F.G.H.
3/14/85

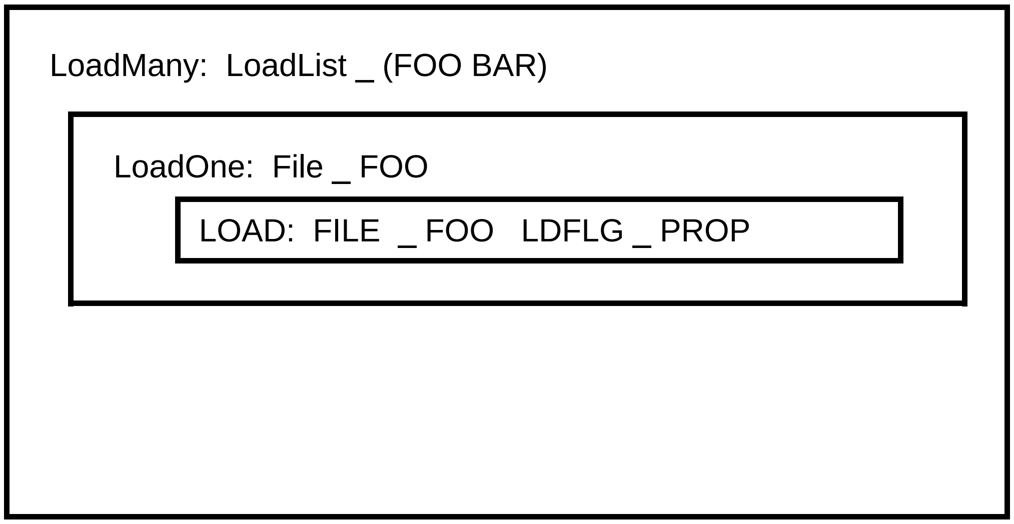
LispCourse #13: Error Handling and the Break Package

Background Concept: The Stack

Consider the following set of functions:

```
(DEFINEQ
    (LoadMany (LAMBDA (LoadList)
        (COND
            (LoadList
                (LoadOne (CAR LoadList))
                (LoadMany (CDR LoadList))))))
    (LoadOne (LAMBDA (File)
        (LOAD File ’PROP))))
```

Trace the following function call down to the first call to LOAD: (LoadMany ’(FOO BAR))



When the Lisp evaluator starts to process the call to LOAD, it is keeping track of three function calls: the call to LoadMany, the call to LoadOne, and the call to LOAD.

The Lisp evaluator needs to keep track of lots of information about these three function calls. For example for each function call the evaluator must record the exact function called, the values of the arguments in the function call, where in the function body it currently is, and so on.

Lisp keeps this kind of information in a data structure called **the stack**.

The stack consists a set of stack frames. Each stack frame contains the information about one function call. The stack frames are ordered from most recent at the top of the stack to oldest at the bottom of the stack.

When a new function call is made, a new frame is added to the top of the stack. When a function call completes, its frame is removed from the top of the stack. Thus the stack grows and shrinks as function calls are made and then completed.

For example, at the start of the LOAD function call the Lisp stack would look something like:

```
LOAD:
        FILE _ FOO
        LDFLG _ PROP
LoadOne:
        File _ FOO
LoadMany:
        LoadList _ (FOO BAR)
```

After FOO is loaded and LoadOne completes, the Lisp stack would be:

```
LoadMany:
        LoadList _ (FOO BAR)
```

LoadMany then recurses to load BAR, so at the point that LOAD is called again the stack would be:

```
LOAD:
        FILE _ BAR
        LDFLG _ PROP
LoadOne:
        File _ BAR
LoadMany:
        LoadList _ (BAR)
LoadMany:
        LoadList _ (FOO BAR)
```

Note that at each point, the stack contains an ordered list of the function calls that are in progress. Each function call on the stack is waiting for function call just above it to complete and return a value. The top of the stack contains information about the function call currently being processed.

In short, at any given time the stack contains a record of the current state of a computation in progress.

When an error occurs, the stack contains valuable information about the state the computation was in when the error occured.

Errors: Break or Abort?

When the Lisp evaluator encounters an error that DWIM/CLISP cannot "correct", it does one of two things:

1) it **aborts** the computation in progress. Unless the aborted program specifies otherwise, control returns to the Lisp exec. (Programs can be written so that they "handle" the abort by "fixing' or ignoring the error.)

2) it suspends the computation in progress and enters a **break.** From the break, the user can "fix" the error and resume (or abort) the computation.

The decision whether to Abort or Break is based on the heuristic that complex computations should be broken while simple computations should be aborted. This heuristic rests on the assumption that simple computations are easier redone rather than fixed and resumed.

"Complex" in this case is based on two factors, "depth" of the computation and time spent so far in the computation.

Depth is basically the number of number of function calls on the stack when the error occurs. If this number passes a given threshold (i.e., the value of HELPDEPTH), the computation will be broken.

Time is measured by how long the computation has been in progress. If this time passes a given threshold (i.e., the value of HELPTIME), then the computation will be broken.

If neither the depth nor the time passes threshold, then the computation will be aborted.

Some programs choose to handle their own aborts using the error processing machinery provided by Interlisp. The computation of "complexity" for these programs is slightly different. The effect is that breaks are less likely to occur and aborts are more likely to occur. When the abort does occur it is up to the program to decide what to do.

> When an error causes an abort, the program can try to "fix" the error or decide to ignore it. If it does either of these, the computation will proceed as if no error had occurred (though the program will probably print out some sort of error message to let you know that the error did occur.) The program can also decide to force a break or it can force a "true" abort and return to the Lisp exec.
>
> Standard packages like TEdit and DEdit almost always process their own aborts. They try to ignore or fix all but the most drastic errors so that the user isn't constantly bombarded by breaks and/or aborts.

When a computation is aborted, an error message is always printed in its TTY window before control is given to the Lisp exec. This message usually includes the type of error and the offending object. For example: "Non-numeric arg: NIL" indicates that one of the arguments to some function was NIL when it was to supposed to be some numeric value.

There are some 50 standard types of error in Interlisp. We will review some of these later.

Examples:

```
12_(PLUS T)

NON-NUMERIC ARG
T

13_(QQQQQ 4)

UNDEFINED CAR OF FORM
QQQQQ

14_(SETQ NIL 5)

ATTEMPT TO SET NIL OR T
5
```

Breaks: the Break Exec and Break Windows

When a Break occurs, the computation is suspended and a *Break window* is opened on the screen. The title of this window is a message describing the error that caused the break. This error message is also printed in the window.

A *Break exec* is started in the Break window. A Break exec is very similar to the Lisp exec except it has an additional set of commands that can be used to "handle" the break.

The Break package is primarily intended as a debugging tool for programmers. Rather than aborting a computation when an error occurs, Interlisp suspends the computation and enters a break. The break gives the user an opportunity to examine the suspended computation, make changes that "fix" the error, and then restart the computation from any point. From a programmer’s point of view this is a fantastic opportunity.

Unfortunately, to make full use of breaks you need to have considerable expertise in Interlisp programming. The average user doesn’t have sufficient expertise and therefore can’t really take advantage of the power of breaks.

But, when an error occurs a break usually occurs. So the all users have to learn to deal with breaks as best they can. We will cover "dealing with breaks" today and pick up the "effective use of breaks" later in the programming part of the course.

What is a break and what is it good for?

When a break occurs the ongoing computation is suspended at the point at which the error occurred. The break provides the tools for examining the state of the computation including its stack, all of the variables its using, the definitions of the functions it calls, etc. Functions can be edited, the values of variables reset, previous events undone, and so on. The computation can then be restarted at any point or it can be aborted completely.

Basically, a break is an opportunity for the user to do what she can to recover from errors, thereby saving any work that has been done by the computaion before the error occurred. The decision to abort the computation if the work can’t be saved or isn’t worth savingis left in the hands of the user rather than the system.

Examples:

If during a COPYFILE you run out of space on your account on the file server, the system will enter a break with a message saying "File system resources exceeded". From the break, you can delete some files on your account to free up some space. You can then type "OK" to restart the COPYFILE. If you can’t find any files to delete and just want to stop the COPYFILE, you can type "^" to abort the COPYFILE and return to the Lisp exec.

If you type "(LOAD ’FOOBAR)" to the Lisp exec and FOOBAR doesn’t exist, the system will (sometimes) enter a break with saying "File not found FOOBAR".   If you really meant to load FUZBAL, you can type "(SETQ FILE ’FUZBAL)" followed by "OK". This will "correct" the typo you made and restart the computation, causing the file FUZBAL to be loaded.

The Break Exec

The Break exec running inside the Break window is a read-eval-print loop just like the Lisp exec. The type-in prompt is a ":" rather than a "_" to distinguish the Break exec from the Lisp exec.

The history event numbers that preceed the prompt are shared between the Lisp exec and the Break exec, i.e., there is only one history list that serves all execs, Lisp and Break alike.

The Break exec contains all of the functionality of the Lisp exec including TTYIN and the Programmer’s Assistant. You can evaluate any Lisp expression, CLISP expression, P.A. command, etc. just as you can in the Lisp exec.

The Break exec contains a number of additional commands for managing the Break, for repairing the computation, and for restarting/aborting the computation. These commands are like the P.A. commands in that you simply type them into exec followed by a <RETURN>. No parentheses etc. are necessary.

BRKEXP: When you enter a break, the value of the variable BRKEXP is the S-expression that caused the error in the evaluator. If you can’t figure out what to do from the error message printed in the break window, you can always examine the value of BRKEXP.

BRKEXP can also be used as the expression to evaluate when leaving the break. Several of the break exec commands involve modifying or re-evaluating the BRKEXP expression.

Example:

In the LOAD example above, the value of BRKEXP was *(OPENFILE FILE ACCESS RECOG BYTESIZE)*. This suggests that the variable *FILE* is the "cause" of the error and should be set to the correct file name. Hence to fix the error, you SETQ *FILE* to the correct name. You then use the OK break command to exit the break and reevaluate the BRKEXP expression.

```
41_(LOAD 'FOOBAR)
FILE NOT FOUND
FOOBAR
(OPENSTREAM broken)
42:BRKEXP
(OPENFILE FILE ACCESS RECOG BYTESIZE)
43:FILE
FOOBAR
44:(SETQ FILE 'FUZBAL]
FUZBAL
45:FILE
FUZBAL
46:OK
{DSK}FUZBAL
47_
```

Break Exec Commands: The following are special commands that can be used to manage breaks from the break exec.

Exiting from a break:

When exiting from a break, you return some value. This is used in by the evaluator as the value of the expression that caused the error. For example, the expression *(SETQ A (PLUS 1 T))* will cause a "non-numeric arg" break while evaluating the *(PLUS 1 T)* expression. The value returned from this break is used in place of the value of the *(PLUS 1 T)* expression and hence is the value that A will be set to.

The following functions exit from a break, returning some value.

GO ž Evaluates the BRKEXP expression, prints the result in the break window, and exits the break returning the evaluation result.

OK ž Like GO, except doesn’t print the result of the evaluation.

RETURN *Form* ž Evaluates *Form* and then exits from the break returning the evaluation result.

The following function, exit from a break by aborting and hence doesn’t return any value:

^ ž exits the break by causing an abort. This will cause a return to the Lisp exec unless aborts are explicitly processed by the broken program as described earlier.

Other Commands:

Many of the break exec commands rely on the value of the variable LASTPOS. LASTPOS points to an entry on the stack, i.e., to a function call somewhere on the stack. Many functions allow you to see various characteristics of the LASTPOS function call. You can move the value of LASTPOS in order to examine various function calls on the stack. LASTPOS starts out at the expression causing the break.

EVAL ž evaluates the BRKEXP expression but does not exit from the break. !VALUE is set to the result of the evaluation. Can be used to see what the break would return, if GO or OK were used.

@ Atom ž moves LASTPOS to the most recent call to the function named by *Atom.* To search for the second most recent call to the function named by atom use "@ *Atom Atom*" and so on.

?= ž prints out the values of the arguments to the function at LASTPOS. For example, in the LOAD break described above with LASTPOS pointing to the OPENFILE function call, ?= would print out the values of FILE, ACCESS, RECOG and BYTESIZE.

BT DUMMYFRAMEPž prints out the names of the function calls on the stack starting from the most recent to the oldest (called a *backtrace*). The DUMMYFRAMEP is optional, but gets a more concise backtrace with less junk in it.

REVERT ž removes from the stack all function calls from the error back through the function call specified by LASTPOS. It then causes a break in the function call specified by LASTPOS. Thus REVERT allows you to back the computation up to some previous point before the error was encountered.

EDIT ž calls DEdit on the function containing the BRKEXP expression. For example: if in the definition of the function FOO there is an expression like *(PLUS 1 T),* a break will occur with the value of BRKEXP being *(PLUS 1 T)*. In this break, EDIT will call DEdit on the function FOO because FOO is the function that contains the incorrect statement *(PLUS 1 T).*

Examples of Break Exec in use

```
99_(COPYFILE (QUOTE {DSK2}ABC)
        (QUOTE {DSK2}XYZ))

FILE NOT FOUND
{DSK2}ABC

(OPENSTREAM broken)
100:BRKEXP
(OPENSTREAM FILE ACCESS RECOG BYTESIZE PARAMETERS)
```

```
1:BT DUMMYFRAMEP
OPENSTREAM
ERRORSET
COPYFILE
EVAL
LISPX
ERRORSET
EVALQT
ERRORSET
T
2:?=
*FILE* = {DSK2}ABC
*ACCESS* = INPUT
*RECOG* = NIL
*BYTESIZE* = NIL
*PARAMETERS* = ((SEQUENTIAL T))
3:(COPYFILE '{DSK2}AAA '{DSK2}ABC]
{DSK2}ABC.;1
4:OK
OPENSTREAM
{DSK2}XYZ.;1
5_
```

```
64_(DEFINEQ
        (EXAMPLE (LAMBDA (A B C)
                (PLUS
                        (EXAMPLEX A)
                        (EXAMPLEX B)
                        (EXAMPLEX C)))))
(EXAMPLE)
65_(DEFINEQ
        (EXAMPLEX (LAMBDA (X)
                (EXAMPLEY X 1))))
```

```
(EXAMPLEX)
66_(DEFINEQ
        (EXAMPLEY (LAMBDA (P Q)
                        (PLUS P Q))))
(EXAMPLEY)
67_(EXAMPLE 2 3 4)
12
...
77_(EXAMPLE 1 2 T)

NON-NUMERIC ARG
T

(PLUS broken)
80:BT DUMMYFRAMEP
PLUS
EXAMPLEY
EXAMPLEX
EXAMPLE
EVAL
LISPX
ERRORSET
EVALQT
ERRORSET
T

81:?=
*ARG1* = T
*ARG2* = 1
82:@ EXAMPLEY
EXAMPLEY
83:?=
P = T
Q = 1
84:@ EXAMPLEX
EXAMPLEX
85:?=
X = T
86:REVERT
EXAMPLEX

(EXAMPLEX broken)
87: BT DUMMYFRAMEP
EXAMPLEX
EXAMPLE
EVAL
LISPX
ERRORSET
EVALQT
ERRORSET
T

88:?=
X = T
89:X
T
90:(SETQ X 55)
55
91:X
55
92:BRKEXP
(PROGN (* fgh: "13-Mar-85 22:54") (EXAMPLEY X 1))
93:EVAL
EXAMPLEX evaluated
```

```
94:!VALUE
56
95:OK
EXAMPLEX
61
96_
```

Break Windows

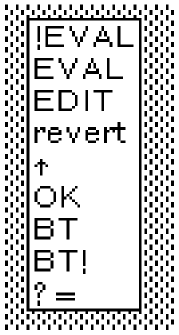
When a break occurs, the system opens a break window and starts up the break exec within this window. All interaction with the break exec takes place in this window, just as all interaction with the Lisp exec takes place in the TTY window.

Break windows provide two other functions besides exec type-in:

1. menu-based access to many of the break exec commands
2. visual displays of the stack and stack frames

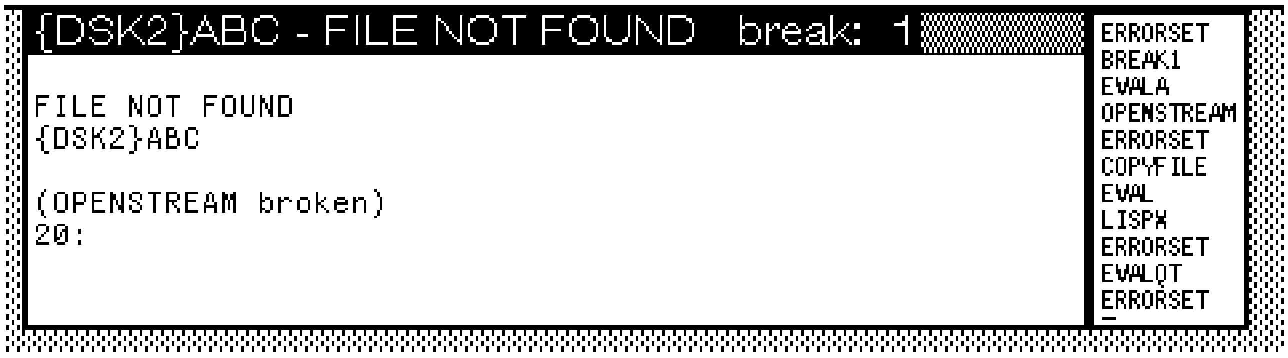
Menu-based access to the break exec:

Clicking on the middle mouse button anywhere in the break window will bring up a menu consisting of 9 items: *!EVAL, EVAL, EDIT, revert,* ^, *OK, BT, BT!, ?=*.



Choosing *EVAL, EDIT, revert,* ^, *OK, or ?=* from the menu simply carries out the corresponding break exec command. For example, choosing the *OK* entry will exit the break after evaluating the BRKEXP expression.

Choosing the *BT* command from this menu will bring up a small window (called the *backtrace window*) attached to the left or right edge of the break window. The stack backtrace will be printed in this window, one item (i.e., function call) per line. The window is scrollable if the whole backtrace doesn't fit in the window.



```
{DSK2}ABC - FILE NOT FOUND   break: 1

FILE NOT FOUND
{DSK2}ABC

(OPENSTREAM broken)
20:
```

```
ERRORSET
BREAK1
EVALA
OPENSTREAM
ERRORSET
COPYFILE
EVAL
LISPX
ERRORSET
EVALQT
ERRORSET
```

The *!EVAL* and *BT!* are simply specialized versions of the *EVAL* and *BT* commands.

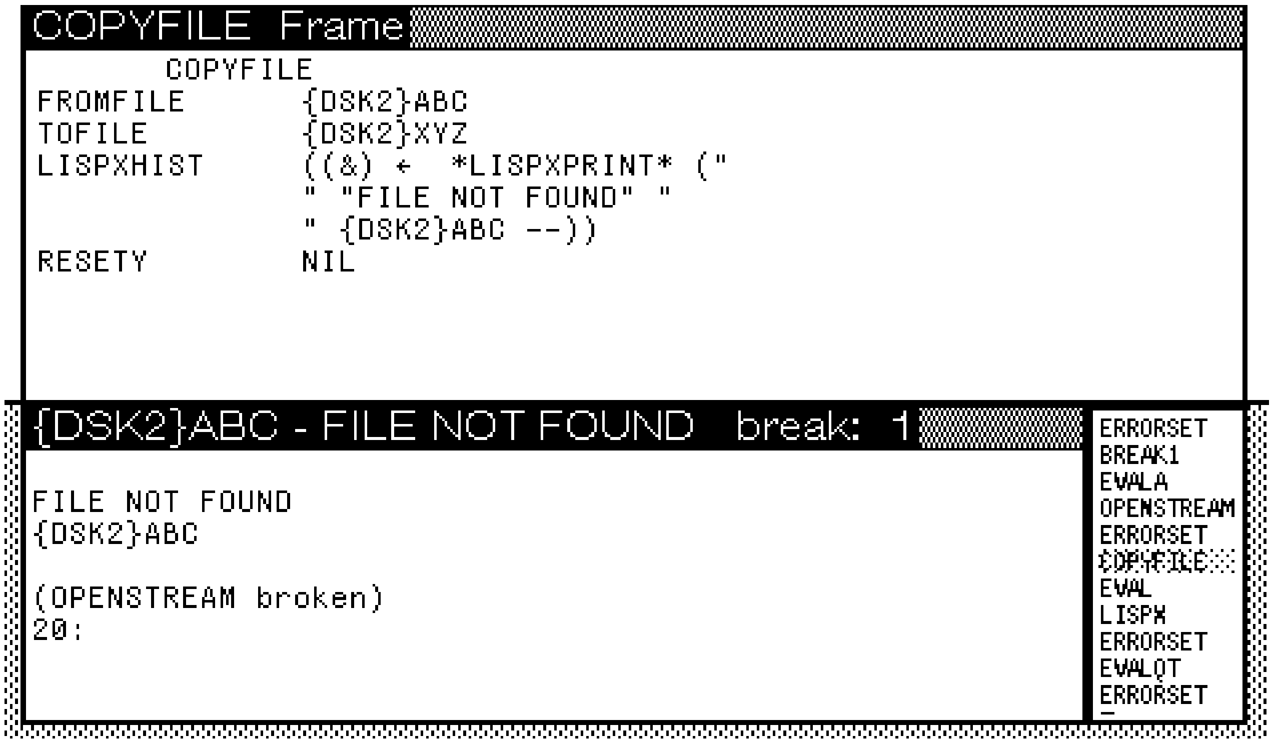
The backtrace window and the frame inspector:

Each item (i.e., function call) in the backtrace window is individually selectable by clicking on the left or middle mouse buttons while the cursor is over the item. The selected item will be highlighted with a gray background.

Selecting an item in the backtrace window has two effects:

1. LASTPOS is set to the corresponding stack entry. This is the menu-based equivalent of the @ command in the break exec.

2. a stack frame inspector is opened on the corresponding stack entry. This stack frame inspector contains the name of the function call and the functions arguments and their values.


```
COPYFILE Frame
        COPYFILE
FROMFILE      {DSK2}ABC
TOFILE        {DSK2}XYZ
LISPXHIST     ((&) ←  *LISPXPRINT* ("
              " "FILE NOT FOUND" "
              " {DSK2}ABC --))
RESETY        NIL
```

```
{DSK2}ABC - FILE NOT FOUND   break: 1

FILE NOT FOUND
{DSK2}ABC

(OPENSTREAM broken)
20:
```

The stack frame inspector is a standard inspector window. It has lots of functionality which will not be covered here. However, there are several operations in the inspector that may be useful. In particular:

To call DEdit the function in the frame:

First select the function name at the top of the inspector window by placing the cursor over the name and clicking the left mouse button.

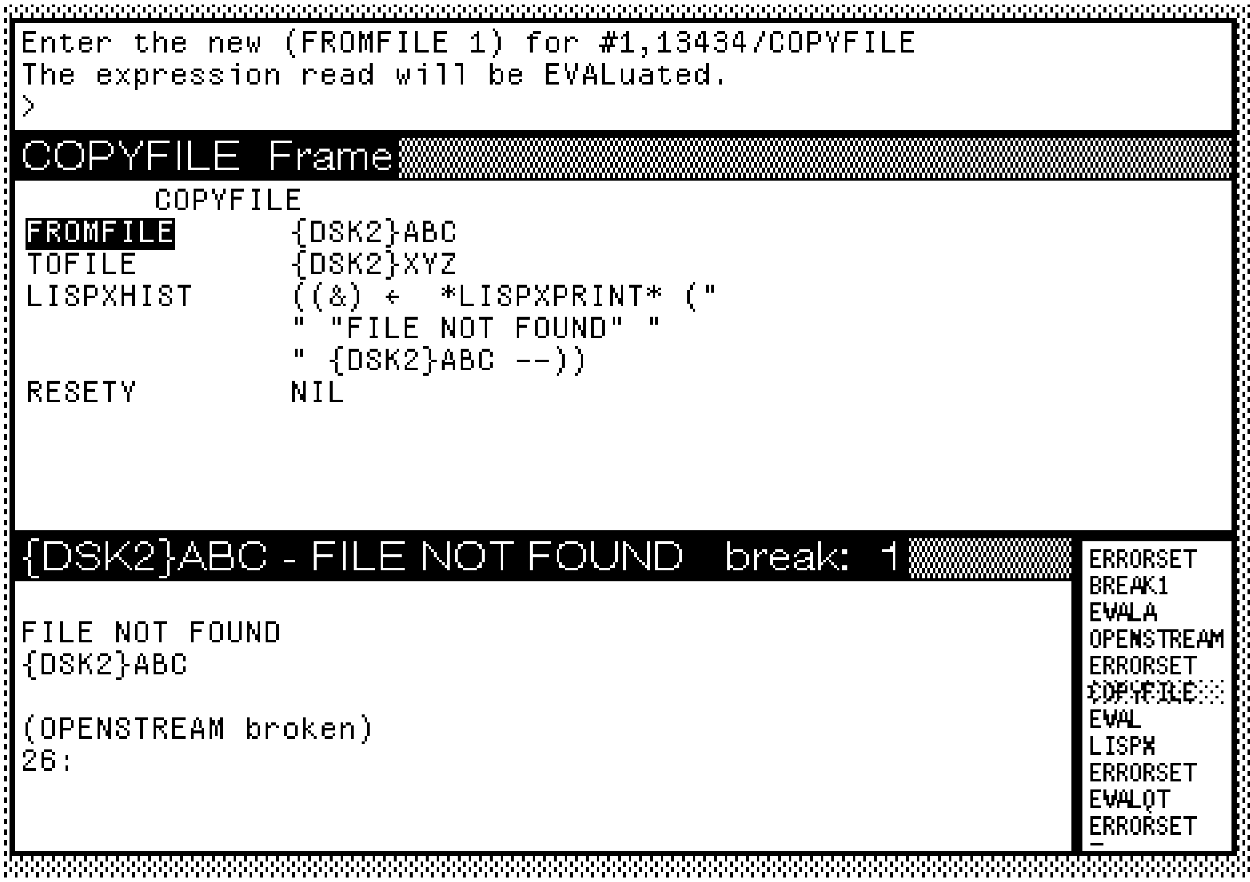
Then click the middle mouse button anywhere in the inspector window. This will bring up a menu with three choices. Choose the middle one: *DisplayEdit*. This will bring up an editor on the function in the inspector window.

To change the value of one of the arguments:

First select the argument to be reset by placing the cursor over the argument name and clicking the left mouse button.

Then click the middle mouse button anywhere in the inspector window. This will bring up a menu with one item. Choose this item: *SET*.

This will bring up type-in window just above the inspector window. Type the new value for the argument into this window. NOTE that the expression you type in will be evaluated before the argument is set. Hence to set the argument to A, you have to type in (QUOTE A) and so on.



```
Enter the new (FROMFILE 1) for #1,13434/COPYFILE
The expression read will be EVALuated.
>
```

COPYFILE Frame

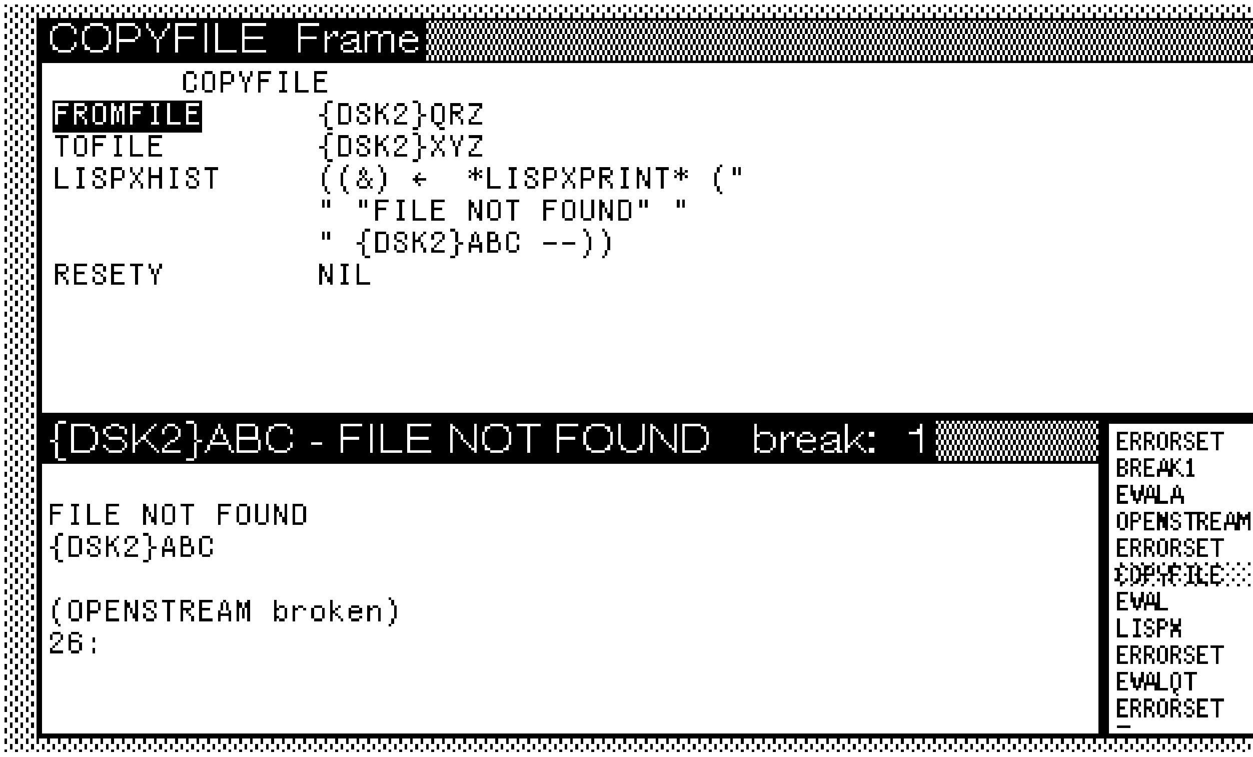
```
        COPYFILE
FROMFILE        {DSK2}ABC
TOFILE          {DSK2}XYZ
LISPXHIST       ((&) ←  *LISPXPRINT* ("
                " "FILE NOT FOUND" "
                " {DSK2}ABC --))
RESETY          NIL
```

{DSK2}ABC - FILE NOT FOUND   break: 1

```
FILE NOT FOUND
{DSK2}ABC

(OPENSTREAM broken)
26:
```

The new setting for the argument will be displayed immediately in the inspector window.



```
COPYFILE Frame
       COPYFILE
FROMFILE       {DSK2}QRZ
TOFILE         {DSK2}XYZ
LISPXHIST      ((&) ← *LISPXPRINT* ("
               " "FILE NOT FOUND" "
               " {DSK2}ABC --))
RESETY         NIL
```

```
{DSK2}ABC - FILE NOT FOUND   break: 1

FILE NOT FOUND
{DSK2}ABC

(OPENSTREAM broken)
26:
```

```
ERRORSET
BREAK1
EVALA
OPENSTREAM
ERRORSET
COPYFILE
EVAL
LISPX
ERRORSET
EVALQT
ERRORSET
```

Breaks within Breaks

When working inside a break frequently evaluates an expression containing an error. If this error causes a break, a second break window will be opened and a break exec started in this window.

Exiting from this second break window using either a normal exit (OK, GO, RETURN) or an abort (^) will simply return to the next higher level break exec, i.e., the break in which the the break being exited occurred.

Breaks within breaks are like DEdits. You can exit from a higher level break only by exiting from the next lower level break first.

If an error occurs in the break exec that would normally cause an abort rather than another break, then the abort error message is simply printed in the break window and control returns to the break exec.

Tailoring the Break Package

The following parameters can be used to tailor various aspects of the error handler, the break exec and break windows.

HELPDEPTH ž an integer used as a threshold for determing whether to break or abort. When an error occurs, if the stack depth is greater than HELPDEPTH then the computation is broken, otherwise its aborted. Initially, 7. A setting of 1 will insure that a break always occurs.

HELPTIME ž an integer indicating the number of milliseconds that a computation has to be in progress for a break to be used instead of an abort when an error occurs. Initially, 1000.

HELPFLAG ž if NIL, no breaks will occur and all errors will cause an abort. If BREAK!, a break will occur for every error. If T, the break or abort decision will be made based on HELPDEPTH and HELPTIME as described above. Initially, T.

Note HELPFLAG must be set with SETTOPVAL as in (SETTOPVAL ’HELPFLAG ’BREAK!). SETQ cannot be used to set this variable.

AUTOBACKTRACEFLG ž if non-NIL, a backtrace window is automatically opened along with every break window. If NIL, then the BT menu command must be used to open the backtrace window. Initially, NIL.

CLOSEBREAKWINDOWFLG ž if NIL, then a break window will remain on the screen when the break is exited. The window must then be closed by hand. If T, then each break window is closed as the break is exited. Initially, T.

Break Documentation

Breaks and the Break Exec are documented in Sections 9.1 thru 9.3 of the IRM. Break Windows are documented in Section 20.3 of the IRM. All of these sections contain a mix of user and programmer material.

The Standard Interlisp Errors

There are currently fifty-plus standard types of errors in the Interlisp system. When an abort or a break occurs, most errors will print the offending expression following the error message for that type of error, e.g., NON-NUMERIC ARG NIL is very common.

All of the standard Interlisp errors and error messages are described (very) briefly in Section 9.8 of the IRM. This section is reproduced in the Appendix.

Error #17 is a sort of generic error. Many packages and system function use the function called ERROR! to indicate an error not on the standard error types list. ERROR! will indicate an generic error and then print a more specific error message given to it by the specific package or function in which the error occurred.

Exercises

Set HELPDEPTH to 1. Then start entering Lisp expression containing errors. When these expressions cause errors, browse around in the break trying to do various things.

Define a few functions with errors and then execute them. Browse around in the resulting break. Try to correct the error and restart the computation.