Figure 5.5
Accumulating a Sum

```
TotalPay  Pay
10.0     20.0

Second iteration
TotalPay  Pay
20.0     30.0

Third iteration
TotalPay  Pay
50.0     35.0

TotalPay
65.0
```

adds the current value of Pay to the sum being accumulated in TotalPay, thereby increasing the value of TotalPay with each loop iteration. Figure 5.5 traces the effect of repeating this statement for the three values of Pay shown in the sample run. Recall that iteration means a pass through the loop.

Program Style
Writing General Loops

Because the loop in Fig. 5.1 uses the loop repetition condition CountEmp < 7, it processes exactly 7 employees. The more general loop in Fig. 5.4 uses the loop repetition condition CountEmp < NumberEmp, so it can process any number of employees. The number of employees to be processed in the latter loop must be read into variable NumberEmp before the while statement executes.

Exercises for Section 5.2

Self-Check

1. What output values are displayed by the following while loop for a data value of 5?

   ```
   WriteLn ('Enter an integer');
   Readln (X);
   Product := 1;
   Count := 0;
   ```
while Count < 4 do
    begin
        Writeln (Product);
        Product := Product * X;
        Count := Count + 1
    end; [while]

2. What values are displayed if the call to Writeln comes at the end of
   the loop instead of at the beginning?

3. What mathematical operation does the following segment compute?
   Write ('Enter X> ');
   Readln (X);
   Write ('Enter Y> ');
   Readln (Y);
   Product := 1;
   while Y > 0 do
       begin
           Product := Product * X;
           Y := Y - 1
       end; [while]
   Writeln ('Result = ', Product)

4. How would you modify the program in Fig. 5.4 to display the average
   employee salary, in addition to the total payroll amount?

Programming

1. When Robin's new baby was born, she opened a savings account with
   $1000.00. On each birthday, starting with the first, the bank added an
   additional 4.5% of the balance and Robin added another $500.00 to the
   account. Write a loop that will determine how much money was in the
   account on the child's 18th birthday.

5.3 Event-Controlled Loops

Programmers use two kinds of loops: counter-controlled loops and event-
controlled loops (or conditional loops). In the former, the loop body repeats
a specified number of times. In an event-controlled loop, loop repetition
stops when a particular event occurs.

For example, if faced with a pile of bills when sitting down to write
monthly checks, you might not know how many checks to write. However,
if you wanted to pay as many bills as possible, you could pay them in order
by amount (smallest bill first) and stop writing checks when the event "ac-
tount is overdrawn" occurs. In other words, you would continue writing
checks as long as your account was not overdrawn, as indicated by the fol-
lowing pseudocode description:
loop initialization step is always performed, even for zero-iteration loops. It should always be written to ensure that a program with a zero-iteration loop generates meaningful results.

The bill-paying program of Fig. 5.6 has a zero-iteration loop when the initial account balance is negative. If initial balance is -120.00, the loop initialization step would assign -120.00 to Balance, the loop would be skipped, and the statements following the loop would execute. The program would be:

```
Enter initial account balance >-120.00
Insufficient funds to pay any more bills!
```

Exercises for Section 5.3

Self-Check

1. What is the least number of times that the body of a while loop executed?
2. When would the output of the following segment be erroneous? How could it be fixed?

```
Total := 0;
Write ('Enter number of items > '); Readln (Num);
Count := 0;
while Count < Num do begin
  Write ('Enter a value > '); Readln (Value);
  Last := Value;
end; [while]
WriteLn ('The last value entered was ', Last)
```

3. Trace the program in Fig. 5.6 for the following data: 150.00, 75.00,
50.00, 25.00, 30.00.
4. a. How would you modify the loop in Fig. 5.6 so that it also determines the number of bills paid (CountBills)?
   b. How would you modify the loop in Fig. 5.6 if it is possible to change the condition (Balance := 0.0) and (CountBills < NumberBills).

Programming

1. There are 9870 people in a town whose population increases by 10% each year. Write a loop that determines how many years (CountYears) it would take for the population to exceed 30,000.
DigitRead := False;  \textit{(Assume no digit character was read.)}
\textbf{while} not DigitRead \textbf{do}
  \begin{align*}
    \text{Write ('Enter another data character >');} \\
    \text{ReadIn (NextChar);} \\
    \text{DigitRead := ('0' \leq NextChar) \text{ and (NextChar} \leq '9')}
  \end{align*}
\textbf{end} \textbf{while}

Inside the loop body, the assignment statement
\text{DigitRead := ('0' \leq NextChar) \text{ and (NextChar} \leq '9')}

assigns a value of True to DigitRead if NextChar is a digit character (with
the range '0' through '9'); otherwise, DigitRead remains False. If DigitRead becomes True, loop exit occurs; if DigitRead remains False, the loop continues to execute until a digit character is finally read.

\textit{Template for a Flag-Controlled Loop}

1. Initialize flag to False
2. \textbf{while} not flag \textbf{do}
   \begin{align*}
     \text{...} \\
     \text{3. Reset flag to True if the event being monitored occurs} \\
     \text{end}
   \end{align*}

The last step in the loop body updates the flag value, setting it to True after
the first occurrence of the event being monitored.

\textbf{Exercises for Section 5.4 Self-Check}

1. When generating a table in a loop, why is it necessary to add format
   specifiers to every output list item including integers (which cannot
   have decimal places)?
2. Why would it be incorrect to move the assignment statement in the
   sentinel-controlled loop of Fig. 5.8 to the end of the loop body?
3. Rewrite the sentinel-controlled loop in Fig. 5.8 as a flag-controlled loop.
   In this case the event being monitored would be "sentinel value was
   read."
4. When would a flag-controlled loop be a better choice than a sentinel-
   controlled loop?

\textbf{Programming}

1. Write a program to maintain a checkbook. It should first ask for a balance,
   then for transactions, and enter deposits as positive values and checks as
   negative values. After each check or deposit, the new balance should be
   printed. Use 0 as a sentinel value.
2. Write a program segment that allows the user to enter values and prints out the number of positive values entered and the number of negative values entered. Use 0 as the sentinel value.

3. Write a while loop that displays all powers of an integer, \( n \), less than the specified value, \( \text{MaxPower} \). On each line of a table, show the power \( 1, 2, \ldots \), and the value of the integer \( n \) raised to that power.

4. Write a loop that prints a table of angle measures along with their sine and cosine values. Assume that the initial and final angle measures (in degrees) are available in \( \text{InitDeg} \) and \( \text{FinalDeg} \) (type Real), respectively, and that the change in angle measure between table entries is given by \( \text{StepDeg} \). Hint: Don't forget to change degrees to radians.

5. Write a flag-controlled loop that continues to read pairs of integers until it reads a pair with the property that the first integer in the pair is evenly divisible by the second.

5.5 The for Statement

In addition to the while statement, Pascal provides another statement for writing counter-controlled loops. The for statement condenses the code of a counter-controlled loop, as you can see by comparing this pseudocode:

**Pseudocode for for Loop**

```
for counter := initial to final do
  begin
    ...
  end (for)
```

**Pseudocode for while Loop**

```
counter := initial;
while counter <= final do
  begin
    counter := count
  end (while)
```

The for loop header (the first line) specifies all manipulation of the variable (the color lines in the while loop). These three operations follow:

- Initialize \( \text{counter} \) to \( \text{initial} \).
- Test if \( \text{counter} \leq \text{final} \).
- Increment \( \text{counter} \) to its next value before each test.

**Example 5.5**

The following two loops produce the same results:
Output Window in Text Mode

Enter number of bars: 8

Output Window in Graphics Mode

set the foreground color to one of 16 colors, based on the value of counter variable \( i \). Similarly, procedure SetFillStyle sets the fill pattern to one of 12 possible patterns. After procedure Bar draws a bar, the statements

\[
X1 := X1 + \text{StepX}; \quad Y1 := Y1 + \text{StepY}; \quad \text{(change top left.)}
\]

\[
X2 := X2 - \text{StepX}; \quad Y2 := Y2 - \text{StepY}; \quad \text{(change bottom right.)}
\]

change the top-left corner (point \( X1, Y1 \)) and the bottom-right corner (point \( X2, Y2 \)) for the next bar, moving them closer together. For interesting effects, try running this program with different values assigned to \( \text{StepX} \) and \( \text{StepY} \). ▲

Exercises for Section 5.5 Self-Check

1. Trace the following program fragments:

\[
J := 10;
\]

\[
\text{for } i := 1 \text{ to } 5 \text{ do begin}
\]

\[
\text{WriteLn } (i, J);
\]

\[
J := J - 2
\]

\[
\text{end; } \text{(for)}
\]
2. What is the minimum number of times the statement that comprises the for loop body can be executed? Give an example program fragment that will execute that minimum number of times.

3. Write for loop headers that process all values of Celsius (type Integer) in the following ranges:
   a. -10 through +10
   b. 100 through 1
   c. 15 through 50
   d. 50 through -75

4. Variables of which data types can be declared as for loop counter variables?

5. What would be drawn by the following fragment?

   \begin{verbatim}
   Radius := 20;
   X := Radius;
   Y := GetMaxY div 2;
   for I := 1 to 10 do
     begin
       ForeColor := I mod 16;
       SetColor (ForeColor);
       Circle (X, Y, Radius);
       Floodfill (X, Y, ForeColor);
       X := X + Radius
     end; [for]
   \end{verbatim}

Programming

1. Write a for statement that computes the sum of the odd integers in the range 0 to 100, inclusive.

2. Write a program fragment with a for statement that accumulates the total number of days for the years 1950 to the year 2000. Remember, any year divisible by 4 is a leap year and has 366 days.

5.6 The repeat Statement

The repeat statement specifies a conditional loop that is repeated until the condition becomes true. Such a loop is called a repeat-until loop. To illustrate, compare two program segments that print the powers of 2 between 1 and 1000:

\begin{verbatim}
repeat statement
Power := 1;
repeat
  Write (Power :5);
  Power := Power * 2
until Power >= 1000

while statement
Power := 1;
while Power < 1000 do
  begin
    Write (Power :5);
    Power := Power * 2
  end [while]
\end{verbatim}
In Fig. 5.16, Count, StartValue, and StopValue must all be the same data type. The assignment statement:

```
Count := Count + 1
```

is used in both the while and repeat loops to update the loop-control variable Count, but is not required in the for loop.

**Exercises for Section 5.6**

**Self-Check**

1. Use DeMorgan's theorem to complement the following conditions:
   a. \((X <= Y) \text{ and } (X >= 15)\)
   b. \((X <= Y) \text{ or } (Z <= 7.5)\)
   c. \((X <= 15) \text{ or } (Z <= 7.5)\) and \((X <= Y)\)
   d. \(\text{Flag or not } (X <= 15.7)\)
   e. \(\text{not Flag and } (X <= 8)\)

2. What would the following `repeat` statement display? What is the difference between `repeat ... until False` and `while False do`?
   ```
   repeat
       Writeln('False conditional example.');
   until False
   ```

3. When would you use a `repeat-until` loop rather than a `while` loop?

**Programming**

1. Write a program fragment that continues to read data values as long as they are not decreasing. The fragment should stop reading whenever a number smaller than the preceding one is entered. Write two versions: one with `repeat` and one with `while`. 
Figure 5.18
Isosceles Triangle Program

Edit Window

program Triangle;
(Draws an isosceles triangle)

const
NumLines = 5;  {number of rows in triangle}
Blank = ' ';   {output characters}
Star = '*';

var
Row,   {loop control for outside}
LeadBlanks, {loop control for first inner
CountStars : Integer; {loop control for second inner

begin (Triangle)
for Row := 1 to NumLines do
begin
for LeadBlanks := NumLines - Row downto 1 do
  Write (Blank);  {Print leading blank}
for CountStars := 1 to 2 * Row - 1 do
  Write (Star);
  Writeln
end {for Row}
end. {Triangle}

Output Window

*
***
*****
*******

Exercises for Section 5.7

Self-Check

1. What is displayed by the following program fragments, assuming N is 5?
   a. for I := 1 to N do begin
       for J := 1 to I do
         Write ('*');
       Writeln
     end; {for I}
   b. for I := N downto 1 begin
       for J := M downto 1 do
         Write ('*');
       Writeln
     end {for I}

2. Show the output printed by the following nested loops:
for Count := StartValue to StopValue do (for loop)
  begin
    Count := StartValue:
    while Count <= StopValue do
      begin
        Count := Count + 1
        end (while)
      end (while)
    Count := StartValue:
    repeat until Count > StopValue
    if StartValue <= StopValue then
      repeat
        Count := Count + 1
      until Count > StopValue
    end (for)

In Fig. 5.16, Count, StartValue, and StopValue must all be the same ordinal type. The assignment statement
Count := Count + 1
is used in both the while and repeat loops to update the loop-control variable Count, but is not required in the for loop.

Exercises for Section 5.6

Self-Check

1. Use DeMorgan's theorem to complement the following conditions:
   a. \( (X \leq Y) \) and \( (X \geq 15) \)
   b. \( (X \leq Y) \) or \( (Z = 7.5) \)
   c. not Flag and \( (X \leq 8) \)
   d. Flag or not \( (X \geq 15.7) \)
   e. not Flag and \( (X \leq 8) \)

2. What would the following repeat statement display? What is the difference between repeat ..., until False and while False do?
   repeat
     write('False conditional example:');
   until False

3. When would you use a repeat-until loop rather than a while loop?

Programming

1. Write a program fragment that continues to read data values as long as they are not decreasing. The fragment should stop reading whenever a number smaller than the preceding one is entered. Write two versions: one with repeat and one with while.
Figure 5.18  Isosceles Triangle Program

Edit Window

program Triangle;
(Draws an isosceles triangle)

const
NumLines := 5;  {number of rows in triangle}
Blank := ' ';   {output characters}
Star := '*';

var
Row,  {loop control for outer loop}
LeadBlanks, {loop control for first inner loop}
CountStars : integer; {loop control for second inner loop

begin {Triangle}
for Row := 1 to NumLines do  
begin  {Draw each row.}
  for LeadBlanks := NumLines - Row downto 1 do
    Write (Blank);  
    {Print leading blanks.
  for CountStars := 1 to 2 * Row - 1 do
    Write (Star);   {Print asterisks.
    WriteLn;  
  end {for Row}
end. {Triangle}

Output Window

* 
***
*****
*******
**********

Exercises for Section 5.7  Self-Check

1. What is displayed by the following program fragments, assuming $M$ is 3 and $N$ is 5?
   a. for $I := 1$ to $N$ do
      begin
      for $J := 1$ to $I$ do
        Write ('*');
        WriteLn;
      end {for J}
    end {for I}
   b. for $I := N$ downto 1 do
      begin
      for $J := M$ downto 1 do
        Write ('*');
        WriteLn;
      end {for J}
    end {for I}

2. Show the output printed by the following nested loops:
for I := 1 to 2 do
begin
    WriteLn ('Outer':5, I:5);
    for J := 1 to 3 do
        WriteLn ('Inner':7, I:3, J:3)
    for K := 2 downto 1 do
        WriteLn ('Inner':7, I:3, K:3)
end {for I}

Programming

1. Write a program fragment that, given an input value N, displays N rows of the form 1 2 ... N 2 3 ... N + 1, and so on—a barber pole pattern of numbers. As an example, for an input value of 5, display:

   1 2 3 4 5
   2 3 4 5 6
   3 4 5 6 7
   4 5 6 7 8
   5 6 7 8 9

2. Write a program that prints a nicely labeled multiplication table for the digits 0 through 9.

5.8 Debugging and Testing Programs

In section 2.9, we described the three general categories of errors: syntax errors, run-time errors, and logic errors. Sometimes the cause of a run-time error or the source of a logic error is apparent and the error can be fixed easily. Often, however, the error is not obvious and may require considerable time to locate.

The first step in locating a hidden error is to examine the program output to determine which part of the program is generating incorrect results. Then you can focus on the statements in that section to determine which are at fault. To find problem areas during debugging, insert extra WriteLn statements to display intermediate results at different points in your program. For example, if the loop in Fig. 5.8 is not computing the correct sum, the following diagnostic WriteLn statement in color will display each value of Score and Sum. The asterisks highlight the diagnostic output in the debugging runs and the diagnostic WriteLn statements in the source program.

ReadLn (Score);
while Score <> 'Sentinel' do
begin
    Sum := Sum + Score;
    WriteLn ('***** score is ', Score, ' sum is ', Sum);
    Write ('Enter the next score > ');
    ReadLn (Score)
end {while}
Restarting the Debugger

If you want to go back to the beginning of your program while in the middle of a debugging session, select Program Reset (Ctrl-F2) from the Run menu. This reinitializes the debugging system and positions the execution bar over the begin line of the main program.

Prior to loading a new program into the Turbo Pascal environment after a debugging session, select Program Reset to verify that the computer memory used by your old program is available for use by your new program. It is important to note that neither loading a new program into the Turbo Pascal system nor selecting Program Reset removes any of the expressions displayed in the Watch window or clears any of the program breakpoints. To remove Watch Expressions from the Watch window, close the Watch window. To clear all breakpoints, select the Clear all button shown in Fig. 5.22. You should do this prior to loading a new program into the Turbo Pascal environment.

Turbo Pascal will offer to restart the debugging session if you make any changes to a program's statements during debugging. For example, if you make a change to a program statement using an Edit command and then press one of the execution command keys (F7, F4, or Ctrl-F9), Turbo Pascal will display an Information dialog box with the message Source has been modified, Rebuild? If you type Y, your program will be compiled again, the execution bar will be placed on the begin line of the main program, and the debugger will be reinitialized (as it would following a Program Reset). If you type N, the current debugging session will continue, and the changes made to your program will have no effect until it is recompiled.

Testing

After you have corrected all errors and the program appears to execute as expected, test it thoroughly to make sure it works. In Section 4.6, we discussed tracing an algorithm and suggested that you provide enough sets of test data to ensure that all possible paths are traced. The same is true for a completed program. Make enough test runs to verify that the program works properly for representative samples of all possible data combinations.

Exercises for Section 5.8  Self-Check

1. For the while loop in the subsection entitled “Off-by-One Loop Errors,” add debugging statements to show the value of the loop-control variable at the start of each repetition. Add additional debugging statements to show the value of Sum at the end of each loop repetition.

2. Why does the following code fragment fail? What are the loop boundaries for this code fragment?
\begin{verbatim}
X := 10;
repeat
  X := X - 1;
  WriteLn (X, Sqr(X))
until X < 0
\end{verbatim}

5.9 Common Programming Errors

Beginners sometimes confuse if and while statements because both statements contain a condition. Always use an if statement to implement a decision step and a while statement to implement a conditional loop.

Be careful when you use tests for inequality to control the repetition of while loop. For instance, the following loop is intended to process all transactions for a bank account while the balance is positive:

\begin{verbatim}
while Balance <> 0.0 do
  Update (Balance)
\end{verbatim}

If the bank balance goes from a positive to a negative amount without being exactly 0.0, the loop will not terminate (an infinite loop). This loop would be safer:

\begin{verbatim}
while Balance > 0.0 do
  Update (Balance)
\end{verbatim}

Be sure to verify that the repetition condition for a while loop will eventually become false. If you use a sentinel-controlled loop, remember to provide a prompt that tells the program user what value to enter as the sentinel. Make sure that the sentinel value cannot be confused with a normal data item.

If the loop body contains more than one statement, remember to bracket it with begin and end (unless it is a repeat-until loop). Otherwise, on the first statement will be repeated, and the remaining statements will be executed when and if the loop is exited. The following loop will not terminate because the step that updates the loop-control variable is not considered part of the loop body. The program will continue to print the initial value of Power until you instruct the computer to terminate its execution. You can terminate a program by pressing Ctrl-C:

\begin{verbatim}
while Power <= 10000 do
  WriteLn ('Next power of N is ', Power;6):
  Power := Power * N
\end{verbatim}

Be sure to initialize to 0 an accumulator variable used for accumulating sum by repeated addition and to initialize to 1 a variable used for accumulating product by repeated multiplication. Omitting this step leads to inaccurate results.

The value of the counter variable in a for statement either increases by (to form) or decreases by 1 (down-to form) after each repetition. If larger
Slope := (Y2 - Y1) / (X2 - X1);
while Slope θ StopValue do
begin
  Writeln ('Slope is ', Slope :5:2);
  Writeln ('Enter four more numbers
> '); Readln (Y2, Y1, X2, X1);
  Slope := (Y2 - Y1) / (X2 - X1);
end [while]
end. [Slopes]

5. Can you always replace a while loop with a for or a repeat loop? Why or why not?
6. Consider the program segment

Count := 0;
for I := 1 to N do
begin
  Read (X);
  if X = 1 then
  begin
    Count := Count + 1
  end [for]
end. [for]

a. Write a while loop equivalent to the for loop.
b. Write a repeat-until loop equivalent to the for loop.

Programming Projects

1. Write a program that will find the smallest, largest, and average values in a collection of N numbers. Read in the value of N before reading each value in the collection of N numbers.

2. Modify Programming Project 1 to compute and display both the range of values in the data collection and the standard deviation of the data collection. To compute the standard deviation, accumulate the sum of the data (Sum) and the sum of the squares of the data values (SumSquares) in the main loop. After loop exit, use the formula

\[
\text{Standard deviation} = \sqrt{\frac{\text{SumSquares} - \text{Sum}^2}{N}}
\]

3. Bunyan Lumber Co. needs to create a table of the engineering properties of its lumber. The dimensions of the wood are given as the base and the height in inches. Engineers need to know the following information about lumber:

- cross-sectional area = base × height
- moment of inertia = (base × height^3)/12
- section modulus = (base × height^2)/6

The height sizes are 2, 4, 6, 8, and 10 inches. Produce a table with appropriate headings to show these values and the computed engineering properties. Do not duplicate a 2-by-6 board with a 6-by-2 board.
4. Write a program to generate a yearly calendar. The program should accept the
year and the day of the week for January 1 of that year (1 = Sunday, 7 = Sat-
day). Remember, February has 29 days if the year is divisible by 4. The calendar
should be printed in the following form (for each month):

<table>
<thead>
<tr>
<th>January</th>
<th>1</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>3</td>
</tr>
<tr>
<td>4</td>
<td>5</td>
</tr>
<tr>
<td>6</td>
<td>7</td>
</tr>
<tr>
<td>8</td>
<td></td>
</tr>
<tr>
<td>9</td>
<td>10</td>
</tr>
<tr>
<td>11</td>
<td>12</td>
</tr>
<tr>
<td>13</td>
<td>14</td>
</tr>
<tr>
<td>15</td>
<td></td>
</tr>
<tr>
<td>16</td>
<td>17</td>
</tr>
<tr>
<td>18</td>
<td>19</td>
</tr>
<tr>
<td>20</td>
<td>21</td>
</tr>
<tr>
<td>22</td>
<td></td>
</tr>
<tr>
<td>23</td>
<td>24</td>
</tr>
<tr>
<td>25</td>
<td>26</td>
</tr>
<tr>
<td>27</td>
<td>28</td>
</tr>
<tr>
<td>29</td>
<td></td>
</tr>
<tr>
<td>30</td>
<td>31</td>
</tr>
</tbody>
</table>

5. a. Write a program to read in a collection of exam scores ranging in value
from 1 to 100. Your program should count and print the number of out-
standing scores (90–100), the number of satisfactory scores (60–89), and
the number of unsatisfactory scores (1–59). It should also display the
category of each score. Test your program on the following data:

63 75 72 72 78 67 80 63 75
90 89 45 59 99 82 12 100

b. Modify your program to display the average exam score (a real number) at
the end of the run.

6. Write a program to process weekly employee time cards for all employees of an
organization. Each employee will have three data items: an identification num-
ber, the hourly wage rate, and the number of hours worked during a given
week. Each employee is to be paid time and a half for all hours worked over
40. A tax amount of 3.625% of gross salary will be deducted. The program out-
put should show the employee’s number and net pay. Display the total payroll
and the average amount paid at the end of the run.

7. Suppose you own a beer distributorship that sells Pils (ID number 1), Coors
(ID number 2), Bud (ID number 3), and Iron City (ID number 4) by the case.
Write a program to
a. read in the case inventory for each brand for the start of the week.
b. process all weekly sales and purchase records for each brand.
c. print out the final inventory.
Each transaction will consist of two data items: the brand identification number
(an integer) followed by the amount purchased (a positive integer value) or the
amount sold (a negative integer value). The weekly inventory for each brand
(for the start of the week) will also consist of two items: the identification num-
ber and the initial inventory for that brand. For now, assume that you always
have sufficient foresight to prevent depletion of your inventory for any brand.
Hint: Your data entry should begin with eight values representing the case in-
ventory, followed by the transaction values.

8. Revise Programming Project 7 to make it a menu-driven program. The
menu operations supported by the revised program should be (E)nter In-
ventory, (P)urchase Case, (S)ell Case, (D)isplay Inventory, and (Q)uit Program. Negative quantities should no longer be used to repre-
sent goods sold.
Writing Boolean Values

Most Boolean expressions appear in control structures, where they determine the sequence in which Pascal statements execute. Because programs do not process Boolean data in the same way that they process numerical data, your programs will rarely read Boolean values as input data or display Boolean values as program results. If necessary, you can display the value of a Boolean variable with procedure Write or WriteLn, but you cannot use procedure ReadLn to read in a Boolean variable. If Switch is False, the statement

\[
\text{WriteLn ('Value of Switch is ', Switch)}
\]

displays the line

\[
\text{Value of Switch is FALSE}
\]

Exercises for Section 4.2

Self-Check

1. Which of the following Boolean expressions is incorrect and why? Assume \(X\) and \(Y\) are Real and \(P\), \(Q\), and \(R\) are Boolean.
   a. \(X < 5.1\) and \(Y > 22.3\)
   b. \(P \text{ or } Q\) or \(Q \text{ and } R\)
2. Draw evaluation trees for the following expressions:
   a. \(A = (B + A - B)\)
   b. \((C = (A + B))\) or not Flag
   c. \((A <> 7)\) and \((C >= 6)\) or Flag
   d. not \((8 <= 12)\) and \((A \text{ mod } 2 = 0)\)
   e. not \(((A > 5) \text{ or } (C < (A + B)))\)
3. Evaluate each expression in Exercise 2 if \(A\) is 5, \(B\) is 10, \(C\) is 15, and Flag is True.

Programming

1. Write a Boolean expression for each of the following relationships.
   a. Age is from 18 to 21 inclusive.
   b. Water is less than 1.5 and is greater than 0.1.
   c. Year is divisible by 4. (Hint: Use mod.)
   d. Speed is not greater than 55.
2. Write Boolean assignment statements for the following.
   a. Assign a value of True to Between if \(N\) is in the range \(-K\) to \(+K\), inclusive; otherwise, assign a value of False.
   b. Assign a value of True to Uppercase if \(Ch\) is an uppercase letter; otherwise, assign a value of False.
   c. Assign a value of True to Divisor if \(M\) is a divisor of \(N\); otherwise, assign a value of False.

4.3 The if Statement

In Pascal, the primary selection control structure consists of an if statement, which always contains a Boolean expression. The if statements determine
If MomOrDad = 'M' then
  Writeln ('Hi Mom');
else
  Writeln ('Hi Dad'); \[error - new statement begins with else\]

**SYNTAX DISPLAY**

**if Statement (One Alternative)**

**Form:** \[if \text{condition} \text{then statement,}\]

**Example:** \[if \; X > 0.0 \text{then}
  \text{PosProd} := \text{PosProd} \times X\]

**Interpretation:** If \text{condition} evaluates to \text{True}, then \text{statement}, (the true task) is executed; otherwise, \text{statement}, is skipped.

**SYNTAX DISPLAY**

**if Statement (Two Alternatives)**

**Form:** \[if \text{condition} \text{then statement,}\]

else \[statement,\]

**Example:** \[if \; X >= 0.0 \text{then}
  \text{Write} ('Positive')\]

else \[Write ('Negative')\]

**Interpretation:** If \text{condition} evaluates to \text{True}, then \text{statement,} (the true task) is executed and \text{statement,} is skipped; otherwise, \text{statement,} is skipped and \text{statement,} (the false task) is executed.

**Program Style**

**Format of the if Statement**

All if statement examples in this text indent \text{statement,} and \text{statement,}. The word else is typed on a separate line, aligned with the word if. The format of the if statement makes its meaning apparent and is used solely to improve program readability; the format makes no difference to the compiler.

**Exercises for Section 4.3**

**Self-Check**

1. What do these statements display?
   a. \[if \; 12 < 12 \text{then}
      \text{Writeln} ('Less')\]
      else
      \[\text{Writeln} ('Not less')\]
   b. \[\text{Var1} := 25.12;\]
      \[\text{Var2} := 15.00;\]
      if \text{Var1} <= \text{Var2} then
      \[\text{Writeln} ('Less or equal')\]
      else
      \[\text{Writeln} ('Greater than')\]
2. What value is assigned to \( X \) when \( Y \) is 15.0?
   a. \( X := 25.0; \)
      \[ \text{if } Y < (X + 10) \text{ then} \]
      \[ X := X - 10.0 \]
      \[ \text{else} \]
      \[ X := X / 2.0 \]
   b. \( X := 5 * Y \)
      \[ \text{if } (Y < 15) \text{ and } (Y > 0.0) \text{ then} \]
      \[ X := 2 * Y \]
      \[ \text{else} \]

**Programming**

1. Write Pascal statements to carry out the following steps:
   a. If \( \text{Item} \) is nonzero, then multiply \( \text{Product} \) by \( \text{Item} \) and save the result in \( \text{Product} \); otherwise, skip the multiplication. In either case, print the value of \( \text{Product} \).
   b. Store the absolute difference of \( X \) and \( Y \) in \( Y \), where the absolute difference is \( (X - Y) \) or \( (Y - X) \), whichever is positive. Do not use the \texttt{Abs} function in your solution.
   c. If \( X \) is 0, add 1 to \texttt{ZeroCount}. If \( X \) is negative, add \( X \) to \texttt{MinusSum}. If \( X \) is greater than 0, add \( X \) to \texttt{PlusSum}.

### 4.4 Syntax Diagrams

In addition to syntax displays, we can use **syntax diagrams** to describe Pascal constructs. Syntax diagrams are sometimes called railroad diagrams because they resemble track layout diagrams for a model railroad.

To learn how to read syntax diagrams, study the syntax diagram for **program** (Fig. 4.6). The diagram consists of a group of syntactic elements connected by arrows, with the category of each syntactic element indicated by its shape:

- Reserved words appear in gray ovals.
- Special symbols (punctuation marks and operators) appear in gray circles.
- Syntactic elements with their own syntax diagrams appear in colored rectangles.

![Syntax Diagram for program](image-url)
EXAMPLE 4.10

As manager of a clothing boutique, you keep records of your checking transactions. When TransType is 'C' in the next if statement, the compound statement following then processes a transaction (TransAmount) that represents a check you wrote to pay for goods received; otherwise, the compound statement following else processes a deposit made into your checking account. Both compound statements display an appropriate message and update the account balance (Balance).

```
if TransType = 'C' then
  begin [check]
    Write ('Check for $', TransAmount :4:2);
    Balance := Balance - TransAmount  {Deduct check.}
  end [check]
else
  begin [deposit]
    Write ('Deposit of $', TransAmount :4:2);
    Balance := Balance + TransAmount  {Add deposit.}
  end [deposit and if]
```

The semicolons in the if statement separate the individual statements in each alternative. A common error would be to insert a semicolon after the first end (end; [check]), causing the if to separate into two statements. Because the second statement can't begin with else, the compiler would display an unexpected symbol error message.

---

Program Style

**Writing if Statements with Compound True or False Statements**

Each if statement in this section contains at least one compound statement bracketed by begin and end. To improve our ability to read and understand the if statement, each compound statement is indented; indentation is ignored by the Pascal compiler.

The comment after each end helps to associate the end with its corresponding begin. The comments are not required but are included to improve program readability.

Semicolons are required between the individual statements within a compound statement, but they should not appear before or after the reserved words then, else, or begin. A semicolon may appear after an end that terminates the entire if statement.

---

Exercises for Section 4.5 **Self-Check**

1. Insert semicolons where needed to avoid syntax errors and indent to improve readability:
4.6 Decision Steps in Algorithms

1. Correct if statement:
   
   ```pascal
   if X > Y then
       begin
       X := X + 10.0
       Writeln ('X Bigger')
       end
   else
       Writeln ('X Smaller')
       Writeln ('Y is ', Y)
   end
   ```

2. Explain why the program will not compile if you remove the bracketing `begin` and `end`.

3. What is the effect of placing a bracketing `begin` and `end` around the last two lines in Exercise 1?

4. Find the syntax error and logic error in the if statement:
   ```pascal
   if Num1 < 0 then
       begin
       Product := Num1 * Num2 * Num3;
       Writeln ('Product is ', Product :1)
       end;
   else
       Sum := Num1 + Num2 + Num3;
       Writeln ('Sum is ', Sum :1)
   end
   ```

5. What syntax diagrams would you use to validate the following if statement? Provide the label of every syntax diagram that describes an element of this statement.
   ```pascal
   if X > 0 then
       begin
       X := 25.0;
       Writeln ('Positive')
       end
   ```

Programming

1. Write an if statement that, given two real values \( x \) and \( y \), will negate the two values if both are negative or both are positive.

2. Write an interactive program that computes the area of a rectangle \( \text{area} = \text{base} \times \text{height} \) or a triangle \( \text{area} = \frac{1}{2} \times \text{base} \times \text{height} \) after prompting the user to type the first character of the figure name (R or T).

4.6 Decision Steps in Algorithms

decision step

An algorithm step that selects one of several actions.

Algorithm steps that select from a choice of actions are called decision steps. The algorithm in the following payroll problem contains decision steps that are coded as Pascal if statements to compute an employee's gross pay and net pay.
Problem Inputs
Premium : Real  \{premium amount\}
NumClaims : Integer  \{number of claims\}

Problem Output
Dividend : Real  \{dividend amount\}

DESIGN ▼
The initial algorithm follows.

Initial Algorithm
1. Display user instructions.
2. Enter premium amount and number of claims.
3. Compute dividend including any bonus dividend.
4. Display dividend.

Algorithm Refinements
The refinement of step 3 is similar to the refinement of step 3 in the payroll problem.

Step 3 Refinement
3.1 if no claims then
   3.2 Compute dividend including a bonus dividend.
   else
3.3 Compute dividend as the basic dividend.

IMPLEMENTATION AND TESTING ▼
We leave the program coding as a programming exercise. The if statement
\{Compute dividend including any bonus dividend.\}
if NumClaims = 0 then
   Dividend := Premium * BasicRate + \{basic dividend\}
   Premium * BonusRate; \{bonus dividend\}
else
   Dividend := Premium * BasicRate \{basic dividend\}
implements algorithm step 3.

Exercises for Section 4.6 Self-Check
1. Change the refinement of step 4 of the payroll problem so that it uses a
decision step with one alternative. \Hint: Assign Gross to Net before the
decision step, and use the decision step to change Net when necessary.
2. Predict the payroll program output when
a. Hours is 30.0, Rate is 5.00.
b. Hours is 20.0, Rate is 4.00.
c. Hours is 50.0, Rate is 2.00.
d. Hours is 50.0, Rate is 6.00.

3. Revise the pizza problem from Section 3.1 so that the user can compute
the unit price of either a circular or a square pizza. Give the algorithm
with refinements. Draw a structure chart with data flow information for
the new pizza problem showing the relationship between the main pro-
gram and its subproblems. Discuss how the role of the variable that
represents pizza area (a problem output) changes for each subproblem.

Programming
1. Modify the program for the payroll problem to deduct union dues of
10% for gross salary over $100.00 and 5% otherwise. Also, deduct a 3%
city wage tax for all employees.
2. Provide the structure chart and complete program for computing insur-
ance company dividends.

4.7 Hand-Tracing an Algorithm

A critical step in algorithm design is to verify the algorithm’s accuracy be-
fore you spend extensive time coding it. These few extra minutes often save
hours of coding and testing time.

A hand trace, or desk check, is a careful, step-by-step simulation on pa-
paper of how a computer would execute an algorithm. The results should
show the effect of each step’s execution using data that are relatively easy to
process by hand. In Section 4.5, we simulated the execution of an if state-
ment that switches the values of two variables. Now we will trace the execu-
tion of the refined algorithm for the payroll problem solved in Section 4.6.

Refined Algorithm
1. Display user instructions.
2. Enter hours worked and hourly rate.
3. Compute gross pay.
   3.1 If no overtime then
   3.2 Compute gross pay without overtime pay.
   else
   3.3 Compute gross pay with overtime pay.
   4.1 If no union dues then
   4.2 Net gets Gross
   else
   4.3 Net gets Gross - Dues
5. Display gross and net pay.
if (X <> 0.0) then
  if (Y / X > 5.0) then

The first condition guards the second and prevents the latter from being evaluated when X is zero. The result of evaluating these conditions is the same for short-circuit or complete evaluation.

Be wary of short-circuit evaluation and avoid writing Boolean expressions that rely on it. If you insert the special comment `$B+$` in a program, the Turbo Pascal compiler generates code that causes complete evaluation of all Boolean expressions. Comments beginning with the symbols `$` are called compiler directives because they provide instructions to the compiler. The `$B+$` compiler directive should precede the first Boolean expression for which complete evaluation is desired. Once enabled, complete Boolean evaluation remains in effect until the comment `$B-$` is encountered.

**Boolean Evaluation Compiler Directive**

**Form:** `$B-$` or `$B+$`

**Default:** `$B-$`

**Interpretation:** In the default state, Turbo Pascal uses short-circuit Boolean evaluation. The compiler directive `$B+$` causes the compiler to generate code for complete evaluation of every operand of a Boolean expression.

**Note:** Writing a Boolean expression that may fail when complete evaluation is used instead of short-circuit evaluation is not good programming practice.

### Exercises for Section 4.8

#### Self-Check

1. Trace the execution of the nested `if` statement in Example 4.14 when `Salary` is 13500.00.
2. In Example 4.16, how many comparisons are required to execute the first `if` statement? What about the second? Which `if` statement is more efficient?
3. Evaluate the following expressions, with and without short-circuit evaluation, if `X` is 6 and `Y` is 7.
   a. `(X > 5)` and `(Y div X <= 10)`
   b. `(X <= 10)` or `(X / (Y - 7) > 3)`

#### Programming

1. Rewrite the `if` statement for Example 4.15 using only the relational operator `<` in all conditions. Test for a failing grade first.
2. Implement the following decision table using a nested `if` statement. Assume that the grade point average is within the range 0.0 through 4.0.
4.9 The case Statement

In addition to the if statement, the case statement can be used in Pascal to select one of several alternatives. The case statement is especially useful when selection is based on the value of a single variable or a simple expression (called the case selector). The case selector must be an ordinal data type, or a data type whose values may all be listed. Data types integer, Boolean, and Char are ordinal types, but data types Real and String are not. (If you tried to list the real numbers between 3.1 and 3.2 as 3.11, 3.12, 3.13, ..., you would be leaving out 3.111, 3.112, 3.113, ...)

EXAMPLE 4.18 ▼

```
The case statement

case MomOrDad of
  'M', 'm' : WriteLn ('Hello Mom - Happy Mother's Day');
  'D', 'd' : WriteLn ('Hello Dad - Happy Father's Day')
end;              
```

behaves the same way as the following if statement when the character stored in MomOrDad is one of the four letters listed (M, m, D, or d):

```
if (MomOrDad = 'M') or (MomOrDad = 'm') then
  WriteLn ('Hello Mom - Happy Mother's Day')
else if (MomOrDad = 'D') or (MomOrDad = 'd') then
  WriteLn ('Hello Dad - Happy Father's Day')
```

The message displayed by the case statement depends on the value of the case selector MomOrDad (type Char). If the case selector value is 'M' or 'm', the first message is displayed. If the case selector value is 'D' or 'd', the second message is displayed. The lists 'M', 'm' and 'D', 'd' are called case labels. ▲
Programming Projects

1. Write procedures to draw a circle, square, and triangle. Then write a program that reads a letter C, S, or T and, depending on the letter chosen, draws either a circle, square, or triangle.

2. Write a program that reads in four words and displays them in increasing alphabetical sequence and also in decreasing alphabetical sequence.

3. While spending the summer as a surveyor's assistant, you decide to write a program that transforms compass headings in degrees (0 to 360) to compass bearings. A compass bearing consists of three items: the direction you face (north or south), an angle between 0 and 90 degrees, and the direction you turn before walking (east or west). For example, to get the bearing for a compass heading of 110.0 degrees, you would first face due south (180 degrees) and then turn 70.0 degrees east (180.0 - 70.0 = 110.0). Therefore, the bearing is south 70.0 degrees east. Be sure to check the input for invalid compass headings.

4. Write a program that reads in a room number, its capacity, and the size of the class enrolled so far and prints an output line showing the classroom number, capacity, number of seats filled, number of seats available, and a message indicating whether the class is filled. Call a procedure to display the following heading before the output line:

   Room  Capacity  Enrollment  Empty seats  Filled/  Not filled

Display each part of the output line under the appropriate heading. Test your program with the following classroom data:

<table>
<thead>
<tr>
<th>Room</th>
<th>Capacity</th>
<th>Enrollment</th>
</tr>
</thead>
<tbody>
<tr>
<td>426</td>
<td>25</td>
<td>25</td>
</tr>
<tr>
<td>327</td>
<td>18</td>
<td>14</td>
</tr>
<tr>
<td>420</td>
<td>20</td>
<td>15</td>
</tr>
<tr>
<td>317</td>
<td>100</td>
<td>90</td>
</tr>
</tbody>
</table>

5. Write a program that determines the additional state tax owed by an employee. The state charges a 4% tax on net income. Determine net income by subtracting a $500 allowance for each dependent from gross income. Your program will read gross income, number of dependents, and tax amount already deducted. It will then compute the actual tax owed and print the difference between tax owed and tax deducted followed by the message 'SEND CHECK' or 'REFUND', depending on whether the difference is positive or negative.

6. Write a program to control a bread machine. Allow the user to input the type of bread as W for white and S for sweet. Ask the user whether the loaf size is double and whether the baking is manual. The program should fail if the user inputs are invalid. The following table details the time chart for the machine for each bread type. Print a statement for each step. If the loaf size is double, increase the baking time by 50%. If baking is manual, stop after the loaf shaping cycle and instruct the user to remove the dough for manual baking. Use a procedure to print program instructions.