The SIGNAL package

Henry Thompson

File: {ivy}<hthompson>lisp>rpc>signal.bravo, .press
Revised: January 22, 1983 2:13 PM

The file SIGNAL.DCOM implements a first pass at the CEDAR/MESA signal mechanism. It allows signals to be raised, caught, examined, and resumed or exited. Catch phrases are introduced with the clisp-word **enable**; signals are raised with the function Signal.

(Signal type arg)

Raises a signal of type *type*. If the signal is resumed, will return with the argument to **sresume**, - see below. Otherwise will not return.

```
(enable
  s1 => a1 a2 a3 ...
        . . .
  sn => n1 n2 n3 ...
 form
  l1 -> la1 la2 ...
   . . .
  ln -> ln1 ln2 ...)
```

The double arrow lines above are called *catch phrases*, the single arrow lines are called *finish phrases*. Evaluates *form* so as to catch signals *s1*, ... *sn* if they are raised during its evaluation. If e.g. *s1* is raised, the forms a1 ... an
(the *catch phrase* for s1) will be evaluated in the context of the call to Signal which raised *s1*, with the addition of the fact that the variables type and arg will be locally bound to *type* and *arg*. For a catch phrase to be well formed, all control paths through it must end with one of the following four *quit forms*:

(**exit**)

Causes the stack to unwind back through the enclosing enable form, which is exited with value NIL.

(**sresume** form)

Returns from the call to Signal with the value of form as the value of that call.

(**goto** label)

Causes the stack to unwind back to the enclosing enable form, where the *finish phrase* for label is evaluated. The value of the last form in the phrase is the value of the enable. The variables $SignalType$, $SignalArg$, and $Exit$ will be bound to *type* and *arg* of the original call to Signal and *label* respectively, but otherwise the environment of the call to Signal is lost.

(**reject**)

Causes the signal handling process to act as if the catch phrase had not been there at all.

> When a signal is raised the stack is scanned upwards for a catch phrase for that signal. If none is found (or if all those found are **reject**ed) an Uncaught Signal break will occur. Otherwise the one of the three other options listed above will occur.
>
> There is one signal name which receives special interpretation. A catch phrase with the name **any** will catch any signal not explicitly caught elsewhere in the enable.
>
> There is a special label whose name is **unwind**, which cannot be gone to and has a special meaning. The finish clause for the **unwind** label will be evaluated as the stack unwinds upwards from a call to Signal to the enable clause which caught it (or to the UserExec if ERROR! is envoked).

At the moment the package does use ERROR! and NLSETQ itself to implement the stack unwinding. This means the package interacts correctly with RESETSAVE, but has the unfortunate consequence that interleaving on the stack of calls to ERRORSET in any of its forms and enable clauses has consequences which are confusing at best.

As there are many ways in which the semantics of signals and catchers are much cleaner than those of errors and errorsets, it is hoped that at some point a more sophisticated implementation of the signal mechanism undoing the resetsaves directly and redefining ERRORSET in terms of enable will be made available.

As an interim measure however, a mechanism for turning errors into signals has been provided. The function (MakeErrorsSignals) will arrange that all errors will be converted into a signal with name LispError and argument an instance of the following record:

```
        (RECORD LispError (eMess eFn eType . ePos))
```

where eMess is the standard error message pair (for printing with ERRORMESS), eFn the name of the function which the old package would have broken, eType its type, and ePos a stack pointer to its frame. If you catch this signal, you should free the stack pointer. If you don't catch it (or catch and reject it), a break *will happen*, regardless of HELPDEPTH, ERRORSETs, etc. That is, BREAKCHECK is *not* called or paid attention to. This is the interim price of having the other benefits of the signal package, and in practice seems to work quite well. Things from the old package that *do* still work are ERRORTYPELST, user interrupts, and the various other funnies to do with hash arrays and EOFs which the error code handles.

To go back to the old way of life, use (MakeErrorsErrorsAgain).

There follows on the next page an toy example of the use of the package, which is in fact included in the file itself - enjoy!

```
(ST
 [LAMBDA NIL                                    (* ht: "20-JAN-83 21:40")
  (enable
     s1 => (PRINT "s1 caught" T)
        (goto s1)
     s2 => (PRINT "s2 caught")
        (sresume 37)
     s3 => (PRINT "s3 caught")
        (reject)
     s4 => (PRINT "s4 caught")
        (exit)
     any => (printout T type " caught by any" T)
        (exit)
    (TestSignals)
     s1 -> (PRINT "s1 unwound")
     unwind -> (PRINT "unwinding"))])

(TestSignals
 [LAMBDA NIL                                    (* ht: "18-JAN-83 16:39")
  (printout T T
        (SELECTQ (PROGN (printout T T ">")
                                          (READ))
              (1 (Signal ’s1 1))
              (2 (Signal ’s2 2))
              (3 (Signal ’s3 3))
              (4 (Signal ’s4 4))
              (5 (Signal ’foo 5))
              6)
      ’_])
```