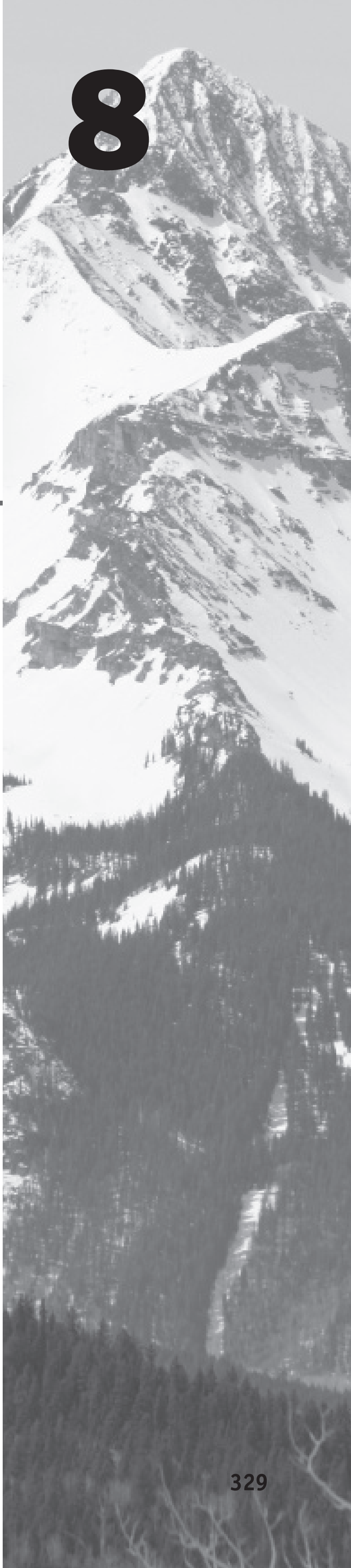

8

Statement-Level Control Structures

The flow of control, or execution sequence, in a program can be examined at several levels. In Chapter 7, the flow of control within expressions, which is governed by operator associativity and precedence rules, was discussed. At the highest level is the flow of control among program units, which is discussed in Chapters 9 and 13. Between these two extremes is the important issue of the flow of control among statements, which is the subject of this chapter.

We begin by giving an overview of the evolution of control statements. This topic is followed by a thorough examination of selection statements, both those for two-way and those for multiple selection. Next, we discuss the variety of looping statements that have been developed and used in programming languages. Next, we take a brief look at the problems associated with unconditional branch statements. Finally, we describe the guarded command control statements.

8.1 Introduction

Computations in imperative-language programs are accomplished by evaluating expressions and assigning the resulting values to variables. However, there are few useful programs that consist entirely of assignment statements. At least two additional linguistic mechanisms are necessary to make the computations in programs flexible and powerful: some means of selecting among alternative control flow paths (of statement execution) and some means of causing the repeated execution of statements or sequences of statements. Statements that provide these kinds of capabilities are called **control statements**.

Computations in functional programming languages are accomplished by evaluating expressions and applying functions to given parameters. Furthermore, the flow of execution among the expressions and functions is controlled by other expressions and functions, although some of them are similar to the control statements in the imperative languages.

The control statements of the first successful programming language, Fortran, were, in effect, designed by the architects of the IBM 704. All were directly related to machine language instructions, so their capabilities were more the result of instruction design rather than language design. At the time, little was known about the difficulty of programming, and, as a result, the control statements of Fortran in the mid-1950s were thought to be entirely adequate. By today's standards, however, they are considered seriously lacking.

A great deal of research and discussion was devoted to control statements in the 10 years between the mid-1960s and the mid-1970s. One of the primary conclusions of these efforts was that, although a single control statement (a selectable goto) is minimally sufficient, a language that is designed *not* to include a goto needs only a small number of different control statements. In fact, it was proven that all algorithms that can be expressed by flowcharts can be coded in a programming language with only two control statements: one for choosing between two control flow paths and one for logically controlled iterations (Böhm and Jacopini, 1966). An important result of this is that the unconditional branch statement is superfluous—potentially useful but nonessential. This fact, combined with the practical problems of using unconditional branches, or gotos, led to a great deal of debate about the goto, as will be discussed in Section 8.4.

Programmers care less about the results of theoretical research on control statements than they do about writability and readability. All languages that have been widely used include more control statements than the two that are minimally required, because writability is enhanced by a larger number and wider variety of control statements. For example, rather than requiring the use of a logically controlled loop statement for all loops, it is easier to write programs when a counter-controlled loop statement can be used to build loops that are naturally controlled by a counter. The primary factor that restricts the number of control statements in a language is readability, because the presence of a large number of statement forms demands that program readers learn a larger language. Recall that few people learn all of the statements of a relatively large language; instead, they learn the subset they choose to use, which is often a different subset from that used by the programmer who wrote the program they are trying to read. On the other hand, too few control statements can require the use of lower-level statements, such as the goto, which also makes programs less readable.

The question as to the best collection of control statements to provide the required capabilities and the desired writability has been widely debated. It is essentially a question of how much a language should be expanded to increase its writability at the expense of its simplicity, size, and readability.

A **control structure** is a control statement and the collection of statements whose execution it controls.

There is only one design issue that is relevant to all of the selection and iteration control statements: Should the control structure have multiple entries? All selection and iteration constructs control the execution of code segments, and the question is whether the execution of those code segments always begins with the first statement in the segment. It is now generally believed that multiple entries add little to the flexibility of a control statement, relative to the decrease in readability caused by the increased complexity. Note that multiple entries are possible only in languages that include gotos and statement labels.

At this point, the reader might wonder why multiple exits from control structures are not considered a design issue. The reason is that all programming languages allow some form of multiple exits from control structures, the rationale being as follows: If all exits from a control structure are restricted to transferring control to the first statement following the structure, where control would flow if the control structure had no explicit exit, there is no harm to readability and also no danger. However, if an exit can have an unrestricted target and therefore can result in a transfer of control to anywhere in the program unit that contains the control structure, the harm to readability is the same as for a goto statement anywhere else in a program. Languages that have a goto statement allow it to appear anywhere, including in a control structure. Therefore, the issue is the inclusion of a goto, not whether multiple exits from control expressions are allowed.

8.2 Selection Statements

A **selection statement** provides the means of choosing between two or more execution paths in a program. Such statements are fundamental and essential parts of all programming languages, as was proven by Böhm and Jacopini.

Selection statements fall into two general categories: two-way and n-way, or multiple selection. Two-way selection statements are discussed in Section 8.2.1; multiple-selection statements are covered in Section 8.2.2.

8.2.1 Two-Way Selection Statements

Although the two-way selection statements of contemporary imperative languages are quite similar, there are some variations in their designs. The general form of a two-way selector is as follows:

```
if control_expression
    then clause
    else clause
```

8.2.1.1 Design Issues

The design issues for two-way selectors can be summarized as follows:

- What is the form and type of the expression that controls the selection?
- How are the then and else clauses specified?
- How should the meaning of nested selectors be specified?

8.2.1.2 The Control Expression

Control expressions are specified in parentheses if the **`then`** reserved word (or some other syntactic marker) is not used to introduce the then clause. In those cases where the **`then`** reserved word (or alternative marker) is used, there is less need for the parentheses, so they are often omitted, as in Ruby.

In C89, which did not have a Boolean data type, arithmetic expressions were used as control expressions. This can also be done in Python, C99, and C++. However, in those languages either arithmetic or Boolean expressions can be used. In other contemporary languages, only Boolean expressions can be used for control expressions.

8.2.1.3 Clause Form

In many languages, the then and else clauses appear as either single statements or compound statements. One variation of this is Perl, in which all then and else clauses must be compound statements, even if they have only one statement. Many languages use braces to form compound statements, which serve as the bodies of then and else clauses. In Python and Ruby, the then and else clauses are statement sequences, rather than compound statements. The complete selection statement is terminated in these languages with a reserved word.

Python uses indentation to specify compound statements. For example,

```
if x > y :
  x = y
  print "case 1"
```

All statements equally indented are included in the compound statement.[1] Notice that rather than **then**, a colon is used to introduce the then clause in Python.

The variations in clause form have implications for the specification of the meaning of nested selectors, as discussed in the next subsection.

8.2.1.4 Nesting Selectors

Recall that in Chapter 3, we discussed the problem of syntactic ambiguity of a straightforward grammar for a two-way selector statement. That ambiguous grammar was as follows:

<if_stmt> → **if** <logic_expr> **then** <stmt>
          | **if** <logic_expr> **then** <stmt> **else** <stmt>

The issue is that when a selection statement is nested in the then clause of a selection statement, it is not clear with which if an else clause should be associated. This problem is reflected in the semantics of selection statements. Consider the following Java-like code:

```
if (sum == 0)
  if (count == 0)
    result = 0;
else
    result = 1;
```

This statement can be interpreted in two different ways, depending on whether the else clause is matched with the first then clause or the second. Notice that the indentation seems to indicate that the else clause belongs with the first then clause. However, with the exceptions of Python and F#, indentation has no effect on semantics in contemporary languages and is therefore ignored by their compilers.

The crux of the problem in this example is that the else clause follows two then clauses with no intervening else clause, and there is no syntactic indicator to specify a matching of the else clause to one of the then clauses. In Java, as in

1. The statement following the compound statement must have the same indentation as the `if`.

many other imperative languages, the static semantics of the language specify that the else clause is always paired with the nearest previous unpaired then clause. A static semantics rule, rather than a syntactic entity, is used to provide the disambiguation. So, in the example, the else clause would be paired with the second then clause. The disadvantage of using a rule rather than some syntactic entity is that although the programmer may have meant the else clause to be the alternative to the first then clause and the compiler found the structure syntactically correct, its semantics is the opposite. To force the alternative semantics in Java, the inner **if** is put in a compound, as in

```
if (sum == 0) {
  if (count == 0)
    result = 0;
}
else
    result = 1;
```

C, C++, and C# have the same problem as Java with selection statement nesting. Because Perl requires that all then and else clauses be compound, it does not. In Perl, the previous code would be written as follows:

```
if (sum == 0) {
  if (count == 0) {
    result = 0;
  }
} else {
    result = 1;
}
```

If the alternative semantics were needed, it would be

```
if (sum == 0) {
  if (count == 0) {
    result = 0;
  }
  else {
    result = 1;
  }
}
```

Another way to avoid the issue of nested selection statements is to use an alternative means of forming compound statements. Consider the syntactic structure of the Java **if** statement. The then clause follows the control expression and the else clause is introduced by the reserved word **else**. When the then clause is a single statement and the else clause is present, although there is no need to mark the end, the **else** reserved word in fact marks the end of the then clause. When the then clause is a compound, it is terminated by a right brace. However, if the last clause in an **if**, whether then or else, is not a compound, there is no syntactic entity to mark the end of the whole selection statement. The use of a special word for this purpose resolves the question of the semantics of nested selectors and also adds to the readability of the statement. This is the design of the selection statement in Ruby. For example, consider the following Ruby statement:

```
if a > b then sum = sum + a
  acount = acount + 1
else sum = sum + b
  bcount = bcount + 1
end
```

The design of this statement is more regular than that of the selection statements of the C-based languages, because the form is the same regardless of the number of statements in the then and else clauses. (This is also true for Perl.) Recall that in Ruby, the then and else clauses consist of statement sequences rather than compound statements. The first interpretation of the selector example at the beginning of this section, in which the else clause is matched to the nested **`if`**, can be written in Ruby as follows:

```
if sum == 0  then
  if count == 0 then
    result = 0
  else
    result = 1
  end
end
```

Because the **`end`** reserved word closes the nested **`if`**, it is clear that the else clause is matched to the inner then clause.

The second interpretation of the selection statement at the beginning of this section, in which the else clause is matched to the outer **`if`**, can be written in Ruby as follows:

```
if sum == 0 then
  if count == 0 then
    result = 0
  end
else
    result = 1
 end
```

The following statement, written in Python, is semantically equivalent to the last Ruby statement above:

```
if sum == 0 :
  if count == 0 :
    result = 0
else:
  result = 1
```

If the line **else:** were indented to begin in the same column as the nested **if**, the else clause would be matched with the inner **if**.

ML does not have a problem with nested selectors because it does not allow else-less **if** statements.

8.2.1.5 Selector Expressions

In the functional languages ML, F#, and LISP, the selector is not a statement; it is an expression that results in a value. Therefore, it can appear anywhere any other expression can appear. Consider the following example selector written in F#:

```
let y =
    if x > 0 then  x
    else 2 * x;;
```

This creates the name y and sets it to either x or 2 * x, depending on whether x is greater than zero.

In F#, the type of the value returned by the then clause of an **if** construct must be the same as that of the value returned by its else clause. If there is no else clause, the then clause cannot return a value of a normal type. In this case, it can only return a **unit** type, which is a special type that means no value. A **unit** type is represented in code as ().

8.2.2 Multiple-Selection Statements

The **multiple-selection** statement allows the selection of one of any number of statements or statement groups. It is, therefore, a generalization of a selector. In fact, two-way selectors can be built with a multiple selector.

The need to choose from among more than two control paths in programs is common. Although a multiple selector can be built from two-way selectors and gotos, the resulting structures are cumbersome, unreliable, and difficult to write and read. Therefore, the need for a special structure is clear.

8.2.2.1 Design Issues

Some of the design issues for multiple selectors are similar to some of those for two-way selectors. For example, one issue is the question of the type of expression on which the selector is based. In this case, the range of possibilities is larger, in part because the number of possible selections is larger. A two-way selector needs an expression with only two possible values. Another issue is whether single statements, compound statements, or statement sequences may be selected. Next, there is the question of whether only a single selectable segment can be executed when the statement is executed. This is not an issue for two-way selectors, because they always allow only one of the clauses to be on a control path during one execution. As we shall see, the resolution of this issue for multiple selectors is a trade-off between reliability and flexibility. Another issue is the form of the case value specifications. Finally, there is the issue of what should result from the selector expression evaluating to a value that does not select one of the segments. (Such a value would be unrepresented among the selectable segments.) The choice here is between simply disallowing the situation from arising and having the statement do nothing at all when it does arise.

The following is a summary of these design issues:

- What is the form and type of the expression that controls the selection?
- How are the selectable segments specified?
- Is execution flow through the structure restricted to include just a single selectable segment?
- How are the case values specified?
- How should unrepresented selector expression values be handled, if at all?

8.2.2.2 Examples of Multiple Selectors

The C multiple-selector statement, **switch**, which is also part of C++, Java, and JavaScript, is a relatively primitive design. Its general form is

```
switch (expression) {
 case constant_expression$_1$:statement$_1$;
 . . .
 case constant$_n$: statement_n;
 [default: statement$_{n+1}$]
}
```

where the control expression and the constant expressions are some discrete type. This includes integer types, as well as characters and enumeration types. The selectable statements can be statement sequences, compound statements, or blocks. The optional **default** segment is for unrepresented values of the control expression. If the value of the control expression is not represented and no default segment is present, then the statement does nothing.

The **switch** statement does not provide implicit branches at the end of its code segments. This allows control to flow through more than one selectable code segment on a single execution. Consider the following example:

```
switch  (index) {
   case  1:
   case  3: odd += 1;
            sumodd += index;
   case  2:
   case  4: even += 1;
            sumeven += index;
   default: printf("Error in switch, index = %d\n", index);
}
```

This code prints the error message on every execution. Likewise, the code for the 2 and 4 constants is executed every time the code at the 1 or 3 constants is executed. To separate these segments logically, an explicit branch must be included. The **break** statement, which is actually a restricted goto, is normally used for exiting **switch** statements. **break** transfers control to the first statement after the compound statement in which it appears.

The following **switch** statement uses **break** to restrict each execution to a single selectable segment:

```
switch (index) {
  case 1:
  case 3: odd += 1;
          sumodd += index;
          break;
  case 2:
  case 4: even += 1;
          sumeven += index;
          break;
  default: printf("Error in switch, index = %d\n", index);
}
```

Occasionally, it is convenient to allow control to flow from one selectable code segment to another. For example, in the example above, the segments for the case values 1 and 2 are empty, allowing control to flow to the segments for 3 and 4, respectively. This is the reason why there are no implicit branches in the **switch** statement. The reliability problem with this design arises when the mistaken absence of a **break** statement in a segment allows control to flow to the next segment incorrectly. The designers of C's **switch** traded a decrease in reliability for an increase in flexibility. Studies have shown, however, that the ability to have control flow from one selectable segment to another is rarely used. C's **switch** is modeled on the multiple-selection statement in ALGOL 68, which also does not have implicit branches from selectable segments.

The C switch statement has virtually no restrictions on the placement of the case expressions, which are treated as if they were normal statement labels. This laxness can result in highly complex structure within the switch body. The following example is taken from Harbison and Steele (2002).

```
switch (x)
  default:
  if (prime(x))
    case 2: case 3: case 5: case 7:
      process_prime(x);
  else
     case 4: case 6: case 8: case 9: case 10:
       process_composite(x);
```

This code may appear to have diabolically complex form, but it was designed for a real problem and works correctly and efficiently to solve that problem.[2]

The Java switch prevents this sort of complexity by disallowing case expressions from appearing anywhere except the top level of the body of the switch.

The C# switch statement differs from that of its C-based predecessors in two ways. First, C# has a static semantics rule that disallows the implicit execution of more than one segment. The rule is that every selectable segment must end with an explicit unconditional branch statement: either a **break**, which transfers control out of the **switch** statement, or a goto, which can transfer control to one of the selectable segments (or virtually anywhere else). For example,

```
switch (value) {
   case -1:
      Negatives++;
      break;
   case 0:
      Zeros++;
      goto case 1;
   case 1:
      Positives++;
   default:
      Console.WriteLine("Error in switch \n");
}
```

Note that `Console.WriteLine` is the method for displaying strings in C#.

The other way C#'s **switch** differs from that of its predecessors is that the control expression and the case statements can be strings in C#.

PHP's **switch** uses the syntax of C's **switch** but allows more type flexibility. The case values can be any of the PHP scalar types—string, integer, or double precision. As with C, if there is no **break** at the end of the selected segment, execution continues into the next segment.

Ruby has two forms of multiple-selection constructs, both of which are called *case expressions* and both of which yield the value of the last expression evaluated. The only version of Ruby's case expressions that is described here is semantically similar to a list of nested if statements:

```
case
when Boolean_expression then expression
. . .
when Boolean_expression then expression
[else expression]
end
```

2. The problem is to call `process_prime` when `x` is prime and `process_composite` when `x` is not prime. The design of the switch body resulted from an attempt to optimize based on the knowledge that `x` was most often in the range of `1` to `10`.

The semantics of this case expression is that the Boolean expressions are evaluated one at a time, top to bottom. The value of the case expression is the value of the first then expression whose Boolean expression is true. The else represents true in this statement, and the else clause is optional. For example,[3]

```
leap = case
       when year % 400 == 0 then true
       when year % 100 == 0 then false
       else year % 4 == 0
       end
```

This case expression evaluates to true if `year` is a leap year.

The other Ruby case expression form is similar to the switch of Java. Perl and Python do not have multiple-selection statements.

8.2.2.3 Implementing Multiple Selection Structures

A multiple selection statement is essentially an n-way branch to segments of code, where n is the number of selectable segments. Implementing such a statement must be done with multiple conditional branch instructions. Consider again the general form of the C switch statement, with breaks:

```
switch (expression) {
 case constant_expression_1: statement_1;
  break;
 . . .
 case constant_n: statement_n;
  break;
 [default: statement_n+1]
}
```

One simple translation of this statement follows:

```
Code to evaluate expression into t
goto branches
label_1: code for statement_1
          goto out
...
label_n: code for statement_n
          goto out
default: code for statement_n+1
          goto out
branches: if t = constant_expression_1 goto label_1
          ...
          if t = constant_expression_n goto label_n
          goto default
out:
```

3. This example is from Thomas et al. (2013).

The code for the selectable segments precedes the branches so that the targets of the branches are all known when the branches are generated. An alternative to these coded conditional branches is to put the case values and labels in a table and use a linear search with a loop to find the correct label. This requires less space than the coded conditionals.

The use of conditional branches or a linear search on a table of cases and labels is a simple but inefficient approach that is acceptable when the number of cases is small, say less than 10. It takes an average of about half as many tests as there are cases to find the right one. For the default case to be chosen, all other cases must be tested. In statements with 10 or more cases, the low efficiency of this form is not justified by its simplicity.

When the number of cases is 10 or greater, the compiler can build a hash table of the segment labels, which would result in approximately equal (and short) times to choose any of the selectable segments. If the language allows ranges of values for case expressions, as in Ruby, the hash method is not suitable. For these situations, a binary search table of case values and segment addresses is better.

If the range of the case values is relatively small and more than half of the whole range of values is represented, an array whose indices are the case values and whose values are the segment labels can be built. Array elements whose indices are not among the represented case values are filled with the default segment's label. Then finding the correct segment label is found by array indexing, which is very fast.

Of course, choosing among these approaches is an additional burden on the compiler. In many compilers, only two different methods are used. As in other situations, determining and using the most efficient method costs more compiler time.

8.2.2.4 Multiple Selection Using `if`

In many situations, a **`switch`** or **`case`** statement is inadequate for multiple selection (Ruby's **`case`** is an exception). For example, when selections must be made on the basis of a Boolean expression rather than some ordinal type, nested two-way selectors can be used to simulate a multiple selector. To alleviate the poor readability of deeply nested two-way selectors, some languages, such as Perl and Python, have been extended specifically for this use. The extension allows some of the special words to be left out. In particular, else-if sequences are replaced with a single special word, and the closing special word on the nested **`if`** is dropped. The nested selector is then called an **else-if clause**. Consider the following Python selector statement (note that else-if is spelled **`elif`** in Python):

```
if  count < 10 :
  bag1 = True
elif count < 100 :
  bag2 = True
elif count < 1000 :
  bag3 = True
```

which is equivalent to the following:

```
if count < 10 :
  bag1 = True
else :
  if count < 100 :
    bag2 = True
  else :
    if count < 1000 :
      bag3 = True
    else :
      bag4 = True
```

The else-if version (the first) is the more readable of the two. Notice that this example is not easily simulated with a **`switch`** statement, because each selectable statement is chosen on the basis of a Boolean expression. Therefore, the else-if statement is not a redundant form of **`switch`**. In fact, none of the multiple selectors in contemporary languages are as general as the if-then-else-if statement. An operational semantics description of a general selector statement with else-if clauses, in which the E's are logic expressions and the S's are statements, is given here:

```
   if $E_1$ goto 1
   if $E_2$ goto 2
   . . .
1: $S_1$
   goto out
2: $S_2$
   goto out
. . .
out: . . .
```

From this description, we can see the difference between multiple selection structures and else-if statements: In a multiple selection statement, all the E's would be restricted to comparisons between the value of a single expression and some other values.

Languages that do not include the else-if statement can use the same control structure, with only slightly more typing.

The Python example if-then-else-if statement above can be written as the Ruby `case` statement:

```
case
when count < 10 then bag1 = true
when count < 100 then bag2 = true
when count < 1000 then bag3 = true
end
```

Else-if statements are based on the common mathematics statement, the conditional expression.

The Scheme multiple selector, which is based on mathematical conditional expressions, is a special form function named `COND`. `COND` is a slightly generalized version of the mathematical conditional expression; it allows more than one predicate to be true at the same time. Because different mathematical conditional expressions have different numbers of parameters, `COND` does not require a fixed number of actual parameters. Each parameter to `COND` is a pair of expressions in which the first is a predicate (it evaluates to either `#T` or `#F`).

The general form of `COND` is

```
 (COND
  (predicate_1  expression_1)
  (predicate_2  expression_2)
  . . .
  (predicate_n  expression_n)
  [(ELSE  expression_n+1)]
)
```

where the `ELSE` clause is optional.

The semantics of `COND` is as follows: The predicates of the parameters are evaluated one at a time, in order from the first, until one evaluates to `#T`. The expression that follows the first predicate that is found to be `#T` is then evaluated and its value is returned as the value of `COND`. If none of the predicates is true and there is an `ELSE`, its expression is evaluated and the value is returned. If none of the predicates is true and there is no `ELSE`, the value of `COND` is unspecified. Therefore, all `COND`s should include an `ELSE`.

Consider the following example call to `COND`:

```
(COND
  ((> x y) "x is greater than y")
  ((< x y) "y is greater than x")
  (ELSE "x and y are equal")
)
```

Note that string literals evaluate to themselves, so that when this call to `COND` is evaluated, it produces a string result.

8.3 Iterative Statements

An **iterative statement** is one that causes a statement or collection of statements to be executed zero, one, or more times. An iterative statement is often called a **loop**. Every programming language from Plankalkül on has included some method of repeating the execution of segments of code. Iteration is the very essence of the power of the computer. If some means of repetitive execution of a statement or collection of statements were not possible, programmers would be required to state every action in sequence; useful programs would be huge and inflexible and take unacceptably large amounts of time to write and mammoth amounts of memory to store.

The first iterative statements in programming languages were directly related to arrays. This resulted from the fact that in the earliest years of computers, computing was largely numerical in nature, frequently using loops to process data in arrays.

Several categories of iteration control statements have been developed. The primary categories are defined by how designers answered two basic design questions:

- How is the iteration controlled?
- Where should the control mechanism appear in the loop statement?

The primary possibilities for iteration control are logical, counting, or a combination of the two. The main choices for the location of the control mechanism are the top of the loop or the bottom of the loop. Top and bottom here are logical, rather than physical, denotations. The issue is not the physical placement of the control mechanism; rather, it is whether the mechanism is executed and affects control before or after execution of the statement's body. A third option, which allows the user to decide where to put the control, at the top, at the bottom, or even within the controlled segment, is discussed in Section 8.3.3.

The **body** of an iterative statement is the collection of statements whose execution is controlled by the iteration statement. We use the term **pretest** to mean that the test for loop completion occurs before the loop body is executed and **posttest** to mean that it occurs after the loop body is executed. The iteration statement and the associated loop body together form an **iteration statement**.

8.3.1 Counter-Controlled Loops

A counting iterative control statement has a variable, called the **loop variable**, in which the count value is maintained. It also includes some means of specifying the **initial** and **terminal** values of the loop variable, and the difference between sequential loop variable values, often called the **stepsize**. The initial, terminal, and stepsize specifications of a loop are called the **loop parameters**.

Although logically controlled loops are more general than counter-controlled loops, they are not necessarily more commonly used. Because counter-controlled loops are more complex, their design is more demanding.

Counter-controlled loops are sometimes supported by machine instructions designed for that purpose. Unfortunately, machine architecture can outlive the prevailing approaches to programming at the time of the architecture design. For example, VAX computers have a very convenient instruction for the implementation of posttest counter-controlled loops, which Fortran had at the time of the design of the VAX (mid-1970s). But Fortran no longer had such a loop by the time VAX computers became widely used (it had been replaced by a pretest loop). Furthermore, no other widely used language of the time had a posttest counting loop.

8.3.1.1 Design Issues

There are many design issues for iterative counter-controlled statements. The nature of the loop variable and the loop parameters provide a number of design issues. The type of the loop variable and that of the loop parameters obviously should be the same or at least compatible, but what types should be allowed? One apparent choice is integer, but what about enumeration, character, and floating-point types? Another question is whether the loop variable is a normal variable, in terms of scope, or whether it should have some special scope. Allowing the user to change the loop variable or the loop parameters within the loop can lead to code that is very difficult to understand, so another question is whether the additional flexibility that might be gained by allowing such changes is worth that additional complexity. A similar question arises about the number of times and the specific time when the loop parameters are evaluated: If they are evaluated just once, loops are simple but less flexible. Finally, what is the value of the loop variable after loop termination, if its scope extends beyond the loop?

The following is a summary of these design issues:

- What are the type and scope of the loop variable?
- Should it be legal for the loop variable or loop parameters to be changed in the loop, and if so, does the change affect loop control?
- Should the loop parameters be evaluated only once, or once for every iteration?
- What is the value of the loop variable after loop termination?

The issue of the value of the loop variable after loop termination is solved in some languages, such as Fortran 90, by making the loop variable undefined after loop termination. Other languages, such as Ada, make the scope of the loop variable the loop itself.

8.3.1.2 The `for` Statement of the C-Based Languages

The general form of C's **`for`** statement is

```
for (expression_1; expression_2; expression_3)
 loop body
```

The loop body can be a single statement, a compound statement, or a null statement.

Because assignment statements in C produce results and thus can be considered expressions, the expressions in a **`for`** statement are often assignment statements. The first expression is for initialization and is evaluated only once, when the **`for`** statement execution begins. The second expression is the loop control and is evaluated before each execution of the loop body. As is usual in C, a zero value means false and all nonzero values mean true. Therefore, if the value of the second expression is zero, the **`for`** is terminated; otherwise, the loop body statements are executed. In C99, the expression also could be a Boolean type. A C99 Boolean type stores only the values 0 or 1. The last expression in the **`for`** is executed after each execution of the loop body. It is often used to increment the loop counter. An operational semantics description of the C **`for`** statement is shown next. Because C expressions can be used as statements, expression evaluations are shown as statements.

```
      expression_1
loop:
      if expression_2 = 0 goto out
      [loop body]
      expression_3
      goto loop
out: . . .
```

Following is an example of a skeletal C **`for`** statement:

```
for (count = 1; count <= 10; count++)
  . . .
}
```

All of the expressions of C's **`for`** are optional. An absent second expression is considered true, so a **`for`** without one is potentially an infinite loop. If the first and/or third expressions are absent, no assumptions are made. For example, if the first expression is absent, it simply means that no initialization takes place.

Note that C's **`for`** need not count. It can easily model counting *and* logical loop structures, as demonstrated in the next section.

The C **`for`** design choices are the following: There is no explicit loop variable and no loop parameters. All involved variables can be changed in the loop body. The expressions are evaluated in the order stated previously. Although it can create havoc, it is legal to branch into a C **`for`** loop body.

C's **`for`** is one of the most flexible, because each of the expressions can comprise multiple expressions, which in turn allow multiple loop variables that can be of any type. When multiple expressions are used in a single expression of a **`for`** statement, they are separated by commas. All C statements have values, and this form of multiple expression is no exception. The value of such a multiple expression is the value of the last component.

Consider the following **`for`** statement:

```
for (count1 = 0, count2 = 1.0;
    count1 <= 10 && count2 <= 100.0;
    sum = ++count1 + count2, count2 *= 2.5);
```

The operational semantics description of this is

```
      count1 = 0
      count2 = 1.0
loop:
      if count1 > 10 goto out
      if count2 > 100.0 goto out
```

```
    count1 = count1 + 1
    sum = count1 + count2
    count2 =  count2 * 2.5
    goto loop
out: . . .
```

The example C **for** statement does not need and thus does not have a loop body. All the desired actions are part of the **for** statement itself, rather than in its body. The first and third expressions are multiple statements. In both of these cases, the whole expression is evaluated, but the resulting value is not used in the loop control.

The **for** statement of C99 and C++ differs from that of earlier versions of C in two ways. First, in addition to an arithmetic expression, it can use a Boolean expression for loop control. Second, the first expression can include variable definitions. For example,

```
for (int count = 0; count < len; count++) { . . . }
```

The scope of a variable defined in the **for** statement is from its definition to the end of the loop body.

The **for** statement of Java and C# is like that of C++, except that the loop control expression is restricted to **boolean**.

In all of the C-based languages, the last two loop parameters are evaluated with every iteration. Furthermore, variables that appear in the loop parameter expression can be changed in the loop body. Therefore, these loops can be complex and potentially unreliable.

8.3.1.3 The `for` Statement of Python

The general form of Python's **for** is

```
for loop_variable in object:
 - loop body
[else:
 - else clause]
```

The loop variable is assigned the value in the object, which is often a range, one for each execution of the loop body. After loop termination, the loop variable has the value last assigned to it. The loop variable can be changed in the loop body, but such a change has no effect on loop operation. The else clause, when present, is executed if the loop terminates normally.

Consider the following example:

```
for count in [2, 4, 6]:
   print count
```

produces

```
2
4
6
```

For most simple counting loops in Python, the **range** function is used. **range** takes one, two, or three parameters. The following examples demonstrate the actions of **range**:

```
range(5) returns [0, 1, 2, 3, 4]
range(2, 7) returns [2, 3, 4, 5, 6]
range(0, 8, 2) returns [0, 2, 4, 6]
```

Note that **range** never returns the highest value in a given parameter range.

8.3.1.4 Counter-Controlled Loops in Functional Languages

Counter-controlled loops in imperative languages use a counter variable, but such variables do not exist in pure functional languages. Rather than iteration to control repetition, functional languages use recursion. Rather than a statement, functional languages use a recursive function. Counting loops can be simulated in functional languages as follows: The counter can be a parameter for a function that repeatedly executes the loop body, which can be specified in a second function sent to the loop function as a parameter. So, such a loop function takes the body function and the number of repetitions as parameters.

The general form of an F# function for simulating counting loops, named `forLoop` in this case, is as follows:

```
let rec forLoop loopBody reps =
    if reps <= 0  then
        ()
    else
         loopBody()
         forLoop loopBody, (reps - 1);;
```

In this function, the parameter `loopBody` is the function with the body of the loop and the parameter `reps` is the number of repetitions. The reserved word **rec** appears before the name of the function to indicate that it is recursive. The empty parentheses do nothing; they are there because in F# an empty statement is illegal and every **if** must have an else clause.

8.3.2 Logically Controlled Loops

In many cases, collections of statements must be repeatedly executed, but the repetition control is based on a Boolean expression rather than a counter. For these situations, a logically controlled loop is convenient. Actually, logically controlled loops are more general than counter-controlled loops. Every counting loop can be built with a logical loop, but the reverse is not true. Also, recall that only selection and logical loops are essential to express the control structure of any flowchart.

8.3.2.1 Design Issues

Because they are much simpler than counter-controlled loops, logically controlled loops have fewer design issues.

- Should the control be pretest or posttest?
- Should the logically controlled loop be a special form of a counting loop or a separate statement?

8.3.2.2 Examples

The C-based programming languages include both pretest and posttest logically controlled loops that are not special forms of their counter-controlled iterative statements. The pretest and posttest logical loops have the following forms:

```
while (control_expression)
  loop body
```

and

```
do
  loop body
while (control_expression);
```

These two statement forms are exemplified by the following C# code segments:

```
sum = 0;
indat = Int32.Parse(Console.ReadLine());
while (indat >= 0) {
  sum += indat;
  indat = Int32.Parse(Console.ReadLine());
}

value = Int32.Parse(Console.ReadLine());
do {
   value /= 10;
   digits ++;
}  while  (value > 0);
```

Note that all variables in these examples are integer type. The `ReadLine` method of the `Console` object gets a line of text from the keyboard. `Int32.Parse` finds the number in its string parameter, converts it to **`int`** type, and returns it.

In the pretest version of a logical loop (**`while`**), the statement or statement segment is executed as long as the expression evaluates to true. In the posttest version (**`do`**), the loop body is executed until the expression evaluates to false. In both cases, the statement can be compound. The operational semantics descriptions of those two statements follow:

```
while

loop:
  if control_expression is false goto out
  [loop body]
  goto loop
out: . . .
```

do-while

```
loop:
  [loop body]
  if control_expression is true goto loop
```

It is legal in both C and C++ to branch into both **while** and **do** loop bodies. The C89 version uses an arithmetic expression for control; in C99 and C++, it may be either arithmetic or Boolean.

Java's **while** and **do** statements are similar to those of C and C++, except the control expression must be **boolean** type, and because Java does not have a goto, the loop bodies cannot be entered anywhere except at their beginnings.

Posttest loops are infrequently useful and also can be somewhat dangerous, in the sense that programmers sometimes forget that the loop body will always be executed at least once. The syntactic design of placing a posttest control physically after the loop body, where it has its semantic effect, helps avoid such problems by making the logic clear.

A pretest logical loop can be simulated in a purely functional form with a recursive function that is similar to the one used to simulate a counting loop statement in Section 8.3.1.5. In both cases, the loop body is written as a function. Following is the general form of a simulated logical pretest loop, written in F#:

```
let rec whileLoop test body =
    if test() then
        body()
        whileLoop test body
     else
         ();;
```

8.3.3 User-Located Loop Control Mechanisms

In some situations, it is convenient for a programmer to choose a location for loop control other than the top or bottom of the loop body. As a result, some languages provide this capability. A syntactic mechanism for user-located loop control can be relatively simple, so its design is not difficult. Such loops have the structure of infinite loops but include one or more user-located loop exits. Perhaps the most interesting question is whether a single loop or several nested loops can be exited. The design issues for such a mechanism are the following:

- Should the conditional mechanism be an integral part of the exit?
- Should only one loop body be exited, or can enclosing loops also be exited?

C, C++, Python, Ruby, and C# have unconditional unlabeled exits (**break**). Java and Perl have unconditional labeled exits (**break** in Java, **last** in Perl).

Following is an example of nested loops in Java, in which there is a break out of the outer loop from the nested loop:

```
outerLoop:
  for (row = 0; row < numRows; row++)
     for  (col = 0; col < numCols; col++) {
       sum += mat[row][col];
       if (sum > 1000.0)
         break outerLoop;
     }
```

C, C++, and Python include an unlabeled control statement, **continue**, that transfers control to the control mechanism of the smallest enclosing loop. This is not an exit but rather a way to skip the rest of the loop statements on the current iteration without terminating the loop construct. For example, consider the following:

```
while (sum < 1000) {
  getnext(value);
  if (value < 0) continue;
  sum += value;
}
```

A negative value causes the assignment statement to be skipped, and control is transferred instead to the conditional at the top of the loop. On the other hand, in

```
while (sum < 1000) {
  getnext(value);
  if (value < 0) break;
  sum += value;
}
```

a negative value terminates the loop.

Both **last** and **break** provide for multiple exits from loops, which may seem to be somewhat of a hindrance to readability. However, unusual conditions that require loop termination are so common that such a statement is justified. Furthermore, readability is not seriously harmed, because the target of all such loop exits is the first statement after the loop (or an enclosing loop) rather than just anywhere in the program. Finally, the alternative of using multiple breaks to leave more than one level of loops is even more detrimental to readability.

The motivation for user-located loop exits is simple: They fulfill a common need for goto statements using a highly restricted branch statement. The target of a goto can be many places in the program, both above and below the goto itself. However, the targets of user-located loop exits must be below the exit and can only follow immediately at the end of a compound statement.

8.3.4 Iteration Based on Data Structures

A general data-based iteration statement uses a user-defined data structure and a user-defined function (the iterator) to go through the structure's elements. The iterator is called at the beginning of each iteration, and each time it is called, the iterator returns an element from a particular data structure in some specific order. For example, suppose a program has a user-defined binary tree of data nodes, and the data in each node must be processed in some particular order. A user-defined iteration statement for the tree would successively set the loop variable to point to the nodes in the tree, one for each iteration. The initial execution of the user-defined iteration statement needs to issue a special call to the iterator to get the first tree element. The iterator must always remember which node it presented last so that it visits all nodes without visiting any node more than once. So an iterator must be history sensitive. A user-defined iteration statement terminates when the iterator fails to find more elements.

The **for** statement of the C-based languages, because of its great flexibility, can be used to simulate a user-defined iteration statement. Once again, suppose the nodes of a binary tree are to be processed. If the tree root is pointed to by a variable named `root`, and if `traverse` is a function that sets its parameter to point to the next element of a tree in the desired order, the following could be used:

```
for (ptr = root; ptr == null; ptr = traverse(ptr)) {
  . . .
}
```

In this statement, `traverse` is the iterator.

User-defined iteration statements are more important in object-oriented programming than they were in earlier software development paradigms, because users of object-oriented programming routinely use abstract data types for data structures, especially collections. In such cases, a user-defined iteration statement and its iterator must be provided by the author of the data abstraction because the representation of the objects of the type is not known to the user.

An enhanced version of the **for** statement was added to Java in Java 5.0. This statement simplifies iterating through the values in an array or objects in a collection that implements the `Iterable` interface. (All of the predefined generic collections in Java implement `Iterable`.) For example, if we had an `ArrayList`[4] collection named `myList` of strings, the following statement would iterate through all of its elements, setting each to `myElement`:

```
for  (String myElement : myList) { . . . }
```

4. An `ArrayList` is a predefined generic collection that is actually a dynamic-length array of whatever type it is declared to store.

This new statement is referred to as "foreach," although its reserved word is **for**.

C# and F# (and the other .NET languages) also have generic library classes for collections. For example, there are generic collection classes for lists, which are dynamic length arrays, stacks, queues, and dictionaries (hash table). All of these predefined generic collections have built-in iterators that are used implicitly with the **foreach** statement. Furthermore, users can define their own collections and write their own iterators, which can implement the `IEnumerator` interface, which enables the use of **foreach** on these collections.

For example, consider the following C# code:

```
List<String> names =  new  List<String>();
names.Add("Bob");
names.Add("Carol");
names.Add("Alice");
. . .
foreach  (String name  in  names)
  Console.WriteLine(name);
```

In Ruby, a **block** is a sequence of code, delimited by either braces or the **do** and **end** reserved words. Blocks can be used with specially written methods to create many useful constructs, including iterators for data structures. This construct consists of a method call followed by a block. A block is actually an anonymous method that is sent to the method (whose call precedes it) as a parameter. The called method can then call the block, which can produce output or objects.

Ruby predefines several iterator methods, such as `times` and `upto` for counter-controlled loops, and `each` for simple iterations of arrays and hashes. For example, consider the following example of using `times`:

```
>> 4.times {puts "Hey!"}
Hey!
Hey!
Hey!
Hey!
=> 4
```

Note that >> is the prompt of the interactive Ruby interpreter and => is used to indicate the return value of the expression. The Ruby `puts` statement displays its parameter. In this example, the `times` method is sent to the object `4`, with the block sent along as a parameter. The `times` method calls the block four times, producing the four lines of output. The destination object, `4`, is the return value from `times`.

The most common Ruby iterator is `each`, which is often used to go through arrays and apply a block to each element.[5] For this purpose, it is convenient to allow blocks to have parameters, which, if present, appear at the beginning of the block, delimited by vertical bars (|). The following example, which uses a block parameter, illustrates the use of `each`:

```
>> list = [2, 4, 6, 8]
=> [2, 4, 6, 8]
>> list.each {|value| puts value}
2
4
6
8
=> [2, 4, 6, 8]
```

In this example, the block is called for each element of the array to which the `each` method is sent. The block produces the output, which is a list of the array's elements. The return value of `each` is the array to which it is sent.

Instead of a counting loop, Ruby has the `upto` method. For example, we could have the following:

```
1.upto(5) {|x| print x, " "}
```

This produces the following output:

```
1 2 3 4 5
```

Syntax that resembles a **`for`** loop in other languages could also be used, as in the following:

```
for x in 1..5
  print x, " "
end
```

Ruby actually has no **`for`** statement—constructs like the above are converted by Ruby into `upto` method calls.

Now we consider how blocks work. The **`yield`** statement is similar to a method call, except that there is no receiver object and the call is a request to execute the block attached to the method call, rather than a call to a method. **`yield`** is only called in a method that has been called with a block. If the block has parameters, they are specified in parentheses in the **`yield`** statement. The value returned by a block is that of the last expression evaluated in the block. It is this process that is used to implement the built-in iterators, such as `times`.

5. This is similar to the map functions discussed in Chapter 15.

Python provides strong support for iteration. Suppose one needs to process the nodes in some user-defined data structure. Further suppose that the structure has a traversal method that goes through the nodes of the structure in the desired order. The following skeletal class definition includes such a traversal method that produces the nodes of an instance of this class, one at a time.

```
class MyStructure:
    # Other method definitions, including a constructor

   def traverse(self):
       # if there is another node:
       #    set nod to next node
       # else:
       #    return
       yield nod
```

The `traverse` method appears to be a regular Python method, but it contains a **`yield`** statement, which dramatically changes the semantics of the method. In effect, the method is run in a separate thread of control. The **`yield`** statement acts like a return. On the first call to `traverse`, **`yield`** returns the initial node of the structure. However, on the second call, it returns the second node. On all but the first call to `traverse`, it begins its execution where it left off on the previous execution. Instead of restarting at its beginning, it is resumed. Any local storage in such a method is maintained across its calls. In the case of `traverse`, subsequent calls begin their execution at the beginning of its code, but in the state that it was in its previous execution. In Python, any method that contains a **`yield`** statement is called a *generator*, because it generates data one element at a time.

Of course, one could also produce all of the nodes of the structure, store them in an array, and process them from the array. However, the number of nodes could be large, requiring a large array to store them. The approach using the iterator is more elegant and is not affected by the size of the data structure.

8.4 Unconditional Branching

An **unconditional branch statement** transfers execution control to a specified location in the program. The most heated debate in language design in the late 1960s was over the issue of whether unconditional branching should be part of any high-level language, and if so, whether its use should be restricted. The unconditional branch, or goto, is the most powerful statement for controlling the flow of execution of a program's statements. However, careless use of the goto can lead to serious problems. The goto has stunning power and great flexibility (all other control structures can be built with goto and a selector),

history note

Although several thoughtful people had pointed out the potential problems of gotos earlier, it was Edsger Dijkstra who gave the computing world the first widely read exposé on the dangers of the goto. In his letter he noted, "The goto statement as it stands is just too primitive; it is too much an invitation to make a mess of one's program" (Dijkstra, 1968a). During the first few years after publication of Dijkstra's views on the goto, a large number of people argued publicly for either outright banishment or at least restrictions on the use of the goto. Among those who did not favor complete elimination was Donald Knuth (1974), who argued that there were occasions when the efficiency of the goto outweighed its harm to readability.

but it is this power that makes its use dangerous. Without usage restrictions, imposed by either language design or programming standards, goto statements can make programs very difficult to read, and as a result, highly unreliable and costly to maintain.

These problems follow directly from a goto's ability to force any program statement to follow any other in execution sequence, regardless of whether that statement precedes or follows the previously executed statement in textual order. Readability is best when the execution order of statements in a program is nearly the same as the order in which they appear—in our case, this would mean top to bottom, which is the order to which we are accustomed. Thus, restricting gotos so they can transfer control only downward in a program partially alleviates the problem. It allows gotos to transfer control around code sections in response to errors or unusual conditions but disallows their use to build any sort of loop.

A few languages have been designed without a goto—for example, Java, Python, and Ruby. However, most currently popular languages include a goto statement. Kernighan and Ritchie (1978) call the goto infinitely abusable, but it is nevertheless included in Ritchie's language, C. The languages that have eliminated the goto have provided additional control statements, usually in the form of loop exits, to code one of the justifiable applications of the goto.

The relatively new language, C#, includes a goto, even though one of the languages on which it is based, Java, does not. One legitimate use of C#'s goto is in the **`switch`** statement, as discussed in Section 8.2.2.2.

All of the loop exit statements discussed in Section 8.3.3 are actually camouflaged goto statements. They are, however, severely restricted gotos and are not harmful to readability. In fact, it can be argued that they improve readability, because to avoid their use results in convoluted and unnatural code that would be much more difficult to understand.

8.5 Guarded Commands

Quite different forms of selection and loop structures were suggested by Dijkstra (1975). His primary motivation was to provide control statements that would support a program design methodology that ensured correctness during development rather than when verifying or testing completed programs. This methodology is described in Dijkstra (1976). Another motivation is the increased clarity in reasoning that is possible with guarded commands. Simply put, a selectable segment of a selection statement in a guarded-command statement can be considered independently of any other part of the statement, which is not true for the selection statements of the common programming languages.

Guarded commands are covered in this chapter because they are the basis for the linguistic mechanism developed later for concurrent programming in

CSP (Hoare, 1978). Guarded commands are also used to define functions in Haskell, as discussed in Chapter 15.

Dijkstra's selection statement has the form

```
if <Boolean expression> -> <statement>
[] <Boolean expression> -> <statement>
[] . . .
[] <Boolean expression> -> <statement>
fi
```

The closing reserved word, **fi**, is the opening reserved word spelled backward. This form of closing reserved word is taken from ALGOL 68. The small blocks, called *fatbars*, are used to separate the guarded clauses and allow the clauses to be statement sequences. Each line in the selection statement, consisting of a Boolean expression (a guard) and a statement or statement sequence, is called a **guarded command**.

This selection statement has the appearance of a multiple selection, but its semantics is different. All of the Boolean expressions are evaluated each time the statement is reached during execution. If more than one expression is true, one of the corresponding statements can be nondeterministically chosen for execution. An implementation might always choose the statement associated with the first Boolean expression that evaluates to be true. But it may choose any statement associated with a true Boolean expression. So, the correctness of the program cannot depend on which statement is chosen (among those associated with true Boolean expressions). If none of the Boolean expressions are true, a run-time error occurs that causes program termination. This forces the programmer to consider and list all possibilities. Consider the following example:

```
if i = 0 -> sum := sum + i
[] i > j -> sum := sum + j
[] j > i -> sum := sum + k
fi
```

If `i = 0` and `j > i`, this statement chooses nondeterministically between the first and third assignment statements. If `i` is equal to `j` and is not zero, a run-time error occurs because none of the conditions are true.

This statement can be an elegant way of allowing the programmer to state that the order of execution, in some cases, is irrelevant. For example, to find the largest of two numbers, we can use

```
if  x >= y -> max := x
[] y >= x -> max := y
fi
```

This computes the desired result without overspecifying the solution. In particular, if `x` and `y` are equal, it does not matter which we assign to `max`. This is a form of abstraction provided by the nondeterministic semantics of the statement.

Now, consider this same process coded in a traditional programming language selector:

```
if (x >= y)
  max = x;
else
  max = y;
```

This could also be coded as follows:

```
if (x > y)
  max = x;
else
  max = y;
```

There is no practical difference between these two statements. The first assigns `x` to `max` when `x` and `y` are equal; the second assigns `y` to `max` in the same circumstance. This choice between the two statements complicates the formal analysis of the code and the correctness proof of it. This is one of the reasons why guarded commands were developed by Dijkstra.

The loop structure proposed by Dijkstra has the form

```
do <Boolean expression> -> <statement>
[] <Boolean expression> -> <statement>
[] . . .
[] <Boolean expression> -> <statement>
od
```

The semantics of this statement is that all Boolean expressions are evaluated on each iteration. If more than one is true, one of the associated statements is nondeterministically (perhaps randomly) chosen for execution, after which the expressions are again evaluated. When all expressions are simultaneously false, the loop terminates.

Consider the following problem: Given four integer variables, `q1`, `q2`, `q3`, and `q4`, rearrange the values of the four so that `q1 <= q2 <= q3 <= q4`. Without guarded commands, one straightforward solution is to put the four values into an array, sort the array, and then assign the values from the array back into the scalar variables `q1`, `q2`, `q3`, and `q4`. While this solution is not difficult, it requires a good deal of code, especially if the sort process must be included.

Now, consider the following code, which uses guarded commands to solve the same problem but in a more concise and elegant way.[6]

```
do q1 > q2 -> temp := q1; q1 := q2; q2 := temp;
[] q2 > q3 -> temp := q2; q2 := q3; q3 := temp;
[] q3 > q4 -> temp := q3; q3 := q4; q4 := temp;
od
```

6. This code appears in a slightly different form in Dijkstra (1975).

Dijkstra's guarded command control statements are interesting, in part because they illustrate how the syntax and semantics of statements can have an impact on program verification and vice versa. Program verification is virtually impossible when goto statements are used. Verification is greatly simplified if (1) only logical loops and selections are used or (2) only guarded commands are used. The axiomatic semantics of guarded commands are conveniently specified (Gries, 1981). It should be obvious, however, that there is considerably increased complexity in the implementation of the guarded commands over their conventional deterministic counterparts.

8.6 Conclusions

We have described and discussed a variety of statement-level control structures. A brief evaluation now seems to be in order.

First, we have the theoretical result that only sequence, selection, and pretest logical loops are absolutely required to express computations (Böhm and Jacopini, 1966). This result has been used by those who wish to ban unconditional branching altogether. Of course, there are already sufficient practical problems with the goto to condemn it without also using a theoretical reason. One of the main legitimate needs for gotos—premature exits from loops—can be met with restricted branch statements, such as **`break`**.

One obvious misuse of the Böhm and Jacopini result is to argue against the inclusion of *any* control structures beyond selection and pretest logical loops. No widely used language has yet taken that step; furthermore, we doubt that any ever will, because of the negative effect on writability and readability. Programs written with only selection and pretest logical loops are generally less natural in structure, more complex, and therefore harder to write and more difficult to read. For example, the C# multiple selection structure is a great boost to C# writability, with no obvious negatives. Another example is the counting loop structure of many languages, especially when the statement is simple.

It is not so clear that the utility of many of the other control structures that have been proposed is worth their inclusion in languages (Ledgard and Marcotty, 1975). This question rests to a large degree on the fundamental question of whether the size of languages must be minimized. Both Wirth (1975) and Hoare (1973) strongly endorse simplicity in language design. In the case of control structures, simplicity means that only a few control statements should be in a language, and they should be simple.

The rich variety of statement-level control structures that have been invented shows the diversity of opinion among language designers. After all the invention, discussion, and evaluation, there is still no unanimity of opinion on the precise set of control statements that should be in a language. Most contemporary languages do, of course, have similar control statements, but there is still some variation in the details of their syntax and semantics. Furthermore, there is still disagreement on whether a language should include a goto; C++ and C# do, but Java and Ruby do not.

SUMMARY

Control statements occur in several categories: selection, multiple selection, iterative, and unconditional branching.

The **`switch`** statement of the C-based languages is representative of multiple-selection statements. The C# version eliminates the reliability problem of its predecessors by disallowing the implicit continuation from a selected segment to the following selectable segment.

A large number of different loop statements have been invented for high-level languages. C's **`for`** statement is the most flexible iteration statement, although its flexibility leads to some reliability problems.

Most languages have exit statements for their loops; these statements take the place of one of the most common uses of goto statements.

Data-based iterators are loop statements for processing data structures, such as linked lists, hashes, and trees. The **`for`** statement of the C-based languages allows the user to create iterators for user-defined data. The **`foreach`** statement of Perl and C# is a predefined iterator for standard data structures. In the contemporary object-oriented languages, iterators for collections are specified with standard interfaces, which are implemented by the designers of the collections.

Ruby includes iterators that are a special form of methods that are sent to various objects. The language predefines iterators for common uses, but also allows user-defined iterators.

The unconditional branch, or goto, has been part of most imperative languages. Its problems have been widely discussed and debated. The current consensus is that it should remain in most languages but that its dangers should be minimized through programming discipline.

Dijkstra's guarded commands are alternative control statements with positive theoretical characteristics. Although they have not been adopted as the control statements of a language, part of the semantics appear in the concurrency mechanisms of CSP and the function definitions of Haskell.

REVIEW QUESTIONS

1. What is the definition of *control structure?*
2. What did Böhm and Jocopini prove about flowcharts?
3. What is the definition of *block?*
4. What is/are the design issue(s) for all selection and iteration control statements?
5. What are the design issues for selection structures?
6. What is unusual about Python's design of compound statements?
7. Under what circumstances must an F# selector have an else clause?
8. What are the common solutions to the nesting problem for two-way selectors?

9. What are the design issues for multiple-selection statements?
10. Between what two language characteristics is a trade-off made when deciding whether more than one selectable segment is executed in one execution of a multiple selection statement?
11. What is unusual about C's multiple-selection statement?
12. On what previous language was C's **`switch`** statement based?
13. Explain how C#'s switch statement is safer than that of C.
14. What are the design issues for all iterative control statements?
15. What are the design issues for counter-controlled loop statements?
16. What is a pretest loop statement? What is a posttest loop statement?
17. What is the difference between the **`for`** statement of C++ and that of Java?
18. In what way is C's **`for`** statement more flexible than that of many other languages?
19. What does the **`range`** function in Python do?
20. What contemporary languages do not include a goto?
21. What are the design issues for logically controlled loop statements?
22. What is the main reason user-located loop control statements were invented?
23. What are the design issues for user-located loop control mechanisms?
24. What advantage does Java's **`break`** statement have over C's **`break`** statement?
25. What are the differences between the **`break`** statement of C++ and that of Java?
26. What is a user-defined iteration control?
27. What Scheme function implements a multiple selection statement?
28. How does a functional language implement repetition?
29. How are iterators implemented in Ruby?
30. What language predefines iterators that can be explicitly called to iterate over its predefined data structures?
31. What common programming language borrows part of its design from Dijkstra's guarded commands?

PROBLEM SET

1. Describe three situations where a combined counting and logical looping statement is needed.
2. Study the iterator feature of CLU in Liskov et al. (1981) and determine its advantages and disadvantages.

3. Compare the set of Ada control statements with those of C# and decide which are better and why.
4. What are the pros and cons of using unique closing reserved words on compound statements?
5. What are the arguments, pros and cons, for Python's use of indentation to specify compound statements in control statements?
6. Analyze the potential readability problems with using closure reserved words for control statements that are the reverse of the corresponding initial reserved words, such as the **`case-esac`** reserved words of ALGOL 68. For example, consider common typing errors such as transposing characters.
7. Use the *Science Citation Index* to find an article that refers to Knuth (1974). Read the article and Knuth's paper and write a paper that summarizes both sides of the goto issue.
8. In his paper on the goto issue, Knuth (1974) suggests a loop control statement that allows multiple exits. Read the paper and write an operational semantics description of the statement.
9. What are the arguments both for and against the exclusive use of Boolean expressions in the control statements in Java (as opposed to also allowing arithmetic expressions, as in C++)?
10. Describe a programming situation in which the else clause in Python's **`for`** statement would be convenient.
11. Describe three specific programming situations that require a posttest loop.
12. Speculate as to the reason control can be transferred into a C loop statement.

PROGRAMMING EXERCISES

1. Rewrite the following pseudocode segment using a loop structure in the specified languages:

```
  k = (j + 13) / 27
loop:
  if k > 10 then goto  out
  k = k + 1
  i = 3 * k - 1
  goto  loop
out: . . .
```

a. C, C++, Java, or C#
b. Python
c. Ruby

Assume all variables are integer type. Discuss which language, for this code, has the best writability, the best readability, and the best combination of the two.

2. Redo Programming Exercise 1, except this time make all the variables and constants floating-point type, and change the statement

```
k = k + 1
```

to

```
k = k + 1.2
```

3. Rewrite the following code segment using a multiple-selection statement in the following languages:

```
if ((k == 1) || (k == 2)) j = 2 * k - 1
if ((k == 3) || (k == 5)) j = 3 * k + 1
if (k == 4) j = 4 * k - 1
if ((k == 6) || (k == 7) || (k == 8)) j = k - 2
```

a. C, C++, Java, or C#
b. Python
c. Ruby

Assume all variables are integer type. Discuss the relative merits of the use of these languages for this particular code.

4. Consider the following C program segment. Rewrite it using no gotos or **break**s.

```
j = -3;
for (i = 0; i < 3; i++) {
  switch  (j + 2) {
    case  3:
    case  2: j--; break;
    case  0: j += 2; break;
    default: j = 0;
  }
  if  (j > 0)  break;
 j = 3 - i
}
```

5. In a letter to the editor of *CACM*, Rubin (1987) uses the following code segment as evidence that the readability of some code with gotos is better than the equivalent code without gotos. This code finds the first row of an *n* by *n* integer matrix named x that has nothing but zero values.

```
for (i = 1; i <= n; i++) {
  for (j = 1; j <= n; j++)
    if (x[i][j] != 0)
     goto reject;
  println ('First all-zero row is:', i);
  break;
reject:
 }
```

Rewrite this code without gotos in one of the following languages: C, C++, Java, or C#. Compare the readability of your code to that of the example code.

6. Consider the following programming problem: The values of three integer variables—`first`, `second`, and `third`–must be placed in the three variables `max`, `mid`, and `min`, with the obvious meanings, without using arrays or user-defined or predefined subprograms. Write two solutions to this problem, one that uses nested selections and one that does not. Compare the complexity and expected reliability of the two.
7. Rewrite the C program segment of Programming Exercise 4 using **`if`** and goto statements in C.
8. Rewrite the C program segment of Programming Exercise 4 in Java without using a **`switch`** statement.
9. Translate the following call to Scheme's `COND` to C and set the resulting value to `y`.

```
(COND
 ((> x 10) x)
 ((< x 5) (* 2 x))
 ((= x 7) (+ x 10))
)
```