Chapter 8

# Nested Selections

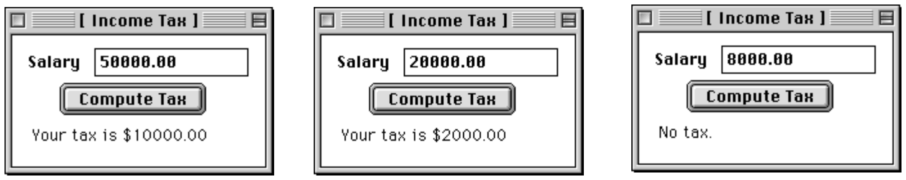
*The concept of nesting*

Nested boxes consist of an outer large box, inside which there is another box, inside which there is another box, and so on to the innermost box. Figure 3.1 is a set of nested boxes that illustrates how procedures are nested in modules, and data is nested in procedures and modules. Nesting is closely related to the concept of abstraction. If the outer large box has a lid, it hides the details of the remaining boxes that are within it. And hiding detail is the essence of abstraction.

## Nested IF statements

Figure 6.11 shows that an IF statement selects one of two alternative statement sequences, depending on the value of a boolean expression. Component Pascal allows either of those alternative statements to contain another IF statement. An IF statement contained in one of the alternatives of another IF statement is called a nested IF statement.

Figure 8.1 shows the dialog box for a program that inputs a salary and calculates an income tax from it. There is no tax at all if the salary is less than or equal to $10,000. Otherwise, the tax is 20% on the salary between $10,000 and $30,000 and 30% on the salary that is in excess of $30,000. You can see from the figure that a single IF statement is not sufficient to compute the tax, because there are three possible outcomes and the single IF statement shown in Figure 6.11 has only two alternatives.



Salary 50000.00
Your tax is $10000.00

Salary 20000.00
Your tax is $2000.00

Salary 8000.00
No tax.

**Figure 8.1**
A dialog box that requires more than two alternative computations.

The program in Figure 8.2 implements the dialog box with a nested IF statement. After the user enters that value for d.salary and clicks the compute button, the outer IF statement in procedure IncomeTax executes. If its boolean expression,

```
d.salary > minTaxable
```

is false, which it will be if the value of d.salary is less than or equal to 10,000.00, the statement sequence containing the nested IF statement is skipped, and the message string is set to the “no tax” message.

```
MODULE Pbox08A;
    IMPORT Dialog, PboxStrings;
    VAR
        d*: RECORD
            salary*: REAL;
            message-: ARRAY 64 OF CHAR
        END;

    PROCEDURE IncomeTax*;
        CONST
            lowRate = 0.20;
            highRate = 0.30;
            minTaxable = 10000.00;
            maxTaxable = 30000.00;
        VAR
            tax: REAL;
            taxString: ARRAY 32 OF CHAR;
    BEGIN
        IF d.salary > minTaxable THEN
            IF d.salary <= maxTaxable THEN
                tax := (d.salary - minTaxable) * lowRate
            ELSE
                tax := (maxTaxable - minTaxable) * lowRate + (d.salary - maxTaxable) * highRate
            END;
            PboxStrings.RealToString(tax, 1, 2, taxString);
            d.message := "Your tax is $" + taxString
        ELSE
            d.message := "No tax."
        END;
        Dialog.Update(d)
    END IncomeTax;

BEGIN
    d.salary := 0.0;
    d.message := ""
END Pbox08A.
```

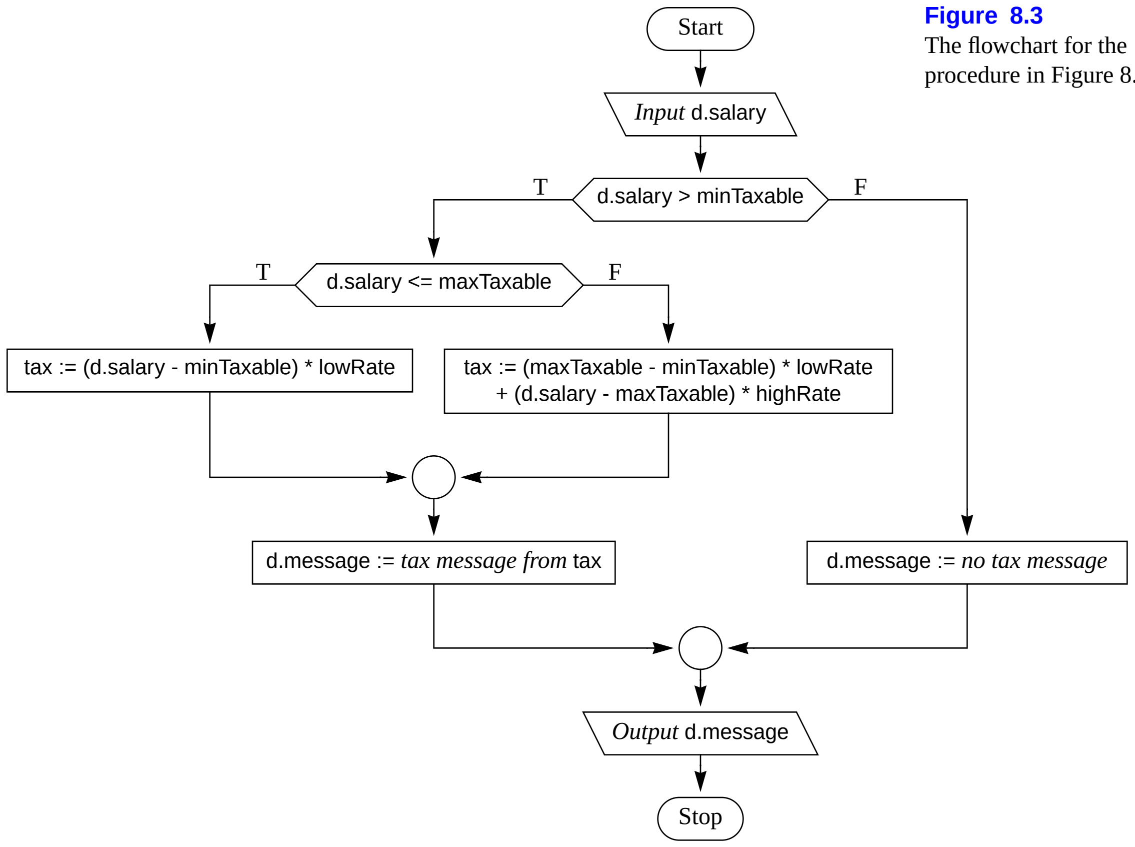
**Figure 8.2**
An income tax computation with a nested IF statement.

If the boolean expression is true, the nested IF executes. It evaluates the boolean expression

```
d.salary <= maxTaxable
```

If this boolean expression is true, it computes the tax according to the low rate, and if it is false according to the high rate on the excess beyond maxTaxable.

Figure 8.3 is the flowchart for the listing in Figure 8.2. It shows the inner IF statement nested in the true alternative of the outer IF statement. You can see from the



Start
*Input* d.salary
d.salary > minTaxable
T
F
d.salary <= maxTaxable
T
F
tax := (d.salary - minTaxable) * lowRate
tax := (maxTaxable - minTaxable) * lowRate + (d.salary - maxTaxable) * highRate
d.message := *tax message from* tax
d.message := *no tax message*
*Output* d.message
Stop

**Figure 8.3**
The flowchart for the procedure in Figure 8.2.

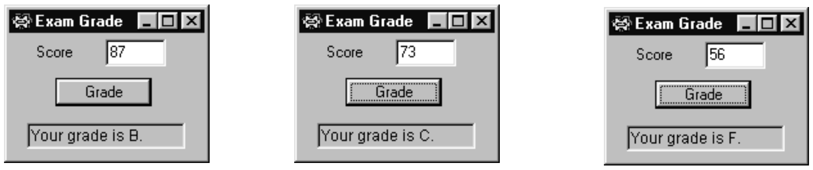
flowchart that the nested condition

```
d.salary <= maxTaxable
```

will never be evaluated if the outer condition is false. The flowchart also shows that each IF statement terminates with the circular collector symbol aligned vertically with its condition. Because this program contains two IF statements, its flowchart contains two circular collector symbols.

## IF statements with an ELSIF part

The previous program had the nested IF statement in the true alternative of the outer IF statement. Component Pascal allows you to nest an IF statement in either the true alternative or the false alternative of the outer IF statement. Figure 8.4 shows a dialog box that computes the letter grade from an integer score according to the traditional 10-point criteria. That is, a score of 90 or more is an A, between 80 and 89 is a B, between 70 and 79 is a C, between 60 and 69 is a D, and less than 60 is an F. The program shown in Figure 8.5 implements this dialog box using IF statements that are nested inside the false alternatives of the outer IF statements.



Score 87
Your grade is B.

Score 73
Your grade is C.

Score 56
Your grade is F.

**Figure 8.4**
A dialog box for computing a letter grade from an exam score.

```
MODULE Pbox08B;
    IMPORT Dialog;
    VAR
        d*: RECORD
            score*: INTEGER;
            message-: ARRAY 64 OF CHAR
        END;

    PROCEDURE TestGrade*;
    BEGIN
        IF d.score >= 90 THEN
            d.message := "Your grade is A."
        ELSE
            IF d.score >= 80 THEN
                d.message := "Your grade is B."
            ELSE
                IF d.score >= 70 THEN
                    d.message := "Your grade is C."
                ELSE
                    IF d.score >= 60 THEN
                        d.message := "Your grade is D."
                    ELSE
                        d.message := "Your grade is F."
                    END
                END
            END
        END;
        Dialog.Update(d)
    END TestGrade;

BEGIN
    d.score := 0;
    d.message := ""
END Pbox08B.
```

**Figure 8.5**
Conversion of an integer exam score into a letter grade.

You can nest IF statements to any level. The last IF statement in this program is nested three levels deep. Each IF statement is nested in the ELSE part of its outer IF statement.

Suppose the value of Score is 93. The boolean expression of the outer IF statement

```
d.score >= 90
```

would be true, and the “Grade of A” message would be output. The ELSE part of the outer IF statement would be skipped. Because the ELSE part is a single large nested IF statement, none of the other boolean expressions is ever tested.

Suppose the value of d.score is 70. The boolean expression of the outer IF statement would be false, and the ELSE part of the outer IF statement would execute. Now the boolean expression

```
d.score >= 80
```

of the second IF statement would be tested as false. So the ELSE part of the second IF statement would execute. This time, the boolean expression

```
d.score >= 70
```

would be true, and the “Grade of C” message would be output. The ELSE part of the third IF statement is skipped. Therefore, the D and F messages are not output.

This pattern of a succession of IF statements nested in the false alternatives occurs so frequently in practice that Component Pascal has a special ELSIF option to perform the equivalent processing with a single IF statement. The same procedure can be written with a single IF statement as:

```
PROCEDURE TestGrade*;
BEGIN
   IF d.score >= 90 THEN
      d.message := "Your grade is A."
   ELSIF d.score >= 80 THEN
      d.message := "Your grade is B."
   ELSIF d.score >= 70 THEN
      d.message := "Your grade is C."
   ELSIF d.score >= 60 THEN
      d.message := "Your grade is D."
   ELSE
      d.message := "Your grade is F."
   END;
   Dialog.Update(d)
END TestGrade;
```

You can think of ELSIF and the last ELSE as a list of conditions that starts with the first IF condition. The boolean expressions in the list are evaluated in order, starting with the first. When a boolean expression is false, the next one in the list is tested. The first boolean expression that tests true causes its alternative to execute and the rest of the ELSIF alternatives in the list to be skipped.

When deciding whether to use this feature of the IF statement, you must be careful to distinguish between nested IF statements and sequential IF statements, which are not nested. The following IF statements are sequential:

```
IF d.score >= 90 THEN
    d.message := "Your grade is A."
END;
IF d.score >= 80 THEN
    d.message := "Your grade is B."
END;
IF d.score >= 70 THEN
    d.message := "Your grade is C."
END;
IF d.score >= 60 THEN
    d.message := "Your grade is D.";
ELSE
    d.message := "Your grade is F.";
END
```

In this code fragment, suppose that d.score gets the value 70 from the dialog box. The first two boolean expressions would be false and the third one would be true. But after d.message gets the C message, the next IF statement would execute. Because d.score >= 60 is true, d.message would get the D message destroying the previously stored C message. The net result would be an erroneous output of

```
Your grade is D.
```

Figure 8.6 shows the difference in flow of control between three sequential IF statements and an IF statement with two ELSIF parts.

## Assertions and invariants

*Assertions*

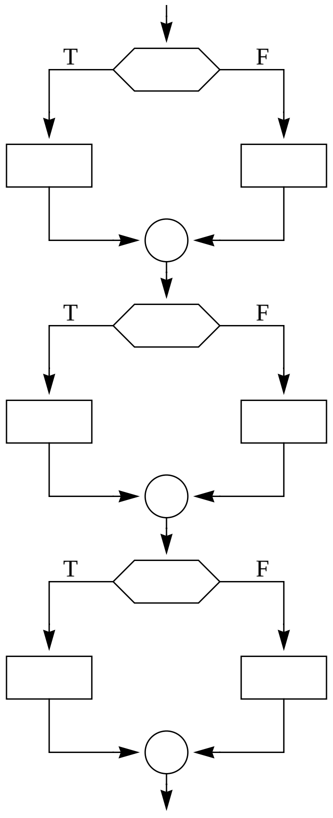
When complex IF statements are nested it is sometimes helpful to formulate assertions to keep track of what is happening in the program. An *assertion* is a condition that is assumed to be true at a given point in a program. One example of an assertion is the precondition $P$ in the Hoare triple $\{P\}S\{Q\}$. It is a condition that is assumed to be true for statement $S$ to execute correctly. If $P$ is true, then after $S$ executes, $Q$ is guaranteed to be true.

*Invariants*

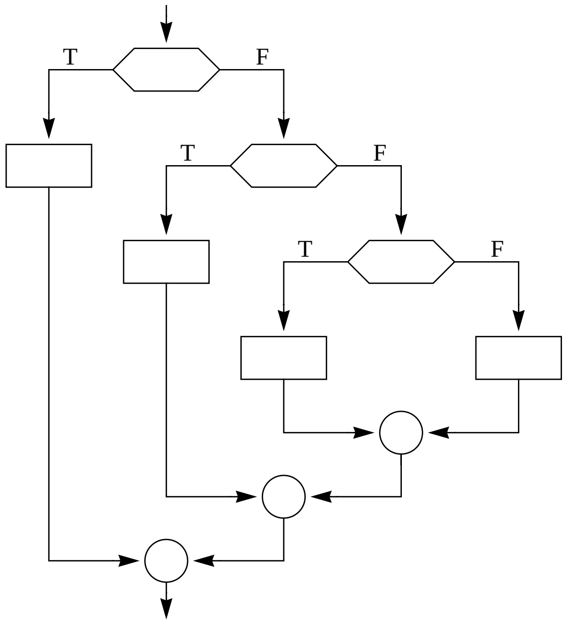
Another example of an assertion is the *invariant*. It differs from a precondition only by its physical placement in a program and by its use in program design. Whereas preconditions are assertions placed at the beginning point of a procedure, invariants are typically placed within procedure code. Furthermore, preconditions are frequently provided in the documentation of server modules as guidelines for use by client programmers. They are especially valuable if the programmer of the server is a different individual from the programmer of the client. Invariants, however, are usually intended as guidelines for a single programmer within a single procedure. They are hidden from the user of the module that contains the procedure.

*The ASSERT procedure*

Component Pascal provides assertions with the ASSERT procedure. ASSERT takes two parameters, a condition and an error number. The condition is a boolean expression. That is, it is an expression that evaluates to one of two possible values, true or false. When you execute ASSERT, it evaluates the condition. If the condition is true nothing happens and the program continues as if the ASSERT statement were not in the program. If the condition is false, however, the program terminates with a



**(a)** Sequential IF statements



**(b)** Nested IF statements

**Figure 8.6**
Flowcharts for sequential IF statements versus a single IF statement with two ELSIF parts.

trap. The error number is the number that appears in the trap window.

With Component Pascal, you can use the ASSERT procedure to implement preconditions and invariants. Recall from Chapter 7 that one of the programming style conventions for Component Pascal in the BlackBox framework is that the error numbers for precondition violations begin with integer 20. Similarly, error numbers for invariant violations should begin with integer 100.

**Example 8.1** The following code fragment is the nested IF statement of Figure 8.2 with invariants. Note that the pseudocode statements in italic may summarize several statements from the original program.

```
IF d.salary > minTaxable THEN
   IF d.salary <= maxTaxable THEN
      ASSERT((minTaxable < d.salary) & (d.salary <= maxTaxable), 100);
      tax := (d.salary - minTaxable) * lowRate
   ELSE
      ASSERT(d.salary > maxTaxable, 101);
      tax := (maxTaxable - minTaxable) * lowRate + (d.salary - maxTaxable) * highRate
   END;
   d.message := tax message from tax
ELSE
   ASSERT(d.salary <= minTaxable, 102);
   d.message := no tax message
END;
```

To see how invariants are formulated, we will begin with the simplest invariant, which can be found just before the statement

d.message := *no tax message*

What condition must be true at this point in the program? In other words, what condition must be true just before this assignment statement executes? The boolean expression of the outer IF statement must be false. But if the expression

d.salary > minTaxable

is false, the expression

d.salary <= minTaxable

must be true, which is the invariant shown in the code fragment.

The next invariant we will consider is the one just before the statement

tax := (d.salary - minTaxable) * lowRate

Why must d.salary be greater than minTaxable and less than or equal to maxTaxable at that point in the program? Because to arrive at that point, the boolean expression of the outer IF statement must be true. Then the boolean expression of the nested IF statement also must be true. The invariant

(minTaxable < d.salary) & (d.salary <= maxTaxable)

is simply reflecting those two conditions.

The remaining invariant is just before the statement

tax := (maxTaxable - minTaxable) * lowRate + (d.salary - maxTaxable) * highRate

To arrive at this point, the boolean expression of the outer IF statement must be true and the boolean expression of the nested IF statement must be false. Therefore, to get to this point in the program, d.salary must satisfy

```
(d.salary > minTaxable) & (d.salary > maxTaxable)
```

So why does the implementation of the invariant in the code fragment

```
ASSERT(d.salary > maxTaxable, 101)
```

seem to ignore the fact that d.salary must be greater than minTaxable?

The answer to this question involves the concept of strong versus weak invariants. One invariant is stronger than another if it places greater limits on the possible values of a variable. In general, stronger invariants are more helpful in analysis of logic than weaker ones, because they give you more information. Suppose you ask your teacher for your score on an exam. If she says, “You scored between 50 and 80,” she is not giving as much information as if she says, “You scored between 73 and 75.” The second statement places a greater limitation on the possible values of your exam score and, therefore, gives you more information.

*Strong invariants*

In this example,

```
d.salary > minTaxable
```

is a valid invariant, because it is a condition guaranteed to be true at this point in the program. However,

```
d.salary > maxTaxable
```

is stronger because it places a greater limitation on the possible values of d.salary.

One way of visualizing strong invariants is with the number line. Figure 8.7 shows the regions of the real number line corresponding to each of the preceding conditions. Recall from mathematics that the AND operation corresponds to the intersection of the regions, while the OR operation corresponds to the union of the regions. The intersection of these two regions is simply the region for

```
d.salary > maxTaxable
```

by itself, which is the stronger invariant.

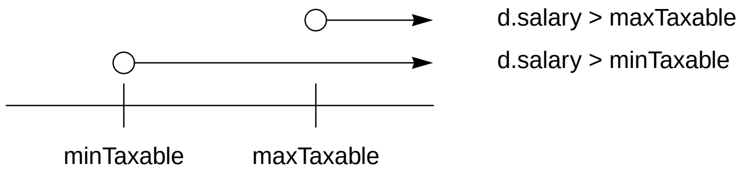


**Figure 8.7** The real number line showing the conditions d.salary > maxTaxable and d.salary > minTaxable.

One purpose of an invariant is to document your analysis of what condition you calculate should be true at a given point of your program. If your analysis is correct, a call to the ASSERT procedure should do nothing. That is, in fact, what the ASSERT procedure does. If the boolean condition in the ASSERT procedure call is true, nothing happens and the program continues to execute. If the boolean condition is false, however, the ASSERT procedure causes the program to abort with a trap. Why would a programmer ever want his program to abort with a trap? He never would! So why would anyone ever put an ASSERT call in his program?

*The purpose of a call to ASSERT to implement an invariant*

The primary purpose of a call to ASSERT to implement an invariant is for testing the correctness of your program. If the analysis of your program is correct, your assertions will never trigger a trap and all will be well for you and the users of your software. But if you make an error in your analysis you will have an error in your program and it will not execute correctly. The trap will show you where your analysis of what should be true at that point of the program is not true after all. You can then correct the program. Better to have a controlled abort of your program when you are testing it than to release it for the users with an error in the program.

To write programs that work correctly, you must be able to analyze the logic of the statements you write. Invariants will help you to think through the logic of your programs. In the beginning, it may seem that invariants make things more complicated than necessary. But after some practice, you will find that you can formulate invariants in your mind as you write your programs. That ability will make it easier for you to write correct programs. Occasionally, it may help to write an ASSERT call in a program to make the program easier to understand.

**Example 8.2** Consider the following code fragment, where age is a variable of type INTEGER.

```
IF age > 65 THEN
    Statement 1
ELSIF age > 21 THEN
    Statement 2
ELSE
    Statement 3
END
```

The logic in this code is identical to that in Figure 8.6(b), where the nesting was consistently in the false part of the IF statements. What are the strongest invariants you can formulate before each statement?

For Statement 1, the condition age > 65 must be true. That is the strongest invariant you can formulate at this point of the program.

For Statement 2, the boolean expression of the outer IF statement, age > 65 must be false. In other words, age <= 65 must be true. Furthermore, the boolean expression of the ELSIF condition, age > 21, also must be true. So the strongest invariant at this point is

```
(21 < age) & (age <= 65)
```

which corresponds to the intersection of the two regions in Figure 8.8.

For Statement 3, both boolean expressions must be false; that is, age <= 65 and age <= 21 must be true. The strongest invariant is

```
Age <= 21
```

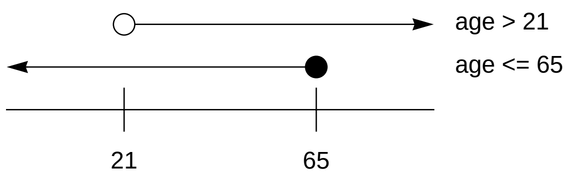


**Figure 8.8**
The number line showing the two conditions age > 21 and age <= 65.

which corresponds to the intersection of the two regions in Figure 8.9. The final code fragment that implements the strongest invariants is

```
IF age > 65 THEN
    ASSERT(age > 65, 100);
    Statement 1
ELSIF age > 21 THEN
    ASSERT((21 < age) & (age <= 65), 101);
    Statement 2
ELSE
    ASSERT(age <= 21, 102);
    Statement 3
END
```

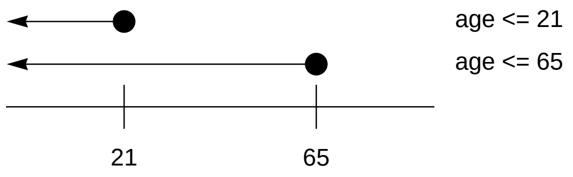


**Figure 8.9**
The number line showing the two conditions age <= 21 and age <= 65.

## Dead code

*Dead code*

Something to avoid when you write nested IF statements is dead code. *Dead code* is a statement that cannot possibly execute. If you ever discover dead code in your own program or in a program that someone else wrote, you can be sure that it was unintentional. Because dead code never executes, there is never a reason to put it in a program except by mistake. Formulating the strongest invariant can help you discover dead code.

*Typical purpose of a call to HALT*

Component Pascal provides a procedure that we will use to indicate dead code. When you execute procedure HALT, the program always terminates with a trap, regardless of the value of any boolean expression. Like the ASSERT procedure, the HALT procedure is used for testing large programs. If a programmer wants to view the values of the variables at a given point of a program, she can insert a call to HALT. When the program reaches that point, it will terminate and show the trap window including the values of all the variables at the time the program was interrupted.

The procedure call

```
HALT(100)
```

is logically equivalent to

```
ASSERT(FALSE, 100)
```

Because the boolean expression in the above ASSERT call is always false, its execution would always generate a trap. But this is precisely the behavior of the call to HALT. The following examples use a call to HALT to indicate the strongest invariant of dead code. In the same way that an ASSERT call with the strongest invariant will not affect the execution of a program, a HALT call with dead code will not affect a program. Because dead code never executes, the HALT procedure will never be called.

*The purpose of a call to HALT in this chapter*

Keep in mind, however, that this use of HALT would never be found in production code because a programmer would never willfully have dead code in her program. The following examples are designed to teach you skill in identifying dead code, so you can root it out of your programs.

**Example 8.3** Consider the following code fragment:

```
IF quantity < 200 THEN
    Statement 1
ELSIF quantity >= 100 THEN
    Statement 2
ELSE
    Statement 3
END
```

Statement 3 can never execute regardless of the value of quantity. To see why, try to formulate a strong invariant at the point just before Statement 3. To get to that point in the program, you must have quantity >= 200 because the first boolean expression must be false. You must also have quantity < 100 because the second boolean expression also must be false. But it is impossible to have quantity greater than or equal to 200 and less than 100 at the same time. So Statement 3 can never execute and is dead code. The code fragment with the strongest invariants implemented using calls to ASSERT and HALT is

```
IF quantity < 200 THEN
    ASSERT(quantity < 200, 100);
    Statement 1
ELSIF quantity >= 100 THEN
    ASSERT(quantity >= 200, 101);
    Statement 2
ELSE
    HALT(102);
    Statement 3
END
```

∎

Do not conclude from this example that dead code is always the last statement in a sequence of ELSIF parts. You must analyze each situation afresh. The general strategy to determine the strongest invariant at a given point is to list the boolean conditions that must be true. This may involve taking the NOT of some expressions if the nesting is in the false part of an IF statement. The intersection of the corresponding regions represents the strongest invariant. If the intersection at a given point is empty, the statement at that point is dead code.

**Example 8.4** Consider the following code fragment, where n has type INTEGER.

```
IF (15 <= n) & (n < 20) THEN
    IF (n > 10) THEN
        Statement 1
    ELSE
        Statement 2
    END
ELSE
    Statement 3
END
```

What is the strongest invariant you can formulate at each statement? The first step is to draw a sketch of the number line with the integer values in their proper order, as in Figure 8.10.

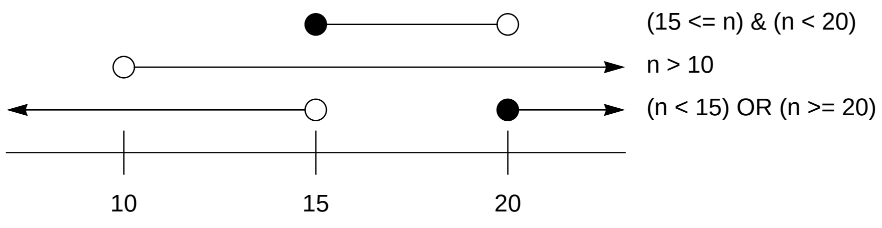


**Figure 8.10** The number line for Example 8.4.

For Statement 1, you can see from the figure that the intersection of the top two lines corresponds to both boolean expressions being true. The strongest invariant is

```
(15 <= n) & (n < 20)
```

For Statement 2, n must be less than or equal to 10 and between 15 and 20, which is impossible. So Statement 2 is dead code.

For Statement 3, the first boolean expression must be false. From De Morgan's law it follows that the strongest invariant is

```
(n < 15) OR (n >= 20)
```

This corresponds to the third region of Figure 8.10, which is that part of the number line not included in the first region. The code with all the invariants implemented with calls to ASSERT and HALT is

```
IF (15 <= n) & (n < 20) THEN
    IF (n > 10) THEN
        ASSERT((15 <= n) & (n < 20), 100);
        Statement 1
    ELSE
        HALT(101);
        Statement 2
    END
ELSE
    ASSERT((n < 15) OR (n >= 20), 102);
    Statement 3
END
```
■

## Using nested IF statements

One of the most common problems beginning programmers have is a failure to recognize the appropriateness of the logic characterized by a sequence of ELSIF parts in an IF statement.

**Example 8.5** Suppose you need to perform three different computations depending on the value of weight, a real variable. The following code:

```
IF weight > 150.0 THEN
    Statement 1
END;
IF (weight > 50.0) & (weight <= 150.0) THEN
    Statement 2
END;
IF (weight <= 50.0) THEN
    Statement 3
END
```

is not as efficient as the equivalent IF statement with a sequence of ELSIF parts:

```
IF weight > 150.0 THEN
    Statement 1
ELSIF weight > 50.0 THEN
    Statement 2
ELSE
    Statement 3
END
```

For example, suppose weight has a value of 200.0. In the first code fragment, every boolean expression must be evaluated because the IF statements are sequential. But in the second fragment, only the first boolean expression is evaluated because the sequence of ELSIFs is skipped. ■

Another tendency when programming with ELSIF logic is to include an unnecessary redundant test at the end.

**Example 8.6** The following code fragment has a redundant test.

```
IF price > 2000 THEN
    Statement 1
ELSIF price > 1000 THEN
    Statement 2
ELSIF price <= 1000 THEN
    Statement 3
END
```

The last boolean expression is redundant. In the following code fragment, you can assert that price <= 1000 when Statement 3 executes.

```
IF price > 2000 THEN
    Statement 1
ELSIF price > 1000 THEN
    Statement 2
ELSE
    Statement 3
END
```

This code fragment executes exactly the same as the previous one, but without the extra test. The redundant test should not be included. ▮

## ★ The guarded command if statement

The guarded command **if** statement can have more than just two guards. For example an **if** statement with four guards has the form

```
if  $B1 \rightarrow S1$
[]  $B2 \rightarrow S2$
[]  $B3 \rightarrow S3$
[]  $B4 \rightarrow S4$
fi
```

Each one of the $B$'s is a boolean guard that must be true for the corresponding statement sequence $S$ to execute. The behavior of the GCL **if** statement is quite different from the CP IF statement in two respects.

First, the CP IF statement has an optional ELSE part. Suppose the IF statement does not have an ELSE part, the condition in the IF part is not true, and none of the conditions in any of the ELSIF parts are true either. Then each condition will be tested, none of the statement sequences will execute, and execution will continue with the statement sequentially following the IF statement. However, suppose that the above GCL **if** statement executes when none of the guards are true. Then the statement aborts, which is the equivalent of a program crash or a trap in CP. In other words, GCL has nothing equivalent to the ELSE part, and requires at least one of the guards to be true to avoid a program abort.

*The if statement aborts when none of the guards are true.*

**Example 8.7** Suppose you want to put the values of $x$ and $y$ in order so that $x$ is guaranteed to be less than or equal to $y$. The GCL statement

```
if  $x > y \to x, y := y, x$
fi
```

works correctly if, for example, the initial state is ($x$, 7), ($y$, 3). In that case, the guard $x > y$ is true and the values are exchanged making the final state ($x$, 3), ($y$, 7). However, if the initial state is ($x$, 4), ($y$, 9) then no guards are true when the **if** statement executes and the program aborts. ■

A second difference between IF and **if** is the order in which the conditions are evaluated. In CP, the conditions are evaluated in order starting with the condition of the IF, then the condition of the first ELSIF if necessary, then the condition of the second ELSIF if necessary, and so on. In GCL, however, you should visualize *all* the guards being evaluated at the same time. If no guard is true the statement aborts. If one guard is true its corresponding statement sequence executes. But if more than one guard is true the computer randomly picks the statement sequence of a true guard to execute. In this case, it may be impossible to predict the exact outcome of the computation.

*The if statement selects at random when more than one guard is true.*

**Example 8.8** Suppose you write the processing of Example 8.6 in GCL as

```
if  $price > 2000 \to S1$
[]  $price > 1000 \to S2$
[]  $price \le 1000 \to S3$
fi
```

This translation from CP to GCL may seem plausible, but it is not correct. There is no problem if the initial state is (*price*, 500), which guarantees that *S*3 will execute. Nor is there a problem if the initial state is (*price*, 1500), which guarantees that *S*2 will execute. With both initial states exactly one guard is true so that the corresponding statement sequence can be determined. Suppose, however, the initial state is (*price*, 2500), when the CP statement in Example 8.6 guarantees that *Statement 1* will execute. The problem is that the above GCL statement has both guards $price > 2000$ and $price > 1000$ true and so will randomly pick either *S*1 or *S*2 to execute. ■

So how do you translate a CP IF statement to a GCL **if** statement? You simply use the strongest invariant as the guard.

*To translate from CP to GCL use the strongest invariant as the guard.*

**Example 8.9** The processing of Example 8.6 is correctly written in GCL as

```
if  $price > 2000 \to S1$
[]  $1000 < price \le 2000 \to S2$
[]  $price \le 1000 \to S3$
fi
```

■

## Exercises

**1.** **(a)** What is an assertion? **(b)** Name two kinds of assertions. **(c)** What is dead code?

**2.** Draw the flowcharts for the following code fragments.

**(a)**
```
IF Condition 1 THEN
    IF Condition 2 THEN
        Statement 1
    ELSE
        Statement 2
    END
ELSE
    Statement 3
END
```

**(b)**
```
IF Condition 1 THEN
    IF Condition 2 THEN
        Statement 1
    ELSE
        Statement 2
    END ;
    Statement 3
ELSE
    Statement 4
END
```

**(c)**
```
IF Condition 1 THEN
    Statement 1 ;
    IF Condition 2 THEN
        Statement 2
    END
ELSE
    Statement 3
END
```

**(d)**
```
IF Condition 1 THEN
    Statement 1
ELSE
    IF Condition 2 THEN
        Statement 2
    END ;
    Statement 3
END
```

**3.** Rewrite the following code fragments with the correct indentation and draw their flow-charts.

**(a)**
```
IF Condition 1 THEN
IF Condition 2 THEN
Statement 1
ELSE
Statement 2
END
END
```

**(b)**
```
IF Condition 1 THEN
IF Condition 2 THEN
Statement 1
END
ELSE
Statement 2
END
```

**4.** Rewrite the following code fragments with the correct indentation and draw their flow-charts.

**(a)**
```
IF Condition 1 THEN
IF Condition 2 THEN
IF Condition 3 THEN
Statement 1
ELSE
Statement 2
END
ELSE
Statement 3
END
END
```

**(b)**
```
IF Condition 1 THEN
IF Condition 2 THEN
IF Condition 3 THEN
Statement 1
ELSE
Statement 2
END
END
ELSE
Statement 3
END
```

**(c)**
```
IF Condition 1 THEN
Statement 1
END ;
IF Condition 2 THEN
Statement 2
ELSE
Statement 3
END
```

**(d)**
```
IF Condition 1 THEN
Statement 1
ELSIF Condition 2 THEN
Statement 2
ELSE
Statement 3
END
```

**5.** Rewrite the following code fragment with only one IF statement. Your revised code fragment must perform the same processing as the original one.

```
IF Condition 1 THEN
    IF Condition 2 THEN
        Statement 1
    END
END;
Statement 2
```

**6.** The following code fragment makes four comparisons. Simplify it so that only two comparisons are needed. age is a variable of type INTEGER.

```
IF age > 64 THEN
    Statement 1
END;
IF age < 18 THEN
    Statement 2
END;
IF (age >= 18) & (age < 65) THEN
    Statement 3
END
```

**7.** Determine the output, if any, of the following code fragment. h, m, and w are variables of type INTEGER. Hint: Rewrite with correct indentation first.

```
IF h > m THEN
IF w > m THEN
StdLog.Int(m)
ELSE
StdLog.Int(h)
END
END
```

**(a)** Assume h = 10, m = 3, and w = 4.
**(b)** Assume h = 10, m = 20, and w = 15.
**(c)** Assume h = 10, m = 5, and w = 3.

**8.** Determine the output, if any, of the following code fragment. x, y, z, and q are variables of type INTEGER. Hint: Rewrite with correct indentation first.

```
IF x > y THEN
StdLog.Int(y)
ELSIF x > z THEN
IF x > q THEN
StdLog.Int(q)
ELSE
StdLog.Int(x)
END
END
```

**(a)** Assume x = 10, y = 5, z = 0, and q = 1.
**(b)** Assume x = 10, y = 20, z = 5, and q = 1.
**(c)** Assume x = 10, y = 10, z = 12, and q = 5.
**(d)** Assume x = 10, y = 5, z = 20, and q = 15.

**9.** Write the strongest possible invariants just before each statement in the following code fragments. Assume that num is a variable of type INTEGER.

**(a)**
```
IF num < 23 THEN
   IF num >= 15 THEN
      Statement 1
   ELSE
      Statement 2
   END
ELSE
   Statement 3
END
```

**(b)**
```
IF num >= 50 THEN
   Statement 1
ELSIF num >= 25 THEN
   Statement 2
ELSE
   Statement 3
END
```

**(c)**
```
IF num >= 60 THEN
   Statement 1
ELSIF num < 80 THEN
   Statement 2
END
```

**(d)**
```
IF (num < 30) OR (num > 40) THEN
   Statement 1
ELSIF num < 35 THEN
   Statement 2
ELSE
   Statement 3
END
```

**10.** Write the strongest possible invariant just before each statement in the code fragment. Use the HALT procedure just before each statement that is dead code. Assume that num is a variable of type INTEGER.

**(a)**
```
IF num < 70 THEN
    IF num >= 80 THEN
        Statement 1
    ELSE
        Statement 2
    END
ELSE
    Statement 3
END
```

**(b)**
```
IF num >= 45 THEN
    Statement 1
ELSIF num <= 35 THEN
    Statement 2
ELSIF num >= 55 THEN
    Statement 3
ELSE
    Statement 4
END
```

**(c)**
```
IF num > 35 THEN
    Statement 1
ELSIF num > 45 THEN
    Statement 2
ELSE
    Statement 3
END
```

**(d)**
```
IF (num < 5) OR (num > 9) THEN
    Statement 1
ELSIF (5 < num) & (num < 9) THEN
    Statement 2
ELSE
    Statement 3
END
```

**(e)**
```
IF (num < 40) OR (num > 50) THEN
    Statement 1
ELSIF num > 30 THEN
    Statement 2
ELSE
    Statement 3
END
```

**(f)**
```
IF (40 <= num) & (num <= 50) THEN
    Statement 1
ELSIF (num < 42) OR (num > 48) THEN
    Statement 2
ELSE
    Statement 3
END
```

**11.** Write the **if** statement in Example 8.7 so that it executes correctly with any initial state.

**12.** For the GCL **if** statement

```
if  $age < 18 \rightarrow S1$
[]  $age \le 21 \rightarrow S2$
[]  $age < 65 \rightarrow S3$
fi
```

tell which statements could possibly execute for each of the following initial states.

**(a)** (*age*, 10) **(b)** (*age*, 19) **(c)** (*age*, 21) **(d)** (*age*, 40) **(e)** (*age*, 70)

**13.** For the GCL **if** statement

```
if  $j < 40 \rightarrow S1$
[]  $20 \le j < 60 \rightarrow S2$
fi
```

tell whether the statement will abort and if not, which statements could possibly execute for each of the following initial states.

**(a)** $(j, 10)$ **(b)** $(j, 30)$ **(c)** $(j, 50)$ **(d)** $(j, 70)$

**14.** Translate each code fragment in Exercise 9 into a single GCL **if** statement.

## Problems

**15.** Write a program to input three integers in a dialog box and print them in descending order on the Log. Your program must contain no local or global variables other than the ones for input in the dialog box. It must use no more than five comparisons and must work correctly even if some of the integers are equal.

**16.** Write a program to input three integers in a dialog box and output the number that is neither the smallest nor the largest in an output field of the dialog box. If two or more of the numbers are equal output that number.

**17.** Write a program to input two integers in a dialog box and output to the dialog box either the larger integer or a message stating that they are equal.

**18.** A salesperson gets a 5% commission on sales of $1000 or less, and a 10% commission on sales in excess of $1000. For example, a sale of $1300 earns him $80; that is, $50 on the first $1000 of the sale and $30 on the $300 in excess of the first $1000. Write a program that inputs a sales figure in a dialog box and outputs the commission to the dialog box. Output an error message if the user enters a negative sales figure.

**19.** The fine for speeding in a 45 MPH zone is $10 for every mile per hour over the speed limit for speeds from 46 to 55 MPH. It is $15 for every additional mile per hour between 56 and 65 MPH. It is $20 for every additional mile per hour over 65 MPH. For example, the fine for driving 57 MPH is $100 for the first 10 MPH plus $30 for the 2 MPH in excess of 55 MPH, for a total of $130. Write a program that inputs the speed in a dialog box as an integer and outputs the fine, or a message that there is no fine, to the dialog box. Output an error message if the user enters a negative speed. Use the smallest possible number of comparisons.

| **Temperature *T*** | **Message** |
|---|---|
| $90 \le T$ | Go swimming |
| $80 \le T < 90$ | Play tennis |
| $70 \le T < 80$ | Study |
| $60 \le T < 70$ | Go to sleep |
| $T < 60$ | Go to Hawaii |

**Figure 8.11**
The table for Problem 21.

**20.** Design a dialog box that has two input fields—an integer field for the temperature and a check box labeled Humid—and one output field. If the temperature is greater than 85 output the message “It is muggy” if the check box is checked or “Dry heat” if the box is not checked. Otherwise output “Cool man”.

**21.** Write a program to input the temperature (integer value) in a dialog box, then output the appropriate message for a given value of temperature to the dialog box, as the table in Figure 8.11 shows. Use the smallest possible number of comparisons.

**22.** The price per Frisbee depends on the quantity ordered, as the table in Figure 8.12 indicates. Write a program to input the quantity requested from a dialog box and output the total cost of an order, including a 6.5% sales tax, to the dialog box. Output an error message if a negative quantity is entered.

| Quantity | Price per Frisbee |
|---|---|
| 0 – 99 | $5.00 |
| 100 – 199 | 3.00 |
| 200 – 299 | 2.50 |
| 300 or more | 2.00 |

**Figure 8.12**
The price schedule for Problem 22.

**23.** You are eligible for a tax benefit if you are married and have an income of $30,000 or less, or unmarried and have an income of $20,000 or less. Design a dialog box that asks for the user’s marital status (check box) and income (real), then outputs a message in the dialog box stating whether the user is eligible for the tax benefit. Output an error message if negative input is entered.

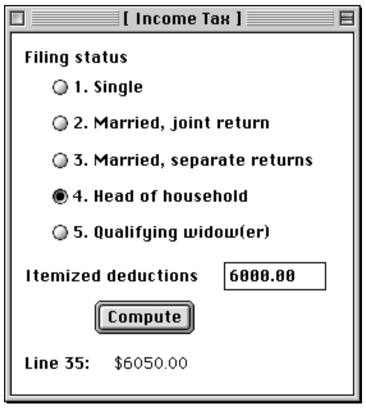
**24.** A year is a leap year if it is divisible by 4 but not by 100. The only exception to this rule is that years divisible by 400 are leap years. Design a dialog box that asks the user to enter a positive integer for the year and displays a message that states whether the year is a leap year.

**25.** The following statements are from the United States Department of Internal Revenue Form 1040 for 1997:

Enter on line 35 the larger of your itemized deductions or the standard deduction shown below for your filing status.

- Single—$4,150
- Married filing jointly or Qualifying widow(er)—$6,900
- Head of household—$6,050
- Married filing separately—$3,450.

Write a program that implements the dialog box of Figure 8.13 to output the value for line 35. Note that the first radio button is labeled 1 for the user, but should have a level number of 0 in your program. Output an error message on Line 35 if the amount entered for the standard deduction is negative.



[ Income Tax ]

Filing status

1. Single

2. Married, joint return

3. Married, separate returns

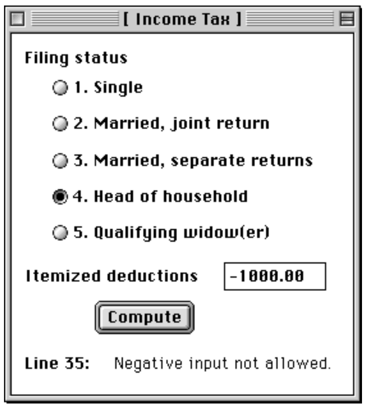
● 4. Head of household

5. Qualifying widow(er)

Itemized deductions 6000.00

Compute

Line 35: $6050.00



[ Income Tax ]

Filing status

1. Single

2. Married, joint return

3. Married, separate returns

● 4. Head of household

5. Qualifying widow(er)

Itemized deductions -1000.00

Compute

Line 35: Negative input not allowed.

**Figure 8.13**
The dialog box for Problem 25.

**26.** Rewrite module Pbox08B in Listing 8.5 using a CASE statement instead of an IF statement. The dialog box should appear as in Figure 8.4 without any radio buttons. Use the fact that if d.score is in the range 70–79, for example, then d.score DIV 10 is 7.