8

Control in Sequential Languages

After looking briefly at the history of jumps and structured control, we will study exceptions and continuations. Exceptions are a form of jump that exits a block or function call, returning to some previously established point for handling the exception. Continuations are a more general form of "return" based on calling a function that is passed into a block for this purpose. The chapter concludes with a discussion of *force* and *delay,* complimentary techniques for delaying computation by placing it inside a function and forcing delayed computation with a function call.

8.1 STRUCTURED CONTROL

8.1.1 Spaghetti Code

In Fortran or assembly code, it is easy to write programs with incomprehensible control structure. Here is a short code fragment that illustrates a few of the possibilities. The fragment includes a Fortran `CONTINUE` statement, which is an instruction that does nothing but is used for the purpose of placing a label between two instructions. If you scan the code from top to bottom, you might get the idea that the instructions between labels 10 and 20 act together to perform some meaningful task. However, then as you scan downward, you can see that it is possible later to jump to instruction 11, which is in the middle of this set of instructions:

```
10 IF (X .GT. 0.000001) GO TO 20
   X = -X
11 Y = X*X - SIN(Y)/(X+1)
   IF (X .LT. 0.000001) GO TO 50
20 IF (X*Y .LT. 0.00001) GO TO 30
   X = X-Y-Y
30 X = X+Y
   . . .
50 CONTINUE
```

Although no short sequence can begin to approximate the Byzantine control flow of many archaic Fortran programs, this example may give you some feel for the kind of confusing control flow that was used in many programs in the early days of computing.

8.1.2 Structured Control

In the 1960s, programmers began to understand that unstructured jumps could make it difficult to understand a program. This was partly a realization about programming style and partly a realization about programming languages: If incomprehensible control flow is bad programming style, then programming languages should provide mechanisms that make it easy to organize the control structure of programs. This led to the development of some constructs that structure jumps:

```
if ... then ... else ... end
while ... do ... end
for ... { ... }
case ...
```

These are now adopted in virtually all modern languages.

In modern programming style, we group code in logical blocks, avoid explicit jumps except for function returns, and cannot jump *into* the middle of a block or function body.

The restriction on jumps into blocks illustrates the value of leaving a construct out of a programming language. If a label is placed in the middle of a function body and a program executes a jump to this label, what should happen? Should an activation record be created for the function call? If not, then local variables will not be meaningful. If so, then how will function parameters stored in the activation record be set? Without executing a call to the function, there are no parameter values to use in the call. Because these questions have no good, convincing, clear answers, it is better to design the compiler to reject programs that might jump into the middle of a function body.

Although most introductory programming books and courses today emphasize the importance of clean control structure and discourage the use of `go to` (if the language even allows it), it took many years of discussion and debate to reach this modern point of view. One reason for the change in perspective over the years is the decreasing importance of instruction-level efficiency. In 1960 and even 1970, there were many applications in which it was useful to save the cost of a test, even if it meant complicating the control structure of the program. Therefore, programmers considered it important to be able to jump out of the middle of a loop, avoiding another test at the top of the loop.

In the 1980s and 1990s, as computer speed increased, the number of applications in which a small change in efficiency would truly matter decreased significantly, to the point at which, in the 1990s, Java was introduced without any `go to` statement. Those interested in history may enjoy reading E.W. Dijkstra's March 1968 letter to the editor of *Communications of the ACM*, "Go To Considered Harmful," later

posted on the Association for Computing Machinery web site as the October 1995 "Classic of the Month." The letter begins with these sentences:

> For a number of years I have been familiar with the observation that the quality of programmers is a decreasing function of the density of **go to** statements in the programs they produce. More recently I discovered why the use of the **go to** statement has such disastrous effects, and I became convinced that the **go to** statement should be abolished from all "higher level" programming languages. . . .

Simple control structures such as if-then-else have now been in common use since the rise of Pascal in the late 1970s. Exceptions and continuations, discussed in the remainder of this chapter, are more recent innovations in programming languages.

8.2 EXCEPTIONS

8.2.1 Purpose of an Exception Mechanism

Exceptions provide a structured form of jump that may be used to exit a construct such as a block or function invocation. The name *exception* suggests that exceptions are to be used in exceptional circumstances. However, programming languages cannot enforce any sort of intention like this. Exceptions are a basic mechanism that can be used to achieve the following effects:

- jump out of a block or function invocation
- pass data as part of the jump
- return to a program point that was set up to continue the computation.

In addition to jumping from one program point to another, there is also some memory management associated with exceptions. Specifically, unnecessary activation records may be deallocated as the result of the jump. We subsequently see how this works.

Exception mechanisms may be found in many programming languages, including Ada, C++, Clu, Java, Mesa, ML, and PL/1. Every exception mechanism includes two constructs:

- a statement or expression form for *raising* an exception, which aborts part of the current computation and causes a jump (transfer of control),
- a *handler* mechanism, which allows certain statements, expressions, or function calls to be equipped with code to respond to exceptions raised during their execution.

Another term for raising an exception is *throwing* an exception; another term for handling an exception is *catching* an exception.

There are several reasons why exceptions have become an accepted language construct. Many languages do not otherwise have a clean mechanism for jumping out of a function call, for example, aborting the call. Rather than using `go to`s, which can be used in unstructured ways, many programmers prefer exceptions, which can be used to jump only out of some part of a program, not into some part of the program that has not been entered yet. Exceptions also allow a programmer to pass data as part of the jump, which is useful if the program tries to recover from some kind of error condition. Exceptions provide a useful dynamic way of determining to where