Chapter 6 - Flow of Execution

6.1 Introduction to Execution Models

A programming language must specify more than the actions that need to be performed. It must specify the order in which they are performed as well. In this chapter, we consider four models for the flow of execution control. These models are classified according to whether or not they involve a test, by the number of possible execution paths they contain, and by the number of execution paths that can actually be followed during a single pass through the structure. The models are summarized by Table 6.1.

<table>
<thead>
<tr>
<th>Model</th>
<th>Test</th>
<th>Path Choices</th>
<th>Paths Followed</th>
</tr>
</thead>
<tbody>
<tr>
<td>Classical control structure</td>
<td>Yes</td>
<td>&gt;1</td>
<td>1</td>
</tr>
<tr>
<td>Unconstrained</td>
<td>No</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>Concurrent</td>
<td>No</td>
<td>&gt;1</td>
<td>&gt;1</td>
</tr>
<tr>
<td>Nondeterministic</td>
<td>No</td>
<td>&gt;1</td>
<td>1</td>
</tr>
</tbody>
</table>

The classical control structures determine one execution path to be followed from multiple possibilities based on the result of a test. Unconstrained structures include no test and give no choice of path at all but rather specify a predetermined path that may or may not be sequential. Concurrent structures are those that specify more than one execution path and follow all of them in a concurrent manner. Nondeterministic structures also specify multiple possible paths, but during execution only one path is followed and that path is determined arbitrarily rather than as a result of a deterministic test.

In this chapter, we consider only the control of execution flow at the statement level. Many of the same issues and corresponding models will appear in Chapter 7 when we look at execution flow in relationship to procedural abstraction.

6.2 Classical Control Structures

Classical control structures in programming languages use a test to select one from among several possible execution paths. This model has two common forms: the conditional structure and the iterative structure. The conditional structure chooses one of several paths based on the result of the test. The iterative structure includes an execution unit that is to be executed repetitively. The test associated with the iterative structure determines whether or not the execution unit will be executed and is repeated after each of the unit’s executions.

6.2.1 Conditional structures

Conditional control structures determine the next block of statements to be executed based on the results of one or more tests. Such structures are usually implemented through an if statement. We examine four forms of such structures.
**Simple conditional** The simplest form of conditional control structure performs a single test, whose result is used to determine whether or not to execute a specified block of statements. The two parts of the structure, therefore, are the Boolean expression that defines the condition and the block of statements that will be performed if the expression evaluates to true.

The typical form of the simple conditional is

```plaintext
if <boolean_expression> then <block_of_statements>
```

The method of blocking statements often varies from language to language. Java and C++, for example, consider the block of statements to be a single statement that can be made into a compound statement by enclosing multiple statements between curly braces (`{}`). In addition, both of these languages omit the `then` keyword delimiter. Ada and Modula 2, however, have specific keywords for ending the block of statements. Ada terminates with `end if` and Modula 2 with `end`. Neither of these languages needs a beginning-of-block marker, because `then` functions in this capacity.

**Two alternative conditional** Most programming languages permit the extension of the simple conditional `if` statement to a two-alternative structure. This direct extension takes the form

```plaintext
if <boolean expression> then
 <block of statements>
else
 <block of statements>
```

The syntax used by C++ and Java to implement this structure leads to the well-known dangling `else` problem in the case of nested conditionals. To illustrate this, consider the following Java fragment.

```java
if (x > 0)
  if (x < 10)
    x=x+1
  else
    x=x-1;
```

The `else` in this statement could be attached to either the first or the second `if`, depending on the way the language interprets this structure. Java, of course, has a rule for defining the semantics of this fragment. It is that the `else` will always be matched with the nearest preceding `if` that is thus far unmatched with an `else`. In the case of our example, Java would match the `else` as our indenting suggests.

The use of required block markers, as in Ada and Modula-2, avoids this confusing syntax completely, because the termination of each conditional block is explicitly denoted. For example, the two ways of interpreting the preceding Java fragment could be written in Ada as

```plaintext
if x>0 then
  if x<10 then
    x:=x+1;
  else
    x:=x-1;
end if;
```
Multialternative conditional Two forms of the multialternative conditional structure are found in most programming languages. The first and more general one permits any sequence of Boolean expressions to be tested, with the first true expression specifying the block of statements chosen for execution. The second form evaluates an expression and chooses to execute the block of statements that corresponds to the value obtained.

Some languages, such as Ada, permit the direct expression of the general multialternative structure through the use of separators for each condition. The general form in Ada is

```plaintext
if <condition-1> then
    <statement-block-1>
elsif <condition-2> then
    <statement-block-2>
...
elsif <condition-n> then
    <statement-block-n>
[else <statement-block-(n+1)>]
end if;
```

The conditions are tested in order until one is evaluated as true. Then, the corresponding statement block is executed. If none of the conditions is true, the statement block following the else, if present, is executed. If no else clause is present and all conditions are false, none of the statement blocks is executed, and control passes to the statement following this entire construct. As soon as one condition is found to be true and its statement block has been executed, execution of the structure is completed, and no further conditions are tested. Such a structure can be simulated in Java and other languages that lack the generalized structure of Ada by use of a nested if-then-else structure.

The switch structure is more restrictive in that it is a special case of the if-elsif-else structure. In other words, any control structure created by a switch statement can be duplicated directly by an if-then-else sequence without adding any complexity to the program.

The general form of a switch structure, using Java syntax, is

```plaintext
switch (_<expression>_)
    case <value1> : <statement_block_1>
    case <value2> : <statement_block_2>
    ...
    case <valuen> : <statement_block_n>
    [default : <statement_block_n+1>]
};
```

where the expression is first evaluated and then the expression is tested against the values until an equality is found. The appropriate statement block or blocks is then executed. Several key issues are pertinent to the implementation of the switch structure in a language.

1. **What types are allowable for the expression?** It is usually necessary to restrict the types allowed in the expression of the switch structure. In Java, for example, the expression is restricted
2. How are multiple values that result in the same action represented? Frequently, more than one value of the expression prescribe the same action. It is helpful if the syntax of the language permits multiple alternative values without duplication of the action specification. Some languages permit ranges and/or lists of values in the value field.

3. What if the value of the expression is not specified among the switch alternatives? Three actions are possible if the value of the expression is not listed. First, the switch construct may do nothing, acting like a null statement. Second, the action may be undefined. Finally, such an unspecified value might result in an error condition. Java chooses the first approach while Ada uses the third. An exhaustive list that avoids this situation is more easily obtained when the language has a specifier which covers all alternatives not specified before. In Java, this is the default alternative.

4. How are the statement blocks executed? There are two basic strategies used in the execution of the statement blocks. The first is to execute only the block associated with the value of the expression. The second strategy executes not only this block, but all blocks that follow it in the switch structure. Ada follows the first strategy, while Java and C++ follow the second. The option of limiting execution to a single statement block is available by ending a block with a break statement in Java and C++.

We conclude this section with a complete description of the conditional structures of Java.

Conditional Structures
Language: Java

if-statement ::=  if (<condition>)
  <statement_block>
[else
  <statement_block>]

switch-statement ::= switch (<expression>)
  { <switch-statement-alternative>
  {<switch-statement-alternative>}
  [default : <sequence_of_statements>]}

switch-statement-alternative ::= case <expression> : <sequence_of_statements>

Allowable types for switch expression: char, byte, short, int

All case expressions for switch expression: char, byte, short, int

If value of condition is not in case list, the default sequence of statements is performed, or, if there is no default clause, no action is performed.

If choices are not mutually exclusive, a compile-time error results

Discuss: Why does Java not permit the use of reals and string types in a switch statement?

Reinforce: For each of the following Java structures, solve the dangling else problem
by indicating what will be printed when initially \( x \) is 10.

(a) if \((x>10)\) 
\[
x:=x+1; \\
if (x<12) x:=x+2; \\
else \\
x:=x-1; \\
System.out.println(x); 
\]

(b) if \((x>10)\) 
\[
\{ \\
if (x<12) x:=x+2; \\
x:=x+1 \}
\]

Reinforce: Express the following Java \texttt{switch} structure as an \texttt{if}-\texttt{else} structure.

\[
\text{switch (x)} \\
\{ \\
  \text{case 1 : process1(); break; } \\
  \text{case 2 } \\
  \text{case 4 : process2(); break; } \\
  \text{case 6 } \\
  \text{case 9 : process3(); break; } \\
  \text{default : error(); } \\
\}
\]

Laboratory: What types are permitted for the control expression of a \texttt{switch} statement in your language?

Laboratory: How does your language react when an unspecified choice is evaluated for the expression controlling a \texttt{switch} statement?

6.2.2 Iterative structures

One of the most powerful features of a programming language is its ability to specify the repetition of a block of statements. Structures for doing this are called \texttt{iteration} structures. These structures are extremely important and raise many interesting issues. For convenience, we will refer to the block of statements to be repeated as the body of the iteration.

Nonterminating iteration The simplest form of iterative structure is a nonterminating iteration, which specifies the indefinite repetition of its body. Often, beginning programmers are cautioned to avoid such iterations because they never terminate and hence execute forever. In practice, however, there are frequently blocks of statements that execute in just this way. A communications program, for example, may have the following nonterminating structure:
do forever
  check for character sent
  if character is sent then process character
end do

Ada has a direct form that expresses a nonterminating iteration. In Ada, one writes

loop
  <sequence-of-statements>
end loop;

In languages like Java that have no special form for the nonterminating iteration, such an iteration can be specified by using a tested iteration with the condition always true.

**Pretest iteration** A fundamental capability of iterations is the ability to terminate based on the result of a test. Two factors can vary in the specification of this test: its placement and its logical direction. The placement of the test can either be before, after, or in the middle of the body of the iteration. We will call these three choices pretest, posttest, and in-test iterations. The logical direction of an iteration test specifies whether the test is a termination test, where a true condition indicates the iteration should halt, or a continuation test, where a false condition completes the iteration. In Java, for example, the while statement indicates a pretest iteration with a continuation test.

An important issue is the syntax for delimiting the statement block that serves as the body of the iteration. As with the if statement, Java assumes the body is one statement, which can be expanded into a compound statement using curly braces. Therefore, the Java form of a pretest loop is

```
while (<condition>)
{
  <statement_block>
}
```

If <condition> is false when the statement is begun, the <statement_block> will not be executed at all.

The technique of specifying a block of statements by special delimiters is used by Ada whose syntax for the pretest iteration is

```
while <condition> loop
  <sequence-of-statements>
end loop;
```

The word loop signifies the beginning of the body, and end loop marks the end.

**Posttest iteration** Whereas a pretest iteration provides a test before entry to the body of the iteration, a posttest iteration places the test after the body. Generally, pretest iteration is preferred over the posttest form, because posttesting permits one execution of the loop body before the test is first performed. In Java, the posttest structure available is the do-while construct. The form of the Java do-while iteration is

```
do
```
The do-while presents an inconsistency in Java because of its method of delimiting the iteration body. This is the only control structure for which Java abandons the compound statement convention in favor of the use of keyword delimiters. In this case, the words do and while are used to delimit the iteration body.

The equivalence of the pretest and posttest iteration constructs is obvious. The inclusion of both constructs in a language is for the convenience of the programmer, who may choose the one most natural for any given iterative structure.

**In-test iteration** Occasionally, it is desirable to perform the test for terminating an iteration neither before nor after the execution of the body but, rather, somewhere in the middle. This form is called an in-test iteration and is a situation where it is often argued that the use of a goto statement (an unconstrained jump out of the iteration body) is justified.

A much more restrictive construct than the goto can be used for this purpose, however—one whose only purpose is to exit from an iteration in the middle of its body. This type of construct has the advantage of permitting a flexible exit from a loop while not allowing indiscriminate branching within the program. This approach also avoids the need to use statement labels.

In Java, the in-test iteration is accomplished by using a break statement within the control of an if statement that tests the termination condition. The break statement results in the immediate exit from the immediately enclosing iteration body. As we saw before, the break statement also enables the immediate exit from a switch body in Java.

The break statement may optionally have an identifier appended, specifying the label of the statement block that is to be terminated. This enables the exit of an iteration body from within another iteration body that is nested within the first one. Consider the following example where a and b are int arrays stored in increasing order. The code determines whether there is any value that is present in both lists.

```java
outer: while(i<a.length)
{ j = 0;
  while(j<b.length)
  { if (a[i] < b[j])
    break;
  else if (a[i] == b[j])
    break outer;
  j++;
  }
  i++;
}
```

The first break, which has no appended identifier, exits from the innermost iteration and goes immediately to the statement that increments i. The second break exits from the outermost iteration, whose statement is labelled outer.

Another common language construct is one that exits from the present execution of the iteration body but, after exiting, executes the iteration test to determine whether the body will be executed again. In Java this is implemented by the continue statement. Java continue statements, like break statements, have an optional identifier to identify the iteration that is being
exited. In the preceding example, the statement

    break;

could have been replaced by

    {
        i++;
        continue outer;
    }

without changing the semantics of the code.

**Fixed count iteration** The oldest of the iteration structures is the **fixed count iteration**, which traces its roots back to the do loop of FORTRAN. This iteration structure is terminated after executing a specified number of times rather than until a specified condition occurs.

Fixed count iterations are controlled by a variable known as the **iteration control variable** (ICV). The general form of such an iteration is

    for <ICV> := <initial> to <final> step <increment> do <body>

Here, ICV is a variable, and initial, final, and increment are expressions of the same type as ICV. Based on this general form, we will address a number of important variations among languages in forming the fixed count iteration.

1. **What types are permitted for the ICV?** In some languages only integers are permitted. In others only numerics are permitted, including integers and reals. Pascal and Ada both permit integer, character and enumerated types, the same as those permitted in the control of a switch structure. These types are permitted, because they possess a built-in stepping function. In other words, each element has a natural successor. Real types do not possess this property and require explicit values for the increment if they are allowed.

2. **What is the scope of the ICV?** Some languages require the ICV to be a variable that is bound in the execution unit containing the iteration. Ada, however, takes a different approach. The scope of an ICV in Ada is the body of the iteration for which it is declared. This means its appearance in the iteration statement is equivalent to its declaration and binds it locally to a location. Upon completion of the iteration, the Ada ICV is no longer bound to that location.

3. **Can the ICV be modified within the iteration body?** The modification of the ICV within the iteration body is dangerous in that it disrupts the sequence of values specified at the beginning of the iteration. For this reason, some languages, such as Ada, disallow modification of the ICV. Other languages place no restrictions on changing the ICV.

4. **What is the value of the ICV after termination of the iteration?** There are four different responses languages give to this question. If the scope of the ICV is the iteration, as in Ada, the answer is obviously that the ICV no longer is bound to a location and hence has no value binding either. If the ICV maintains its location binding after termination of the iteration, it could be bound either to the value it had during the last iteration, to one increment beyond its value during the last iteration, or to an undetermined value. This last option means that, unlike with Ada, the ICV will have some value, but no guarantees are made as to what that value might be.

5. **When are the final and increment expressions evaluated?** This becomes an important
issue when variables in these two expressions are modified inside the iteration. For example, the following Pascal program fragment would raise this issue:

```pascal
define i := 1 to n do
    n := n + 1;
```

If the final value of `n` is re-evaluated each time the iteration body is executed, this iteration will run forever for `n` initially positive. It turns out that for Pascal, as for most languages, both the final and the increment expressions are evaluated once, prior to the initial entry into the iteration. The changing of `n` in the iteration body would, therefore, have no effect on the number of times the body is executed, and for positive `n`, the above fragment is equivalent to the simple statement

```pascal
n := 2 * n;
```

6. Is an increment other than successor permitted? Although an increment expression was specified in the general form, some languages do not allow such a specification. Pascal and Ada, for example, permit only an increment of one for a numeric ICV. Nonnumeric ICVs are required to be of types where each element has a defined successor, making the implied effect of an increment setting the ICV to the successor element within the type.

7. How is iteration backward through a range specified? In languages that permit increment specification, a negative increment is usually used to indicate backward iteration. Pascal, which has no explicit increment, replaces the keyword `to` with `downto` as in

```pascal
for i := 6 downto 1 do ...
```

Ada composes the initial and final expressions together into a range where initial must always be less than or equal to final. For the ICV to proceed backward through this range, the keywords in `reverse` must be appended--for example,

```pascal
for L in reverse 1..6 loop ...
```

8. Is transfer into the iteration permitted? Because the parameters for a fixed count iteration are evaluated and fixed when it is initially entered, branching to the interior of such an iteration without executing these initial evaluations can be very dangerous. Therefore, some languages, like Ada, disallow such transfers. Pascal, among others, allows these transfers to occur, though the results are highly unpredictable.

9. How is the iteration body delimited? As with other iterations, the two approaches are (1) to allow the body to be a compound statement, or (2) to use keywords to delimit the block of statements forming the body. Java, Pascal, and C++ follow the compound statement philosophy. Ada utilizes the keywords `loop` and `end loop` to delimit as before.

Java, and other languages in the C family, implement fixed-count iteration with a more general iteration form known as the `for` statement, which we will describe here.

The syntax of the Java `for` iteration is

```java
for ("[<init_part>]";["<boolean_expression>"];["<update_part>"])<iteration_body>
```
where the iteration_body can be a compound statement or a single statement.

The init_part can be either of two constructs. It can be a comma-separated list of statements or a variable declaration binding one or more variables. If the init_part is a list of statements, those statements are executed before the first execution of the iteration_body. If it is a variable declaration, the scope of that variable will be the iteration_body, and the variables declared are initialized before the body is executed for the first time. If a variable declared in the for statement is already bound in the block containing the for statement, a compile-time error occurs.

The boolean_expression, if present, is evaluated immediately prior to each execution of the iteration_body and if it evaluates to true or is not present, the iteration body will be executed one more time.

The update_part is a comma-separated list of statements, possibly empty, that is executed after each execution of the iteration_body and before the boolean_expression. The general process of the for statement structure can be described by

```
{<init_part>
 while (<boolean_expression>)
 {
   <iteration_body>
   <update_part>
 }
}
```

Note that an empty <init_part> and <update_part> makes the Java for equivalent to the while.

The most common use of the for iteration in Java is fixed-count iteration, though the form is more general. For example, the common iteration over int values from start to end is

```
for (lcv=start; lcv<=end; lcv++)
    do_something(lcv);
```

Using this form of the Java for, we answer the nine pertinent questions for fixed-count iterations that were listed earlier.

1. **What types are permitted for the ICV?** Because all three components are general, any Java type or class may be used for iteration control. For example, the following is a valid iteration:

```
for (double d=0.0; d<1.0; d=d+0.1) ...
```

2. **What is the scope of the ICV?** Java permits local variable declarations in the <init_part>. The scope of such a variable is the scope of the for statement. A variable declared in the containing block may also be used as an ICV, in which case its scope is the containing block.

3. **Can the ICV be modified within the iteration body?** The ICV can be modified without restriction in the iteration body.

4. **What is the value of the ICV after termination of the iteration?** The ICV, if its scope extends beyond the iteration body, will retain the value it had following the final execution of the <update_part>.

5. **When are the final and increment expressions evaluated?** These values may be changed
as the iteration progresses. For example,

```java
int inc = 1;
int n = 10;
for (int i=0; i<n; i=i+inc) {
    System.out.println(i);  // prints 3
    inc = inc + 1;  // prints 6
    n = n - 1;
}
```

Although such modifications of the final, increment, and ICV within the iteration are possible, they are obviously detrimental to code understandability.

6. Is an increment other than successor permitted? Absolutely.
7. How is iteration backward through a range specified? This is done by use of an appropriate <update_part>.
8. Is transfer into the iteration permitted? This is not permitted in Java.
9. How is the iteration body delimited? The iteration body is either a single statement, or it is delimited by curly braces ({}).

The Java `for` iteration can also be used as a nonterminating iteration when given in the form

```java
for (;;)
```

In this case, a break is required somewhere in the body of the iteration to enable termination.

Reinforce: Ada has no posttest iteration form directly built in to the language. Design one which would remain consistent with Ada's other iteration constructs.

Reinforce: Design the syntax for a completely general iteration structure that permits both pretest and posttest and both continuation and termination logic. Write the EBNF for your construct. Does your answer permit more than one test for the same iteration?

Reinforce: Give examples of applications where each of the following are natural iteration forms:
   a. Nonterminating
   b. Termination, pretest
   c. Continuation, pretest
   d. Termination, posttest
   e. Continuation, posttest
   f. Termination, in-test
   g. Continuation, in-test

Reinforce: Rewrite the following `for` loop as a do-while loop.

```java
for (index = 10; index>=5; index--)
    do something_or_other();
```
Laboratory:

a. Does your language permit unconditional branching into or out of the body of an iteration?
b. What is the value of the ICV after completion of an iteration?
c. When is the final expression evaluated in a fixed count iteration?
d. Does your language permit an ICV to be of a real type?
e. Does your language permit an iteration to have multiple exit tests?
f. Does your language permit modification of the ICV in the body of a fixed count iteration?

6.2.3 Unconstrained control statements

The most controversial of all control structures are those that are unconstrained—those that permit branching to any program unit without restriction. These are generally known as goto constructs, and their use is of questionable value, as we will discuss shortly. Nevertheless, many programming languages provide a goto statement. In this section, we will examine this simple, yet powerful, control construct.

The general format of the goto statement is

\[
goto \ <\text{statement-label}>;\]

FORTRAN contains alternate forms of the goto, but these are really just other ways of expressing a multialternative, switch-like structure. SNOBOL permits a goto or two to be attached to every statement as a suffix.

The interesting issues and variations with goto constructs arise when the format of labels and the impact of scope are considered.

Statement labels There is great variation in the way statement labels are provided. BASIC, for example, requires a label to be present for every statement. Other languages make labeling a statement optional. Optional statement labels are frequently separated from the statement by a colon. Ada is an exception, requiring that the label be enclosed between << and >>, since the colon is used in Ada to label iterations.

In order to better understand the use of labels and different languages’ approaches, we return to the data object model we used in Section 4.1. For our discussion here, we consider the data object to be a source program statement that is bound at load time to the location in memory where the corresponding executable instructions are stored. The approach most languages use for labels is illustrated by Figure 6.1. Here, the value is the element of the storage space where the instruction is located. This binding occurs at load time. The name binding occurs at compile time and binds the data object to an element of the label identifier space. This space may or may not be the same as the variable identifier space. In Java, C++, and Ada, the label and variable identifier spaces are the same. But languages often select label identifiers from the space of integers, as in Pascal and FORTRAN. Another consideration is whether the label type binding is made explicitly through a label declaration (Pascal) or implicitly through attachment of the label identifier to a statement (Java, C++, Ada, and FORTRAN). When a label is declared, there are implications for the scope of the label, which we will discuss later.
Figure 6.1 Statement labels as constants

Figure 6.2 illustrates the use of labels as variables. The best example of this is found in PL/I, which permits the declaration of an identifier to be a label variable. That variable can be bound to any legal label constant as its value, permitting such interesting activities as passing labels as parameters and forming arrays of labels. The languages SNOBOL and APL extend this idea even further by permitting calculated expressions to have their values assigned to labels. There is a price to be paid for this interesting extension--namely, the loss of program readability. We know that goto statements themselves can be detrimental to program readability, but when a single goto statement can branch to virtually any labeled statement--with the choice of statement dependent upon some nonlocal run-time action--the readability factor sinks to new lows. For this reason, the implementation of label variables is not included in most languages.
Scope issues There are several important issues related to the scope of labels. The first is the scope of the name binding to a labeled statement. In general, this follows the scoping rules for variables, that is, the binding holds in the present block and all contained blocks. Redefining a label identifier inside a nested block, if allowed by a language, could result in the "hole-in-scope" problem as with variables.

To illustrate the above point, consider the following Ada fragment.

```ada
OUTER: declare
  ...
INNER: declare
  ...
  <<INSIDE>> goto OUTSIDE;  --this is legal
  ...
end INNER;
  ...
<<OUTSIDE>> goto INSIDE;  --this is not legal
end OTHER;
```

The `goto` in the `INNER` block is legal because `OUTSIDE` is bound from the containing block. The `goto` in the `OUTER` block is not permitted because `INSIDE` is bound only in the context of the `INNER` block. By the way, note that `OUTER` and `INNER` are block labels, while `INSIDE` and
outside are statement labels. It is illegal to use block labels in an Ada goto statement.

One further type of scoping block beyond those that apply to variables can be logically defined for labels. This is the block of statements making up the body of a control structure. It is necessary to make these blocks scoping blocks for labels to prevent branching to the interior of a control structure's body without executing the test condition. For example, the Ada fragment below is illegal.

\[
\text{loop } \ldots \text{<<INSIDE>>} \ldots \text{end loop; } \ldots \text{goto INSIDE } --\text{illegal to branch into structure body}
\]

Similarly, branching to a statement inside the body of a conditional or any other iteration structure from outside that body is strictly prohibited in Ada. In defining the scope of labels, the bodies of control structures are thus treated the same as any other scoping block.

Another important observation can be made about the situation where a goto statement branches from inside a block to a statement in a containing block. This action is legal in most languages, but its implementation is not as simple as it might appear. Branching out of a block is actually a termination of that block and requires the removal of that block's activation record from the run-time stack. This may require popping several activation records if the branch is out of several layers of nested blocks as we shall see in Chapter 7.

Java has no unconstrained control statement. Its use of statement labels is for their reference in break and continue statements. Statement labels, while chosen from the same name space as all other Java identifiers, are not in conflict with other identifiers. This means, for example, a statement label and a variable, both with the same name, can exist within the same execution unit with no conflict.

The goto controversy The little goto statement has been the subject of a major controversy in the field of computer science, initiated by Dijkstra (1968) and rekindled by Rubin (1987). This controversy is actually about programming practices rather than programming languages, but inasmuch as language has an impact on practice, programming languages have become a part of this discussion. We limit our discussions to the impact the presence of the goto statement has on the capabilities of a language.

Three facts about control structures are important considerations here.

1. *Simple conditionals and goto statements are sufficient to replace any control structures.*

Each control structure we discussed in this chapter can be replaced by a construct using only simple conditionals and goto statements. For example, the Pascal while construct of the form

\[
\text{while <condition> do} \\
\text{<statement>};
\]

can be replaced by
10: if <condition> then
    begin
    <statement>;
    goto 10;
    end;

Several exercises at the end of this chapter require you to replace other control structures with
these two simple constructs.

2. The two-alternative conditional and pretest loop constructs are sufficient to replace any
control structure.

This result is far less obvious than the first, but it has been proven by Boehm and Jacopini
(1966). One consequence of this result is that a language without a goto could duplicate the pro-
grams written in a language containing the goto. In other words, there are no programs that
require the use of a goto for their construction.

3. The goto is the most powerful control structure.

This result has meaning only if the word powerful is defined. Kosaraju (1974) has proven
that the goto is the most powerful control structures in the sense that replacing a goto with
other structures might require additional variables, whereas replacing other structures with a
goto will never require additional variables. Based on this result we can say that programs
expressed without the goto are more complex than those expressed with it.

What are the implications of these three results for programming languages? First, a pro-
gramming language without a goto statement can express all the programs that can be expressed
by the same language with a goto added. Second, there are situations where programs can be
more simply represented by the use of a goto.

On the other hand, in the same way powerful automobiles or powerful weapons give
greater capability but are accompanied by greater danger, there is an increased danger with the use
of the goto. This danger is an increased ability to generate unreadable programs. Many computer
scientists believe the dangers of using the goto far outweigh the advantages inherent in its power.
Many programming language designers have reacted to this controversy by continuing to provide
a goto statement, while also providing a sufficiently rich set of weaker control structures to make
the usage of the goto unnecessary. This presents the programmer with the final choice of
whether or not to use the goto.

Discuss: How would the usefulness of a language be limited if it contained no control
structures?

Discuss: Why do you think Java’s do-while uses an approach to blocking that differs
from that of all other control structures in the language? What might be some nega-
tive consequences of this?

Discuss: What are the advantages and disadvantages of Java’s compound statement
philosophy of blocking control structures?

Discuss: Give an argument for permitting the modification of the ICV inside a loop
body.
Discuss: What are some reasons for requiring that labels be declared? What are some reasons for not doing so?

Discuss: Consider the Pascal code below:

```pascal
program GotoQuestion (input, output);
  label 99;
  procedure ReadUntil (match: integer);
    var potential: integer;
    begin
      while true do
        begin
          read(potential);
          if potential = match goto 99;
        end;
    end;
  begin
    ReadUntil(42);
  99:    writeln('Got a 42!!')
  end.
```

Not only does the goto above "break" a loop, it also "breaks" a procedure by jumping outside of that procedure. Discuss the legality, advisability, and ramifications of using mechanisms such as this.

Reinforce: Show how a simple conditional and a goto can be used to simulate the actions of

a. a two-alternative conditional (if-then-else)
b. a multi-alternative conditional (switch)
c. a pretest iteration
d. a posttest iteration
e. fixed count iteration

Reinforce: If the break was not available in Java, how might you simulate the action of a break statement?


Reinforce: In Basic, the on ... goto statement evaluates an expression in much the same way a switch statement does. For example, executing the statement

```
25 ON (X) GOTO 100, 200, 300
```

will cause the computer to jump to lines 100, 200 or 300 if \( x \) is either 1, 2, or 3. If \( x \) is none of these, no jump is done.

a. Rewrite the above statement as a switch statement in Java.
b. Rewrite the above statement as an if statement in Java.
Laboratory: Does your language permit an unconditional branch into the block of statements executed under control of a conditional?

6.3 Control Structures for Concurrent Execution

Concurrent control structures are those structures that allow multiple components of a program to execute at the same time. These control structures have taken many forms but can be grouped into two categories: those that represent extensions of classical control structures and those that represent unique or experimental designs.

It should be noted here that concurrency is often characterized at the procedural level rather than at the statement level. We will cover procedural concurrency when we discuss procedural abstraction in Chapter 9.

6.3.1 Concurrency as Semantic Extensions

6.3.1.1 Concurrent Blocks

The simplest concurrent control structure is called a concurrent block. This control structure is an extension of the concept of a block of statements, where each statement in the block executes in parallel. The concurrent block was introduced in 1963 by Dijkstra in the language CoPascal \[\text{ref}\]. In CoPascal, the keywords COBEGIN and COEND were used to define the boundaries of the concurrent block. Therefore, in the block

\begin{verbatim}
COBEGIN
  x := x + 2;
  y := sqrt(x) + 15;
  compute_quadratic(a,b);
COEND
\end{verbatim}

Each of the three statements -- two assignments and a function call -- would execute in parallel. This type of control structure illustrates the concepts of forking and joining, where concurrent execution forks into multiple threads, and then joins back together to form one thread of execution. In CoPascal, the COBEGIN statement represents the fork; the COEND statement is the joining statement. Note that, since not all statements between the COBEGIN and the COEND statements will terminate at the same time, it is possible to wait at the COEND for a while (even infinitely) until all statements terminate.

A more recent equivalent of CoPascal’s concurrent block statements is the \texttt{PAR} statement used in Occam \[\text{ref}\]. In Occam, the \texttt{PAR} statement introduces a section where all statements in that section are run in parallel. Occam also has a \texttt{SEQ} statement that forces sequential execution on a sequence of statements. Thus, we could give the following set of statements:

\begin{verbatim}
PAR
  SEQ
    x = x + 2
    y = sqrt(x) + 15
    compute_quadratic(a,b)
\end{verbatim}

In this fragment, the assignment statements would run as a group in parallel with the procedure call, but as a group, they would run one after the other.
It should be noted here that concurrent control structures typically ignore dependencies between data elements. In the CoPascal example above, it is not clear which “x” is used by the “sqrt” function -- the one from the previous statement or the one that existed before the COBEGIN block was started. Since the statements execute in parallel, we might expect the “x” from before the concurrent block will be used. However, unless data is explicitly protected, no assumptions should be made about the effects of manipulating data shared between concurrent pieces of a program. The Occam example solves this problem by executing the dependent statements in parallel. We cover these data issues in Chapter 9.

### 6.3.1.2 Concurrent Conditional Structures

A concurrent extension to the conditional structure was first suggested by Dijkstra [ref] and has been implemented by several languages. Since extending a simple conditional really does not make sense for concurrent execution, this extension extends the multialternative conditional as follows:

```
if <condition-1> -> <statement-sequence-1>
when <condition-2> -> <statement-sequence-2>
... 
when <condition-n> -> <statement-sequence-n>
fi
```

This construct differs in several ways from the if-elsif-else type of construct. The above form evaluates all conditions at the same time, while the if-elsif-else construct evaluates each condition in turn until it finds one that evaluates to true. The condition in the concurrent conditional that evaluates to true has its corresponding statement sequence executed. For the concurrent conditional, the conditions are typically referred to as guards.

A question remains as to what happens when two or more guards evaluate to true. If there are rules that govern the choice of the statement sequence to execute, this is said to be a deterministic concurrent conditional structure. Typical selection rules are to honor the first guard to evaluate to true or the first in the sequence of specifications to evaluate to true. If there are no such selection rules and the choice may be different for different executions of the same code, this type of structure is said to be a nondeterministic concurrent conditional structure. Ada uses a deterministic approach, one that we look at further in Chapter 18. Occam uses a nondeterministic approach.

### 6.3.1.3 Concurrent Iterative Structures

Iterative control structures can be extended in much the same way conditional structures are extended. Dijkstra also suggested the following syntax for this:

```
do
  when <condition-1> -> <statement-sequence-1>
  when <condition-2> -> <statement-sequence-2>
  ... 
  when <condition-n> -> <statement-sequence-n>
od
```
As with concurrent conditional structures, all conditionals in the iterative structure are evaluated in parallel. The statement sequence whose guard evaluates to true is executed. When more than one guard evaluates to true, a choice must be made, and this choice can be deterministic (as in Ada) or nondeterministic (as in Occam). The iterative structure repeats as long as at least one guard is true and terminates whenever none are true.

Discuss: If the “GOTO” structure is considered dangerous in sequential code, it can be outright damaging in concurrent code. Discuss why languages that implement concurrency never implement a unconstrained concurrent “goto” structure.

Discuss: Discuss the difficulties that concurrent while iterations might have that concurrent for iterations would not have.

6.3.2 Novel Concurrent Control Structures

Instead of simply extending the semantics of existing statements to accommodate concurrency, some languages have invented their own statements and their own semantics. Typically, the invented semantics adheres to the model of the language, and therefore the concurrent control structures are custom tailored for a certain language. We itemize a small sample below:

- **Data flow languages** typically allow all branches of conditionals and all iterations of loops to execute at the same time. Guards that evaluate to false shut down the execution of their corresponding sequence of statements. This type of execution can make for some very complicated execution patterns, and some languages curtail this by forcing forking and joining semantics to apply to conditionals or to loops.

- **Nonprocedural languages** allow as much concurrency as possible typically by allowing all components of programs to execute in parallel and relying on syntax to make various program block depend on the execution of other program blocks. Languages such as Lucid [ref] use keywords such as FOLLOWEDBY and AFTERWARD to impose block ordering on concurrency. All statements execute in parallel, except where directly specified by such restrictions.

- **Nonimperative language paradigms.** Several language paradigms are non based on the ordering of statements as the imperative paradigm is. Paradigms such as functional programming (see Chapter 11) and logic programming (see Chapter 12) are naturally concurrent in their basic concepts. This languages use no control structures to identify concurrent portions but might use constructs to tie pieces of a program together, as in nonprocedural programming.

6.4 Nondeterministic Control Structures

An important extension to the multialternative conditional has been suggested by Dijkstra [Dijkstra (1975)]. The general form proposed is
This construct differs in several ways from the if-elsif-else construct. The above form evaluates all conditions while the if-elsif-else evaluates only until a true condition occurs. Furthermore, in the switch where more than one of the conditions is true, the alternative whose statement sequence is executed is chosen nondeterministically. This means that there is no rule for choosing among several possibilities, and any one of them could be chosen. If none of the conditions is true, the statement is considered to be in error. The conditions are often called guards, and the entire construct is called a guarded command.

Although Ada does not implement this construct in the generality of Dijkstra's definition, it does have a specialized version of it which is used to implement concurrency control. We will study this in Chapter 18.

The nondeterministic conditional can be extended to form a nondeterministic iteration of the following form:

\[
do
  \text{when } \langle \text{condition-1} \rangle \rightarrow \langle \text{statement-sequence-1} \rangle \\
  \text{when } \langle \text{condition-2} \rangle \rightarrow \langle \text{statement-sequence-2} \rangle \\
  \ldots
  \text{when } \langle \text{condition-n} \rangle \rightarrow \langle \text{statement-sequence-n} \rangle
\od
\]

As before, the choice of statement sequence to be executed will be made nondeterministically from among those whose guard conditions are true. The iterative form repeats as long as at least one guard condition is true and terminates whenever none are true. Again, we will see this form implemented for concurrent control in Chapter 18.

Discuss: Discuss the types of problems that would have different results using a nondeterministic concurrent control structure. Are these algorithms to be avoided or can they be used to an advantage?
nonterminating iteration
termination test
continuation test
pretest iteration
posttest iteration
in-test iteration
fixed count iteration
iteration control variable (ICV)
goto