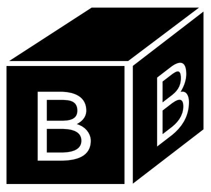


Chapter *13*

Function Procedures

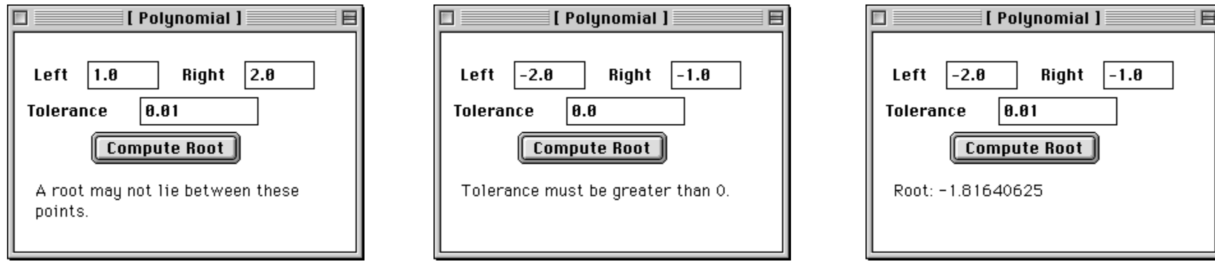


Figure 13.1
Three executions of the bisection algorithm of Figure 13.2.

```
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;
```

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

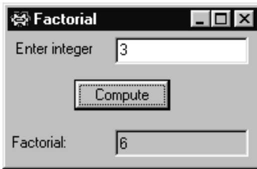
```
    PROCEDURE ComputeRoot*;
        VAR
            left, mid, right: REAL;
            fLeft, fMid: REAL;
            rootString: ARRAY 32 OF CHAR;
    BEGIN
        IF d.tolerance <= 0.0 THEN
            d.message := "Tolerance must be greater than 0."
        ELSIF F(d.left) * F(d.right) > 0 THEN (* 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.
```

Allocation for a function procedure

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

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.



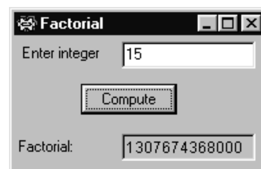


Figure 13.3
The dialog box for the factorial function of Listing 13.4.

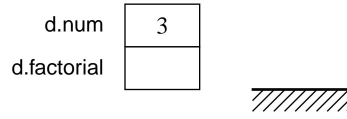
```
MODULE Pbox13B;
    IMPORT Dialog;
    VAR
        d*: RECORD
            num*: INTEGER;
            factorial-: LONGINT
        END;

    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;
```

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

```
    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.
```

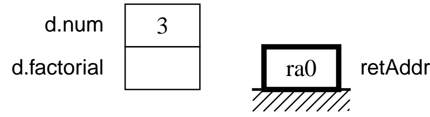
```
PROCEDURE Factorial (n: INTEGER): LONGINT;
   VAR
      i: INTEGER;
      fact: LONGINT;
BEGIN
   ASSERT((0 <= n) & (n <= 20), 20);
   fact := 1;
   FOR i := 1 TO n DO
      fact := fact * i
   END;
   RETURN fact
END Factorial;

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;
```

Global variables **Run-time stack**



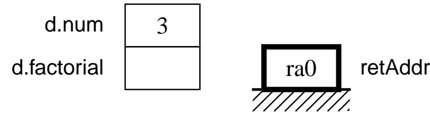
```
PROCEDURE Factorial (n: INTEGER): LONGINT;
   VAR
      i: INTEGER;
      fact: LONGINT;
BEGIN
   ASSERT((0 <= n) & (n <= 20), 20);
   fact := 1;
   FOR i := 1 TO n DO
      fact := fact * i
   END;
   RETURN fact
END Factorial;

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;
```

Global variables **Run-time stack**



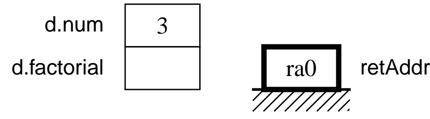
```
PROCEDURE Factorial (n: INTEGER): LONGINT;
   VAR
      i: INTEGER;
      fact: LONGINT;
BEGIN
   ASSERT((0 <= n) & (n <= 20), 20);
   fact := 1;
   FOR i := 1 TO n DO
      fact := fact * i
   END;
   RETURN fact
END Factorial;

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;
```

Global variables **Run-time stack**



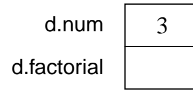
```
PROCEDURE Factorial (n: INTEGER): LONGINT;
   VAR
      i: INTEGER;
      fact: LONGINT;
BEGIN
   ASSERT((0 <= n) & (n <= 20), 20);
   fact := 1;
   FOR i := 1 TO n DO
      fact := fact * i
   END;
   RETURN fact
END Factorial;

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;
```

Global variables **Run-time stack**

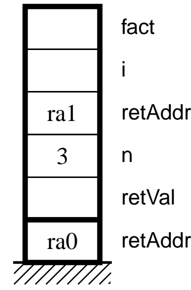


```
PROCEDURE Factorial (n: INTEGER): LONGINT;
   VAR
      i: INTEGER;
      fact: LONGINT;
BEGIN
   ASSERT((0 <= n) & (n <= 20), 20);
   fact := 1;
   FOR i := 1 TO n DO
      fact := fact * i
   END;
   RETURN fact
END Factorial;

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;
```

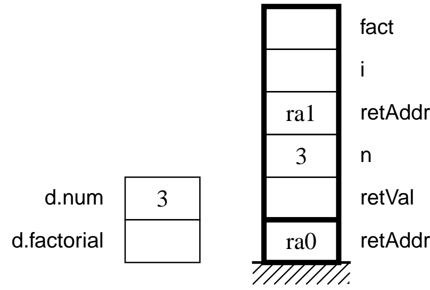
Global variables



Run-time stack

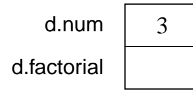


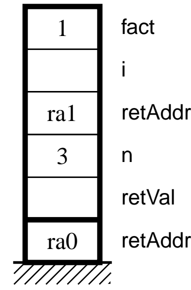
```
PROCEDURE Factorial (n: INTEGER): LONGINT;
   VAR
      i: INTEGER;
      fact: LONGINT;
BEGIN
   ASSERT((0 <= n) & (n <= 20), 20);
   fact := 1;
   FOR i := 1 TO n DO
      fact := fact * i
   END;
   RETURN fact
END Factorial;

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;
```



```
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;
```

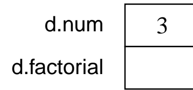
Global variables **Run-time stack**

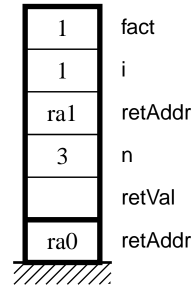




```
PROCEDURE Factorial (n: INTEGER): LONGINT;
   VAR
      i: INTEGER;
      fact: LONGINT;
BEGIN
   ASSERT((0 <= n) & (n <= 20), 20);
   fact := 1;
   FOR i := 1 TO n DO
      fact := fact * i
   END;
   RETURN fact
END Factorial;

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;
```

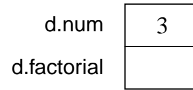
Global variables **Run-time stack**

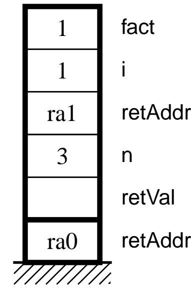




```
PROCEDURE Factorial (n: INTEGER): LONGINT;
   VAR
      i: INTEGER;
      fact: LONGINT;
BEGIN
   ASSERT((0 <= n) & (n <= 20), 20);
   fact := 1;
   FOR i := 1 TO n DO
      fact := fact * i
   END;
   RETURN fact
END Factorial;

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;
```

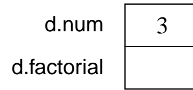
Global variables **Run-time stack**

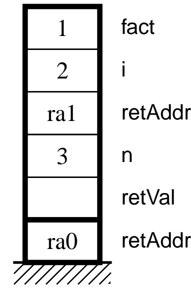




```
PROCEDURE Factorial (n: INTEGER): LONGINT;
   VAR
      i: INTEGER;
      fact: LONGINT;
BEGIN
   ASSERT((0 <= n) & (n <= 20), 20);
   fact := 1;
   FOR i := 1 TO n DO
      fact := fact * i
   END;
   RETURN fact
END Factorial;

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;
```

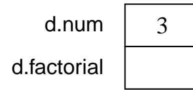
Global variables **Run-time stack**

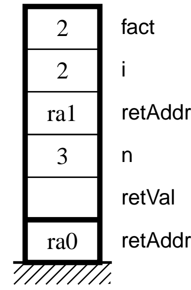




```
PROCEDURE Factorial (n: INTEGER): LONGINT;
   VAR
      i: INTEGER;
      fact: LONGINT;
BEGIN
   ASSERT((0 <= n) & (n <= 20), 20);
   fact := 1;
   FOR i := 1 TO n DO
      fact := fact * i
   END;
   RETURN fact
END Factorial;

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;
```

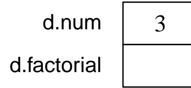
Global variables **Run-time stack**



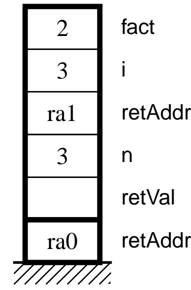


```
PROCEDURE Factorial (n: INTEGER): LONGINT;
   VAR
      i: INTEGER;
      fact: LONGINT;
BEGIN
   ASSERT((0 <= n) & (n <= 20), 20);
   fact := 1;
   FOR i := 1 TO n DO
      fact := fact * i
   END;
   RETURN fact
END Factorial;

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;
```

Global variables

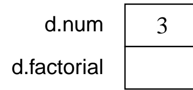


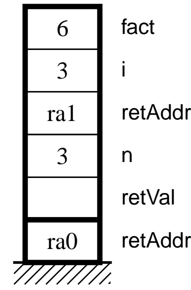
Run-time stack



```
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;
```

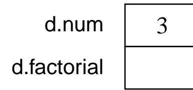
Global variables **Run-time stack**

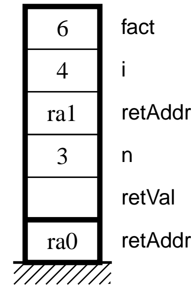




```
PROCEDURE Factorial (n: INTEGER): LONGINT;
    VAR
        i: INTEGER;
        fact: LONGINT;
BEGIN
    ASSERT((0 <= n) & (n <= 20), 20);
    fact := 1;
    FOR i := 1 TO n DO
        fact := fact * i
    END;
    RETURN fact
END Factorial;

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;
```

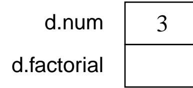
Global variables **Run-time stack**

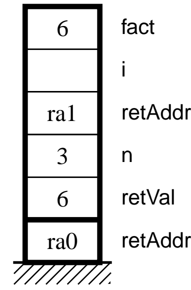




```
PROCEDURE Factorial (n: INTEGER): LONGINT;
   VAR
      i: INTEGER;
      fact: LONGINT;
BEGIN
   ASSERT((0 <= n) & (n <= 20), 20);
   fact := 1;
   FOR i := 1 TO n DO
      fact := fact * i
   END;
   RETURN fact
END Factorial;

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;
```

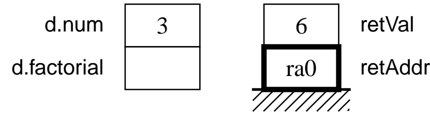
Global variables **Run-time stack**





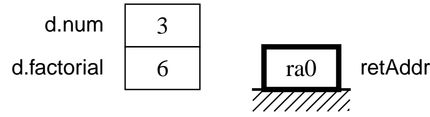
```
PROCEDURE Factorial (n: INTEGER): LONGINT;
   VAR
      i: INTEGER;
      fact: LONGINT;
BEGIN
   ASSERT((0 <= n) & (n <= 20), 20);
   fact := 1;
   FOR i := 1 TO n DO
      fact := fact * i
   END;
   RETURN fact
END Factorial;

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;
```

Global variables **Run-time stack**



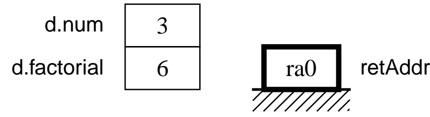
```
PROCEDURE Factorial (n: INTEGER): LONGINT;
   VAR
      i: INTEGER;
      fact: LONGINT;
BEGIN
   ASSERT((0 <= n) & (n <= 20), 20);
   fact := 1;
   FOR i := 1 TO n DO
      fact := fact * i
   END;
   RETURN fact
END Factorial;

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;
```

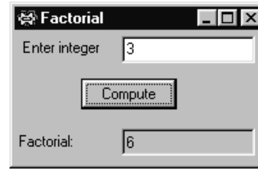
Global variables **Run-time stack**



```
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;
```

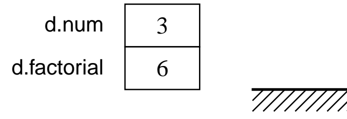
Global variables **Run-time stack**

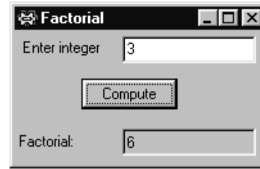


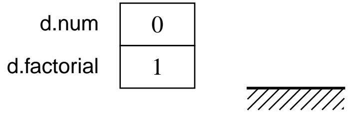


```
PROCEDURE Factorial (n: INTEGER): LONGINT;
   VAR
      i: INTEGER;
      fact: LONGINT;
BEGIN
   ASSERT((0 <= n) & (n <= 20), 20);
   fact := 1;
   FOR i := 1 TO n DO
      fact := fact * i
   END;
   RETURN fact
END Factorial;

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;
```

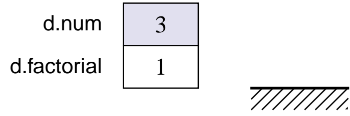
Global variables **Run-time stack**



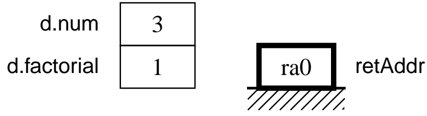




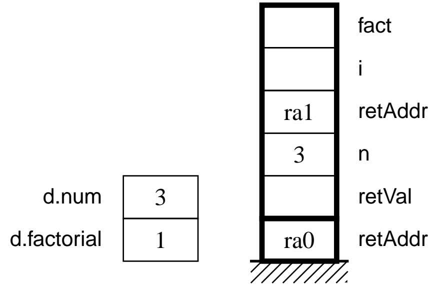
(a) After loading module Pbox13B



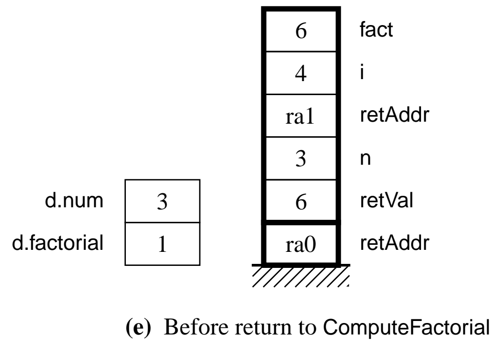
(b) After user enters d.num



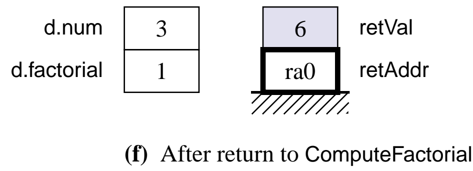
(c) After call to ComputeFactorial



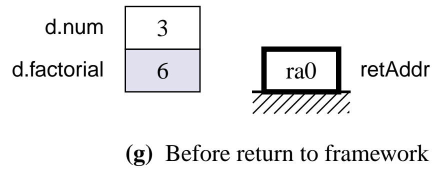
(d) After call to Factorial(d.num)



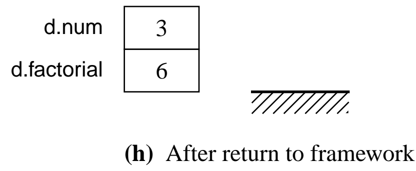
(e) Before return to ComputeFactorial



(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.

[Wage]
Hours 46.0
Rate 10.0
Compute
Wage: $490.00

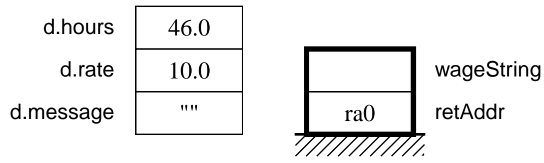
[Wage]
Hours -46.0
Rate 10.0
Compute
Neither hours nor rate
can be negative.

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

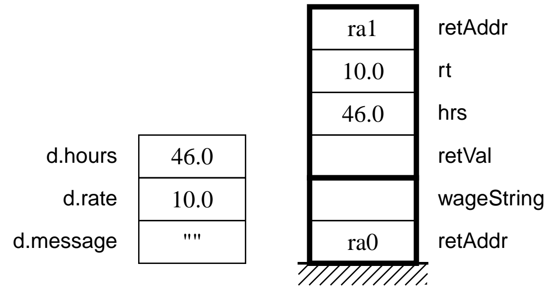
```
MODULE Pbox13C;
   IMPORT Dialog, PboxStrings;
   VAR
      d*: RECORD
         hours*, rate*: REAL;
         message-: ARRAY 64 OF CHAR
      END;

   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;
```

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

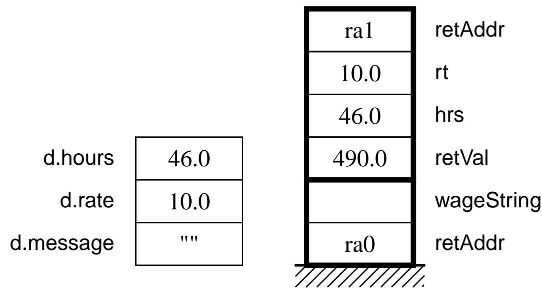
```
    PROCEDURE ComputeWages*;
        VAR
            wageString: ARRAY 16 OF CHAR;
    BEGIN
        IF (d.hours >= 0.0) & (d.rate >= 0.0) THEN
            PboxStrings.RealToString(Wages(d.hours, d.rate), 1, 2, wageString);  (* ra1 *)
            d.message := "Wage: $" + wageString
        ELSE
            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.
```



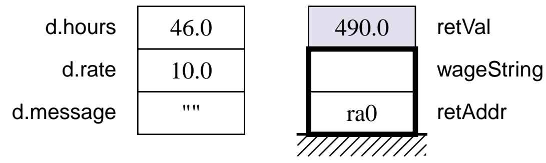
(a) After call to ComputeWages



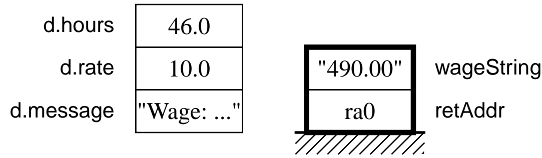
(b) After call to Wages(d.hours, d.rate)



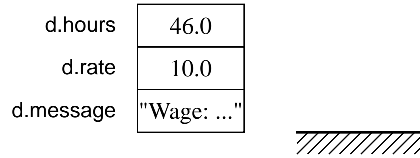
(c) Before return to ComputeWages



(d) After return to ComputeWages



(e) Before return to framework



(f) After return to framework

Figure 13.8
Memory allocation for the program in Figure 13.7.