3 Nonlocal Exit

This chapter is about the commands `catch` and `throw`. These commands work together as a kind of super-`stop` command, which you can use to stop several levels of procedure invocation at once.

Quiz Program Revisited

In Chapter 4 of the first volume, which was about predicates, I posed the problem of a quiz program that would allow three tries to answer each question. Here is the method I used then:

```
to ask.thrice :question :answer
repeat 3 [if ask.once :question :answer [stop]]
print sentence [The answer is] :answer
end

to ask.once :question :answer
print :question
if equalp readlist :answer [print [Right!] output "true]
print [Sorry, that's wrong.]
output "false
end
```

I remarked that `ask.once` acts like a command, in that it has an effect (printing stuff), but it's also an operation, which outputs `true` or `false`. What it *really* wants to do is not output a value but instead be able to stop not only itself but also its calling procedure `ask.thrice`. Here is another version that allows just that:

```
to qa :question :answer
catch "correct [ask.thrice :question :answer]
end

to ask.thrice :question :answer
repeat 3 [ask.once :question :answer]
print sentence [The answer is] :answer
end

to ask.once :question :answer
print :question
if equalp readlist :answer [print [Right!] throw "correct]
print [Sorry, that's wrong.]
end
```

To understand this group of procedures, start with `ask.thrice` and suppose the player keeps getting the wrong answer. Both `ask.once` and `ask.thrice` are straightforward commands; the `repeat` instruction in `ask.thrice` is simpler than it was in the other version.

Now what if the person answers correctly? `Ask.once` then evaluates the instruction

```
throw "correct
```

`Throw` is a command that requires one input, which must be a word, called a "tag." The effect of `throw` is to stop the current procedure, like `stop`, and to keep stopping higher-level procedures until it reaches an active `catch` whose first input is the same as the input to `throw`.

If that sounds confusing, don't give up; it's because I haven't explained `catch` and you have to understand them together. The description of `catch` is deceptively simple: `Catch` is a command that requires two inputs. The first must be a word (called the "catch tag"), the second a list of Logo instructions. The effect of `catch` is the same as that of `run`—it evaluates the instructions in the list. `Catch` pays no attention to its first input. That input is there only for the benefit of `throw`.

In this example program `qa` invokes `catch`; `catch` invokes `ask.thrice`, which invokes `repeat`, which invokes `ask.once`. To understand how `throw` works, you have to remember that primitive procedures are just as much procedures as user-defined ones. That's something we're sometimes lax about. A couple of paragraphs ago, I said that `ask.once` evaluates the instruction

```
throw "correct
```

if the player answers correctly. That wasn't really true. The truth is that `ask.once` evaluates the instruction

```
if equalp readlist :answer [print [Right!] throw "correct]
```

by invoking `if`. It is the procedure `if` that actually evaluates the instruction that invokes `throw`. I made a bit of a fuss about this fine point when we first met `if`, but I've been looser about it since then. Now, though, we need to go back to thinking precisely. The point is that there is a `catch` procedure in the collection of active procedures (`qa`, `catch`, `ask.thrice`, and so on) at the time `throw` is invoked.

(In Chapter 9 of the first volume, I made the point that primitives count as active procedures and that `stop` stops the lowest-level invocation of a user-defined procedure. I said that it would be silly for `stop` to stop only the `if` that invoked it, but that you could imagine `stop` stopping a `repeat`. I gave

```
repeat 100 [print "hello if equalp random 5 0 [stop]]
```

as an example of something that doesn't work. But we can make it work this way:

```
catch "done [repeat 100 [print "hello
                         if equalp random 5 0 [throw "done]]]
```

The `throw` stops the `if`, the `repeat`, and the `catch`. Here's a little quiz for you: Why don't I say that the `throw` stops the `equalp`?)

Nonlocal Exit and Modularity

`Throw` is called a "nonlocal exit" because it stops not only the (user-defined) procedure in which it is used but also possibly some number of superprocedures of that one. Therefore, it has an effect on the program as a whole that's analogous to the effect of changing the value of a variable that is not local to the procedure doing the changing. If you see a `make` command used in some procedure, and the variable whose name is the first input isn't local to the same procedure, it becomes much harder to understand what that procedure is really doing. You can't just read that procedure in isolation; you have to think about all its superprocedures too. That's why I've been discouraging you from using global variables.

`Throw` is an offense against modularity in the same way. If I gave you `ask.once` to read, without having shown you the rest of the program, you'd have trouble understanding it. The point may not seem so important when you're reading the small example programs in this book, but when you are working on large projects, with 30 or 300 procedures in them, it becomes much more important.

If I were going to use `catch` and `throw` in this quiz project, one thing I might do is rename `ask.thrice` and `ask.once` as `qa1` and `qa2`. These names would make it clear that the three procedures are meant to work together and indicate which is a subprocedure of which. That name change would help a reader of the program. (Remember that `qa` and its friends are not the whole project; they're all subprocedures of a higher-level `quiz` procedure. So grouping them with similar names really does distinguish them from something else.)

Nonlocal Output

Consider this procedure that takes a list of numbers as its input and computes the product of all the numbers:

```
to multiply :list
if emptyp :list [output 1]
output (first :list) * (multiply butfirst :list)
end
```

Suppose that we intend to use this procedure with very large lists of numbers, and we have reason to believe that many of the lists will include a zero element. If any number in the list is zero, then the product of the entire list must be zero; we can save time by giving an output of zero as soon as we discover this:

```
to multiply :list
if emptyp :list [output 1]
if equalp first :list 0 [output 0]
output (first :list) * (multiply butfirst :list)
end
```

This is an improvement, but not enough of one. To see why, look at this trace of a typical invocation:

```
? trace "multiply
? print multiply [4 5 6 0 1 2 3]
( multiply [4 5 6 0 1 2 3] )
 ( multiply [5 6 0 1 2 3] )
  ( multiply [6 0 1 2 3] )
   ( multiply [0 1 2 3] )
   multiply outputs 0
  multiply outputs 0
 multiply outputs 0
multiply outputs 0
0
```

Each of the last three lines indicates an invocation of `multiply` in which the zero output by a lower level is multiplied by a number seen earlier in the list: first 6, then 5, then 4. It would be even better to avoid those extra multiplications:

```
to multiply :list
output catch "zero [mul1 :list]
end

to mul1 :list
if emptyp :list [output 1]
if equalp first :list 0 [(throw "zero 0)]
output (first :list) * (mul1 butfirst :list)
end

? trace [multiply mul1]
? print multiply [4 5 6 0 1 2 3]
( multiply [4 5 6 0 1 2 3] )
 ( mul1 [4 5 6 0 1 2 3] )
  ( mul1 [5 6 0 1 2 3] )
   ( mul1 [6 0 1 2 3] )
    ( mul1 [0 1 2 3] )
multiply outputs 0
0
```

This time, as soon as `mul1` sees a zero in the list, it arranges for an immediate return to `multiply`, without completing the other three pending invocations of `mul1`.

In the definition of `mul1`, the parentheses around the invocation of `throw` are required, because in this situation we are giving `throw` an optional second input. When given a second input, `throw` acts as a super-`output` instead of as a super-`stop`. That is, `throw` finds the nearest enclosing matching `catch`, as usual, but arranges that that matching `catch` outputs a value, namely the second input to `throw`. In this example, the word `zero` is the catch tag, and the number `0` is the output value.

The same trick that I've used here for efficiency reasons can also be used to protect against the possibility of invalid input data. This time, suppose that we want to multiply a list of numbers, but we suspect that occasionally the user of the program might accidentally supply an input list that includes a non-numeric member. A small modification will prevent a Logo error message:

```
to multiply :list
output catch "early [mul1 :list]
end

to mul1 :list
if emptyp :list [output 1]
if not numberp first :list [(throw "early "non-number)]
if equalp first :list 0 [(throw "early 0)]
output (first :list) * (mul1 butfirst :list)
end

? print multiply [781 105 87 foo 24 13 6]
non-number
```

I've changed the catch tag, even though Logo wouldn't care, because using the word `zero` as the tag is misleading now that it also serves the purpose of catching non-numeric data.

Catching Errors

On the other hand, if we don't expect to see invalid data very often, then checking every list member to make sure it's a number is needlessly time-consuming; also, this "defensive" test makes the program structure more complicated and therefore harder for people to read. Instead, I'd like to be able to multiply the list members, and let Logo worry about possible non-numeric input. Here's how:

```
to multiply :list
catch "error [output mul1 :list]
output "non-number
end

to mul1 :list
if emptyp :list [output 1]
output (first :list) * (mul1 butfirst :list)
end
```

```
? print multiply [3 4 5]
60
? print multiply [3 four 5]
non-number
```

To understand how this works, you must know what Logo does when some primitive procedure (such as `*` in this example) complains about an error. The Logo error handler automatically carries out the instruction

```
throw "error
```

If this `throw` "unwinds" the active procedures all the way to top level without finding a corresponding `catch`, then Logo prints the error message. If you do catch the error, no message is printed.

If you are paused (see Chapter 15 of the first volume), the situation is a little more complicated. Imagine that there is a procedure called `pause.loop` that reads and evaluates the instructions you type while paused. The implicit `throw` on an error can be caught by a `catch` that is invoked "below" that interactive level. That is, during the pause you can invoke a procedure that catches errors. But if you don't do that, `pause.loop` will catch the error and print the appropriate message. (You understand, I hope, that this is an imaginary procedure. I've just given it a name to make the point that the interactive instruction evaluator that is operating during a pause is midway through the collection of active procedures starting with the top-level one and ending with the one that caused the error.) What all this means, more loosely, is that an error during a pause can't get you all the way back to top level, but only to where you were paused.

You should beware of the fact that stopping a program by typing control-C or command-period, depending on the type of computer you're using, is handled as if it were an error. That is, it can be caught. So if you write a program that catches errors and never stops, you're in trouble. You may have to turn the computer off and start over again to escape!

If you use the `item` primitive to ask for more items than are in the list, it's an error. Here are two versions of `item` that output the empty list instead:

```
to safe.item1 :number :list
if :number < (1+count :list) [output item :number :list]
output []
end
```

```
to safe.item2 :number :list
catch "error [output item :number :list]
output []
end
```

The first version explicitly checks, before invoking `item`, to make sure the item number is small enough. The second version goes ahead and invokes `item` without checking, but it arranges to catch any error that happens. If there is no error, the `output` ends the running of the procedure. If we get to the next instruction line, we know there must have been an error. The second version of the procedure is a bit faster because it doesn't have to do all that arithmetic before trying `item`. Also, the first version only tests for one possible error; it will still bomb out, for example, if given a negative item number. The second version is safe against *any* bad input.

This technique works well if the instruction list `output`s or `stop`s. But what if we want to do something like

```
catch "error [make "variable item 7 :list]
```

and we want to put something special in the variable if there is an error? In this example, the procedure will continue to its next instruction whether or not an error was caught. We need a way to ask Logo about any error that might have happened. For this purpose we use the operation `error`. This operation takes no inputs. It outputs a list with information about the most recently caught error. If no error has been caught, it outputs the empty list. Otherwise it outputs a list of four members: a numeric error code, the text of the error message that would otherwise have been printed, the name of the procedure in which the error happened, and the instruction line that was being evaluated.

```
to sample
catch "error [print :nonexistent]
show error
end

? sample
[11 [nonexistent has no value] sample
    [catch "error [print :nonexistent]]]
```

But for now all that matters is that the output will be nonempty if an error was caught. So I can say

```
catch "error [make "variable item 7 :list]
if not emptyp error [make "variable []]
```

This will put an empty list into the variable if there is an error in the first line.

You can only invoke `error` once for each caught error. If you invoke `error` a second time, it will output the empty list. That's so that you don't get confused by trying to catch an error twice and having an error actually happen the first time but not the second time. If you'll need to refer to the contents of the `error` list more than once, put it in a variable.

Just in case you've previously caught an error without invoking `error`, it's a good idea to use the instruction

```
ignore error
```

before catching an error and invoking `error` to test whether or not the error occurred. `Ignore` is a Berkeley Logo primitive that takes one input and does nothing with it; the sole purpose of the instruction is to "use up" any earlier caught error so that the next invocation of `error` will return an empty list if no error is caught this time.

Ending It All

You can stop all active procedures and return to top level by evaluating the instruction

```
throw "toplevel
```

This is a special kind of `throw` that can't be caught.

You've seen this instruction before, in the first volume, where I mentioned it as a way to get out of a pause. That's where it's most useful. Before you use it in a procedure, though, you should be sure that you *really* want to stop everything. For example, suppose you're writing a game program. If the player gets zapped by an evil Whatzit, he's dead and the game is over. So you write

```
to zap.player
print [You're dead!]
throw "toplevel
end
```

because `zap.player` might be invoked several levels deep, but you want to stop everything. But one day you decide to take three different games you've written and combine them into a single program:

```
to play
local "gamename
print [You can play wumpus, dungeon, or rummy.]
print [Which do you want?]
make "gamename first rl
if :gamename = "wumpus [wumpus]
if :gamename = "dungeon [dungeon]
if :gamename = "rummy [rummy]
if not memberp :gamename [wumpus dungeon rummy] [print [No such game!]]
play
end
```

Now your game is no longer the top-level procedure. `play` wants to keep going after a game is over. By throwing to toplevel in the game program, you make that impossible.