096 16384 NIL)**, which yields a printout of the form:

```
variable-datum free list:
le 4     26 items;   104 cells.
le 16    72 items;   783 cells.
le 64    36 items;   964 cells.
le 256   28 items;  3155 cells.
le 1024   3 items;  1175 cells.
le 4096   5 items;  8303 cells.
le 16384  3 items; 17067 cells.
others    1 items; 17559 cells.
```

This information can be useful in determining if the variable-length data space is fragmented. If most of the free space is composed of small items, then the allocator may not be able to find room for large items, and will extend the variable datum space. If this is extended too much, this could cause an **ARRAYS FULL** error, even if there is a lot of space left in little chunks.

(STORAGE.LEFT) [Function]

Provides a programmatic way of determining how much storage is left in the major data areas in the virtual address space. Returns a list of the form (*MDSFREE MDSFRAC 8MBFRAC ATOMFREE ATOMFRAC*), where the elements are interpreted as follows:

MDSFREE The number of free pages left in the main data space (which includes both fixed-length and variable-length data types).

MDSFRAC The fraction of the total possible main data space that is free.

8MBFRAC The fraction of the total main data space that is free, relative to eight megabytes.

This number is useful when using Interlisp-D on some early computers where the hardware limits the address space to eight megabytes. The function **32MBADDRESSABLE** returns non-**NIL** if the currently running Interlisp-D system can use the full 32 megabyte address space.

ATOMFREE The number of free pages left in the litatom space.

ATOMFRAC The fraction of the total litatom space that is free.

Note: Another important space resource is the amount of the virtual memory backing file in use (see **VMEMSIZE**, page 12.11). The system will crash if the virtual memory file is full, even if the address space is not exhausted.

22.2 Variable Bindings

Different implementations of lisp use different methods of accessing free variables. The binding of variables occurs when a function or a **PROG** is entered. For example, if the function **FOO** has the definition **(LAMBDA (A B)** *BODY*), the variables **A** and **B** are bound so that any reference to **A** or **B** from *BODY* or any function called from *BODY* will refer to the arguments to the function **FOO** and not to the value of **A** or **B** from a higher level function. All variable names (litatoms) have a top level value cell which is used if the variable has not been bound in any function. In discussions of variable access, it is useful to distinquish

between three types of variable access: local, special and global. Local variable access is the use of a variable that is bound within the function from which it is used. Special variable access is the use of a variable that is bound by another function. Global variable access is the use of a variable that has not been bound in any function. We will often refer to a variable all of whose accesses are local as a "local variable." Similarly, a variable all of whose accesses are global we call a "global variable."

In a "deep" bound system, a variable is bound by saving on the stack the variable's name together with a value cell which contains that variable's new value. When a variable is accessed, its value is found by searching the stack for the most recent binding (occurrence) and retrieving the value stored there. If the variable is not found on the stack, the variable's top level value cell is used.

In a "shallow" bound system, a variable is bound by saving on the stack the variable name and the variable's old value and putting the new value in the variable's top level value cell. When a variable is accessed, its value is always found in its top level value cell.

The deep binding scheme has one disadvantage: the amount of cpu time required to fetch the value of a variable depends on the stack distance between its use and its binding. The compiler can determine local variable accesses and compiles them as fetches directly from the stack. Thus this computation cost only arises in the use of variable not bound in the local frame ("free" variables). The process of finding the value of a free variable is called free variable lookup.

In a shallow bound system, the amount of cpu time required to fetch the value of a variable is constant regardless of whether the variable is local, special or global. The disadvantages of this scheme are that the actual binding of a variable takes longer (thus slowing down function call), the cells that contain the current in use values are spread throughout the space of all litatom value cells (thus increasing the working set size of functions) and context switching between processes requires unwinding and rewinding the stack (thus effectively prohibiting the use of context switching for many applications).

Interlisp-D uses deep binding, because of the working set considerations and the speed of context switching. The free variable lookup routine is microcoded, thus greatly reducing the search time. In benchmarks, the largest percentage of free variable lookup time was 20 percent of the total ellapsed time; the normal time was between 5 and 10 percent.

One consequence of Interlisp-D's deep binding scheme is that users may significantly improve performance by declaring global variables in certain situations. If a variable is declared global, the

compiler will compile an access to that variable as a retrieval of its top level value, completely bypassing a stack search. This should be done only for variables that are never bound in functions, such as global databases and flags.

Global variable declarations should be done using the **GLOBALVARS** file package command (page 17.37). Its form is **(GLOBALVARS** VAR_1 ... VAR_N**)**.

Another way of improving performance is to declare variables as local within a function. Normally, all variables bound within a function have their names put on the stack, and these names are scanned during free variable lookup. If a variable is declared to be local within a function, its name is not put on the stack, so it is not scanned during free variable lookup, which may increase the speed of lookups. The compiler can also make some other optimizations if a variable is known to be local to a function.

A variable may be declared as local within a function by including the form **(DECLARE (LOCALVARS** VAR_1 ... VAR_N**))** following the argument list in the definition of the function. Note: local variable declarations only effect the compilation of a function. Interpreted functions put all of their variable names on the stack, regardless of any declarations.

22.3 Performance Measuring

This section describes functions that gather and display statistics about a computation, such as as the elapsed time, and the number of data objects of different types allocated. **TIMEALL** and **TIME** gather statistics on the evaluation of a specified form. **BREAKDOWN** gathers statistics on individual functions called during a computation. These functions can be used to determine which parts of a computation are consuming the most resources (time, storage, etc.), and could most profitably be improved.

(TIMEALL *TIMEFORM NUMBEROFTIMES TIMEWHAT INTERPFLG* —**)** [NLambda Function]

Evaluates the form *TIMEFORM* and prints statistics on time spent in various categories (elapsed, keyboard wait, swapping time, gc) and data type allocation.

For more accurate measurement on small computations, *NUMBEROFTIMES* may be specified (its default is 1) to cause *TIMEFORM* to be executed *NUMBEROFTIMES* times. To improve the accuracy of timing open-coded operations in this case, **TIMEALL** compiles a form to execute *TIMEFORM* *NUMBEROFTIMES* times (unless *INTERPFLG* is non-**NIL**), and then times the execution of the compiled form.

Note: If **TIMEALL** is called with *NUMBEROFTIMES*>1, the dummy form is compiled with compiler optimizations on. This means that it is not meaningful to use **TIMEALL** with very simple forms that are optimized out by the compiler. For example, **(TIMEALL '(IPLUS 2 3) 1000)** will time a compiled function which simply returns the number 5, since **(IPLUS 2 3)** is optimized to the integer 5.

TIMEWHAT restricts the statistics to specific categories. It can be an atom or list of datatypes to monitor, and/or the atom **TIME** to monitor time spent. Note that ordinarily, **TIMEALL** monitors all time and datatype usage, so this argument is rarely needed.

TIMEALL returns the value of the last evaluation of *TIMEFORM*.

(TIME *TIMEX TIMEN TIMETYP***)** [NLambda Function]

TIME evaluates the form *TIMEX*, and prints out the number of **CONS** cells allocated and computation time. Garbage collection time is subtracted out. This function has been largely replaced by **TIMEALL**.

If *TIMEN* is greater than 1, *TIMEX* is executed *TIMEN* times, and **TIME** prints out (number of conses)/*TIMEN*, and (computation time)/*TIMEN*. If *TIMEN* = **NIL**, it defaults to 1. This is useful for more accurate measurement on small computations.

If *TIMETYP* is 0, **TIME** measures and prints total *real* time as well as computation time. If *TIMETYP* = 3, **TIME** measures and prints garbage collection time as well as computation time. If *TIMETYP* = **T**, **TIME** measures and prints the number of pagefaults.

TIME returns the value of the last evaluation of *TIMEX*.

(BOXCOUNT *TYPE N***)** [Function]

Returns the number of data objects of type *TYPE* allocated since this Interlisp system was created. *TYPE* can be any data type name (see **TYPENAME**, page 8.20). If *TYPE* is **NIL**, it defaults to **FIXP**. If *N* is non-**NIL**, the corresponding counter is reset to *N*.

(CONSCOUNT *N***)** [Function]

Returns the number of **CONS** cells allocated since this Interlisp system was created. If *N* is non-**NIL**, resets the counter to *N*. Equivalent to **(BOXCOUNT 'LISTP** *N***)**.

(PAGEFAULTS) [Function]

Returns the number of page faults since this Interlisp system was created.

22.3.1 BREAKDOWN

TIMEALL collects statistics for whole computations. **BREAKDOWN** is available to analyze the breakdown of computation time (or any other measureable quantity) function by function.

(BREAKDOWN FN_1 ... FN_N**)** [NLambda NoSpread Function]

The user calls **BREAKDOWN** giving it a list of function names (unevaluated). These functions are modified so that they keep track of various statistics.

To remove functions from those being monitored, simply **UNBREAK** (page 15.7) the functions, thereby restoring them to their original state. To add functions, call **BREAKDOWN** on the new functions. This will not reset the counters for any functions not on the new list. However **(BREAKDOWN)** will zero the counters of all functions being monitored.

The procedure used for measuring is such that if one function calls other and both are "broken down", then the time (or whatever quantity is being measured) spent in the inner function is *not* charged to the outer function as well.

Note: **BREAKDOWN** will *not* give accurate results if a function being measured is not returned from normally, e.g., a lower **RETFROM** (or **ERROR**) bypasses it. In this case, all of the time (or whatever quantity is being measured) between the time that function is entered and the time the next function being measured is entered will be charged to the first function.

(BRKDWNRESULTS *RETURNVALUESFLG***)** [Function]

BRKDWNRESULTS prints the analysis of the statistics requested as well as the number of calls to each function. If *RETURNVALUESFLG* is non-**NIL**, **BRKDWNRESULTS** will not to print the results, but instead return them in the form of a list of elements of the form (*FNNAME #CALLS VALUE*).

Example:

```
←(BREAKDOWN SUPERPRINT SUBPRINT COMMENT1)
(SUPERPRINT SUBPRINT COMMENT1)
←(PRETTYDEF '(SUPERPRINT) 'FOO)
FOO.;3
←(BRKDWNRESULTS)
FUNCTIONS  TIME   #CALLS  PER CALL  %
SUPERPRINT 8.261  365    0.023     20
SUBPRINT   31.910 141    0.226     76
COMMENT1   1.612  8      0.201     4
TOTAL      41.783 514    0.081
NIL
```

```
←(BRKDWNRESULTS T)
((SUPERPRINT 365 8261) (SUBPRINT 141 31910) (COMMENT1 8
1612))
```

BREAKDOWN can be used to measure other statistics, by setting the following variables:

BRKDWNTYPE [Variable]

To use **BREAKDOWN** to measure other statistics, before calling **BREAKDOWN**, set the variable **BRKDWNTYPE** to the quantity of interest, e.g., **TIME**, **CONSES**, etc, or a list of such quantities. Whenever **BREAKDOWN** is called with **BRKDWNTYPE** not **NIL**, **BREAKDOWN** performs the necessary changes to its internal state to conform to the new analysis. In particular, if this is the first time an analysis is being run with a particular statistic, a measuring function will be defined, and the compiler will be called to compile it. The functions being broken down will be redefined to call this measuring function. When **BREAKDOWN** is through initializing, it sets **BRKDWNTYPE** back to **NIL**. Subsequent calls to **BREAKDOWN** will measure the new statistic until **BRKDWNTYPE** is again set and a new **BREAKDOWN** performed.

BRKDWNTYPES [Variable]

The list **BRKDWNTYPES** contains the information used to analyze new statistics. Each entry on **BRKDWNTYPES** should be of the form (*TYPE FORM FUNCTION*), where *TYPE* is a statistic name (as would appear in **BRKDWNTYPE**), *FORM* computes the statistic, and *FUNCTION* (optional) converts the value of form to some more interesting quantity. For example, **(TIME (CLOCK 2) (LAMBDA (X) (FQUOTIENT X 1000)))** measures computation time and reports the result in seconds instead of milliseconds. **BRKDWNTYPES** currently contains entries for **TIME**, **CONSES**, **PAGEFAULTS**, **BOXES**, and **FBOXES**.

Example:

```
←(SETQ BRKDWNTYPE '(TIME CONSES))
(TIME CONSES)
←(BREAKDOWN MATCH CONSTRUCT)
(MATCH CONSTRUCT)
←(FLIP '(A B C D E F G H C Z) '(.. $1 .. #2 ..) '(.. #3 ..))
(A B D E F G H Z)
←(BRKDWNRESULTS)
FUNCTIONS TIME  #CALLS PER CALL  %
MATCH    0.036  1     0.036     54
CONSTRUCT 0.031  1     0.031     46
TOTAL    0.067  2     0.033
FUNCTIONS CONSES #CALLS PER CALL  %
```

```
MATCH     32    1    32.000   40
CONSTRUCT 49    1    49.000   60
TOTAL     81    2    40.500
NIL
```

Occasionally, a function being analyzed is sufficiently fast that the overhead involved in measuring it obscures the actual time spent in the function. If the user were using **TIME**, he would specify a value for *TIMEN* greater than 1 to give greater accuracy. A similar option is available for **BREAKDOWN**. The user can specify that a function(s) be executed a multiple number of times for each measurement, and the average value reported, by including a number in the list of functions given to **BREAKDOWN**, e.g., **BREAKDOWN(EDITCOM EDIT4F 10 EDIT4E EQP)** means normal breakdown for **EDITCOM** and **EDIT4F** but executes (the body of) **EDIT4E** and **EQP** 10 times each time they are called. Of course, the functions so measured must not cause any harmful side effects, since they are executed more than once for each call. The printout from **BRKDWNRESULTS** will look the same as though each function were run only once, except that the measurement will be more accurate.

Another way of obtaining more accurate measurement is to expand the call to the measuring function in-line. If the value of **BRKDWNCOMPFLG** is non-**NIL** (initially **NIL**), then whenever a function is broken-down, it will be redefined to call the measuring function, and then recompiled. The measuring function is expanded in-line via an appropriate macro. In addition, whenever **BRKDWNTYPE** is reset, the compiler is called for *all* functions for which **BRKDWNCOMPFLG** was set at the time they were originally broken-down, i.e. the setting of the flag at the time a function is broken-down determines whether the call to the measuring code is compiled in-line.

22.4 GAINSPACE

Users with large programs and data bases may sometimes find themselves in a situation where they needs to obtain more space, and are willing to pay the price of eliminating some or all of the context information that the various user-assistance facilities such as the programmer's assistant, file package, CLISP, etc., have accumulated during the course of his session. The function **GAINSPACE** provides an easy way to selectively throw away accumulated data:

(GAINSPACE) [Function]

Prints a list of deletable objects, allowing the user to specify at each point what should be discarded and what should be retained. For example:

```
←(GAINSPACE)
purge history lists ? Yes
purge everything, or just the properties, e.g., SIDE, LISPXPRINT,
etc. ?
just the properties
discard definitions on property lists ? Yes
discard old values of variables ? Yes
erase properties ? No
erase CLISP translations? Yes
```

GAINSPACE is driven by the list **GAINSPACEFORMS**. Each element on **GAINSPACEFORMS** is of the form (*PRECHECK MESSAGE FORM KEYLST*). If *PRECHECK*, when evaluated, returns **NIL**, **GAINSPACE** skips to the next entry. For example, the user will not be asked whether or not to purge the history list if it is not enabled. Otherwise, **ASKUSER** (page 26.12) is called with the indicated *MESSAGE* and the (optional) *KEYLST*. If the user responds **No**, i.e., **ASKUSER** returns **N**, **GAINSPACE** skips to the next entry. Otherwise, *FORM* is evaluated with the variable **RESPONSE** bound to the value of **ASKUSER**. In the above example, the *FORM* for the "**purge history lists**" question calls **ASKUSER** to ask "**purge everything, ...**" only if the user had responded **Yes**. If the user had responded with **Everything**, the second question would not have been asked.

The "**erase properties**" question is driven by a list **SMASHPROPSMENU**. Each element on this list is of the form (*MESSAGE . PROPS*). The user is prompted with *MESSAGE* (by **ASKUSER**), and if he responds **Yes**, *PROPS* is added to the list **SMASHPROPS**. The "**discard definitions on property lists**" and "**discard old values of variables**" questions also add to **SMASHPROPS**. The user will not be prompted for any entry on **SMASHPROPSMENU** for which all of the corresponding properties are already on **SMASHPROPS**. **SMASHPROPS** is initially set to the value of **SMASHPROPSLST**. This permits the user to specify in advance those properties which he always wants to be discarded, and not be asked about them subsequently. After finishing all the entries on **GAINSPACEFORMS**, **GAINSPACE** checks to see if the value of **SMASHPROPS** is non-**NIL**, and if so, does a **MAPATOMS**, i.e., looks at every atom in the system, and erases the indicated properties.

Note that the user can change or add new entries to **GAINSPACEFORMS** or **SMASHPROPSMENU**, so that **GAINSPACE**

can also be used to purge structures that the user's programs have accumulated.

22.5 Using Data Types Instead of Records

If a program uses large numbers of large data structures, there are several advantages to representing them as user data types rather than as list structures. The primary advantage is increased speed: accessing and setting the fields of a data type can be significantly faster than walking through a list with repeated **CAR**s and **CDR**s. Also, compiled code for referencing data types is usually smaller. Finally, by reducing the number of objects created (one object against many list cells), this can reduce the expense of garbage collection.

User data types are declared by using the **DATATYPE** record type (page 8.9). If a list structure has been defined using the **RECORD** record type (page 8.7), and all accessing operations are written using the record package's **fetch**, **replace**, and **create** operations, changing from **RECORD**s to **DATATYPE**s only requires editing the record declaration (using **EDITREC**, page 8.16) to replace declaration type **RECORD** by **DATATYPE**, and recompiling.

Note: There are some minor disadvantages with allocating new data types: First, there is an upper limit on the number of data types which can exist. Also, space for data types is allocated a page at a time, so each data type has at least one page assigned to it, which may be wasteful of space if there are only a few examples of a given data type. These problems should not effect most applications programs.

22.6 Using Incomplete File Names

Currently, Interlisp allows you to specify an open file by giving the file name. If the file name is incomplete (it doesn't have the device/host, directory, name, extension, and version number all supplied), the system converts it to a complete file name, by supplying defaults and searching through directories (which may be on remote file servers), and then searches the open streams for one corresponding to that file name. This file name-completion process happens whenever any I/O function is given an incomplete file name, which can cause a serious performance problem if I/O operations are done repeatedly. In general, it is much faster to convert an incomplete file name to a stream once, and use the stream from then on. For example,

suppose a file is opened with **(SETQ STRM (OPENSTREAM 'MYNAME 'INPUT))**. After doing this, **(READC 'MYNAME)** and **(READC STRM)** would both work, but **(READC 'MYNAME)** would take longer (sometimes orders of magnitude longer). This could seriously effect the performance if a program which is doing many I/O operations.

Note: At some point in the future, when multiple streams are supported to a single file, the feature of mapping file names to streams will be removed. This is yet another reason why programs should use streams as handles to open files, instead of file names.

For more information on efficiency considerations when using files, see page 24.13.

22.7 Using "Fast" and "Destructive" Functions

Among the functions used for manipulating objects of various data types, there are a number of functions which have "fast" and "destructive" versions. The user should be aware of what these functions do, and when they should be used.

"Fast" functions: By convention, a function named by prefixing an existing function name with **F** indicates that the new function is a "fast" version of the old. These usually have the same definitions as the slower versions, but they compile open and run without any "safety" error checks. For example, **FNTH** runs faster than **NTH**, however, it does not make as many checks (for lists ending with anything but **NIL**, etc). If these functions are given arguments that are not in the form that they expect, their behavior is unpredictable; they may run forever, or cause a system error. In general, the user should only use "fast" functions in code that has already been completely debugged, to speed it up.

"Destructive" functions: By convention, a function named by prefixing an existing function with **D** indicates the new function is a "destructive" version of the old one, which does not make any new structure but cannibalizes its argument(s). For example, **REMOVE** returns a copy of a list with a particular element removed, but **DREMOVE** actually changes the list structure of the list. (Unfortunately, not all destructive functions follow this naming convention: the destructive version of **APPEND** is **NCONC**.) The user should be careful when using destructive functions that they do not inadvertantly change data structures.

[This page intentionally left blank]

23. PROCESSES

The Interlisp-D Process mechanism provides an environment in which multiple Lisp processes can run in parallel. Each executes in its own stack space, but all share a global address space. The current process implementation is cooperative; i.e., process switches happen voluntarily, either when the process in control has nothing to do or when it is in a convenient place to pause. There is no preemption or guaranteed service, so you cannot run something demanding (e.g., Chat) at the same time as something that runs for long periods without yielding control. Keyboard input and network operations block with great frequency, so processes currently work best for highly interactive tasks (editing, making remote files).

In Interlisp-D, the process mechanism is already turned on, and is expected to stay on during normal operations, as some system facilities (in particular, most network operations) require it. However, under exceptional conditions, the following function can be used to turn the world off and on:

(PROCESSWORLD *FLG***)** [Function]

Starts up the process world, or if *FLG* = **OFF**, kills all processes and turns it off. Normally does not return. The environment starts out with two processes: a top-level **EVALQT** (the initial "tty" process) and the "background" process, which runs the window mouse handler and other system background tasks.

Note: **PROCESSWORLD** is intended to be called at the top level of Interlisp, not from within a program. It does not toggle some sort of switch; rather, it constructs some new processes in a new part of the stack, leaving any callers of **PROCESSWORLD** in a now inaccessible part of the stack. Calling **(PROCESSWORLD 'OFF)** is the only way the call to **PROCESSWORLD** ever returns.

(HARDRESET) [Function]

Resets the whole world, and rebuilds the stack from scratch. This is "harder" than doing **RESET** to every process, because it also resets system internal processes (such as the keyboard handler).

HARDRESET automatically turns the process world on (or resets it if it was on), unless the variable **AUTOPROCESSFLG** is **NIL**.

23.1 Creating and Destroying Processes

(ADD.PROCESS *FORM* $PROP_1$ $VALUE_1$ ... $PROP_N$ $VALUE_N$**)** [NoSpread Function]

Creates a new process evaluating *FORM*, and returns its process handle. The process's stack environment is the top level, i.e., the new process does not have access to the environment in which **ADD.PROCESS** was called; all such information must be passed as arguments in *FORM*. The process runs until *FORM* returns or the process is explicitly deleted. An untrapped error within the process also deletes the process (unless its **RESTARTABLE** property is **T**), in which case a message is printed to that effect.

The remaining arguments are alternately property names and values. Any property/value pairs acceptable to **PROCESSPROP** may be given, but the following two are directly relevant to **ADD.PROCESS**:

NAME Value should be a litatom; if not given, the process name is taken from **(CAR** *FORM***)**. **ADD.PROCESS** may pack the name with a number to make it unique. This name is solely for the convenience of manipulating processes at Lisp typein; e.g., the name can be given as the *PROC* argument to most process functions, and the name appears in menus of processes. However, programs should normally only deal in process handles, both for efficiency and to avoid the confusion that can result if two processes have the same defining form.

SUSPEND If the value is non-**NIL**, the new process is created but then immediately suspended; i.e., the process does not actually run until woken by a **WAKE.PROCESS** (below).

(PROCESSPROP *PROC PROP NEWVALUE***)** [NoSpread Function]

Used to get or set the values of certain properties of process *PROC*, in a manner analogous to **WINDOWPROP**. If *NEWVALUE* is supplied (including if it is **NIL**), property *PROP* is given that value. In all cases, returns the old value of the property. The following properties have special meaning for processes; all others are uninterpreted:

NAME Value is a litatom used for identifying the process to the user.

FORM Value is the Lisp form used to start the process (readonly).

RESTARTABLE Value is a flag indicating the disposition of the process following errors or hard resets:

NIL or **NO** (the default): If an untrapped error (or control-E or control-D) causes its form to be exited, the process is deleted. The process is also deleted if a **HARDRESET** (or control-D from **RAID**) occurs, causing the entire Process world to be reinitialized.

T or **YES**: The process is automatically restarted on errors or **HARDRESET**. This is the normal setting for persistent

"background" processes, such as the mouse process, that can safely restart themselves on errors.

HARDRESET: The process is deleted as usual if an error causes its form to be exited, but it *is* restarted on a **HARDRESET**. This setting is preferred for persistent processes for which an error is an unusual condition, one that might repeat itself if the process were simply blindly restarted.

RESTARTFORM If the value is non-**NIL**, it is the form used if the process is restarted (instead of the value of the **FORM** property). Of course, the process must also have a non-**NIL RESTARTABLE** prop for this to have any effect.

BEFOREEXIT If the value is the atom **DON'T**, it will not be interrupted by a **LOGOUT**. If **LOGOUT** is attempted before the process finishes, a message will appear saying that Interlisp is waiting for the process to finish. If you want the **LOGOUT** to proceed without waiting, you must use the process status window (from the background menu) to delete the process.

AFTEREXIT Value indicates the disposition of the process following a resumption of Lisp after some exit (**LOGOUT**, **SYSOUT**, **MAKESYS**). Possible values are:

DELETE: Delete the process.

SUSPEND: Suspend the process; i.e., do not let it run until it is explicitly woken.

An event: Cause the process to be suspended waiting for the event (page 23.7).

INFOHOOK Value is a function or form used to provide information about the process, in conjunction with the **INFO** command in the process status window (page 23.16).

WINDOW Value is a window associated with the process, the process's "main" window. Used to switch the tty process to this process when the user clicks in this window (see page 23.12).

Note: Setting the **WINDOW** property does not set the primary i/o stream (**NIL**) or the terminal i/o stream (**T**) to the window. When a process is created, i/o operations to the **NIL** or **T** stream will cause a new window to appear. **TTYDISPLAYSTREAM** (page 28.29) should be used to set the terminal i/o stream of a process to a specific window.

TTYENTRYFN Value is a function that is applied to the process when the process is made the tty process (page 23.13).

TTYEXITFN Value is a function that is applied to the process when the process ceases to be the tty process (page 23.13).

(THIS.PROCESS) [Function]

Returns the handle of the currently running process, or **NIL** if the Process world is turned off.

(DEL.PROCESS *PROC* **—)** [Function]

Deletes process *PROC*. *PROC* may be a process handle (returned by **ADD.PROCESS**), or its name. Note that if *PROC* is the currently running process, **DEL.PROCESS** does not return!

(PROCESS.RETURN *VALUE***)** [Function]

Terminates the currently running process, causing it to "return" *VALUE*. There is an implicit **PROCESS.RETURN** around the *FORM* argument given to **ADD.PROCESS**, so that normally a process can finish by simply returning; **PROCESS.RETURN** is supplied for earlier termination.

(PROCESS.RESULT *PROCESS WAITFORRESULT***)** [Function]

If *PROCESS* has terminated, returns the value, if any, that it returned. This is either the value of a **PROCESS.RETURN** or the value returned from the form given to **ADD.PROCESS**. If the process was aborted, the value is **NIL**. If *WAITFORRESULT* is true, **PROCESS.RESULT** blocks until *PROCESS* finishes, if necessary; otherwise, it returns **NIL** immediately if *PROCESS* is still running. Note that *PROCESS* must be the actual process handle returned from **ADD.PROCESS**, not a process name, as the association between handle and name disappears when the process finishes (and the process handle itself is then garbage collected if no one else has a pointer to it).

(PROCESS.FINISHEDP *PROCESS***)** [Function]

True if *PROCESS* has terminated. The value returned is an indication of how it finished: **NORMAL** or **ERROR**.

(PROCESSP *PROC***)** [Function]

True if *PROC* is the handle of an active process, i.e., one that has not yet finished.

(RELPROCESSP *PROCHANDLE***)** [Function]

True if *PROCHANDLE* is the handle of a deleted process. This is analogous to **RELSTKP**. It differs from **PROCESS.FINISHEDP** in that it never causes an error, while **PROCESS.FINISHEDP** can cause an error if its *PROC* argument is not a process at all.

(RESTART.PROCESS *PROC***)** [Function]

Unwinds *PROC* to its top level and reevaluates its form. This is effectively a **DEL.PROCESS** followed by the original **ADD.PROCESS**.

(MAP.PROCESSES *MAPFN***)** [Function]

Maps over all processes, calling *MAPFN* with three arguments: the process handle, its name, and its form.

(FIND.PROCESS *PROC ERRORFLG***)** [Function]

If *PROC* is a process handle or the name of a process, returns the process handle for it, else **NIL**. If *ERRORFLG* is **T**, generates an error if *PROC* is not, and does not name, a live process.

23.2 Process Control Constructs

(BLOCK *MSECSWAIT TIMER***)** [Function]

Yields control to the next waiting process, assuming any is ready to run. If *MSECSWAIT* is specified, it is a number of milliseconds to wait before returning, or **T**, meaning wait forever (until explicitly woken). Alternatively, *TIMER* can be given as a millisecond timer (as returned by **SETUPTIMER**, page 12.17) of an absolute time at which to wake up. In any of those cases, the process enters the *waiting* state until the time limit is up. **BLOCK** with no arguments leaves the process in the *runnable* state, i.e., it returns as soon as every other runnable process of the same priority has had a chance.

BLOCK can be aborted by interrupts such as control-D, control-E, or control-B. Note that **BLOCK** will return before its timeout is completed, if the process is woken by **WAKE.PROCESS**, **PROCESS.EVAL**, or **PROCESS.APPLY**.

(DISMISS *MSECSWAIT TIMER NOBLOCK***)** [Function]

DISMISS is used to dismiss the current process for a given period of time. Similar to **BLOCK**, except that (1) **DISMISS** is guaranteed not to return until the specified time has elapsed; (2) *MSECSWAIT* cannot be **T** to wait forever; and (3) If *NOBLOCK* is **T**, **DISMISS** will not allow other processes to run, but will busy-wait until the amount of time given has elapsed.

(WAKE.PROCESS *PROC STATUS***)** [Function]

Explicitly wakes process *PROC*, i.e., makes it *runnable*, and causes its call to **BLOCK** (or other waiting function) to return *STATUS*.

This is one simple way to notify a process of some happening; however, note that if **WAKE.PROCESS** is applied to a process more than once before the process actually gets its turn to run, it sees only the latest *STATUS*.

(SUSPEND.PROCESS *PROC***)** [Function]

Blocks process *PROC* indefinitely, i.e., *PROC* will not run until it is woken by a **WAKE.PROCESS**.

The following three functions allow access to the stack context of some other process. They require a little bit of care, and are computationally non-trivial, but they do provide a more powerful way of manipulating another process than **WAKE.PROCESS** allows.

(PROCESS.EVALV *PROC VAR***)** [Function]

Performs **(EVALV** *VAR***)** in the stack context of *PROC*.

(PROCESS.EVAL *PROC FORM WAITFORRESULT***)** [Function]

Evaluates *FORM* in the stack context of *PROC*. If *WAITFORRESULT* is true, blocks until the evaluation returns a result, else allows the current process to run in parallel with the evaluation. Any errors that occur will be in the context of *PROC*, so be careful. In particular, note that

(PROCESS.EVAL *PROC* **'(NLSETQ (FOO)))**

and

(NLSETQ (PROCESS.EVAL *PROC* **'(FOO)))**

behave quite differently if **FOO** causes an error. And it is quite permissible to intentionally cause an error in proc by performing

(PROCESS.EVAL *PROC* **'(ERROR!))**

If errors are possible and *WAITFORRESULT* is true, the caller should almost certainly make sure that *FORM* traps the errors; otherwise the caller could end up waiting forever if *FORM* unwinds back into the pre-existing stack context of *PROC*.

Note: After *FORM* is evaluated in *PROC*, the process *PROC* is woken up, even if it was running **BLOCK** or **AWAIT.EVENT**. This is necessary because an event of interest may have occurred while the process was evaluating *FORM*.

(PROCESS.APPLY *PROC FN ARGS WAITFORRESULT***)** [Function]

Performs **(APPLY** *FN ARGS***)** in the stack context of *PROC*. Note the same warnings as with **PROCESS.EVAL**.

23.3 Events

An "event" is a synchronizing primitive used to coordinate related processes, typically producers and consumers. Consumer processes can "wait" on events, and producers "notify" events.

(CREATE.EVENT *NAME***)** [Function]

Returns an instance of the **EVENT** datatype, to be used as the event argument to functions listed below. *NAME* is arbitrary, and is used for debugging or status information.

(AWAIT.EVENT *EVENT TIMEOUT TIMERP***)** [Function]

Suspends the current process until *EVENT* is notified, or until a timeout occurs. If *TIMEOUT* is **NIL**, there is no timeout. Otherwise, timeout is either a number of milliseconds to wait, or, if *TIMERP* is **T**, a millisecond timer set to expire at the desired time using **SETUPTIMER** (see page 12.16).

(NOTIFY.EVENT *EVENT ONCEONLY***)** [Function]

If there are processes waiting for *EVENT* to occur, causes those processes to be placed in the running state, with *EVENT* returned as the value from **AWAIT.EVENT**. If *ONCEONLY* is true, only runs the first process waiting for the event (this should only be done if the programmer knows that there can only be one process capable of responding to the event at once).

The meaning of an event is up to the programmer. In general, however, the notification of an event is merely a hint that something of interest to the waiting process has happened; the process should still verify that the conceptual event actually occurred. That is, *the process should be written so that it operates correctly even if woken up before the timeout and in the absence of the notified event.* In particular, the completion of **PROCESS.EVAL** and related operations in effect wakes up the process in which they were performed, since there is no secure way of knowing whether the event of interest occurred while the process was busy performing the **PROCESS.EVAL**.

There is currently one class of system-defined events, used with the network code. Each Pup and NS socket has associated with it an event that is notified when a packet arrives on the socket; the event can be obtained by calling **PUPSOCKETEVENT** (page 31.29) or **NSOCKETEVENT** (page 31.37), respectively.

23.4 Monitors

It is often the case that cooperating processes perform operations on shared structures, and some mechanism is needed to prevent more than one process from altering the structure at the same time. Some languages have a construct called a monitor, a collection of functions that access a common structure with mutual exclusion provided and enforced by the compiler via the use of monitor locks. Interlisp-D has taken this implementation notion as the basis for a mutual exclusion capability suitable for a dynamically-scoped environment.

A monitorlock is an object created by the user and associated with (e.g., stored in) some shared structure that is to be protected from simultaneous access. To access the structure, a program waits for the lock to be free, then takes ownership of the lock, accesses the structure, then releases the lock. The functions and macros below are used:

(CREATE.MONITORLOCK *NAME* —**)** [Function]

Returns an instance of the **MONITORLOCK** datatype, to be used as the lock argument to functions listed below. *NAME* is arbitrary, and is used for debugging or status information.

(WITH.MONITOR *LOCK* $FORM_1$ ... $FORM_N$**)** [Macro]

Evaluates $FORM_1$ ... $FORM_N$ while owning *LOCK*, and returns the value of $FORM_N$. This construct is implemented so that the lock is released even if the form is exited via error (currently implemented with **RESETLST**).

Ownership of a lock is dynamically scoped: if the current process already owns the lock (e.g., if the caller was itself inside a **WITH.MONITOR** for this lock), **WITH.MONITOR** does not wait for the lock to be free before evaluating $FORM_1$ ... $FORM_N$.

(WITH.FAST.MONITOR *LOCK* $FORM_1$ ... $FORM_N$**)** [Macro]

Like **WITH.MONITOR**, but implemented without the **RESETLST**. User interrupts (e.g., control-E) are inhibited during the evaluation of $FORM_1$ ... $FORM_N$.

Programming restriction: the evaluation of $FORM_1$ ... $FORM_N$ must not error (the lock would not be released). This construct is mainly useful when the forms perform a small, safe computation that never errors and need never be interrupted.

(MONITOR.AWAIT.EVENT *RELEASELOCK EVENT TIMEOUT TIMERP***)** [Function]

For use in blocking inside a monitor. Performs **(AWAIT.EVENT** *EVENT TIMEOUT TIMERP***)**, but releases *RELEASELOCK* first, and reobtains the lock (possibly waiting) on wakeup.

Typical use for **MONITOR.AWAIT.EVENT**: A function wants to perform some operation on *FOO*, but only if it is in a certain state. It has to obtain the lock on the structure to make sure that the state of the structure does not change between the time it tests the state and performs the operation. If the state turns out to be bad, it then waits for some other process to make the state good, meanwhile releasing the lock so that the other process can alter the structure.

```
(WITH.MONITOR FOO-LOCK
  (until CONDITION-OF-FOO
   do (MONITOR.AWAIT.EVENT FOO-LOCK
EVENT-FOO-CHANGED TIMEOUT))
    OPERATE-ON-FOO)
```

It is sometimes convenient for a process to have **WITH.MONITOR** at its top level and then do all its interesting waiting using **MONITOR.AWAIT.EVENT**. Not only is this often cleaner, but in the present implementation in cases where the lock is frequently accessed, it saves the **RESETLST** overhead of **WITH.MONITOR**.

Programming restriction: There must not be an **ERRORSET** between the enclosing **WITH.MONITOR** and the call to **MONITOR.AWAIT.EVENT** such that the **ERRORSET** would catch an **ERROR!** and continue inside the monitor, for the lock would not have been reobtained. (The reason for this restriction is that, although **MONITOR.AWAIT.EVENT** won't itself error, the user could have caused an error with an interrupt, or a **PROCESS.EVAL** in the context of the waiting process that produced an error.)

On rare occasions it may be useful to manipulate monitor locks directly. The following two functions are used in the implementation of **WITH.MONITOR**:

(OBTAIN.MONITORLOCK *LOCK DONTWAIT UNWINDSAVE*) [Function]

Takes possession of *LOCK*, waiting if necessary until it is free, unless *DONTWAIT* is true, in which case it returns **NIL** immediately. If *UNWINDSAVE* is true, performs a **RESETSAVE** to be unwound when the enclosing **RESETLST** exits. Returns *LOCK* if *LOCK* was successfully obtained, **T** if the current process already owned *LOCK*.

(RELEASE.MONITORLOCK *LOCK EVENIFNOTMINE*) [Function]

Releases *LOCK* if it is owned by the current process, and wakes up the next process, if any, waiting to obtain the lock.

If *EVENIFNOTMINE* is non-**NIL**, the lock is released even if it is not owned by the current process.

When a process is deleted, any locks it owns are released.

23.5 Global Resources

The biggest source of problems in the multi-processing environment is the matter of global resources. Two processes cannot both use the same global resource if there can be a process switch in the middle of their use (currently this means calls to **BLOCK**, but ultimately with a preemptive scheduler means anytime). Thus, user code should be wary of its own use of global variables, if it ever makes sense for the code to be run in more than one process at a time. "State" variables private to a process should generally be bound in that process; structures that are shared among processes (or resources used privately but expensive to duplicate per process) should be protected with monitor locks or some other form of synchronization.

Aside from user code, however, there are many *system* global variables and resources. Most of these arise historically from the single-process Interlisp-10 environment, and will eventually be changed in Interlisp-D to behave appropriately in a multi-processing environment. Some have already been changed, and are described below. Two other resources not generally thought of as global variables—the keyboard and the mouse—are particularly idosyncratic, and are discussed in the next section.

The following resources, which are global in Interlisp-10, are allocated per process in Interlisp-D: primary input and output (the streams affected by **INPUT** and **OUTPUT**), terminal input and output (the streams designated by the name **T**), the primary read table and primary terminal table, and dribble files. Thus, each process can print to its own primary output, print to the terminal, read from a different primary input, all without interfering with another process's reading and printing.

Each process begins life with its primary and terminal input/output streams set to a dummy stream. If the process attempts input or output using any of those dummy streams, e.g., by calling **(READ T)**, or **(PRINT & T)**, a tty window is automatically created for the process, and that window becomes the primary input/output and terminal input/output for the process. The default tty window is created at or near the region specified in the variable **DEFAULTTTYREGION**.

A process can, of course, call **TTYDISPLAYSTREAM** explicitly to give itself a tty window of its own choosing, in which case the automatic mechanism never comes into play. Calling **TTYDISPLAYSTREAM** when a process has no tty window not only sets the terminal streams, but also sets the primary input and output streams to be that window, assuming they were still set to the dummy streams.

(HASTTYWINDOWP *PROCESS*) [Function]

Returns **T** if the process *PROCESS* has a tty window; **NIL** otherwise. If *PROCESS* is **NIL**, it defaults to the current process.

Other system resources that are typically changed by **RESETFORM**, **RESETLST**, or **RESETVARS** are all global entities. In the multiprocessing environment, these constructs are suspect, as there is no provision for "undoing" them when a process switch occurs. For example, in the current release of Interlisp-D, it is not possible to set the print radix to 8 inside only one process, as the print radix is a global entity.

Note that **RESETFORM** and similar expressions are perfectly valid in the process world, and even quite useful, when they manipulate things strictly within one process. The process world is arranged so that deleting a process also unwinds any **RESETxxx** expressions that were performed in the process and are still waiting to be unwound, exactly as if a control-D had reset the process to the top. Additionally, there is an implicit **RESETLST** at the top of each process, so that **RESETSAVE** can be used as a way of providing "cleanup" functions for when a process is deleted. For these, the value of **RESETSTATE** (page 14.26) is **NIL** if the process finished normally, **ERROR** if it was aborted by an error, **RESET** if the process was explicitly deleted, and **HARDRESET** if the process is being restarted after a **HARDRESET** or a **RESTART.PROCESS**.

23.6 Typein and the TTY Process

There is one global resource, the keyboard, that is particularly problematic to share among processes. Consider, for example, having two processes both performing **(READ T)**. Since the keyboard input routines block while there is no input, both processes would spend most of their time blocking, and it would simply be a matter of chance which process received each character of typein.

To resolve such dilemmas, the system designates a distinguished process, termed the *tty process*, that is assumed to be the process that is involved in terminal interaction. Any typein from the keyboard goes to that process. If a process other than the tty process requests keyboard input, it blocks until it becomes the tty process. When the tty process is switched (in any of the ways described further below), any typeahead that occurred before the switch is saved and associated with the current tty process. Thus, it is always the case that keystrokes are sent to the process

that is the tty process at the time of the keystrokes, regardless of when that process actually gets around to reading them.

It is less immediately obvious how to handle keyboard interrupt characters, as their action is asynchronous and not always tied to typein. Interrupt handling is described on page 23.14.

23.6.1 Switching the TTY Process

Any process can make itself be the tty process by calling **TTY.PROCESS**.

(TTY.PROCESS *PROC***)** [Function]

Returns the handle of the current tty process. In addition, if *PROC* is non-**NIL**, makes it be the tty process. The special case of *PROC* = **T** is interpreted to mean the executive process; this is sometimes useful when a process wants to explicitly give up being the tty process.

(TTY.PROCESSP *PROC***)** [Function]

True if *PROC* is the tty process; *PROC* defaults to the running process. Thus, **(TTY.PROCESSP)** is true if the caller is the tty process.

(WAIT.FOR.TTY *MSECS NEEDWINDOW***)** [Function]

Efficiently waits until **(TTY.PROCESSP)** is true. **WAIT.FOR.TTY** is called internally by the system functions that read from the terminal; user code thus need only call it in special cases.

If *MSECS* is non-**NIL**, it is the number of milliseconds to wait before timing out. If **WAIT.FOR.TTY** times out before **(TTY.PROCESSP)** is true, it returns **NIL**, otherwise it returns **T**. If *MSECS* is **NIL**, **WAIT.FOR.TTY** will not time out.

If *NEEDWINDOW* is non-**NIL**, **WAIT.FOR.TTY** opens a TTY window for the current process if one isn't already open.

WAIT.FOR.TTY spawns a new mouse process if called under the mouse process (see **SPAWN.MOUSE**, page 23.15).

In some cases, such as in functions invoked as a result of mouse action or a user's typed-in call, it is reasonable for the function to invoke **TTY.PROCESS** itself so that it can take subsequent user type in. In other cases, however, this is too undisciplined; it is desirable to let the user designate which process typein should be directed to. This is most conveniently done by mouse action.

The system supports the model that "to type to a process, you click in its window." To cooperate with this model, any process desiring keyboard input should put its process handle as the

PROCESS property of its window(s). To handle the common case, the function **TTYDISPLAYSTREAM** does this automatically when the ttydisplaystream is switched to a new window. A process can own any number of windows; clicking in any of those windows gives the process the tty.

This mechanism suffices for most casual process writers. For example, if a process wants all its input/output interaction to occur in a particular window that it has created, it should just make that window be its tty window by calling **TTYDISPLAYSTREAM**. Thereafter, it can **PRINT** or **READ** to/from the **T** stream; if the process is not the tty process at the time that it calls **READ**, it will block until the user clicks in the window.

For those needing tighter control over the tty, the default behavior can be overridden or supplemented. The remainder of this section describes the mechanisms involved.

There is a window property **WINDOWENTRYFN** that controls whether and how to switch the tty to the process owning a window. The mouse handler, before invoking any normal **BUTTONEVENTFN**, specifically notices the case of a button going down in a window that belongs to a process (i.e., has a **PROCESS** window property) that is not the tty process. In this case, it invokes the window's **WINDOWENTRYFN** of one argument (*WINDOW*). **WINDOWENTRYFN** defaults to **GIVE.TTY.PROCESS**:

(GIVE.TTY.PROCESS *WINDOW***)** [Function]

If *WINDOW* has a **PROCESS** property, performs **(TTY.PROCESS (WINDOWPROP** *WINDOW* **'PROCESS))** and then invokes *WINDOW*'s **BUTTONEVENTFN** function (or **RIGHTBUTTONFN** if the right button is down).

There are some cases where clicking in a window does not always imply that the user wants to talk to that window. For example, clicking in a text editor window with a shift key held down means to "shift-select" some piece of text into the input buffer of the *current* tty process. The editor supports this by supplying a **WINDOWENTRYFN** that performs **GIVE.TTY.PROCESS** if no shift key is down, but goes into its shift-select mode, without changing the tty process, if a shift key is down. The shift-select mode performs a **BKSYSBUF** of the selected text when the shift key is let up, the **BKSYSBUF** feeding input to the current tty process.

Sometimes a process wants to be notified when it becomes the tty process, or stops being the tty process. To support this, there are two process properties, **TTYEXITFN** and **TTYENTRYFN**. The actions taken by **TTY.PROCESS** when it switches the tty to a new process are as follows: the former tty process's **TTYEXITFN** is called with two arguments (*OLDTTYPROCESS NEWTTYPROCESS*);

the new process is made the tty process; finally, the new tty process's **TTYENTRYFN** is called with two arguments (*NEWTTYPROCESS OLDTTYPROCESS*). Normally the **TTYENTRYFN** and **TTYEXITFN** need only their first argument, but the other process involved in the switch is supplied for completeness. In the present system, most processes want to interpret the keyboard in the same way, so it is considered the responsibility of any process that changes the keyboard interpretation to restore it to the normal state by its **TTYEXITFN**.

A window is "owned" by the last process that anyone gave as the window's **PROCESS** property. Ordinarily there is no conflict here, as processes tend to own disjoint sets of windows (though, of course, cooperating processes can certainly try to confuse each other). The only likely problem arises with that most global of windows, **PROMPTWINDOW**. Programs should not be tempted to read from **PROMPTWINDOW**. This is not usually necessary anyway, as the first attempt to read from **T** in a process that has not set its **TTYDISPLAYSTREAM** to its own window causes a tty window to be created for the process (see page 23.10).

23.6.2 Handling of Interrupts

At the time that a keyboard interrupt character (page 30.1) is struck, any process could be running, and some decision must be made as to which process to actually interrupt. To the extent that keyboard interrupts are related to typein, most interrupts are taken in the tty process; however, the following are handled specially:

RESET (initially control-D)
ERROR (initially control-E)

These interrupts are taken in the mouse process, if the mouse is not in its idle state; otherwise they are taken in the tty process. Thus, control-E can be used to abort some mouse-invoked window action, such as the Shape command. As a consequence, note that if the mouse invokes some lengthy computation that the user thinks of as "background", control-E still aborts it, even though that may not have been what the user intended. Such lengthy computations, for various reasons, should generally be performed by spawning a separate process to perform them.

The **RESET** interrupt in a process other than the executive is interpreted exactly as if an error unwound the process to its top level: if the process was designated **RESTARTABLE** = **T**, it is restarted; otherwise it is killed.

HELP (initially control-G)

A menu of processes is presented to the user, who is asked to select which one the interrupt should occur in. The current tty process appears with a * next to its name at the top of the menu. The menu also includes an entry "[Spawn Mouse]", for the common case of needing a mouse because the mouse process is

currently tied up running someone's **BUTTONEVENTFN**; selecting this entry spawns a new mouse process, and no break occurs.

BREAK (initially control-B) Performs the **HELP** interrupt in the mouse process, if the mouse is not in its idle state; otherwise it is performed in the tty process.

RUBOUT (initially DELETE) This interrupt clears typeahead in *all* processes.

RAID
STACK OVERFLOW
STORAGE FULL These interrupts always occur in whatever process was running at the time the interrupt struck. In the cases of **STACK OVERFLOW** and **STORAGE FULL**, this means that the interrupt is more likely to strike in the offending process (especially if it is a "runaway" process that is not blocking). Note, however, that this process is still not necessarily the guilty party; it could be an innocent bystander that just happened to use up the last of a resource prodigiously consumed by some other process.

23.7 Keeping the Mouse Alive

Since the window mouse handler runs in its own process, it is not available while a window's **BUTTONEVENTFN** function (or any of the other window functions invoked by mouse action) is running. This leads to two sorts of problems: (1) a long computation underneath a **BUTTONEVENTFN** deprives the user of the mouse for other purposes, and (2) code that runs as a **BUTTONEVENTFN** cannot rely on other **BUTTONEVENTFN**s running, which means that there some pieces of code that run differently from normal when run under the mouse process. These problems are addressed by the following functions:

(SPAWN.MOUSE —) [Function]

Spawns another mouse process, allowing the mouse to run even if it is currently "tied up" under the current mouse process. This function is intended mainly to be typed in at the Lisp executive when the user notices the mouse is busy.

(ALLOW.BUTTON.EVENTS) [Function]

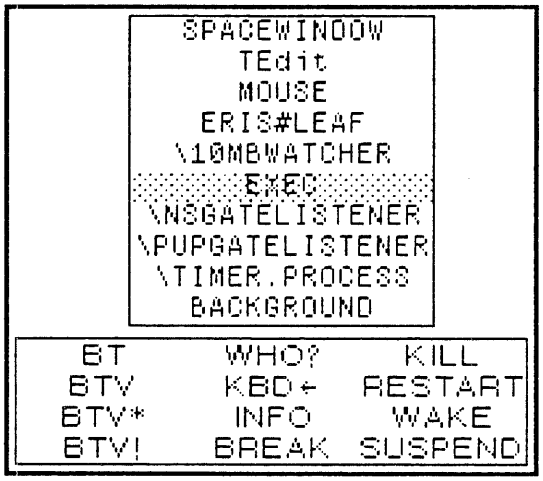
Performs a **(SPAWN.MOUSE)** only when called underneath the mouse process. This should be called (once, on entry) by any function that relies on **BUTTONEVENTFN**s for completion, if there is any possibility that the function will itself be invoked by a mouse function.

It never hurts, at least logically, to call **SPAWN.MOUSE** or **ALLOW.BUTTON.EVENTS** needlessly, as the mouse process

arranges to quietly kill itself if it returns from the user's **BUTTONEVENTFN** and finds that another mouse process has sprung up in the meantime. (There is, of course, some computational expense.)

23.8 Process Status Window

The background menu command **PSW** (page 28.6) and the function **PROCESS.STATUS.WINDOW** (below) create a "Process Status Window", that allows the user to examine and manipulate all of the existing processes:



The window consists of two menus. The top menu lists all the processes at the moment. Commands in the bottom menu operate on the process selected in the top menu (**EXEC** in the example above). The commands are:

BT, BTV, BTV*, BTV! Displays a backtrace of the selected process.

WHO? Changes the selection to the tty process, i.e., the one currently in control of the keyboard.

KBD← Associates the keyboard with the selected process; i.e., makes the selected process be the tty process.

INFO If the selected process has an **INFOHOOK** property, calls it. The hook may be a function, which is then applied to two arguments, the process and the button (**LEFT** or **MIDDLE**) used to invoke **INFO**, or a form, which is simply **EVAL**'ed. The **APPLY** or **EVAL** happens in the context of the selected process, using **PROCESS.APPLY** or **PROCESS.EVAL**. The **INFOHOOK** process property can be set using **PROCESSPROP** (page 23.2).

BREAK Enter a break under the selected process. This has the side effect of waking the process with the value returned from the break.

KILL Deletes the selected process.

RESTART Restarts the selected process.

WAKE Wakes the selected process. Prompts for a value to wake it with (see **WAKE.PROCESS**).

SUSPEND Suspends the selected process; i.e., causes it to block indefinitely (until explicitly woken).

(PROCESS.STATUS.WINDOW *WHERE*) [Function]

Puts up a process status window that provides several debugging commands for manipulating running processes. If the window is already up, **PROCESS.STATUS.WINDOW** refreshes it. If *WHERE* is a position, the window is placed in that position; otherwise, the user is prompted for a position.

Currently, the process status window runs under the mouse process, like other menus, so if the mouse is unavailable (e.g., a mouse function is performing an extensive computation), you may be unable to use the process status window (you can try **SPAWN.MOUSE**, of course).

23.9 Non-Process Compatibility

This section describes some considerations for authors of programs that ran in the old single-process Interlisp-D environment, and now want to make sure they run properly in the Multi-processing world. The biggest problem to watch out for is code that runs underneath the mouse handler. Writers of mouse handler functions should remember that in the process world the mouse handler runs in its own process, and hence (a) you cannot depend on finding information on the stack (stash it in the window instead), and (b) while your function is running, the mouse is not available (if you have any non-trivial computation to do, spawn a process to do it, notify one of your existing processes to do it, or use **PROCESS.EVAL** to run it under some other process).

The following functions are meaningful even if the process world is not on: **BLOCK** (invokes the system background routine, which includes handling the mouse); **TTY.PROCESS**, **THIS.PROCESS** (both return **NIL**); and **TTY.PROCESSP** (returns **T**, i.e., anyone is allowed to take tty input). In addition, the following two functions exist in both worlds:

(EVAL.AS.PROCESS *FORM*) [Function]

Same as **(ADD.PROCESS** *FORM* **'RESTARTABLE 'NO)**, when processes are running, **EVAL** when not. This is highly

recommended for mouse functions that perform any non-trivial activity.

(EVAL.IN.TTY.PROCESS *FORM WAITFORRESULT***)** [Function]

Same as **(PROCESS.EVAL (TTY.PROCESS)** *FORM WAITFORRESULT***)**, when processes are running, **EVAL** when not.

Most of the process functions that do not take a process argument can be called even if processes aren't running. **ADD.PROCESS** creates, but does not run, a new process (it runs when **PROCESSWORLD** is called).

[This page intentionally left blank]