Chapter *13*

Function Procedures

Component Pascal provides both native and programmer-defined procedures. For example, the module in Figure 12.5 contains the statement

```
n := SHORT(ENTIER(x + 0.5))
```

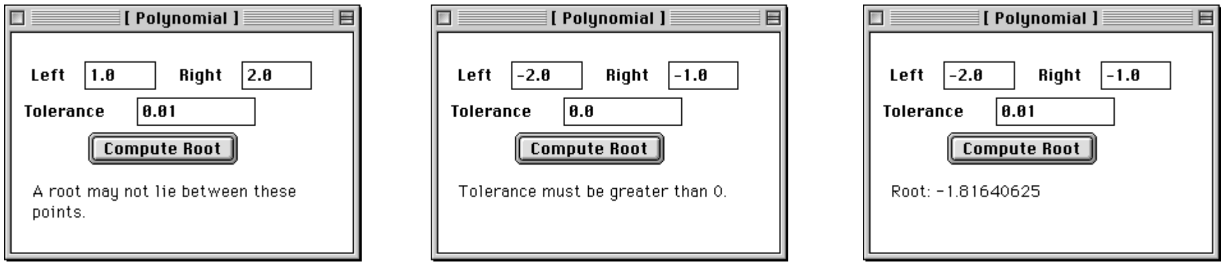
Both the ENTIER function, which truncates a real value and returns a long integer, and the SHORT function, which converts a LONGINT value to an INTEGER value, are native. That is, they are provided by the Component Pascal language, and do not need to be imported from another module or defined by the programmer. If you need a function procedure that is not provided by the language or the framework, you must define your own. This chapter shows how to define function procedures that contain parameter lists.

Function procedures have some characteristics in common with proper procedures and some characteristics that differ. One common characteristic is that a calling procedure makes a procedure call, the called procedure executes, then control is passed back to the calling procedure. The details of the procedure call and return differ, however. Another common characteristic is that storage is allocated on the run-time stack with both proper and function procedures. Again, however, the details differ.

The bisection algorithm

Procedure ComputeRoot in Figure 10.11 computed one root of a cubic equation with the bisection algorithm. The following program improves procedure ComputeRoot in several ways. Figure 10.8 shows that the cubic equation has three real roots—one between –2 and –1, one between 0 and 1, and one between 2 and 3. But the original version of ComputeRoot found only the root between 2 and 3. One improvement would be to allow the user to enter the starting values of left and right, which would permit finding any of the roots.

Another shortcoming of the original version of ComputeRoot is that it permits the user to enter a value of 0.0 for the tolerance. The problem is that the loop will execute endlessly if d.tolerance has the value 0.0. It would be better if the program guarded against this possibility. Figure 13.1 shows the dialog box for an improved version of ComputeRoot. The dialog box has additional input fields for the user to enter starting values for left and right. You can see from the figure that the program must be able to prove to itself that a root must lie between the two values entered by the user before it will proceed.



Left 1.0 Right 2.0
Tolerance 0.01
A root may not lie between these points.

Left -2.0 Right -1.0
Tolerance 0.0
Tolerance must be greater than 0.

Left -2.0 Right -1.0
Tolerance 0.01
Root: -1.81640625

Figure 13.1
Three executions of the bisection algorithm of Listing 13.2.

The module in Figure 13.2 uses two procedures to implement these improvements to the program. The first procedure, ComputeRoot, is the same kind of procedure we have always used to link to a button in a dialog box. The second procedure, F, is a programmer defined function procedure. The procedure declaration part of F is

```
PROCEDURE F (x: REAL): REAL;
    CONST
        a3 = 1.0; a2 = -1.0; a1 = -4.0; a0 = 2.0;
BEGIN
    RETURN ((a3 * x + a2) * x + a1) * x + a0
END F;
```

These are the statements that define the function. The parameter x in this declaration is the formal parameter. You can tell that F is a function procedure by the type ": REAL" that follows the formal parameter, which indicates that the function returns a real value.

Figure 13.2
The bisection algorithm with a programmer-defined function.

```
MODULE Pbox13A;
    IMPORT Dialog, PboxStrings;
    VAR
        d*: RECORD
            left*, right*: REAL;
            tolerance*: REAL;
            message-: ARRAY 64 OF CHAR;
        END;

    PROCEDURE F (x: REAL): REAL;
        CONST
            a3 = 1.0; a2 = -1.0; a1 = -4.0; a0 = 2.0;
    BEGIN
        RETURN ((a3 * x + a2) * x + a1) * x + a0
    END F;
    PROCEDURE ComputeRoot*;
        VAR
            left, mid, right: REAL;
            fLeft, fMid: REAL;
            rootString: ARRAY 32 OF CHAR;
    BEGIN
        IF d.tolerance <= 0.0 THEN
            d.message := "Tolerance must be greater than 0."
        ELSIF F(d.left) * F(d.right) > 0 THEN (* ra1 *)
            d.message := "A root may not lie between these points."
        ELSE
            left := d.left; right := d.right; fLeft := F(left); (* ra2 *)
            (* Assert: root is between left and right *)
            WHILE ABS(left - right) > d.tolerance DO
                mid := (left + right) / 2.0;
                fMid := F(mid); (* ra3 *)
                IF fLeft * fMid > 0.0 THEN
                    (* Assert: root is between mid and right *)
                    left := mid;
                    fLeft := fMid
                ELSE
                    (* Assert: root is between left and mid *)
                    right := mid
                END
            END;
            PboxStrings.RealToString((left + right) / 2, 1, 0, rootString);
            d.message := "Root: " + rootString
        END;
        Dialog.Update(d)
    END ComputeRoot;

BEGIN
    d.left := 0.0; d.right := 0.0; d.tolerance := 1.0;
    d.message := ""
END Pbox13A.
```

Figure 13.2
Continued.

One of the statements that calls the function from the main program is

```
fLeft := F(left)
```

F(left) is a function call. In this example, it occurs on the right side of an assignment statement. The EBNF for an assignment statement is

Statement = Designator “ := ” Expr

which shows that an expression occurs on the right side of an assignment statement. So, a function call is an example of an expression. In general, a function can occur within a more complicated expression.

Advantages of function procedures

One advantage of this version of the bisection algorithm is the ability to easily modify the function. In Figure 10.11, modifying the function would require changing the statement in two different places in the main program. In Figure 13.2, you would only need to modify the function definition once. That one modification affects the computation of the function from all calling points in the main program.

Another advantage of this version is its readability. In the original version, the function was buried in the code of the main program, and the coefficients, a3, a2, a1, and a0, were separated from the function computation. In this version, the coefficients and the code for the function are all together. That makes the program easier to understand.

Function procedure calls

When a proper procedure is called, three items are pushed onto the run-time stack—the parameters, the return address, and storage for the local variables. Because the purpose of a function procedure is to return a value, it must allocate an extra cell on the run-time stack for the value returned. The cell for the returned value is allocated before any of the other items on the stack. Allocation takes place on the run-time stack in the following order when you call a function procedure:

Allocation for a function procedure

- Push storage for the return value.
- Push the parameters.
- Push the return address.
- Push storage for the local variables.

Like storage for the local variables, the cell for the return value has some undefined value when the function is called. This is in contrast to the parameters, which get values of or references to the actual parameters, and the return address, which gets the address of the statement to execute when the function terminates.

Another difference in detail between proper procedures and function procedures is flow of control when the procedure terminates. The difference is:

Flow of control with proper procedures and function procedures

- When a proper procedure terminates, control is passed to the statement *following* the calling statement.
- When a function procedure terminates, control is passed to the *calling* statement.

This difference is related to the difference between the way a proper procedure is called and the way a function procedure is called. When you call a proper procedure, the call is its own statement. For example, the proper procedure call

Proper procedure calls stand alone.

```
StdLog.String("Mr. K. Kong")
```

stands alone, apart from any other statement. However, when you call a function procedure, it is always part of another statement. For example, in the function procedure call to ABS

Function procedure calls are part of another statement.

```
x := ABS(-3.7)
```

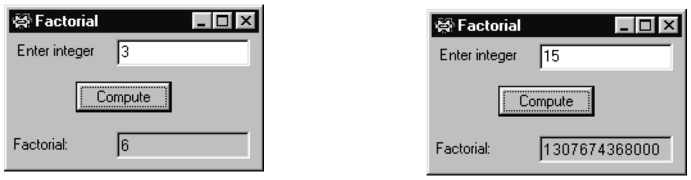
the function call is part of an assignment statement, while in the call

```
StdLog.Real(ABS(-3.7))
```

the function call is part of a procedure call. You can see from the proper procedure call that when the call terminates, there is nothing left for the calling statement to do. The next statement to execute will be the one following the procedure call. Also, from the function call you can see that when the call terminates, the value 3.7 is returned. The statement that makes the call must do something with this returned value, either assign it to x or output it to the Log. Therefore, control must be returned to the calling statement.

A factorial function

The next program illustrates allocation on the run-time stack. It computes the factorial of an integer entered by the user as shown in Figure 13.3. The factorial function is not defined for values less than zero. If the user enters a number less than zero the program changes the number entered to zero. Also, the factorial function produces very large values even for small parameters. If the number entered is greater than 20, its factorial exceeds the range of a LONGINT, which is 9,223,372,036,854,775,807. If the user enters a number greater than 20, the program guards against numerical overflow by changing it to zero. Otherwise, an overflow trap would occur.



Enter integer 3
Factorial: 6

Enter integer 15
Factorial: 1307674368000

Figure 13.3 The dialog box for the factorial function of Listing 13.4.

The program, which is shown in Figure 13.4, contains two procedures—ComputeFactorial, which is called by the framework when the user clicks the button in the dialog box, and Factorial, which is called by ComputeFactorial.

Figure 13.5 shows the storage allocation for the program in Figure 13.4. The proper procedure ComputeFactorial has no parameters and no local variables. So, when it is called only the return address is stored on the run-time stack. The function procedure Factorial has one parameter and two local variables. So, when it is called five items are stored on the run-time stack—storage for the returned value, one parameter, the return address, and storage for the two local variables.

Figure 13.5(a) shows the storage allocated for global variables d.num and d.factorial when module Pbox13B is loaded. Their values are set by the initialization code for the module. These storage cells remain in memory until the module is unloaded.

Figure 13.5(b) shows what happens when the user enters the value 3 in the dialog box. The text field of the dialog box is linked to the interactor field d.num. So, d.num gets the value 3.

```
MODULE Pbox13B;
    IMPORT Dialog;
    VAR
        d*: RECORD
            num*: INTEGER;
            factorial-: LONGINT
        END;

    PROCEDURE Factorial (n: INTEGER): LONGINT;
        VAR
            i: INTEGER;
            fact: LONGINT;
    BEGIN
        ASSERT((0 <= n) & (n <= 20), 20);
        fact := 1;
        FOR i := 1 TO n DO
            fact := fact * i
        END;
        RETURN fact
    END Factorial;

    PROCEDURE ComputeFactorial*;
    BEGIN
        IF (0 <= d.num) & (d.num <= 20) THEN
            d.factorial := Factorial(d.num)  (* ra1 *)
        ELSE
            d.num := 0;
            d.factorial := 1
        END;
        Dialog.Update(d)
    END ComputeFactorial;

BEGIN
    d.num := 0;
    d.factorial := 1
END Pbox13B.
```

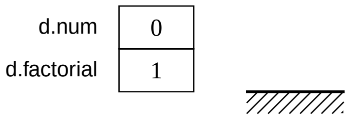
Figure 13.4
A program to compute the factorial of an integer with a function.

Figure 13.5(c) shows what happens when the user clicks the button in the dialog box. The button of the dialog box is linked to procedure ComputeFactorial, so the framework calls ComputeFactorial. The return address ra0 is the location of some instruction within the framework not visible in the program of Figure 13.4. After the IF statement executes, the next statement is

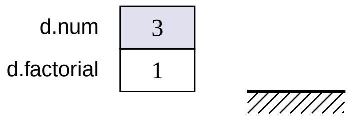
```
d.factorial := Factorial(d.num);  (* ra1 *)
```

which is a call to the function procedure Factorial.

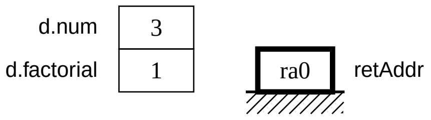
Figure 13.5(d) shows the storage allocated on the run-time stack immediately after the call to Factorial. Five items are pushed onto the stack in this order: storage for the returned value labeled retVal, the value of the actual parameter labeled n, the



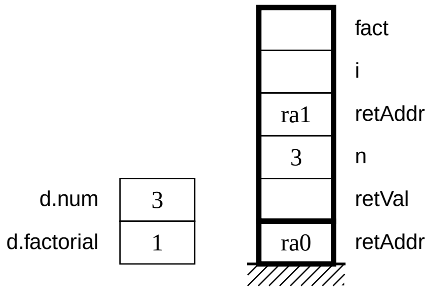
(a) After loading module Pbox13B



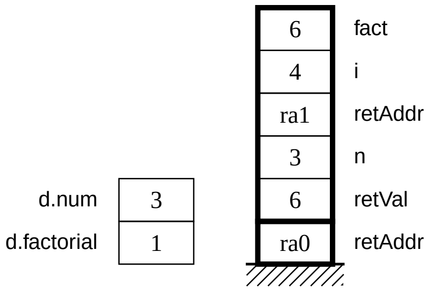
(b) After user enters d.num



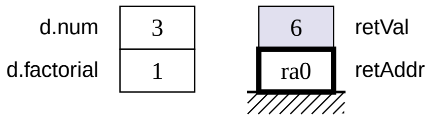
(c) After call to ComputeFactorial



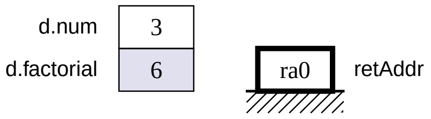
(d) After call to Factorial(d.num)



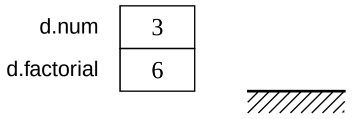
(e) Before return to ComputeFactorial



(f) After return to ComputeFactorial



(g) Before return to framework



(h) After return to framework

Figure 13.5
Memory allocation for the program in Figure 13.4.

return address labeled retAddr, the first local variable labeled i, and the second local variable labeled fact. When you call a function procedure, after the procedure executes control returns to the same statement that called it. Therefore, the return address stored on the stack is the address of the statement that made the call. The program listing has the comment (* ra1 *) on the same line as the calling statement to indicate the return address for the function procedure. As usual, the stack frame is outlined in bold on the stack in the figure.

Figure 13.5(e) shows the values on the stack after Factorial has executed and just before its return to ComputeFactorial. Factorial has used local variable i to compute the value 6 for fact. The statement

```
RETURN fact
```

assigns the value of fact to the retVal storage location. Then the return address ra1 is used to determine which instruction to execute next.

Figure 13.5(f) shows the run-time stack immediately after the return from Factorial. The value returned is available to the calling procedure. Control is returned to the instruction at ra1, which is the location of the statement

```
d.factorial := Factorial(d.num);  (* ra1 *)
```

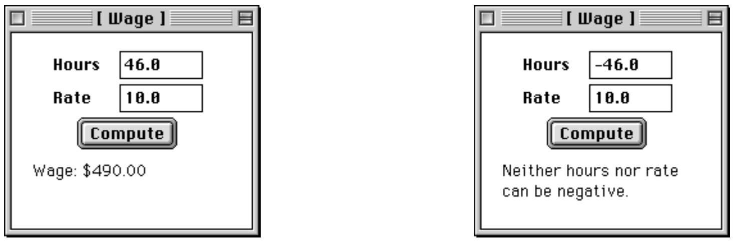
The entire stack frame is deallocated except for the value returned. This statement now completes its execution by assigning the value returned to d.factorial.

Figure 13.5(g) shows the allocated storage just before the return from ComputeFactorial. The dialog box has been updated so the computed value of d.factorial is visible on the screen.

Figure 13.5(h) shows the run-time stack immediately after the return from ComputeFactorial. Control has returned to ra0, which is the address of some statement in the framework. Storage for variables d.num and d.factorial will remain allocated with their current values until the values are changed by the program or the module is unloaded.

A function to compute wages

The next program is yet another example of computing a wage with possibility of overtime. The computation is performed in a function with preconditions that are guaranteed to be met by the calling procedure. Figure 13.6 shows the dialog box.



Hours 46.0
Rate 10.0
Wage: $490.00

Hours -46.0
Rate 10.0
Neither hours nor rate can be negative.

Figure 13.6
The dialog box for the wage function of Figure 13.7.

```
MODULE Pbox13C;
   IMPORT Dialog, PboxStrings;
   VAR
      d*: RECORD
         hours*, rate*: REAL;
         message-: ARRAY 64 OF CHAR
      END;

   PROCEDURE Wages (hrs, rt: REAL): REAL;
   BEGIN
      ASSERT(hrs >= 0.0, 20);
      ASSERT(rt >= 0.0, 21);
      IF hrs <= 40.0 THEN
         RETURN hrs * rt
      ELSE
         RETURN 40.0 * rt + (hrs - 40.0) * 1.5 * rt
      END
   END Wages;

   PROCEDURE ComputeWages*;
      VAR
         wageString: ARRAY 16 OF CHAR;
   BEGIN
      IF (d.hours >= 0.0) & (d.rate >= 0.0) THEN
         PboxStrings.RealToString(Wages(d.hours, d.rate), 1, 2, wageString);  (* ra1 *)
         d.message := "Wage: $" + wageString
      ELSE
         d.message := "Neither hours nor rate can be negative."
      END;
      Dialog.Update(d)
   END ComputeWages;

BEGIN
   d.hours := 0.0; d.rate := 0.0;
   d.message := ""
END Pbox13C.
```

Figure 13.7
A program to compute wages with possible overtime with a function procedure.

Figure 13.7 is the program for the dialog box in Figure 13.6. As usual, the compute button is linked to an exported procedure ComputeWages. The actual computation is performed by the function procedure Wages, which has two formal parameters, hrs and rt. It is typical for the formal parameters to be an abbreviation of the actual parameters. hrs is an abbreviation for d.hours and rt is an abbreviation for d.rate.

Figure 13.8 shows the storage allocation for the program. Figure 13.8(a) assumes the user has entered 46.0 in the dialog box for d.hours and 10.0 in the dialog box for d.rate. When the user clicks the button, the framework calls ComputeWages, which has no parameters and one local variable. The stack frame consists of the return address of some statement in the framework and storage for the one local variable.

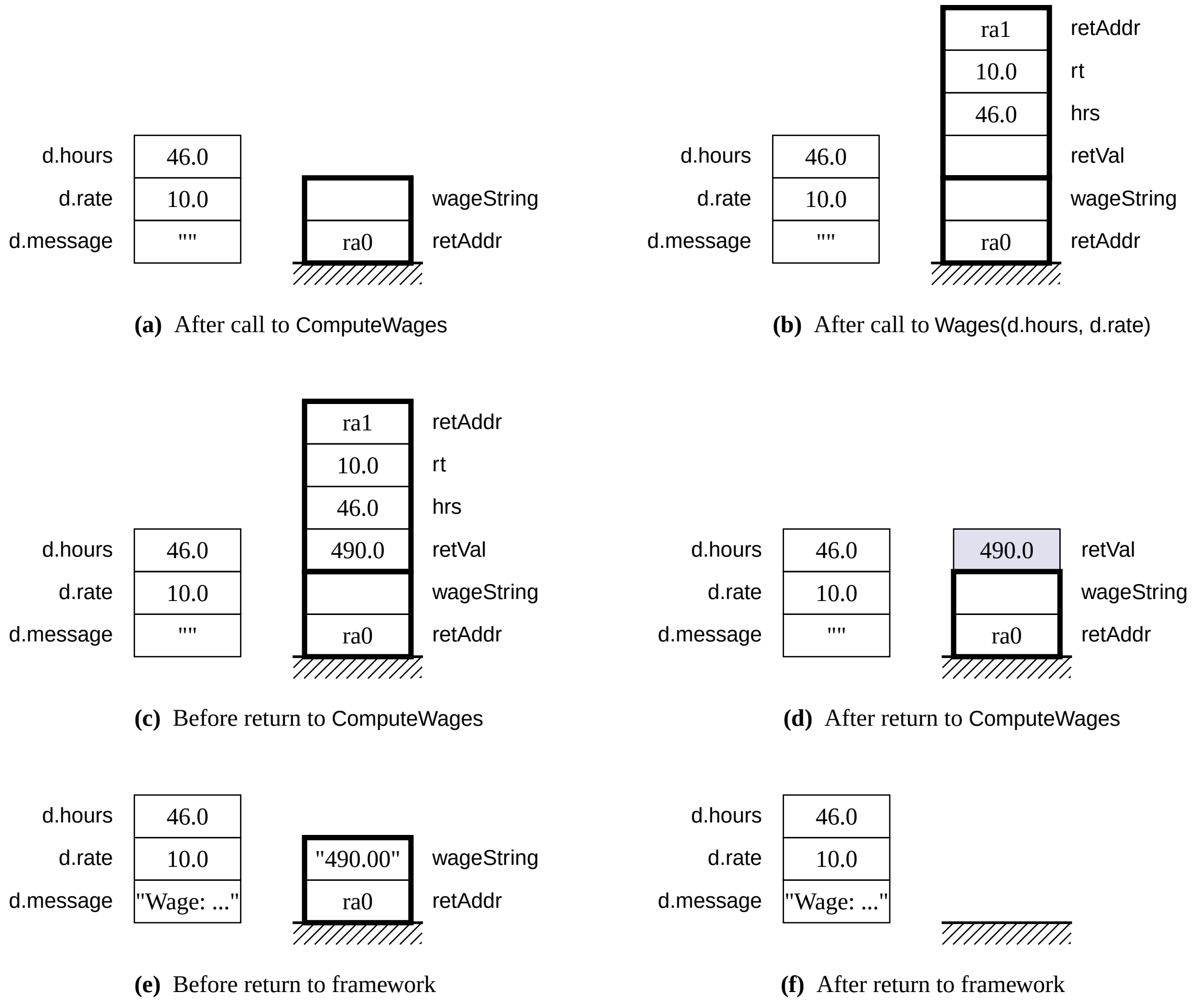

d.hours
d.rate
d.message
46.0
10.0
""
ra0
wageString
retAddr
(a) After call to ComputeWages
ra1
10.0
46.0
retAddr
rt
hrs
retVal
wageString
retAddr
(b) After call to Wages(d.hours, d.rate)
490.0
(c) Before return to ComputeWages
(d) After return to ComputeWages
"490.00"
"Wage: ..."
(e) Before return to framework
(f) After return to framework

Figure 13.8 Memory allocation for the program in Figure 13.7.

Figure 13.8(b) shows allocation on the run-time stack when function procedure Wages is called. First on the stack is storage for the returned value labeled retVal. Then the formal parameters hr and rt are pushed. Because they are called by value, the cells contain the values of the actual parameters d.hours and d.rate. Next on the stack is the return address ra1, which is the address of the calling statement

```
PboxStrings.RealToString(Wages(d.hours, d.rate), 1, 2, wageString)
```

Function Wages verifies that the preconditions are met. It tests for the possibility of overtime by comparing hrs with 40.0. There are two RETURN statements within the function, only one of which will execute. When the computer encounters a RETURN within a procedure, the procedure terminates immediately without executing any other statements that may follow the RETURN statement. This program also shows that a RETURN statement can be followed by an arbitrary expression of the correct type. It need not be followed by a single variable as was the RETURN in function Factorial in Figure 13.4. In Figure 13.8(c) the value of the expression to be returned is stored in the cell labeled retVal for the returned value.

Figure 13.8(d) shows the run-time stack after the return to the calling procedure. The returned value of 490.0 is given to the calling statement, which proceeds to use it as the actual parameter in a call to the proper procedure PboxStrings.RealToString. The stack frame for the call to this procedure is not shown in Figure 13.8. The call to the proper procedure give the string value to wageString as shown in Figure 13.8(e). Finally, after the dialog box is updated to show the result, control is returned to the framework in Figure 13.8(f).

Using function procedures

All the examples of call by result and call by reference so far have been with proper procedures instead of function procedures. Component Pascal permits function procedures to use call by result and call by reference. However, such parameters with function procedures are usually inappropriate. The purpose of a function is to return a single value. In mathematics, you give a function a value, *x*, and it returns a value *f*(*x*). The function depends on *x* but does not change the value of *x*. Programs are easier to understand if functions behave as they do in mathematics and do not change their actual parameters. This book has no examples of functions with parameters called by result or called by reference.

Do not use call by result or call by reference with functions

Because the formal parameters of function procedures are usually called by value, the actual parameters may be expressions.

Example 13.1 The function call

```
d.factorial := Factorial(2 * d.num + 1)
```

where d.factorial, d.num, and Factorial are declared as in Figure 13.4, is legal. If d.num has value 3, then formal parameter n would get the value of 7 at the start of function execution. ▮

In general, a function call can be placed anywhere an expression is allowed. Figure 13.4 has the function call, Factorial(d.num), on the right side of an assignment statement. The function call can just as easily be part of a larger expression.

Example 13.2 The statement

```
num := Math.Pi * Math.IntPower(x, 2) / Factorial(i)
```

is legal, where num and x are real variables, i is an integer variable, and Factorial is declared as in Figure 13.4. ▮

The RETURN statement for a function can occur anywhere, even within the body of a loop. When it is encountered, the function terminates immediately.

Example 13.3 Suppose that i, max, and testDivisor are integer variables, and the following code executes from within a function procedure with return type BOOLEAN.

```
WHILE i < max DO
    IF i MOD testDivisor = 0 THEN
        RETURN TRUE
    END;
    INC(i)
END;
RETURN FALSE
```

If at any time during execution of the loop i gets a value that makes the test of the IF statement true, the function will terminate immediately and will return true. Otherwise, the loop will terminate because the WHILE test will become false, and the function will return false. ■

Exercises

1. If the function procedure Exr1 is defined as

```
PROCEDURE Exr1 (a, b: INTEGER): INTEGER;
BEGIN
    IF a < b THEN
        RETURN 2 * a
    ELSE
        RETURN 2 * b
    END
 END Exr1
```

then what does each of the following code fragments output to the Log?

(a)
```
i := 12;
j := 3;
StdLog.Int(Exr1(i, j))
```

(b)
```
i := 4;
j := Exr1(2 * i + 1, 10);
StdLog.Int(j)
```

(c)
```
StdLog.Int(Exr1(Exr1(3, 2), Exr1(4, 5)))
```

2. Assume that the user enters 2 for d.a and 3 for d.b. Draw a picture of the memory allocation as in Figure 13.5 produced by the module listed below for the following times.

(a) After user enters d.a and d.b.
(b) After call to ComputeProb2.
(c) After call to Prob2.
(d) Before return to ComputeProb2.
(e) After return to ComputeProb2.
(f) Before return to framework.
(g) After return to framework.

```
MODULE Pbox13Prob2;
    IMPORT Dialog;
    VAR
        d*: RECORD
            a*, b*: INTEGER;
            c-: INTEGER
        END;

    PROCEDURE Prob2 (e, f: INTEGER): INTEGER;
        VAR
            i, j: INTEGER;
    BEGIN
        i := e; j := f;
        INC(i); INC(j);
        IF i > 2 THEN
            RETURN 2 * i
        ELSE
            RETURN 3 * j
        END
    END Prob2;

    PROCEDURE ComputeProb2*;
    BEGIN
        d.c := Prob2(d.a, d.b);  (* ra1 *)
        Dialog.Update(d)
    END ComputeProb2;

BEGIN
    d.a := 0; d.b := 0;
    d.c := 0
END Pbox13Prob2.
```

Problems

3. For Chapter 6, Problem 13, write the program to find the sales commission. Declare

```
PROCEDURE Commission (sales: REAL): REAL
```

to compute the commission from the amount of sales. Implement a precondition for the function that the entered number cannot be less than 0. Insure that the precondition is met in the calling procedure. If it is not met, set the entered sales to $0.00 and the displayed commission to $0.00.

4. Write the bowling prize program of Chapter 6, Problem 14. Declare

```
PROCEDURE BowlingPrize (scr1, scr2, scr3: INTEGER): REAL
```

to compute the prize from the three scores, scr1, scr2, and scr3. Implement a precondition for the function that none of the scores can be less than 0 or greater than 300. Insure that the precondition is met in the calling procedure. If it is not met, set the entered scores to 0 and the displayed prize to $0.

5. For Chapter 6, Problem 16, write the program to find the grade point average. Declare

```
PROCEDURE GPA (numA, numB, numC, numD, numF: INTEGER): REAL
```

to compute the grade point average from the letter grades. Implement a precondition for the function that none of the entered numbers can be less than 0. Insure that the precondition is met in the calling procedure. If it is not met, set the entered numbers to 0 and the displayed grade point average to a blank field.

6. For Chapter 8, Problem 19, write the program to determine the traffic fine. Declare

```
PROCEDURE TrafficFine (speed: INTEGER): REAL
```

to compute the traffic fine from the speed. Implement a precondition for the function that the entered speed cannot be less than 0. Insure that the precondition is met in the calling procedure. If it is not met, set the entered speed to 0 and the displayed fine to 0.

7. Write the Frisbee program of Chapter 8, Problem 22. Declare

```
PROCEDURE OrderCost (numFr: INTEGER): REAL
```

to compute the cost of the order from the number of Frisbees ordered. Implement a precondition for the function that the entered number of frisbees cannot be less than 0. Insure that the precondition is met in the calling procedure. If it is not met, set the entered numbers to 0 and the displayed cost of the order to $0.00.

8. Write the schedule program of Chapter 6, Problem 19. Declare

```
PROCEDURE RegPeriod (lastIntl: CHAR): INTEGER
```

to compute the registration period from the user's last initial. Allow the user to input uppercase or lowercase letters. Implement a precondition for the function that lastIntl is alphabetic. Insure that the precondition is met in the calling procedure. If it is not met, output an error message in the message field.

9. For Chapter 8, Problem 24, write the program to determine whether a given year is a leap year. Declare

```
PROCEDURE IsLeapYear (Year: INTEGER): BOOLEAN
```

to determine the leap year from the year. Implement a precondition for the function that the year entered cannot be less than 0. Insure that the precondition is met in the calling procedure. If it is not met, set the entered year to 0 and the displayed message to a blank field.

10. For Chapter 10, Problem 35, write the program to raise a number to a power. Declare

```
PROCEDURE Power (base: REAL; expon: INTEGER): REAL
```

to compute the base raised to the exponent. There are no preconditions for the function. If the exponent is 0, return 1 for the function even if the base is 0.0.

11. For Chapter 10, Problem 36, write the program to estimate the value of the base of the natural logarithms, *e*. Declare

```
PROCEDURE EstE (numTrm: INTEGER): REAL
```

to compute the estimate from the number of terms. Implement a precondition for the function that the number of terms numTrm cannot be less than 2. Insure that the precondition is met in the calling procedure. If it is not met, set the entered number to 2 and the displayed estimate to 2.

12. For Chapter 10, Problem 38, write the program to determine if a number is prime. Declare

```
PROCEDURE IsPrime (n: INTEGER): BOOLEAN
```

to determine whether the number is prime. Implement a precondition for the function that n cannot be less than 1. Insure that the precondition is met in the calling procedure. If it is not met, set the entered number to 1 and the displayed message to a blank field.