CHAPTER 9

ERRORS AND BREAK HANDLING

Occasionally, while a program is running, an error may occur which will stop the computation. A coding mistake may have caused the wrong arguments to be passed to a function, or the programmer may have not forseen a particular unusual situation which came up, causing a function to try doing something illegal. Interlisp provides extensive facilities for detecting and handling error conditions, to enable testing, debugging,and revising of imperfect programs.

Errors can be caused in dierent ways. As mentioned above, an Interlisp primitive function may signal an error if given illegal arguments; 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 18.1; those for Interlisp- 10 on page 22.1). Finally, as an aid to debugging, the programmer can specify that certain functions should cause an error automatically whenever they are entered (see page 10.1). This allows examination of the context within the computation.

When an error occurs, the system can either[1] 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. Within a break, Interlisp oers an extensive set of ''break commands'', which assist with debugging.

This chapter explains what happens when errors occur. Breaks and break commands are given which allow the user to handle program errors. Finally, advanced facilities for modifying and extending the error mechanism are presented.

9.1   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 a ect the course of the computation. A break appears to the user like a top- level executive, except that a break uses the prompt character '':'' to indicate it is ready to accept input(s), in the same way that ''_'' is used at the top- level. 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.

Note: In Interlisp- D, the break package has been extended to include window operations (see page 20.10).

---

[1]The mechanism used for deciding whether to unwind the stack or to go into a break is described on page 9.10. The user can modify this mechanism.

Breaks

Breaks may be entered in several di erent ways. Some interrupt characters (page 9.17) automatically cause a break to be entered whenever they are typed. Functions errors may also cause a break, depending on the depth of the computation (see page 9.10). Finally, Interlisp provides functions 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 10.4).

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 even type in commands to the programmer's assistant (page 8.1), e.g. to redo or undo previously executed events, including break commands.

Similarly, the user can prettyprint functions, de ne new functions or rede ne old ones, load a le, 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 7.1), 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 ow of the computation, and the computation will not proceed without speci c 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. [2]

The basic function of the break package is `BREAK1`. Note that `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. The value returned by `BREAK1` is called ''the value of the break.'' The user can specify this value explicitly by using the `RETURN` command described below. 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,'' `BRKEXP`, which is one of the arguments to `BREAK1`. For more information on the function `BREAK1`, see page 9.11.

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 de nition 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 de nition of the broken function (see page 10.4). For functions broken with `BREAKIN`, using `BEFORE` or `AFTER`, `BRKEXP` is `NIL`. For `BREAKIN AROUND`, `BRKEXP` is the indicated expression (see page 10.5).

`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

---

[2]Except that `BREAK1` does not ''turn o '' control-D, i.e., a control-D will force an immediate return back to the top level.

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 return 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 rst 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).[3]

@ treats the rest of the teletype line as its argument(s). It rst 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) specify 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 speci ed by (the value of) FOO.

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

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

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 15.13). If the search is still unsuccessful, @ types (FN NOT FOUND), and then aborts.

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

@ can be used on BRKCOMS (see page 9.12). In this case, the *next* command on BRKCOMS is treated the same as the rest of the teletype line.

---

[3]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.

?=   [Break Command]

This is a multi-purpose command. [4] 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).[5] 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 examing 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 9.12, 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.

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
```

[4]In fact, ?= 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 17.37), and a read-macro (actually ? is the read-macro character) which does the same for the current level list being read.

[5]The value of each variable is printed with the function SHOWPRINT (page 6.17), so that if SYSPRETTYFLG= T, the value will be prettyprinted.

PB is also a programmer's assistant command (page 8.14) 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 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 6.17), 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 SUBRs, local variables, and temporaries of the interpreter, i.e. eval blips (see page 7.10).

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 functional arguments to 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 SUBRP will skip all SUBRs, BTV (LAMBDA (X) (NOT (MEMB X FOOFNS))) will skip all but those functions on FOOFNS. If used on BRKCOMS (page 9.12) 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 "ᶜʳ", is printed to delimit the output of ?= and backtrace commands. This can be reset (e.g. to ",") for more linear output.

ARGS [Break Command]
Prints the names of the variables bound at LASTPOS, i.e., (VARIABLES LASTPOS) (page 7.5). 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 rst 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 9.12). Form is (ORIGINAL . COMS). COMS are executed without regard for BREAKMACROS. Useful for rede ning a break command in terms of itself.

The following two commands are for use only with unbound atoms or unde ned 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 [AR GS].

-> EXPR   [Break Command]

Can be used in a break following either with unbound atom error, or an unde ned 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 unde ned function breaks, the user can specify a function *and* initial arguments, e.g.,

UNDEFINED CAR OF FORM

```
(MEMBERX BROKEN)
:-> MEMBER X
```

Note that in the case of a unde ned function error occurring immediately following a call to `APPLY` (e.g., `(APPLY X Y)` where the value of `X` is `FOO` and `FOO` is unde ned), 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 o ending 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 complicated in its implementation 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 simpli ed 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 rst 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-COMPUT ATION )` were handled by `EDIT`, `BIG-COMPUT ATION` 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 nd the form headed by `FOO`, and also nd *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 in any way informed `EDIT` what to do about the immediate problem, i.e., the incorrect call to `PUTD`. However, if `FOO` were *interpreted* `EDIT` would nd the `PUTD` form itself, so that when the user corrected that form, `EDIT` could use the new corrected

form to reset the break expression. The two cases are shown below:

If `FOO` is compiled:

FOO *compiled*	FOO *interpreted*
| ```
ARG NOT ATOM
(FUM)
(PUTD BROKEN)
:EDIT
IN FIE...
(FOO X)
EDIT
*(2 (CAR X))
*OK
NOTE: BRKEXP NOT CHANGED
FIE
:?=
U = (FUM)
:(SETQ U (CAR U))
FUM
:OK
PUTD
``` | ```
ARG NOT ATOM
(PUTD BROKEN)
:EDIT
IN FOO...
(PUTD X)
EDIT
*(2 (CAR X))
*OK
:OK
PUTD
``` |

IN?   [Break Command]

> Similar to `EDIT`, but just prints parent form, and superform, but does not call 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`. The following two protocol's with and without the use of `EDIT`, illustrate this:

Without `EDIT`:	*With* `EDIT`:
| ```
(FOO BROKEN)
:?=
X = (A B C)
Y = D
:BT

FOO
SETQ
COND
PROG
FIE
``` | ```
(FOO BROKEN)
:?=
X = (A B C)
Y = D
:EDIT
*(SW 2 3)
*OK
FIE 6
:OK
FOO
``` |

```
COND                nd   which function
                    FOO  is called from
                    (aborted  with  ^E)
:EDITF(FIE)
EDIT
*F FOO P
(FOO V U)           edit it
*(SW 2 3)
*OK
FIE
:(SETQ Y X)         reset X and Y
(A B C)
:(SETQQ X D)
D
:?=
X = D
Y = (A B C)         check them
:OK
FOO
```

9.2   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; su ce it to say that the parameters a ecting this decision have been adjusted empirically so that trivial type-in errors do not cause breaks, but deep errors do.

(BREAKCHECK ERR ORPOS ERXN)   [Function]
         BREAKCHECK is called by the error routine to decide whether or not to induce a break when a error occurs. ERR ORPOS 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

---

[6]X and Y have not been changed, but BRKEXP has.

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 (ERRORSETs indicate how far back unwinding is to take place when an error occurs, see page 9.15). At the same time, it counts the number of internal calls to EVAL. As soon as (if) 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 14.10). 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 e ect 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 SETQing 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.

9.3   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 interprets break commands (page 9.3).

(BREAK1 BRKEXP   BRKWHEN   BRKFN   BRK COMS   BRKTYPE   ERR ORN)   [NLambda Function]
If BRKWHEN   is NIL, BRKEXP   is evaluated and returned as the value of BREAK1. Otherwise a break occurs and commands are then taken from BRK COMS   or the terminal and interpreted. All inputs not recognized by BREAK1 are simply passed on to the programmer's assistant.

When a break occurs, if ERR ORN   is a list whose CAR is a number, ERRORMESS is called to print an identifying message. If ERR ORN   is a list whose CAR is not a number, ERRORMESS1 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 in nite 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 de ned 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 bu er. If the break returns a value (i.e., is not aborted via ^ or control-D) the input bu er will be restored.

The fourth argument to BREAK1 is BRK COMS , a list of break commands that BREAK1 interprets and executes as though they were keyboard input. One can think of BRK COMS as another input le which always has priority over the keyboard. Whenever BRK COMS = NIL, BREAK1 reads its next command from the keyboard. Whenever BRK COMS is not NIL, BREAK1 takes (CAR BRKCOMS) as its next command and sets BRK COMS 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 BRK COMS = (EVAL (PRINT X) OK), which would have the desired e ect. Note that if BRK COMS 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 10.4) uses BRK COMS : it sets up a break with two commands; the rst one prints the arguments of the function, or whatever the user speci es, 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 BRK COMS commands, BRK COMS is set to NIL, and a full interactive break occurs.

The break package has a facility for redirecting ouput to a le. All output resulting from BRK COMS will be output to the value of the variable BRKFILE, which should be the name of an open le. Output due to user typein is not a ected, 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 rst 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 BRK COMS , 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 nally, treats it as a function or variable as before. [7]

Example: The command ARGS could be de ned 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).

---

[7]If the command is not the name of a de ned 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.

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 de ned 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 o echoing, etc.), debugging them by breaking or tracing may be di cult, 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 9.19). 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 rst restores the state of the system with respect to the various expressions on BREAKRESETFORMS. When (if) 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,[8] BREAK1 again restores state. Thus the net e ect 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.

As mentioned earlier, BREAK1 detects ''Break within a break'' situations, and avoids in nite 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.

9.4   ERROR FUNCTIONS

(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

---

[8]All user type-in is scanned in order to make the operations undoable as described on page 8.22. At this point, RETFROMs and RETEVALs 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 e ects of BREAKRESETFORMS will *not* be reversed.

(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 9.10). 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 calls ERROR!.[9] 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.

(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 9.22) and EXP is the expression which was (would have
been) 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).

---

[9]unless the value of HELPFLAG is BREAK!, in which case a break will always occur (see page 9.11).

(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 N)   [Function]

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

(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.
>
> Note that 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 (described below) are more useful.
>
> The argument FLAG controls the printing of error messages if an error occurs:
>
> If FLAG= T, the error message is printed; if FLAG= NIL it is not (unless NLSETQGAG is NIL, see below). Note that if a *break* occurs below an ERRORSET, the message is printed regardless of the value of FLAG.
>
> If FLAG= INTERNAL, this ERRORSET is ignored for the purpose of deciding whether or not to break or print a message (see page 9.10). However, the ERRORSET is in eect for the purpose of ow 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. 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.

(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 eectively changes all NLSETQs to ERSETQs. NLSETQGAG is initially T.

9.5 ERROR HANDLING BY ERROR TYPE

Occasionally the user may want to treat certain types of errors di erently 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 of the form (NUM FORM $_1$ FORM $_N$), where NUM is one of the error numbers (page 9.22).   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 o ender, and the function causing the error is reentered.

Within ERRORTYPELST entries, the following variables may be useful:

ERRORMESS   [Variable]

> CAR is the error number, CADR the ''o ender'', 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 rst 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 message for various errortypes 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 *e ect* 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 15.20). If SPELLFILE is successful, the operation will be reexecuted with the new (corrected) le name.

9.6   INTERRUPT CHARACTERS

Errors and breaks can be caused by errors within functions, or by explicitly breaking a function. The user can also indicate his desire to go into a break at while a program is running by typing certain control characters known as ''interrupt characters''. The interrupt characters in Interlisp- D are listed on page 18.1; those in Interlisp- 10 are listed on page 22.1.

The user can disable and/or rede ne Interlisp interrupt characters, as well as de ne new interrupt characters. Interlisp- 10 is initialized with 9 interrupt channels: RESET (control- D), ERROR (control- E), BREAK (control- B), HELP (control- H), PRINTLEVEL (control- P), CONTROL– T (control- T), RUBOUT (del), STORAGE (control- S), and OUTPUTBUFFER (control- O). Interlisp- D does not have the STORAGE and OUTPUTBUFFER interrupt channels, and has the additional channel RAID (control- C). Each of these channels independently can be disabled, or have a new interrupt character assigned to it via the function INTERRUPTCHAR described below. In addition, the user can enable up to 9 new interrupt channels, and associate with each channel an interrupt character and an expression to be evaluated when that character is typed.

User interrupts can be either ''hard'' or ''soft''. A ''hard'' interrupt is like control- E or control- D: it takes place as soon as it is typed. A soft interrupt is like control- H; it does not occur until the next function call. Soft interrupts can always be safely continued from. Hard interrupts rip the system out of the function currently being executed and unwind back to the last function call, i.e. part of the computation that was interrupted is lost and cannot be continued.

Hard interrupts are implemented by generating error number 43, and retrieving the corresponding form from the list USERINTERRUPTS once inside of ERRORX. Soft interrupts are implemented by calling INTERRUPT with an appropriate third argument, and then obtaining the corresponding form from USERINTERRUPTS. As soon as a soft interrupt character is typed, Interlisp clears and saves the input bu ers, and then rings the bell. After the interrupt form is evaluated, the input bu ers are restored. In either case, if a character is enabled as a user interrupt, but for some reason it is not found on USERINTERRUPTS, an UNDEFINED USER INTERRUPT error will be generated.

(INTERRUPTCHAR CHAR TYP/F ORM HARDFL G)   [Function]
>De nes CHAR as an interrupt character. If CHAR was previously de ned as an interrupt character, that interpretation is disabled.
>
>CHAR is either a character or a character code (as returned by CHCON1). TENEX requires that interrupt characters be one of control- A, B,...,Z, space, esc(alt- mode), rubout(delete), or break.
>
>If TYP/F ORM = NIL, CHAR is disabled.
>
>If TYP/F ORM = T, the current state of CHAR is returned without changing or disabling it.
>
>If TYP/F ORM is one of the 8 literal atoms HELP, PRINTLEVEL, STORAGE, RUBOUT,

ERROR, RESET, OUTPUTBUFFER, or BREAK, then INTERRUPTCHAR assigns CHAR to the indicated Interlisp interrupt channel, (reenabling the channel if previously disabled).

If TYP/FORM is any other literal atom, CHAR is enabled as an interrupt character that when typed causes the atom TYP/FORM to be *immediately* set to T.

If TYP/FORM is a list, CHAR is enabled as a user interrupt character, and TYP/FORM is the form that is evaluated when CHAR is typed. The interrupt will be hard if HARDFLG = T, otherwise soft.

(INTERRUPTCHAR T) restores all Interlisp channels to their original state, and disables all user interrupts.

INTERRUPTCHAR returns an expression which, when given as an argument to INTERRUPTCHAR, will restore things as they were before the call to INTERRUPTCHAR. Therefore, INTERRUPTCHAR can be used in conjunction with RESETFORM or RESETLST (page 9.20).

INTERRUPTCHAR is undoable.

(RESET.INTERRUPTS PERMITTEDINTERRUPTS SAVECURRENT?)   [Function]
PERMITTEDINTERRUPTS is a list of interrupt character settings to be performed, each of the form (CHAR . TYP/FORM). The eect of RESET.INTERRUPTS is as if (INTERRUPTCHAR CHAR TYP/FORM) were performed for each item on PERMITTEDINTERRUPTS, and (INTERRUPTCHAR OTHER CHAR NIL) were performed on every other existing interrupt character.

If SAVECURRENT? is non-NIL, then RESET.INTERRUPTS returns the current state of the interrupts in a form that could be passed to RESET.INTERRUPTS, otherwise it returns NIL. This can be used with a RESET.INTERRUPTS that appears in a RESETFORM, so that the list is built at ''entry'', but not upon ''exit''.

(INTERRUPTABLE FLAG)   [Function]
if FLAG = NIL, turns interrupt o. If FLAG = T, turns interrupt on. Value is previous setting. INTERRUPTABLE compiles open.

Note: Any interrupt character typed while interrupts are o is treated the same as any other character, i.e. placed in the input buer, and will not cause an interrupt when interrupts are turned back on.

(INTERRUPTABLEP)   [Function]
(Interlisp-10) Returns T if interrupts are enabled; NIL if disabled.

9.7   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 ''dierent state'', e.g., dierent radix, input le, readtable, etc. but want to ''protect'' the calling environment, i.e., be able to restore the state

when the computation has completed. While program errors and control-E can be ''caught'' by errorsets, control-D is not.[10] 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 do not and cannot handle the situation where their environment is exited via anything other than a normal return, an error, or a reset. E.g. 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). This use is somewhat anachronistic in Interlisp-10 in that in a shallow bound system, it is su cient to simply rebind the variable. Furthermore, if there are any rebindings, the RESETSAVE will *not* a ect the most recent binding but will change only the top level value, and therefore probably not have the intended e ect.
>
> 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 e ect 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,

---

[10]Note that the program could rede ne control-D as a user interrupt (page 9.17), check for it, reenable it, and call RESET or something similar.

```
[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 still ''evaluated'' by *applying* its `CAR` to its `CDR`.

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 NEWV ALUE FORM)   [NLambda Function]
Simpli ed   form of `RESETLST` and `RESETSAVE` for resetting and restoring global variables.[11] Equivalent to (RESETLST (RESETSAVE VAR NEWV ALUE) 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 the variables in VARSLST are global variables. In a shallow bound system (Interlisp-10) `RESETVARS` and `PROG` are identical.[12] In a deep bound system, each variable is ''rebound'' using `RESETSAVE`.

`RESETVARS`, like `GETATOMVAL` and `SETATOMVAL` (page 2.6), is provided to permit compatibility (i.e. transportablility) between a shallow bound and deep bound system with respect to conceptually global variables.

(RESETFORM RESETF ORM   FORM $_1$ FORM $_2$ FORM $_N$)   [NLambda NoSpread Function]
Simpli ed 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 RESETF ORM) 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 di erent depending on whether the computation completed successfully or was aborted by an error or control-D. To facilitate this, while the restoration operation is being performed, the value of `RESETSTATE` will be bound to `NIL`, `ERROR`, or `RESET`,

---

[11]Unnecessarily expensive in a shallow bound system as the variable can simply be rebound.

[12]Except that the compiler insures that variables bound in a `RESETVARS` are declared as `SPECVARS` (see page 12.4).

depending on whether the exit was normal, due to an error, or reset (i.e., control-D, or in Interlisp-10, control-C followed by reenter). For example,

```
(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 le (to FL) and execute some forms, but reset the primary input le 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 explicitly 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 APPLYing AND to (RESETSTATE (INPUT OLDVALUE)) is the same as EVALing (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 8.25) 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 e ects of the computations performed under the RESETLST.

9.8   ERROR LIST

There are currently fty-plus types of errors in the Interlisp system.   Some of these errors are implementation dependent, i.e., appear in Interlisp-10 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 o ending 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 9.10).

The errors are listed below by error number:

0 - JSYS ERROR (Interlisp- 10) Occurs following a trap in a JSYS. As described on page 22.6, TRAP AT LOCATION is printed, followed by the JSYS diagnostic, and control returns to the operating system executive. The user can then safely CONTINUE, and the Interlisp error, JSYS ERROR is then generated. A TRAP AT LOCATION can also occur if an illegal instruction is executed. In this case, the operating system also prints ILLEGAL INSTRUCTION. This can happen for example if the user is programming directly in ASSEMBLE code, or if his system somehow got smashed. In the latter case, 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 abandon this system as soon as possible. (If the user does elect to CONTINUE, Interlisp will (try to) generate a JSYS ERROR and unwind. In some cases, however, the system may be so badly smashed that the error message won't even print.) Note that in some cases, e.g. illegal instruction trap while in the garbage collector, Interlisp will print out CAN'T CONTINUE, because traps under those conditions are fatal. The user *may* be able to reenter his sytem via the START command, and, if lucky, dump some data or functions before the system totally collapses.

In Interlisp- D, this error is named SYSTEM ERROR.

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.

In Interlisp- 10, the garbage collector uses the same stack as the rest of the system, so that if a garbage collection occurs when deep in a computation, the stack can over ow (particularly if there is a lot of list structure that is deep in the CAR direction). If this does happen, the garbage collector will ush the stack used by the computation in order that the garbage collection can complete. Afterwards, the error message STACK OVERFLOW IN GC - COMPUTATION LOST is printed, followed by a (RESET), i.e., return to top level.

3 - ILLEGAL RETURN
Call to RETURN when not inside of an interpreted PROG.

4 - ARG NOT LIST E.g., RPLACA called on a non- list.

5 - HARD DISK ERROR
(Interlisp- D) 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 INFILE or OUTFILE, page 6.2.

10 - NON-NUMERIC ARG

> A numeric function e.g., IPLUS, ITIMES, IGREATERP, expected a number.

11 - ATOM TOO LONG

> Attempted to create a litatom (via PACK, or typing one in, or reading from a le) with too many characters. In Interlisp- D, the maximum number of characters in a litatom is 255. In Interlisp- 10, the maximum is 127 characters.

12 - ATOM HASH TABLE FULL

> No room for any more (new) atoms.
>
> In Interlisp- 10, the atom hash table will automatically expand by a speci ed number of pages each time it lls up until an upper limit of 32K atoms is reached.

13 - FILE NOT OPEN

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

14 - ARG NOT LITATOM

> E.g., SETQ, PUTPROP, GETTOPVAL, etc., given a non- atomic arg.

15 - TOO MANY FILES OPEN

> 30, excluding the terminal.

16 - END OF FILE   From an input function, e.g., READ, READC, RATOM. After the error, the le will then be closed.

> Note: The entries on ERRORTYPELST (page 9.16) are processed before the le is closed, so that the user can intercept and process this error via an entry on ERRORTYPELST, thereby preventing the le from being closed. It is also possible to use an ERRORTYPELST entry 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 9.14).

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 speci ed a stack position with a function name, and that function was not found

on the stack. See page 7.1.

20 - FAULT IN EVAL

Artifact of bootstrap. Never occurs after FAULTEVAL has been de ned as described earlier.

21 - ARRAYS FULL   System will rst initiate a garbage collection of array space, and if no array space is reclaimed, will then generate this error.

22 - FILE SYSTEM RESOURCES EXCEEDED

(Interlisp- 10) Includes no more disk space, disk quota exceeded, directory full, too many jfbs, job full.

23 - FILE NOT FOUND

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

Interlisp is initialized with an entry on ERRORTYPELST (page 9.16) to call SPELLFILE for error 23. 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 le is not a sysout le at all (see page 14.3).

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 2.35.

27 - ILLEGAL ARG   Catch- all error.   Currently used by PUTD, EVALA, ARG, FUNARG, ALLOCATE, RPLSTRING, etc.

28 - ARG NOT ARRAY

ELT or SETA given an argument that is not a pointer to the beginning of an array (see page 2.33).

29 - ILLEGAL OR IMPOSSIBLE BLOCK

(Interlisp- 10) From GETBLK or RELBLK (see page 22.20).

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 7.1).

31 - STORAGE FULL

Following a garbage collection, if a su cient   amount of words has not 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 eld of a user data type is changed, the type of the item is rst checked

to be sure that it is of the expected type. If not, this error is generated (see page 3.14).

33 - ILLEGAL DATA TYPE NUMBER

> The argument is not a valid user data type number (see page 3.14).

34 - DATA TYPES FULL

> All available user data types have been allocated. (see page 3.14).

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 9 user channels are currently enabled (see page 9.17).

37 - READ-MACRO CONTEXT ERROR

> (Interlisp-10) Occurs when a READ is executed from within a read-macro function and the next token is a) or a] (see page 6.36).

38 - ILLEGAL READTABLE

> The argument was expected to be a valid readtable (see page 6.32).

39 - ILLEGAL TERMINAL TABLE

> The argument was expected to be a valid terminal table (see page 6.40).

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. See SETSBSIZE, page 22.26.

41 - PROTECTION VIOLATION

> (Interlisp-10) 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. in Interlisp-10, two ;'s etc.

43 - USER BREAK   Error corresponding to ''hard'' user-interrupt character. See page 9.17.

44 - UNBOUND ATOM

> Unbound atom error. When this occurs, a variable (atom) 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 atom. 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. When is occurs, a form was 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  AR GS )
```

47 - `CONTROL-E`   The user typed Control- E.

48 - `FLOATING  UNDERFLOW`
                (Interlisp- D) Under ow   during   oating- point operation.

49 - `FLOATING  OVERFLOW`
                (Interlisp- D) Over ow   during   oating- point operation.

50 - `OVERFLOW`   (Interlisp- D) Over ow   during integer operation.

51 - `ARG  NOT  HARRAY`
                (Interlisp- D) Signaled by hash array operations when given an argument that is not
                a hash array. (In Interlisp- 10, this still triggers error 28, `ARG  NOT  ARRAY`).

52 - `TOO  MANY  ARGUMENTS`
                (Interlisp- D) Signaled when too many arguments   are given to a lambda- spread,
                lambda- nospread, or nlambda- spread function.

In addition, many system functions, e.g., `DEFINE`, `ARGLIST`, `ADVISE`, `LOG`, `EXPT`, etc, also generate
errors with appropriate   messages by calling `ERROR` (see page 9.14) which causes error number 17.