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.

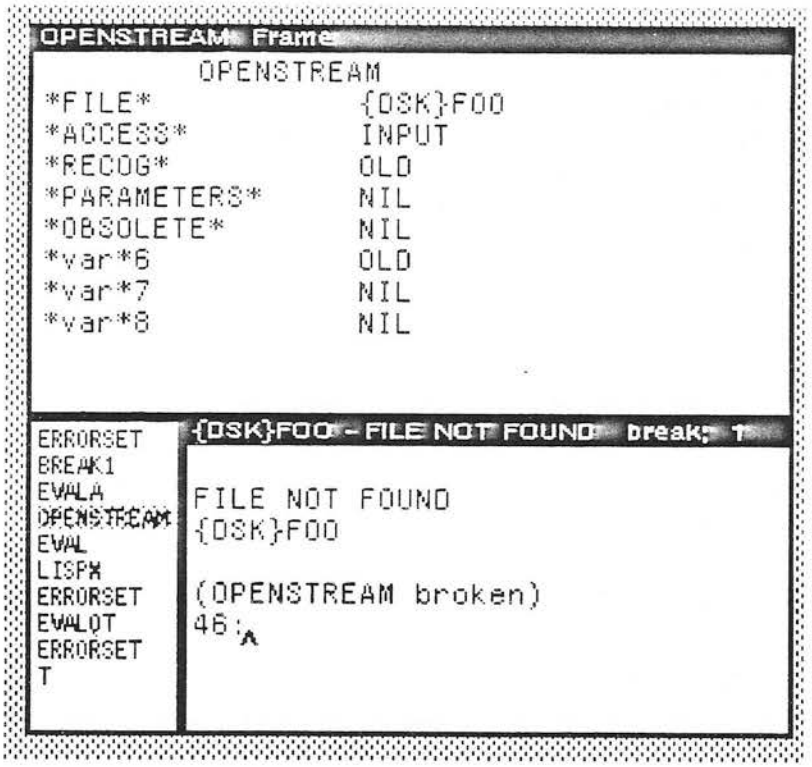
14.2 Break Windows

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 **(** $(NAME_1\ COM_{11} \ldots COM_{1n})$ $(NAME_2\ COM_{21} \ldots COM_{2n})$ **...)**. 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* **.** *ME .S2***)))**, 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.