CHAPTER 3

Recursion

This chapter introduces recursion, a very powerful programming tool that is often misunderstood by beginning students of programming. We define recursion, introduce its use in Java, and present several examples. We also examine an implementation of recursion using stacks. Finally, we discuss the advantages and disadvantages of using recursion in problem solving.

3.1 RECURSIVE DEFINITION AND PROCESSES

Many objects in mathematics are defined by presenting a process to produce the object. For example, π is defined as the ratio of the circumference of a circle to its diameter. This is equivalent to the set of instructions: obtain the circumference of a circle and its diameter, divide the former by the latter, and call the result π. Clearly, the process specified must terminate with a definite result.

Factorial Function

Another example of a definition specified by a process is that of the factorial function, which plays an important role in mathematics and statistics. Given a positive integer n, ***n factorial*** is defined as the product of all integers between n and 1. For example, 5 factorial equals $5 * 4 * 3 * 2 * 1 = 120$, and 3 factorial equals $3 * 2 * 1 = 6$. 0 factorial is defined as 1. In mathematics, the exclamation mark (!) is often used to denote the factorial function. We may therefore write the definition of this function as follows:

```
n! = 1                                  if n = 0
n! = n * (n - 1) * (n - 2) * ... * 1    if n > 0
```

The three dots are really a shorthand for all the numbers between $n - 3$ and 2 multiplied together. In order to avoid this shorthand in the definition of $n!$, we would have to list a formula for $n!$ for each value of n separately, as follows:

```
0! = 1
1! = 1
```

```
2! = 2 * 1
3! = 3 * 2 * 1
4! = 4 * 3 * 2 * 1
.....
```

Of course, we cannot hope to list a formula for the factorial of very integer. In order to define the function precisely without using shorthand or an infinite set of definitions, we present an algorithm that accepts an integer n and returns the value of $n!$.

```
prod = 1;
for (x = n; x > 0; x--)
     prod = prod * x;
return prod;
```

Such an algorithm is called ***iterative*** because it calls for the explicit repetition of some process until a certain condition is met. This algorithm can be translated readily into a Java method that returns $n!$ when n is input as a parameter. An algorithm may be thought of as a program for an "ideal" machine without any of the practical limitations of a real computer and may therefore be used to define a mathematical function. A Java method, however, cannot serve as the mathematical definition of the factorial function because of such limitations as precision and the finite size of a real machine.

Let us look more closely at the definition of $n!$ that lists a separate formula for each value of n. We may note, for example, that 4! equals 4 * 3 * 2 * 1, which equals 4 * 3!. In fact, for any $n > 0$, we see that $n!$ equals $n * (n - 1)!$. Multiplying n by the product of all integers from $n - 1$ to 1 yields the product of all integers from n to 1. We may therefore define:

```
0! = 1
1! = 1 * 0!
2! = 2 * 1!
3! = 3 * 2!
4! = 4 * 3!
  ...
```

or, using the mathematical notation used earlier:

```
n! = 1              if n == 0
n! = n * (n - 1)!   if n > 0.
```

This definition may appear quite strange, since it defines the factorial function in terms of itself. It seems to be a circular definition and totally unacceptable until we realize that the mathematical notation is only a concise way of writing out the infinite number of equations necessary to define $n!$ for each n. 0! is defined directly as 1. Once 0! has been defined, defining 1! as 1 * 0! is not circular at all. Similarly, once 1! has been defined, defining 2! as 2 * 1! is equally straightforward. It may be argued that the latter notation is more precise than the definition of $n!$ as $n * (n - 1) * \ldots * 1$ for $n > 0$ because it does not resort to three dots to be filled in by the hopefully logical intuition of

the reader. Such a definition, which defines an object in terms of a simpler case of itself, is called a ***recursive definition***.

Let us see how the recursive definition of the factorial function may be used to evaluate 5!. The definition states that 5! equals 5 * 4!. Thus, before we can evaluate 5!, we must first evaluate 4!. Using the definition once more, we find that 4! = 4 * 3!. Therefore, we must evaluate 3!. Repeating this process, we have:

```
1 5! = 5 * 4!
2      4! = 4 * 3!
3               3! = 3 * 2!
4                        2! = 2 * 1!
5                                 1! = 1 * 0!
6                                          0! = 1
```

Each case is reduced to a simpler case until we reach the case of 0!, which is defined directly as 1. At line (6) we have a value that is defined directly and not as the factorial of another number. We may therefore backtrack from line (6) to line (1), returning the value computed in one line to evaluate the result of the previous line. This produces:

```
6'  0! = 1
5'  1! = 1 * 0! = 1 * 1 = 1
4'  2! = 2 * 1! = 2 * 1 = 2
3'  3! = 3 * 2! = 3 * 2 = 6
2'  4! = 4 * 3! = 4 * 6 = 24
1'  5! = 5 * 4! = 5 * 24 = 120
```

Let us attempt to incorporate this process into an algorithm. Again, we want the algorithm to input a nonnegative integer n and to compute in a variable *fact* the nonnegative integer that is n factorial.

```
1 if (n == 0)
2       fact = 1;
3 else {
4       x = n - 1;
5       find the value of x!. Call it y;
6       fact = n * y;
7 } // end else
```

This algorithm exhibits the process used to compute $n!$ by the recursive definition. The key to the algorithm is, of course, line 5, where we are told to "find the value of $x!$." This requires re-executing the algorithm with input x, since the method for computing the factorial function is the algorithm itself. To see that the algorithm eventually halts, note that x equals $n - 1$ at the start of line 5. Each time the algorithm is executed, its input is one less than the preceding time, so (since the original input n was a nonnegative integer) 0 is eventually input to the algorithm. At that point, the algorithm simply returns 1. This value is returned to line 5, which asked for the evaluation of 0!. The multiplication of y (which equals 1) by n (which equals 1) is then executed and the result is

returned. This sequence of multiplications and returns continues until the original $n!$ has been evaluated. In the next section, we will see how to convert this algorithm into a Java program.

Of course, it is much simpler and more straightforward to use the iterative method for evaluation of the factorial function. We present the recursive method as a simple example to introduce recursion, not as a more effective method of solving this particular problem. Indeed, all the problems in this section can be solved more efficiently by iteration. However, later in this chapter and in subsequent chapters, we will come across examples that are more easily solved by recursive methods.

Multiplication of Natural Numbers

Another example of a recursive definition is the definition of multiplication of natural numbers. The product $a * b$, where a and b are positive integers, may be defined as a added to itself i times. This is an iterative definition. An equivalent recursive definition is:

```
a * b = a                    if b == 1
a * b = a * (b - 1) + a      if b > 1.
```

To evaluate 6 * 3 by this definition, we first evaluate 6 * 2 and then add 6. To evaluate 6 * 2, we first evaluate 6 * 1 and add 6. But 6 * 1 equals 6 by the first part of the definition. Thus

```
6 * 3 = 6 * 2 + 6 = 6 * 1 + 6 + 6 = 6 + 6 + 6 = 18.
```

The reader is urged to convert the above definition to a recursive algorithm as a simple exercise.

Note the pattern that exists in recursive definitions. A simple case of the term to be defined is defined explicitly (in the case of factorial, 0! was defined as 1; in the case of multiplication, $a * 1 = a$). The other cases are defined by applying some operation to the result of evaluating a simpler case. Thus $n!$ is defined in terms of $(n - 1)!$ and $a * b$ in terms of $a * (b - 1)$. Successive simplifications of any particular case must eventually lead to the explicitly defined trivial case. In the case of the factorial function, successively subtracting 1 from n eventually yields 0. In the case of multiplication, successively subtracting 1 from b eventually yields 1. If this were not the case, the definition would be invalid. For example, if we defined

```
n! = (n + 1)!/(n + 1)
```

or

```
a * b = a * (b + 1) - a
```

we would be unable to determine the value of 5! or 6 * 3. (You are invited to attempt to determine these values using the above definitions.) This is true despite the fact that the two equations are valid. Continually adding 1 to n or b does not eventually produce an explicitly defined case. Even if 100! was defined explicitly, how could the value of 101! be determined?

Fibonacci Sequence

Let us examine a less familiar example. The **Fibonacci sequence** is the sequence of integers:

$$0, 1, 1, 2, 3, 5, 8, 13, 21, 34, \ldots$$

Each element in this sequence is the sum of the two preceding elements (e.g., $0 + 1 = 1, 1 + 1 = 2, 1 + 2 = 3, 2 + 3 = 5, \ldots$). If we let $fib(0) = 0, fib(1) = 1$, etc., then we may define the Fibonacci sequence by the following recursive definition:

```
fib(n) = n                              if n == 0 or n == 1
fib(n) = fib(n - 2) + fib(n - 1)        if n >= 2
```

To compute *fib*(6), for example, we may apply the definition recursively to obtain:

```
fib(6) = fib(4) + fib(5) = fib(2) + fib(3) + fib(5) =
fib(0) + fib(1) + fib(3) + fib(5) = 0 + 1 + fib(3) + fib(5) =
1 + fib(1) + fib(2) + fib(5) =
1 + 1 + fib(0) + fib(1) + fib(5) =
2 + 0 + 1 + fib(5) =
3 + fib(3) + fib(4) =
3 + fib(1) + fib(2) + fib(4) =
3 + 1 + fib(0) + fib(1) + fib(4) =
4 + 0 + 1 + fib(2) + fib(3) =
5 + fib(0) + fib(1) + fib(3) =
5 + 0 + 1 + fib(1) + fib(2) =
6 + 1 + fib(0) + fib(1) =
7 + 0 + 1 = 8
```

Note that the recursive definition of the Fibonacci numbers differs from the recursive definitions of the factorial function and multiplication. The recursive definition of *fib* refers to itself twice. For example, $fib(6) = fib(4) + fib(5)$, so that in computing *fib*(6), *fib* must be applied recursively twice. However, the computation of *fib*(5) also involves determining *fib*(4), so that a great deal of computational redundancy occurs in applying the definition. In the above example, *fib*(3) is computed three separate times. It would be much more efficient to "remember" the value of *fib*(3) the first time it is evaluated and reuse it each time it is needed. An iterative method of computing *fib*(*n*) such as the following is much more efficient:

```
if (n <= 1)
    return n;
lofib = 0;
hifib = 1;
for (i = 2; i <= n; i++) {
    x = lofib;
    lofib = hifib;
    hifib = x + lofib;
} // end for
return hifib;
```

Compare the number of additions (not including increments of the index variable i) that are performed in computing $fib(6)$ by this algorithm and by using the recursive definition. In the case of the factorial function, the same number of multiplications must be performed in computing $n!$ by the recursive and iterative methods. The same is true of the number of additions in the two methods of computing multiplication. However in the case of the Fibonacci numbers, the recursive method is far more expensive than the iterative. We shall have more to say about the relative merits of the two methods in a later section.

Binary Search

You may have received the erroneous impression that recursion is a very handy tool for defining mathematical functions but has no influence in more practical computing activities. The next example illustrates an application of recursion to one of the most common activities in computing: searching.

Consider an array of elements in which objects have been placed in some order. For example, a dictionary or a telephone book may be thought of as an array whose entries are in alphabetical order. A company payroll file may be in the order of employees' social security numbers. Suppose such an array exists and we wish to find a particular element in it. For example, we wish to look up a name in a telephone book, a word in a dictionary, or an employee in a personnel file. The process used to find such an entry is called a ***search***.

Since searching is such a common activity in computing, it is desirable to find an efficient method for performing it. Perhaps the crudest search method is the ***sequential*** or ***linear*** search, in which each item of the array is examined in turn and compared to the item being searched for until a match occurs. If the list is unordered and haphazardly constructed, a linear search may be the only way to find anything in it (unless, of course, the list is first rearranged). However, you would never use this method in looking up a name in a telephone book. Rather, you would open the book to a random page and examine the names on it. Since the names are ordered alphabetically, the examination would determine whether the search should proceed in the first or second half of the book.

Let us apply this idea to searching an array. If the array contains only one element, the problem is trivial. Otherwise, compare the item being searched for with the item at the middle of the array. If they are equal, the search has been completed successfully. If the middle element is greater than the item being searched for, the search process is repeated in the first half of the array (since if the item appears anywhere, it must appear in the first half); otherwise, the process is repeated in the second half. Note that each time a comparison is made, the number of elements yet to be searched is cut in half. For large arrays, this method is superior to a sequential search in which each comparison reduces the number of elements yet to be searched by only one. Because of the division of the array to be searched into two equal parts, this search method is called a ***binary search***.

Note that we have quite naturally defined a binary search recursively. If the item being searched for is not equal to the middle element of the array, the instructions are to search a subarray using the same method. Thus the search method is defined in

terms of itself with a smaller array as input. We are sure that the process will terminate because the input arrays become smaller and smaller, and a search of a one-element array is defined nonrecursively, since the middle element of such an array is its only element.

We now present a recursive algorithm to search a sorted array a for an element x between $a[low]$ and $a[high]$. The algorithm returns an *index* of a such that $a[index]$ equals x if such an *index* exists between *low* and *high*. If x is not found in that portion of the array, *binsrch* returns -1 (in Java, no element $a[-1]$ can exist).

```
1  if (low > high)
2     return -1;
3  mid = (low + high) / 2;
4  if (x == a[mid])
5     return mid;
6  if (x < a[mid])
7     search for x in a[low] to a[mid - 1];
8  else
9     search for x in a[mid + 1] to a[high];
```

Since the possibility of an unsuccessful search is included (i.e., the element may not exist in the array), the trivial case has been altered somewhat. A search on a one-element array is not defined directly as the appropriate index. Instead that element is compared to the item being searched for. If the two items are not equal, the search continues in the "first" or "second" half—each of which contains no elements. This case is indicated by the condition $low > high$, and its result is defined directly as -1.

Let us apply this algorithm to an example. Suppose the array a contains the elements 1, 3, 4, 5, 17, 18, 31, 33 in that order, and we wish to search for 17 (i.e., x equals 17) between item 0 and item 7 (i.e., *low* is 0, *high* is 7). Applying the algorithm, we have

Line 1: Is $low > high$? It is not, so execute line 3.
Line 3: $mid = (0 + 7)/2 = 3$.
Line 4: Is $x==a[3]$? 17 is not equal to 5, so execute line 6.
Line 6: Is $x < a[3]$? 17 is not less than 5, so perform the else clause at line 8.
Line 9: Repeat the algorithm with $low = mid + 1 = 4$ and $high = high = 7$, i.e., search the upper half of the array.
Line 1: Is $4 > 7$? No, so execute line 3.
Line 3: $mid = (4 + 7)/2 = 5$.
Line 4: Is $x==a[5]$? 17 does not equal 18, so execute line 6.
Line 6: Is $x < a[5]$? Yes, since $17 < 18$, so search for x in $a[low]$ to $a[mid - 1]$.
Line 7: Repeat the algorithm with $low = low = 4$ and $high = mid - 1 = 4$. We have isolated x between the **fourth** and the **fourth** elements of a.
Line 1: Is $4 > 4$? No, so execute line 3.
Line 3: $mid = (4 + 4)/2 = 4$.
Line 4: Since $a[4] == 17$, return $mid = 4$ as the answer. 17 is indeed the fourth element of the array.

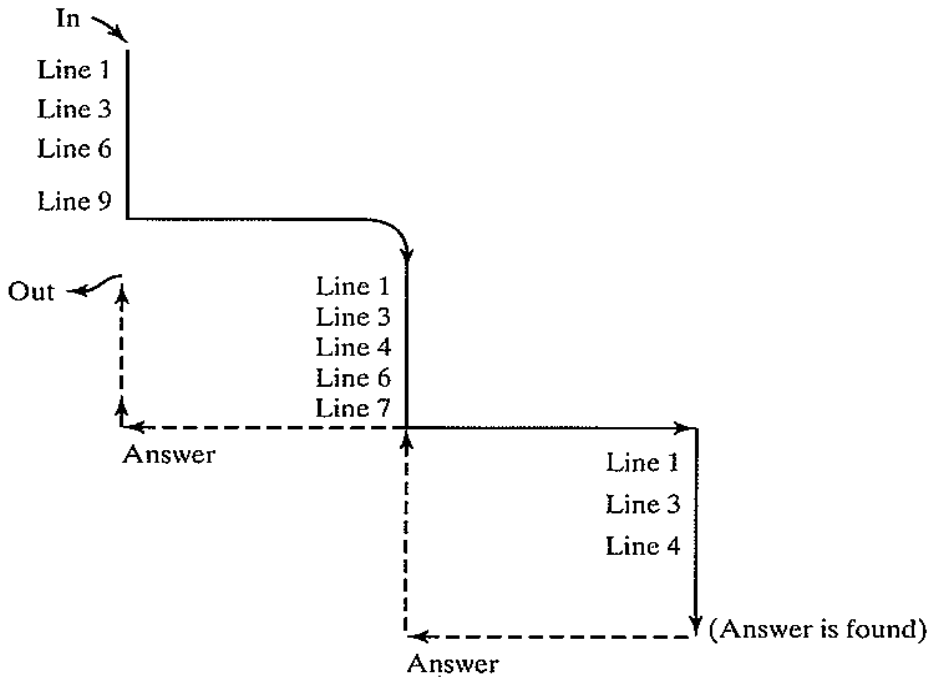

FIGURE 3.1.1 Diagrammatic representation of the binary search algorithm.

Note the pattern of calls to and returns from the algorithm. A diagram tracing this pattern appears in Figure 3.1.1. The solid arrows indicate the flow of control through the algorithm and the recursive calls. The dotted lines indicate returns. Since there are no steps to be executed in the algorithm after line 7 or 8, the returned result is returned intact to the previous execution. Finally, when control returns to the original execution, the answer is returned to the caller.

Let us examine how the algorithm searches for an item that does not appear in the array. Assume the array a as in the previous example, and assume that it is searching for an x that equals 2.

Line 1: Is $low > high$? 0 is not greater than 7, so execute line 3.
Line 3: $mid = (0 + 7)/2 = 3$.
Line 4: Is $x==a[3]$? 2 does not equal 5, so execute line 6.
Line 6: Is $x < a[3]$? Yes, $2 < 5$, so search for x in $a[low]$ to $a[mid - 1]$.
Line 7: Repeat the algorithm with $low = low = 0$ and $high = mid - 1 = 2$. If 2 appears in the array, it must appear between $a[0]$ and $a[2]$ inclusive.
Line 1: Is $0 > 2$? No, execute line 3.
Line 3: $mid = (0 + 2)/2 = 1$.
Line 4: Is $2==a[1]$? No, execute line 6.
Line 6: Is $2 < a[1]$? Yes, since $2 < 3$. Search for x in $a[low]$ to $a[mid - 1]$.
Line 7: Repeat the algorithm with $low = low = 0$ and $high = mid - 1 = 0$. If x exists in a, it must be the first element.
Line 1: Is $0 > 0$? No, execute line 3.

Line 3: $mid = (0 + 0)/2 = 0$.
Line 4: Is $2 == a[0]$? No, execute line 6.
Line 6: Is $2 < a[0]$? 2 is not less than 1, so perform the else clause at line 8.
Line 9: Repeat the algorithm with $low = mid + 1 = 1$ and $high = high = 0$.
Line 1: Is $low > high$? 2 is greater than 1, so -1 is returned. The item 2 does not exist in the array.

Properties of Recursive Definitions or Algorithms

Let us summarize what is involved in a recursive definition or algorithm. In order to be correct, a recursive algorithm must not generate an infinite sequence of calls on itself. Clearly, any algorithm that does generate such a sequence can never terminate. For at least one argument or group of arguments, a recursive function f must be defined in terms that do not involve f. There must be a "way out" of the sequence of recursive calls. In the examples in this section, the nonrecursive portions of the definitions were:

```
factorial:          0! = 1
multiplication:     a * 1 = a
Fibonacci seq.:     fib(0) = 0;   fib(1) = 1
binary search:      if (low > high)
                        return -1;
                    if (x == a[mid])
                        return mid;
```

No recursive function can be computed without a nonrecursive exit of this kind. Every instance of a recursive definition or invocation of a recursive algorithm must eventually reduce to a manipulation of one or more simple nonrecursive cases.

EXERCISES

3.1.1 Write an iterative algorithm to evaluate $a * b$ by using addition, where a and b are nonnegative integers.

3.1.2 Write a recursive definition of $a + b$, where a and b are nonnegative integers, in terms of the successor method *succ* defined as:

```
succ(int x) {
    return x++;
} // end succ
```

3.1.3 Let a be an array of integers. Present recursive algorithms to compute:

a. the maximum element of the array
b. the minimum element of the array
c. the sum of the elements of the array
d. the product of the elements of the array
e. the average of the elements of the array

3.1.4 Evaluate each of the following, using both the iterative and recursive definitions:

a. 6!
b. 9!
c. 100 * 3
d. 6 * 4
e. fib(10)
f. fib(11)

3.1.5 Assume that an array of ten integers contains the elements

1, 3, 7, 15, 21, 22, 36, 78, 95, 106

Use the recursive binary search to find each of the following items in the array

a. 1
b. 20
c. 36

3.1.6 Write an iterative version of the binary search algorithm. (*Hint*: modify the values of *low* and *high* directly.)

3.1.7 Ackerman's function is defined recursively on the nonnegative integers as follows:

$$\begin{array}{ll} a(m, n) = n + 1 & \text{if } m == 0 \\ a(m, n) = a(m - 1, 1) & \text{if } m \neq 0, n == 0 \\ a(m, n) = a(m - 1, a(m, n - 1)) & \text{if } m \neq 0, n \neq 0 \end{array}$$

a. Using the above definition, show that $a(2, 2)$ equals 7.
b. Prove that $a(m, n)$ is defined for all nonnegative integers m and n.
c. Can you find an iterative method of computing $a(m, n)$?

3.1.8 Count the number of additions necessary to compute $fib(n)$ for $0 \leq n \leq 10$ by the iterative and recursive methods. Does a pattern emerge?

3.1.9 If an array contains n elements, what is the maximum number of recursive calls made by the binary search algorithm? $n/2$

3.2 RECURSION IN JAVA

Factorial in Java

The Java language allows a programmer to write methods that call themselves. Such routines are termed ***recursive***.

The recursive algorithm to compute $n!$ may be directly translated into a Java method as follows:

```
public long fact(int n) {
  int x;
  long y;
```

```
    if (n == 0)
      return 1;
    x = n - 1;
    y = fact(x);
    return n * y;
  } // end fact
```

In the statement $y = fact(x)$; the method *fact* calls itself. This is the essential ingredient of a recursive routine. The programmer assumes that the method being computed has already been written and uses it in its own definition. However, the programmer must ensure that this does not lead to an endless series of calls.

Let us examine the execution of this method when it is called by another program. For example, suppose an applet contains the statement

```
output.setText(Long.toString(fact(4)));
```

When the calling applet calls *fact*, the parameter n is set equal to 4. Since n is not 0, x is set equal to 3. At this point, *fact* is called a second time with an argument of 3. Therefore, the method *fact* is reentered and the local variables (x and y) and parameter (n) of the block are reallocated. Since execution has not yet left the first call of *fact*, the first allocation of these variables remains. Thus two generations of each of these variables are in existence simultaneously. Only the most recent copy of these variables can be referenced from any point within the second execution of *fact*.

In general, each time the method *fact* is entered recursively, a new set of local variables and parameters is allocated, and only this new set may be referenced within that call of *fact*. When a return from *fact* to a point in a previous call takes place, the most recent allocation of these variables is freed and the previous copy is reactivated. This previous copy is the one that was allocated upon the original entry to the previous call and is local to that call.

This description suggests the use of a stack to keep the successive generations of local variables and parameters. The stack is maintained by the Java system and is invisible to the user. Each time a recursive method is entered, a new allocation of its variables is pushed on top of the stack. Any reference to a local variable or parameter is through the current top of the stack. When the method returns, the stack is popped, the top allocation is freed, and the previous allocation becomes the current stack top to be used for referencing local variables. This mechanism is examined more closely in Section 4, but for now, let us see how it is applied in computing the factorial method.

Figure 3.2.1 contains a series of snapshots of the stacks for the variables n, x, and y as execution of the *fact* method proceeds. Initially, the stacks are empty, as illustrated by Figure 3.2.1a. After the first call on *fact* by the calling applet, the situation is as shown in Figure 3.2.1b, with n equal to 4. The variables x and y are allocated but not initialized. Since n does not equal 0, x is set to 3 and *fact*(3) is called (Figure 3.2.1c). The new value of n does not equal 0, so x is set to 2, and *fact*(2) is called (Figure 3.2.1d).

This continues until n equals 0 (Figure 3.2.1f). At this point, the value 1 is returned from the call to *fact*(0). Execution resumes from the point at which *fact*(0) was called, which is the assignment of the returned value to the copy of y declared in *fact*(1). This is illustrated by the status of the stack shown in Figure 3.2.1g, where the variables allocated for *fact*(0) have been freed and y is set to 1.

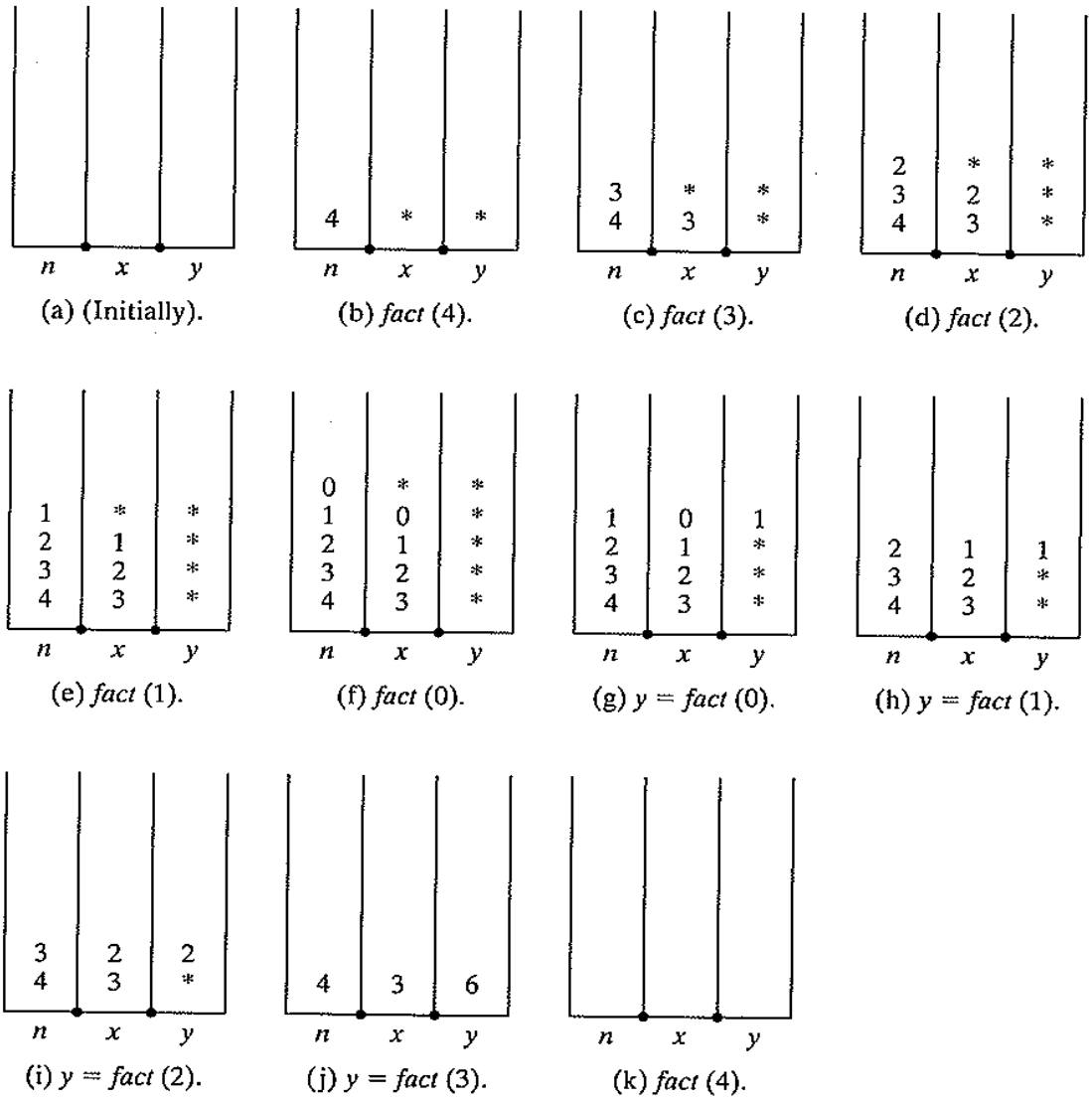

FIGURE 3.2.1 Stack at various times during execution. (An asterisk indicates an uninitialized value.)

The statement *return n * y* is then executed, multiplying the top values of *n* and *y* to obtain 1, and returning this value to *fact*(2) (Figure 3.2.1h). This process is repeated twice more, until finally the value of *y* in *fact*(4) equals 6 (Figure 3.2.1j). The statement *return n * y* is executed one more time. The product 24 is returned to the calling class, where it is displayed on the applet by the statement

```
output.setText(Long.toString(fact(4)));
```

(The *toString* method converts the result, which is of type *long*, into a *string* so that it may be sent to the *setText* method for placement on the *output* field of the applet. Both methods may be imported from the *java.awt.** package. *output* is of type *label.*)

Note that each time a recursive routine returns, it returns to the point immediately following the point from which it was called. Thus, the recursive call to *fact*(3) returns to the assignment of the result to *y* within *fact*(4), but the recursive call to *fact*(4) returns to the method which displays the result on the applet.

Let us transform some of the other recursive definitions and processes from the previous section into recursive Java programs. It is difficult to conceive of a Java programmer writing a function to compute the product of two positive integers in terms of addition, since an asterisk performs the multiplication directly. Nevertheless, such a function can serve as another illustration of recursion in Java. Following closely the definition of multiplication in the previous section, we may write:

```
public long mult(long a, long b) {
   return b == 1 ? a : mult(a, b - 1) + a;
} // end mult
```

Note how similar this method is to the recursive definition in the last section. We leave it as an exercise for you to trace through the execution of this method when it is called with two positive integers. The use of stacks is a great aid in the tracing process.

This example illustrates that a recursive method may invoke itself even within a statement assigning a value to the function. Similarly, we could have written the recursive *fact* function more compactly as:

```
public long fact(long n) {
   return n == 0 ? 1 : n * fact(n - 1);
} // end fact
```

This compact version avoids the explicit use of local variables x (to hold the value of $n - 1$) and y (to hold the value of $fact(x)$). However, temporary locations are set aside anyway for these two values upon each invocation of the function. These temporaries are treated just like any explicit local variable. Thus, in tracing the action of a recursive routine, it may be helpful to declare all temporary variables explicitly. See if it is any easier to trace the following more explicit version of *mult*:

```
public long mult(long a, long b) {
  long c, d, sum;

  if (b == 1)
      return a;
  c = b - 1;
  d = mult(a, c);
  sum = d + a;
  return sum;
} // end mult
```

Another point worth noting is that it is very important to check the validity of input parameters in a recursive routine. For example, let us examine the execution of the *fact* method when it is invoked by a statement such as

```
output.setText(Long.toString(fact(-1)));
```

Of course, the *fact* method is not designed to produce a meaningful result for negative input. However, one of the most important things for programmers to learn is that every method will inevitably be presented at some time with invalid input, and unless provision is made for this, the resultant error may be very difficult to trace.

For example, when -1 is passed as a parameter to *fact* so that n equals -1, x is set to -2 and -2 is passed to a recursive call on *fact*. Another set of n, x, and y is allocated, n is set to -2, and x becomes -3. This process continues until the program either runs out of time or space or the value of x becomes too small. No message indicating the true cause of the error is produced.

If *fact* were originally called with a complicated expression as its argument, and the expression erroneously evaluated to a negative number, a programmer might spend hours searching for the cause of the error. The problem can be remedied by revising the *fact* method to check its input explicitly, as follows:

```
public long fact(long n) throws NegativeFactorialException {
  long x, y;
  if (n < 0)
        throw new NegativeFactorialException();
  if (n == 0)
        return 1;
  x = n - 1;
  y = fact(x);
  return n * y;
} // end fact
```

NegativeFactorialException may be defined as a public class which extends the *java.lang.IllegalArgumentException*. It would be caught by the applet displaying, in turn, an appropriate error message.

Similarly, the method *mult* must guard against a nonpositive value in the second parameter.

The Fibonacci Numbers in Java

We now turn our attention to the Fibonacci sequence. A Java program to compute the nth Fibonacci number can be modeled closely on the recursive definition:

```
public long fib(long n) throws NegativeFibonacciException {
  long x, y;
  if (n < 0)
        throw new NegativeFibonacciException();
  if (n <= 1)
      return n;
  x = fib(n - 1);
  y = fib(n - 2);
  return x + y;
} // end fib
```

Let us trace through the action of this function in computing the sixth Fibonacci number. You may compare the action of the routine with the manual computation we performed in the last section to compute *fib*(6). The stacking process is illustrated in Figure 3.2.2. When the program is first called, the variables n, x, and y are allocated, and n is set to 6 (Fig. 3.2.2a). Since $n > 1$, $n - 1$ is evaluated and *fib* is called recursively.

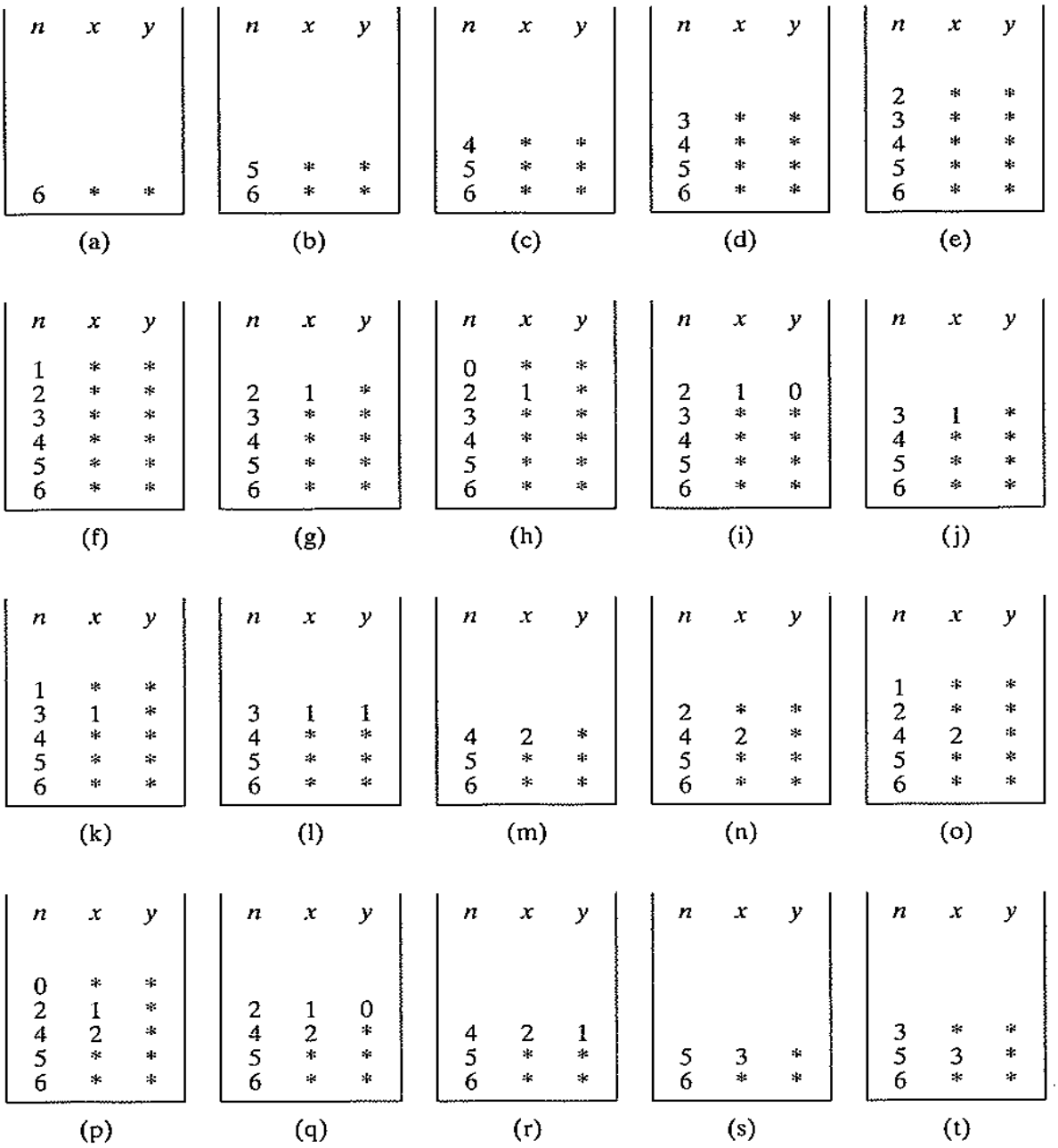

FIGURE 3.2.2 The recursion stack of the Fibonacci function.

A new set of n, x, and y is allocated, and n is set to 5 (Fig. 3.2.2b). This process continues (Figs. 3.2.2c–f) with each successive value of n being 1 less than its predecessor, until *fib* is called with n equal to 1. The sixth call to *fib* returns 1 to its caller so the fifth allocation of x is set to 1 (Fig. 3.2.2g).

The next sequential statement $y = fib(n - 2)$ is then executed. The value of n that is used is the most recently allocated one, which is 2. Thus we again call on *fib* with an argument of 0 (Fig. 3.2.2h). The value of 0 is immediately returned, so that y in $fib(2)$ is set to 0 (Fig. 3.2.2i). Since each recursive call results in a return to the point of call, the call of $fib(1)$ returns to the assignment to x, while the call of $fib(0)$ returns to the assignment to y. The next statement to be executed in $fib(2)$ is the statement that returns

$x + y = 1 + 0 = 1$ to the statement that calls *fib*(2) in the generation of the function calculating *fib*(3). This is the assignment to x, so that x in *fib*(3) is given the value $fib(2) = 1$ (Fig. 3.2.2j). The process of calling and pushing and returning and popping continues until finally the routine returns to the main program for the last time, with the value 8. Figure 3.2.2 shows the stack up to the point where *fib*(5) calls on *fib*(3) so that its value can be assigned to y. The reader is urged to complete the picture by drawing the stack states for the remainder of the program execution.

This program illustrates that a recursive routine may call itself a number of times with different arguments. In fact, as long as a recursive routine uses only local variables, the programmer can use the routine just the same as any other and can assume that it will perform its function and produce the desired value. There is no need to worry about the underlying stacking mechanism.

The Binary Search in Java

Let us now present a Java program for a binary search. A method to do this accepts a sorted array a and an element x as input and returns the index i in a such that $a[i]$ equals x, or -1 if no such i exists. Thus the method *binsrch* might be invoked in a statement such as

```
i = binsrch(a, x)
```

However, in looking at the binary search algorithm in Section 3.1 as a model for a recursive Java method, we note that two other parameters are passed in the recursive calls. Lines 7 and 9 of the algorithm call for a binary search on only part of the array. Thus, in order for the method to be recursive, the bounds between which the array is to be searched must also be specified. The method is written as follows:

```
public int binsrch(int a[], int x, int low, int high)  {
  int mid;

  if (low > high)
      return -1;
  mid = (low+high) / 2;
  return x == a[mid] ? mid : x < a[mid] ?
                                    binsrch(a, x, low, mid-1)  :
                                    binsrch(a, x, mid+1, high) ;
} // end binsrch
```

When *binsrch* is invoked by another method to search for x in a sorted array declared by

```
int a[arraySize];
```

of which the first n elements are occupied, it is called by the statement

```
i = binsrch(a, x, 0, n - 1);
```

You are urged to trace the execution of this routine and follow the stacking and unstacking using the example from the preceding section, where a is an array of eight elements ($n = 8$) containing 1, 3, 4, 5, 17, 18, 31, 33 in that order. The value being searched for is 17 (x equals 17). Note that the array a is stacked for each recursive call. The values of *low* and *high* are, respectively, the lower and upper bounds of the array a.

In the course of tracing through the *binsrch* routine, you may have noticed that the values of the two parameters *a* and *x* do not change throughout its execution. Each time that *binsrch* is called, the same array is searched for the same element; it is only the upper and lower bounds of the search that change. It therefore seems wasteful to stack and unstack these two parameters each time the method is called recursively.

One solution is to rewrite the *binsrch* method so that the parameters *a* and *x* are not defined within its header. Since *a* and *x* are not defined within the *binsrch* method, they are said to be global variables, which enables *binsrch* to access *a* and *x* without allocating additional space for them.

The method may then be invoked by a statement such as

```
i = binsrch(0, n-1);
```

In this case, all references to *a* and *x* are to the global allocations of *a* and *x* declared at the beginning of the class. All multiple allocations and freeings of space for these parameters are eliminated.

We may rewrite the *binsrch* function as follows:

```
public int binsrch(int low, int high)  {
  int mid;

  if (low > high)
       return -1;
  mid = (low + high) / 2;
  return x == a[mid] ? mid : x < a[mid] ?
                                    binsrch(low, mid-1)   :
                                    binsrch(mid+1, high);
} // end binsrch
```

Using this scheme, the variables *a* and *x* are not passed with each recursive call to *binsrch*. *a* and *x* do not change their values and are not stacked. The programmer wishing to make use of *binsrch* in a program only needs to pass the parameters *low* and *high*. The method could be invoked with a statement such as

```
i = binsrch(low, high);
```

Recursive Chains

A recursive function need not call itself directly. Rather, it may call itself indirectly, as in the following example:

```
a(formal parameters) {          b(formal parameters) {
        .                               .
        .                               .
        .                               .
  b(arguments);                   a(arguments);
        .                               .
} // end a                      } // end b
```

In this example, method *a* calls *b*, which may in turn call *a*, which may again call *b*. Thus both *a* and *b* are recursive, since they indirectly call on themselves. However, the fact

that they are recursive is not evident from examining the body of either of the methods individually. The method *a* seems to be calling a separate method *b*, and it is impossible to determine, by examining *a* alone, that it may call itself indirectly.

More than two methods may participate in a ***recursive chain***. Thus a method *a* may call *b*, which calls *c* ..., which calls *z*, which calls *a*. Each method in the chain may potentially call itself and is therefore recursive. Of course, the programmer must ensure that such a program does not generate an infinite sequence of recursive calls.

Recursive Definition of Algebraic Expressions

As an example of a recursive chain, consider the following recursive group of definitions:

1. An ***expression*** is a *term* followed by a *plus sign* followed by a *term*, or a *term* alone.
2. A ***term*** is a *factor* followed by an *asterisk* followed by a *factor*, or a *factor* alone.
3. A ***factor*** is either a *letter* or an *expression* enclosed in *parentheses*.

Before looking at some examples, note that none of the above three items is defined directly in terms of itself. However, each is defined in terms of itself indirectly. An expression is defined in terms of a term, a term in terms of a factor, and a factor in terms of an expression. Similarly, a factor is defined in terms of an expression, which is defined in terms of a term, which is defined in terms of a factor. Thus the entire set of definitions forms a recursive chain.

Let us now give some examples. The simplest form of a factor is a letter. Thus *A, B, C, Q, Z, M* are all factors. They are also terms, because a term may be a factor alone. They are also expressions, because an expression may be a term alone. Since *A* is an expression, (*A*) is a factor and therefore a term as well as an expression. *A* + *B* is an example of an expression that is neither a term nor a factor. (*A* + *B*), however, is all three. *A* * *B* is a term and therefore an expression, but it is not a factor. *A* * *B* + *C* is an expression that is neither a term nor a factor. *A* * (*B* + *C*) is a term and an expression but not a factor.

Each of the above examples is a valid expression. This can be shown by applying the definition of an expression to each of them. Consider, however, the string *A* + * *B*. It is neither an expression, term, nor factor. It would be instructive for you to attempt to apply the definitions of expression, term, and factor to see that none of them describe the string *A* + * *B*. Similarly, (*A* + *B* *)*C* and *A* + *B* + *C* are not valid expressions according to the preceding definitions.

Let us write a program that reads and prints a character string and then prints "valid" if it is a valid expression and "invalid" if it is not. We begin by defining a class *expression* that contains methods for recognizing expressions, terms, and factors. An object of this class consists of a variable *str* which contains the input character string, and an integer *pos* whose value is the position in *str* from which we last obtained a character. A helper method, *getsymb*, is used to return the next character to be examined. If *pos* < *str.length*(), then *getsymb* returns the character located at *pos* and increments *pos* by 1. If *pos* >= *str.length*(), then *getsymb* returns a blank. Helper methods like *getsymb* are usually declared to be *private* because they are designed to be used only by other methods of that class.

The class constructor *expression* receives a string from the calling applet, copies the string in *str*, and initializes *pos* to zero.

The method that recognizes an expression is called *expr*. It returns *true* if a valid expression begins at position *pos* of *str* and *false* otherwise. It also resets *pos* to the position following the longest expression it can find.

The methods *factor* and *term* are much like *expr* except that they are responsible for recognizing factors and terms respectively. They also reposition *pos* to the position following the longest factor or term they can find within the string *str*. In order to determine whether the specified character is a letter, *factor* uses a method *isLetter(char)* which is defined in the class *Character*.

The code for these methods adheres closely to the definitions given earlier. Each of the methods attempts to satisfy one of the criteria for the entity being recognized. If one of these criteria is satisfied, then *true* is returned. If none of these criteria are satisfied, then *false* is returned. The entire expression is then declared to be valid or not valid by the method *valid*.

```
public class Expression  {
  String str;
  int pos;

  public Expression(String input) {
     pos = 0;
     str = new String(input);
  } // end Expression

  public boolean expr() {
     // look for a term
     if (!term())
          return false;
     // We have found a term; look at the next symbol
     if (getsymb() != '+') {
          // We have found the longest expression (a single term).
          // Reposition pos so it refers to the last position of
          the expression.
          pos--;
          return true;
     } // end if
     // At this point, we have found a term and a plus sign.
     // We must look for another term.
     return term();
  } // end expr

  public boolean term() {
     if (!factor())
          return false;
     if (getsymb() != '*') {
          pos--;
          return true;
     } // end if
     return factor();
  } // end term
```

```
   public boolean factor() {
      char c;

      if ((c = getsymb()) != '(')
         return Character.isLetter(c);
      return expr() && getsymb() == ')';
   } // end factor

   private char getsymb() {
      char c;

      if (pos < str.length())
         c = str.charAt(pos);
      else
         c = ' ';
      pos++;
      return c;
   } // end getsymb

   public boolean valid() {
      if (expr() && pos >= str.length())
         return true;
      else
         return false;
   } // end valid

} // end Expression class
```

All three primary methods of the class are recursive because each may call itself indirectly. For example, if you trace through the actions of the program for the input string "$(a * b + c * d) + (e * (f) + g)$", you will find that each of the methods *expr*, *term*, and *factor* calls on itself.

Having described the class *Expression*, we can write the applet that places the results on the browser.

```
// An applet to determine whether an algebraic expression is
valid or not
import java.awt.*;
import java.applet.Applet;

public class AlgebraicExpression extends Applet {
   Label prompt1, prompt2;          // labels the input and output
                                    // values
   TextField input1, output1;       // input and output the value
   Expression str;
   // setup applet gui
   public void init() {

      prompt1 = new Label("Enter an algebraic expression");
      prompt2 = new Label("The expression is");
      input1 = new TextField(30);
```

```
        output1 = new TextField(30);
        output1.setEditable(false);

        add(prompt1);                    // place i/o fields on applet
        add(input1);
        add(prompt2);
        add(output1);
    } // end init

    // process the user's input
    public boolean action(Event e, Object o){

        str = new Expression(o.toString());     // get expression
        showStatus("Calculating...");           // display result
        if (str.valid)
              output1.setText("valid");
      else
              output1.setText("invalid");
      showStatus("Done.");
      return true;
    } // end action

} // end class AlgebraicExpression
```

EXERCISES

3.2.1 Determine what the following recursive Java method computes. Write an iterative method to accomplish the same purpose.

```
public int method(int n) {
  if (n == 0)
      return 0;
  return n + method(n-1);
} // end method
```

3.2.2 The Java expression *m* % *n* yields the remainder of *m* upon division by *n*. Define the ***greatest common divisor*** (***gcd***) of two integers *x* and *y* by:

```
gcd(x, y) = y                    if (y <= x && x % y == 0)
gcd(x, y) = gcd(y, x)            if (x < y)
gcd(x, y) = gcd(y, x % y)        otherwise
```

Write a recursive Java method to compute *gcd*(*x*, *y*). Find an iterative method for computing this function.

3.2.3 Let *comm*(*n*, *k*) represent the number of different committees of *k* people that can be formed, given *n* people from whom to choose. For example, *comm*(4, 3) = 4, since given four people A, B, C, and D there are four possible three-person committees: ABC, ABD, ACD, and BCD. Prove the identity:

```
comm(n, k) = comm(n - 1, k) + comm(n - 1, k - 1)
```

Write and test a recursive Java program to compute *comm*(*n*, *k*) for *n*, *k* >= 1.

3.2.4 Define a ***generalized Fibonacci sequence*** of $f0$ and $f1$ as the sequence $gfib(f0, f1, 0)$, $gfib(f0, f1, 1)$, $gfib(f0, f1, 2)$, ..., where

```
gfib(f0, f1, 0) = f0
gfib(f0, f1, 1) = f1
gfib(f0, f1, n) = gfib(f0, f1, n - 1) + gfib(f0, f1, n - 2) if n
> 1
```

Write a recursive Java method to compute $gfib(f0, f1, n)$. Find an iterative method for computing this function.

3.2.5 Write a recursive Java method to compute the number of sequences of n binary digits that do not contain two 1's in a row. (*Hint*: compute how many such sequences exist that start with 0, and how many exist that start with a 1.)

3.2.6 An ***order n matrix*** is an $n \times n$ array of numbers. For example,

(3)

is a 1×1 matrix,

$$\begin{matrix} 1 & 3 \\ -2 & 8 \end{matrix}$$

is a 2×2 matrix and

$$\begin{matrix} 1 & 3 & 4 & 6 \\ 2 & -5 & 0 & 8 \\ 3 & 7 & 6 & 4 \\ 2 & 0 & 9 & -1 \end{matrix}$$

is a 4×4 matrix. Define the ***minor*** of an element x in a matrix as the submatrix formed by deleting the row and column containing x. In the above example of a 4×4 matrix, the minor of the element 7 is the 3×3 matrix

$$\begin{matrix} 1 & 4 & 6 \\ 2 & 0 & 8 \\ 2 & 9 & -1 \end{matrix}$$

Clearly, the order of a minor of any element is 1 less than the order of the original matrix. Denote the minor of an element $a[i, j]$ by $minor(a[i, j])$.

Define the ***determinant*** of a matrix a (written $det(a)$) recursively as follows:

1. If a is a 1×1 matrix (x), then $det(a) = x$.
2. If a is of an order greater than 1, compute the determinant of a as follows:

   a. Choose any row or column. For each element $a[i, j]$ in this row or column, form the product:

   ```
   power(-1,i + j) * a[i, j] * det(minor(a[i, j]))
   ```

   where i and j are the row and column positions of the element chosen, $a[i, j]$ is the element chosen, $det(minor(a[I, j]))$ is the determinant of the minor of $a[i, j]$, and $power(m, n)$ is the value of m raised to the nth power.

b. $det(a)$ = sum of all these products.
(More concisely, if n is the order of a,

$det(a) = \Sigma_i\ power(-1,\ i + j) * a[i,\ j] * det(minor(a[i,\ j]))$, for any j

or

$det(a) = \Sigma_j\ power(-1,\ i + j) * a[i,\ j] * det(minor(a[i,\ j]))$, for any i).

Write a Java program that reads a, prints a in matrix form, and prints the value of $det(a)$, where *det* is a method that computes the determinant of a matrix.

3.2.7 Write a recursive Java program to sort an array a as follows:

1. Let k be the index of the middle element of the array.
2. Sort the elements up to and including $a[k]$.
3. Sort the elements past $a[k]$.
4. Merge the two subarrays into a single sorted array.

This method is called a ***merge sort***.

3.2.8 Show how to transform the following iterative method into a recursive method. $f(i)$ is a method returning a Boolean value based on the value of i, and $g(i)$ is a method that returns a value with the same attributes as i.

```
public void iter(int n) {
   int i;

   i = n;
   while(f(i)) {
     // any group of Java statements that
     // does not change the value of i
     i = g(i);
   } // end while
} // end iter
```

3.3 WRITING RECURSIVE PROGRAMS

In the last section we saw how to transform a recursive definition or algorithm into a Java program. It is much more difficult to develop a recursive Java solution to a problem specification whose algorithm is not supplied. It is not only the program but also the original definitions and algorithms that must be developed. There is generally no reason to look for a recursive solution when faced with the task of writing a program to solve a problem. Most problems can be solved in a straightforward manner using nonrecursive methods. However, some problems can be solved logically and most elegantly by recursion. In this section we shall identify the kinds of problems that can be solved recursively, develop a technique for finding recursive solutions, and present some examples.

Let us reexamine the factorial function. Since the iterative solution is so direct and simple, factorial is a prime example of a problem that should not be solved recursively. However, let us examine the elements that make the recursive solution work. First of all, we can recognize a large number of distinct cases to solve. That is, we want

to write a program to compute 0!, 1!, 2!, etc. We can also identify a "trivial" case for which a nonrecursive solution is directly obtainable. This is the case of 0!, which is defined as 1. The next step is to find a method of solving a "complex" case in terms of a "simpler" case. This allows the reduction of a complex problem to a simpler problem. The transformation of the complex case to the simpler case should eventually result in the trivial case. This would mean that the complex case is ultimately defined in terms of the trivial case.

Let us examine what this means when applied to the factorial function. 4! is a more "complex" case than 3!. The transformation that is applied to the number 4 to obtain the number 3 is simply the subtraction of 1. Repeatedly subtracting 1 from 4 eventually results in 0, which is a "trivial" case. Thus, if we are able to define 4! in terms of 3!, and in general $n!$ in terms of $(n - 1)!$, we will be able to compute 4! by first working our way down to 0! and then working our way back up to 4! using the definition of $n!$ in terms of $(n - 1)!$. In the case of the factorial function, we have such a definition, since

```
n! = n * (n - 1)!
```

Thus $4! = 4 * 3! = 4 * 3 * 2! = 4 * 3 * 2 * 1! = 4 * 3 * 2 * 1 * 0! = 4 * 3 * 2 * 1 * 1 = 24$.

These are the essential ingredients of a recursive routine—being able to define a "complex" case in terms of a "simpler" case and having a directly solvable (nonrecursive) "trivial" case. Once this has been done, one can develop a solution using the assumption that the simpler case has already been solved. The Java version of the factorial function assumes that $(n - 1)!$ is defined and uses that quantity in computing $n!$.

Let us see how these ideas apply to other examples from the previous sections. In defining $a * b$, the case of $b = 1$ is trivial, since in that case $a * b$ is defined as a. In general, $a * b$ may be defined in terms of $a * (b - 1)$ by the definition $a * b = a * (b - 1) + a$. Again the complex case is transformed into a simpler case by subtracting 1, eventually leading to the trivial case of $b = 1$. Here the recursion is based solely on the second parameter, b.

In the case of the Fibonacci function, two trivial cases were defined: $fib(0) = 0$ and $fib(1) = 1$. A complex case, $fib(n)$, is then reduced to two simpler cases: $fib(n - 1)$ and $fib(n - 2)$. It is because of the definition of $fib(n)$ as $fib(n - 1) + fib(n - 2)$ that two trivial cases directly defined are necessary. $fib(1)$ cannot be defined as $fib(0) + fib(-1)$, because the Fibonacci function is not defined for negative numbers.

The binary search method is an interesting case of recursion. The recursion is based on the number of elements in the array that must be searched. Each time the routine is called recursively, the number of elements to be searched is halved (approximately). The trivial case is the one in which there are either no elements to be searched or the element being searched for is at the middle of the array. If $low > high$, then the first of these two conditions holds and -1 is returned. If $x = a[mid]$, the second condition holds, and *mid* is returned as the answer. In the more complex case of $high - low + 1$ elements to be searched, the search is reduced to taking place in one of two subregions,

1. The lower half of the array from *low* to $mid - 1$
2. The upper half of the array from $mid + 1$ to *high*

Thus a complex case (a large area to be searched) is reduced to a simpler case (an area to be searched that is approximately half the size of the original area). This eventually reduces to a comparison with a single element ($a[mid]$) or a search within an array of no elements.

The Towers of Hanoi Problem

Thus far we have been looking at recursive definitions and examining how they fit the pattern we have established. Let us now look at a problem that is not specified in terms of recursion and see how we can use recursive techniques to produce a logical and elegant solution. The problem is the Towers of Hanoi problem. The initial setup is shown in Figure 3.3.1. There are three pegs, A, B, and C. Five disks of differing diameters are placed on peg A so that a larger disk is always below a smaller disk. The object is to move the five disks to peg C using peg B as auxiliary. Only the top disk on any peg may be moved to any other peg, and a larger disk may never rest on a smaller one. See if you can produce a solution. Indeed, it is not even apparent that a solution is possible.

Let us see if we can develop a solution. Instead of focusing our attention on a solution for five disks, let us consider the general case of n disks. Suppose we had a solution for $n - 1$ disks and could state a solution for n disks in terms of the solution for $n - 1$ disks. Then the problem would be solved. This is true because the solution is simple in the trivial case of one disk (continually subtracting 1 from n will eventually produce 1): merely move the single disk from peg A to peg C. Therefore, we will have developed a recursive solution if we can state a solution for n disks in terms of $n - 1$. See if you can find such a relationship. In particular, for the case of five disks, suppose we knew how to move the top four disks from peg A to another peg according to the rules. How could we then complete the job of moving all five? Recall that there are three pegs available.

Suppose we could move four disks from peg A to peg C. Then we could move them just as easily to B, using C as auxiliary. This would result in the situation depicted in Figure 3.3.2a. We could then move the largest disk from A to C (Figure 3.3.2b) and finally again apply the solution for four disks to move the four disks from B to C, using the now empty peg A as an auxiliary (Figure 3.3.2c). Thus we may state a recursive solution to the Towers of Hanoi problem as follows:

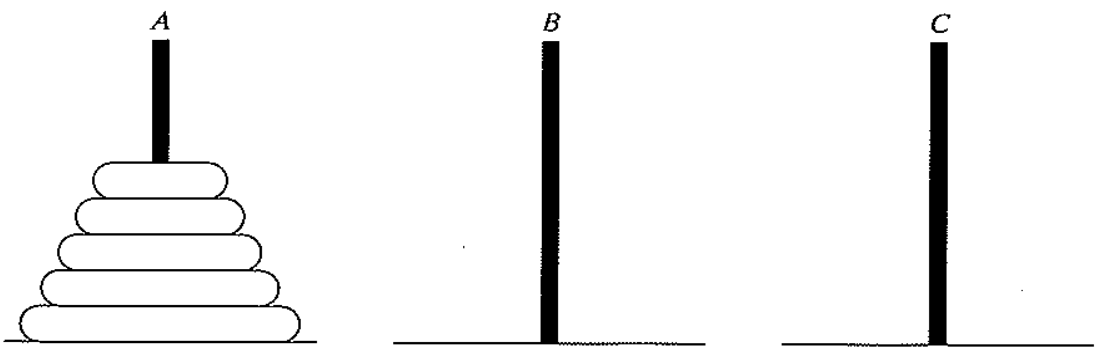

FIGURE 3.3.1 Initial setup of the Towers of Hanoi.

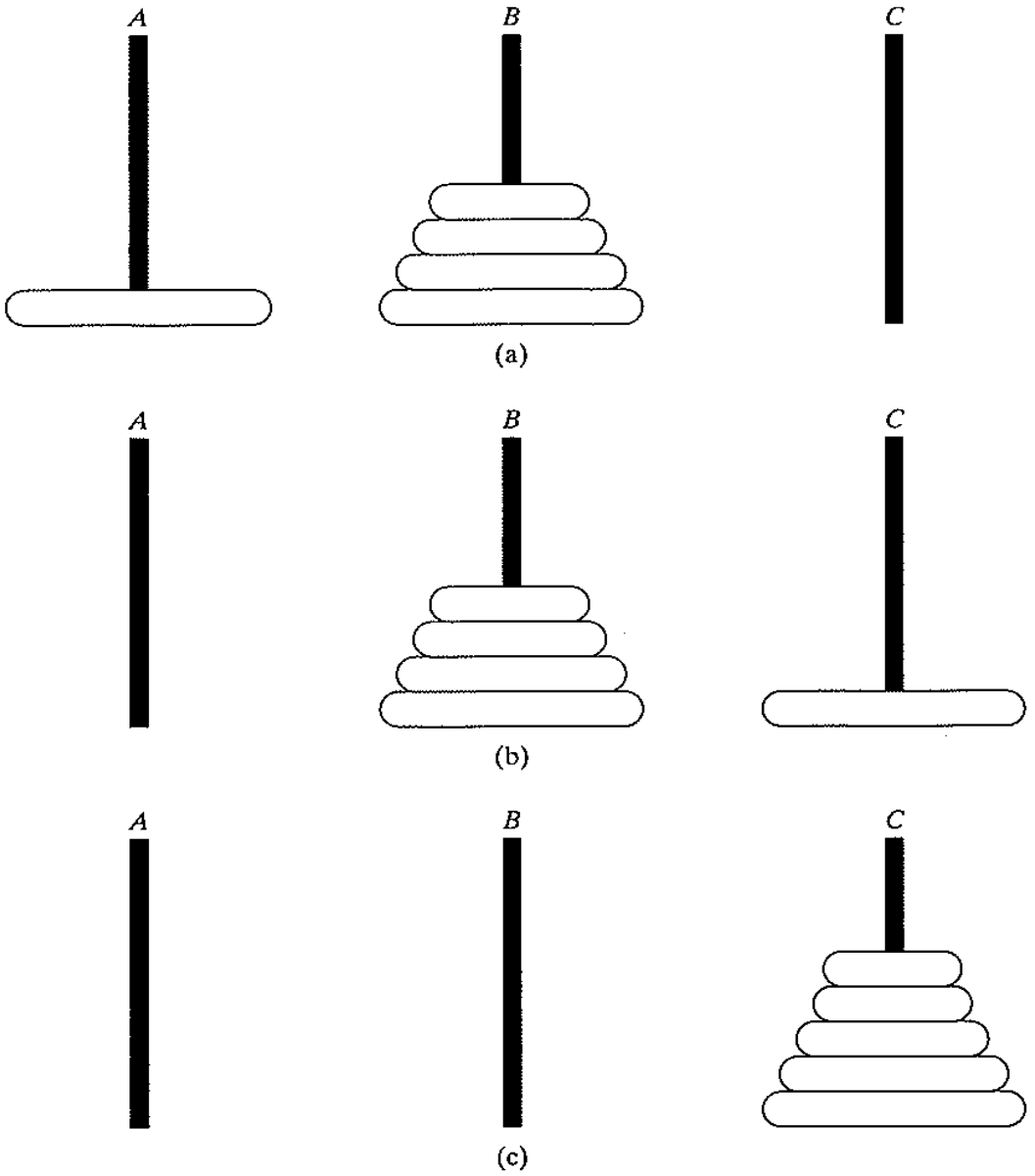

FIGURE 3.3.2 Recursive solution to the Towers of Hanoi.

To move n disks from A to C, using B as auxiliary:

1. If $n == 1$, then move the single disk from A to C and stop.
2. Move the top $n - 1$ disks from A to B, using C as auxiliary.
3. Move the remaining disk from A to C.
4. Move the $n - 1$ disks from B to C, using A as auxiliary.

We are sure that this algorithm will produce a correct solution for any value of n. If $n == 1$, step 1 will result in the correct solution. If $n == 2$, we know that we already have a solution for $n - 1 == 1$, so that steps 2 and 4 will perform correctly. Similarly, when $n == 3$, we have already produced a solution for $n - 1 == 2$, so that steps 2 and 4 can be performed. In this fashion, we can show that the solution works for

$n == 1, 2, 3, 4, 5, \ldots$ up to any value for which we desire a solution. Note that we developed the solution by identifying a trivial case ($n == 1$) and a solution for a general complex case (n) in terms of a simpler case ($n - 1$).

How can this solution be converted into a Java applet? We are no longer dealing with a mathematical function such as factorial, but with concrete actions such as "move a disk." How are we to represent such actions in the computer? The problem is not completely specified. What are the inputs to the program? What are its outputs to be? Whenever you are told to write a program, you must receive specific instructions as to exactly what the program is expected to do. A problem statement such as "Solve the Towers of Hanoi problem" is quite insufficient. Specifying a problem of this kind usually entails that the inputs and outputs as well as the program must be so designed as to reasonably correspond to the problem description. The design of inputs and outputs is an important phase of a solution and should be given as much attention as the rest of the program. There are two reasons for this. First, the user (who must ultimately evaluate and pass judgment on your work) will not see the elegant method you incorporated in your program but will struggle mightily to decipher the output or to adapt the input data to your input conventions. Failures to agree early on input and output details have been the cause of much grief to programmers and users. Second, a slight change in the input or output format may make the program much simpler to design. Thus the programmer can make the job much easier by designing an input or output format compatible with the algorithm. Of course these two considerations, convenience to the user and convenience to the programmer, often conflict sharply, and some happy medium must be found. However, the user must be a full participant with the programmer in the decisions on input and output formats.

Let us, then, proceed to design the inputs and outputs for this program. The only input needed is the value of n, the number of disks. At least that may be the programmer's view. The user may want the names of the disks (e.g., "red", "blue", "green") and perhaps the names of the pegs (e.g., "left", "right", "middle") as well. The programmer can probably convince the user that naming the disks $1, 2, 3, \ldots, n$ and the pegs *A, B, C* is just as convenient. If the user is adamant, the programmer can write a small function to convert the user's names to his own, and vice versa.

A reasonable form for the output would be a list of statements, such as:

```
move disk nnn from peg yyy to peg zzz
```

where *nnn* is the number of the disk to be moved, and *yyy* and *zzz* are the names of the pegs involved. The action to be taken for a solution would be to perform each of the output statements in the order that they appear in the output. (We use the *append* method of the *TextArea* object in order to place the multiline output on the applet. Objects of type *TextArea* contain scrollbars that allow the user to move forward and backward through the output produced by our applet.)

The programmer then decides to write a method *towers* to print the above output (but is purposely vague about the parameters at this point). The *action* method of the applet would then invoke the *towers* method by

```
n = Integer.parseInt(input1.getText());    // get the number of disks
towers(parameters);
```

Let us assume that the user will be satisfied to name the disks 1, 2, 3, . . . , n and the pegs A, B, and C. What should the parameters to *towers* be? Clearly, they should include n, the number of disks to be moved. This includes information about how many disks there are and what their names are. The programmer then notices that in the recursive algorithm, it will be necessary to move $n - 1$ disks using a recursive call to *towers*. Thus, on the recursive call, the first parameter to *towers* will be $n - 1$. But this implies that the top $n - 1$ disks are numbered 1, 2, 3, . . . , $n - 1$, and that the smallest disk is numbered 1. This is a good example of programming convenience determining problem representation. There is no a priori reason for labeling the smallest disk 1; logically the largest disk could have been labeled 1 and the smallest disk n. However, since it leads to a simpler and more direct program, we choose to label the disks so that the smallest disk has the smallest number.

What are the other parameters to *towers*? At first glance, it might appear that no additional parameters are necessary since the pegs are named A, B, and C by default. However, a closer look at the recursive solution leads us to the realization that on the recursive calls disks will not be moved from A to C using B as auxiliary but from A to B using C (step 2) or from B to C using A (step 4). We therefore include three more parameters in *towers*. The first, *fromPeg*, represents the peg from which we are removing disks; the second, *toPeg*, represents the peg to which we will take the disks; and the third, *auxPeg*, represents the auxiliary peg. This situation is one which is quite typical of recursive methods; additional parameters are necessary to handle the recursive call situation. We already saw one example of this in the binary search applet, where the parameters *low* and *high* were necessary.

The complete applet to solve the Towers of Hanoi problem, closely following the recursive solution, may be written as follows:

```
// An Applet to solve the Towers of Hanoi problem
import java.awt.*;
import java.applet.Applet;
public class TowersOfHanoi extends Applet {
  Label prompt1, prompt2;       // labels the input and output values
  TextField input1;             // input the value
  TextArea output1;             // output the solution
  int n;

  // setup applet gui
  public void init() {

        prompt1 = new Label("Enter the number of disks");
        input1 = new TextField(5);
        prompt2 = new Label("The solution is");
        output1 = new TextArea(10,35);
        output1.setEditable(false);

        add(prompt1);                    // place i/o fields on applet
        add(input1);
        add(prompt2);
        add(output1);
  } // end init
```

```
    // process the user's input
    public boolean action(Event e, Object o) {
      n = Integer.parseInt(input1.getText());    // get the number of disks
      showStatus("Calculating...");              // display result

      towers(n, 'A', 'C', 'B');

      output1.append("\n\n");
      showStatus("Done.");
      return true;
    } // end action

    void towers(int n, char fromPeg, char toPeg, char auxPeg) {
          // If only one disk, make the move and return
          if (n == 1) {
                output1.append("\nmove disk 1 from peg " + fromPeg +
                                                    " to peg " + toPeg);
                return;
          } // end if
          // Move top n - 1 disks from A to B, using C as auxiliary
          towers(n - 1, fromPeg, auxPeg, toPeg);
          output1.append("\nmove disk " + n + " from peg " + fromPeg +
                                                    " to peg " + toPeg);
          // Move n - 1 disk from B to C using A as auxiliary
          towers(n - 1, auxPeg, toPeg, fromPeg);
    } // end towers

} // end class TowersOfHanoi
```

Trace the actions of the above program when it reads the value 4 for *n*. Be careful to keep track of the changing values of the parameters *fromPeg*, *auxPeg*, and *toPeg*. Verify that it produces the following output:

```
move disk 1 from peg A to peg B
move disk 2 from peg A to peg C
move disk 1 from peg B to peg C
move disk 3 from peg A to peg B
move disk 1 from peg C to peg A
move disk 2 from peg C to peg B
move disk 1 from peg A to peg B
move disk 4 from peg A to peg C
move disk 1 from peg B to peg C
move disk 2 from peg B to peg A
move disk 1 from peg C to peg A
move disk 3 from peg B to peg C
move disk 1 from peg A to peg B
move disk 2 from peg A to peg C
move disk 1 from peg B to peg C
```

Verify that the above solution actually works and does not violate any of the rules.

Translation from Prefix to Postfix Using Recursion

Let us examine another problem for which recursion offers the most direct and elegant solution. This is the problem of converting a prefix expression to a postfix. Prefix and postfix notations were discussed in the last chapter. Briefly, prefix and postfix notations are methods of writing mathematical expressions without parentheses. In prefix notation each operator immediately precedes its operands. In postfix notation each operator immediately follows its operands. To refresh your memory, here are a few conventional (infix) mathematical expressions with their prefix and postfix equivalents:

infix	*prefix*	*postfix*
$A + B$	$+AB$	$AB+$
$A + B * C$	$+A*BC$	$ABC*+$
$A * (B + C)$	$*A+BC$	$ABC+*$
$A * B + C$	$+*ABC$	$AB*C+$
$A + B * C + D - E * F$	$-++A*BCD*EF$	$ABC*+D+EF*-$
$(A + B) * (C + D - E) * F$	$^{\circ\circ}+AB-+CDEF$	$AB+CD+E-*F*$

The most convenient way to define postfix and prefix is by using recursion. Assuming no constants and using only single letters as variables, a prefix expression is a single letter or an operator followed by two prefix expressions. A postfix expression may be similarly defined as a single letter or as an operator preceded by two postfix expressions. The above definitions assume that all operations are binary—that is, each requires two operands. Examples of such operations are addition, subtraction, multiplication, division, and exponentiation. It is easy to extend the above definitions of prefix and postfix to include unary operations, such as negation or factorial, but in the interest of simplicity we will not do so here. Verify that each of the above prefix and postfix expressions is valid by showing that they satisfy the definitions, and make sure that you can identify the two operands of each operator.

We will put these recursive definitions to use in a moment, but first let us return to our problem. Given a prefix expression, how can we convert it into a postfix expression? We can immediately identify a trivial case: a prefix expression that consists of only a single variable is its own postfix equivalent. That is, an expression such as A is valid as both a prefix and a postfix expression.

Now consider a longer prefix string. If we knew how to convert any shorter prefix string to postfix, could we convert a longer prefix string? The answer is yes, with one proviso. Every prefix string longer than a single variable contains an operator, a first operand, and a second operand (remember, we are assuming binary operators only). Assume we are able to identify the first and second operands, which are necessarily shorter than the original string. We can then convert the long prefix string to postfix by first converting the first operand to postfix, then converting the second operand to postfix and appending it to the end of the first converted operand, and finally appending the initial operator to the end of the resultant string. Thus we have developed a recursive algorithm for converting a prefix string to postfix with the single provision that

we must specify a method for identifying the operands in a prefix expression. We can summarize our algorithm as follows:

1. If the prefix string is a single variable, it is its own postfix equivalent.
2. Let *op* be the first operator of the prefix string.
3. Find the first operand *opnd*1 of the string. Convert it to postfix and call it *post*1.
4. Find the second operand *opnd*2 of the string. Convert it to postfix and call it *post*2.
5. Concatenate *post*1, *post*2, and *op*.

One operation that will be required by this program is concatenation. For example, if two strings represented by *a* and *b* represent the strings "abcde" and "xyy" respectively, the statement

```
a = a + b;
```

places into *a* the string "abcdexyz" (i.e., the string consisting of all the elements of *a* followed by all the elements of *b*). Alternatively, we could have used the *string.concat*(*String str*) method, which returns a string that represents the concatenation of *string*'s characters followed by the *str*'s characters.

We also make use of the *length* method. *str.length*() returns the length of the string *str*.

This application will also require the *substring* method. *string.substring*(*int beginIndex*, *int endIndex*) returns a new string that is a substring of this string. The substring begins at the specified *beginIndex* and extends to the character at index *endIndex* - 1. Thus the $s2 = s1.substring(i, j)$ statement sets the string *s*2 to the substring of *s*1 starting at position *i* containing $j - i$ characters. For example, after executing *s* = "abcd".*substr*(1, 3), *s* equals "bc". The methods *concat*, *length*, and *substring* are defined as part of the *String* class.

Before transforming the conversion algorithm into a Java application, let us examine its inputs and outputs. We wish to write a method *convert* that accepts a character string. This string represents a prefix expression in which all variables are single letters and the allowable operators are '+', '−', '*', and '/'. The method produces a string that is the postfix equivalent of the prefix parameter.

Assume the existence of a function *find* which accepts a string and returns an integer that is the length of the longest prefix expression contained within the input string that starts at the beginning of that string. For example, *find*("*A + CD*") returns 1, because "*A*" is the longest prefix string starting at the beginning of "*A + CD*". *find*("+ * *ABCD + GH*") returns 5 because "+ * *ABC*" is the longest prefix string starting at the beginning of "+ * *ABCD + GH*". If no such prefix string exists within the input string starting at the beginning of the input string, *find* returns 0. (For example, *find*("* + *AB*") returns 0.) This function is used to identify the first and second operands of a prefix operator. *convert* also calls the *Character* method *isLetter*, which determines whether its parameter is a letter. Thus the statement

```
if (Character.isLetter(prefix.charAt(0)))
```

determines whether the character at the head of the *prefix* string is a letter.

We now turn our attention to the method *find*, which accepts a character string and a starting position and returns the length of the longest prefix string which is contained in that input string starting at that position. The word "longest" in this definition is superfluous, since there is at most only one substring starting at a given position of a given string that is a valid prefix expression.

We first show that there is at most one valid prefix expression starting at the beginning of a string. To see this, note that it is trivially true in a string of length 1. Assume it is true for a short string. Then a long string that contains a prefix expression as an initial substring must begin with either a variable, in which case the variable is the desired substring, or with an operator. Deleting the initial operator, the remaining string is shorter than the original string and can therefore have at most a single initial prefix expression. This expression is the first operand of the initial operator. Similarly, the remaining substring (after the first operand is deleted) can only have a single initial substring that is a prefix expression. This expression must be the second operand. Therefore, we have uniquely identified the operator and operands of the prefix expression starting at the first character of an arbitrary string, if such an expression exists. Since there is at most one valid prefix string starting at the beginning of any string, there is at most one such string starting at any position of an arbitrary string. This is obvious when we consider the substring of the given string starting at the given position. Note that this proof has given us a recursive method for finding a prefix expression in a string.

We now present a class *Expression* that implements the conversion from prefix to postfix:

```
import java.io.IOException;  // To delay ending of program

public class Expression {

  public final static int MAXCOLS = 80;

  public static void main(String[] args) throws IOException{
        String inString;
        String outString = new String();

        System.out.println("Enter a prefix string: ");
        inString = readString();
        System.out.println("The original prefix expression is " +
        inString);
        outString = convert(inString);
        System.out.println("In postfix it is " + outString);
  } // end main

  public static String readString() throws IOException {
        char[] charArray = new char[MAXCOLS];
        int position = 0;
        char c;

        while ((c = (char) System.in.read()) != '\n')
              charArray[position++] = c;
```

```
        return  String.copyValueOf(charArray,0,position-1);
} // end readString

public static String convert(String prefix) {
        String opnd1, opnd2;
        String post1, post2;
        String temp;
        char op;
        int m, n, length;

        if ((length = prefix.length()) == 1) {
                if (Character.isLetter(prefix.charAt(0))) {
                        // The prefix string is a single letter
                return new String(prefix);
                } // end if
                System.out.println("Illegal prefix string");
                System.exit(1);
        } // end if

        // The prefix string is longer than a single character.
        // Extract the operator and the two operand lengths.
        op = prefix.charAt(0);

        temp = new String(prefix.substring(1, length));
        m = find(temp);
        temp = new String(prefix.substring(m+1, length));
        n = find(temp);

        if ((op != '+' && op != '-' && op != '*' && op != '/')
                        || (m == 0) || (n == 0) || (m+n+1 != length)) {
                System.out.println("Illegal prefix string");
                System.exit(1);
        } // end if
        opnd1 = new String(prefix.substring(1, m+1));
        opnd2 = new String(prefix.substring(m+1, length));
        post1 = new String();
        post2 = new String();
        post1 = convert(opnd1);
        post2 = convert(opnd2);
        post1 = post1 + post2;
        post1 = post1 + op;
        return (post1.toString()).substring(0, length);
} // end convert

public static int find(String str) {
        String temp;
        int m, n, length;

        if ((length = str.length()) == 0)
                return 0;
        if (Character.isLetter(str.charAt(0)))
                // First character is a letter.
```

```
            // That letter is the initial substring.
            return 1;
        if (str.length() < 2)
            return 0;
        // otherwise find the first operand
        temp = str.substring(1, length);
        m = find(temp);
        if (m == 0 || str.length() == m)
            // no valid prefix operand or no second operand
            return 0;
        temp = str.substring(m+1, length);
        n = find(temp);
        if (n == 0)
            return 0;
        return m+n+1;
    } // end find

} // end class Expression
```

Note that several checks have been incorporated into *convert* to ensure that the parameter is a valid prefix string. One of the most difficult classes of errors to detect are those resulting from invalid inputs and the programmer's neglect to check for validity.

Make sure that you understand how these methods work by tracing their actions on both valid and invalid prefix expressions. More important, make sure that you understand how they were developed and how logical analysis led to a natural recursive solution that was directly translatable into a Java application.

EXERCISES

3.3.1 Suppose that another provision were added to the Towers of Hanoi problem: that one disk may not rest on another disk that is more than one size larger (e.g., disk 1 may only rest on disk 2 or on the ground, disk 2 may only rest on disk 3 or on the ground). Why does the solution in the text fail to work? What is faulty about the logic that led to it under the new rules?

3.3.2 Prove that the number of moves performed by *towers* in moving n disks equals $2^n - 1$. Can you find a method of solving the Towers of Hanoi problem in fewer moves? Either find such a method for some n or prove that none exists.

3.3.3 Define a postfix and prefix expression to include the possibility of unary operators. Write a program to convert a prefix expression possibly containing the unary negation operator (represented by the symbol '@') to postfix.

3.3.4 Rewrite the method *find* in the text so that it is nonrecursive and computes the length of a prefix string by counting the number of operators and single-letter operands.

3.3.5 Write a recursive method that accepts a prefix expression consisting of binary operators and single-digit integer operands and returns the value of the expression.

3.3.6 Consider the method *convert*, for converting a prefix expression to postfix, defined for the *Expression* class given below.

```
public class Expression {
  private String prefix;

  public Expression(String str) {
        prefix = str;
  }

  public Expression() {
        prefix = new String();
  }

  public String convert() {
        String first = prefix.substring(0, 1);
        String postfix = new String();

        prefix = prefix.substring(1);
        char ch = first.charAt(0);
        if (ch == '+' || ch == '*' || ch == '-' || ch == '/') {
                String t1 = convert();
                String t2 = convert();
                t1 = t1 + t2;
                t1 = t1 + first;
                postfix = t1;
                return postfix;
        } // end if
        postfix = first;
        return postfix;
  } // end convert
} // end Expression
```

The method *convert* would be invoked by:

```
Expression preString = new Expression(inString);
postString = preString.convert();
```

Explain how the method works. Is it better or worse than the method in the text? What happens if the method is called with an invalid prefix string as input? Can you incorporate a check for such an invalid string in *convert*? Can you design such a check for the calling application after *convert* has returned?

3.3.7 Develop a recursive method (and program it) to compute the number of different ways in which an integer k can be written as a sum each of whose operands is less than n.

3.3.8 Consider an array a containing positive and negative integers. Define *contigsum*(i, j) as the sum of the contiguous elements $a[i]$ through $a[j]$ for all array indexes $i <= j$. Develop a recursive method that determines i and j such that *contigsum*(i, j) is maximized. The recursion should consider the two halves of the array a.

3.3.9 Write a recursive Java application to find the k^{th} smallest element of an array a of numbers by choosing any element $a[i]$ of a and partitioning a into those elements smaller than, equal to, and greater than $a[i]$.

3.3.10 In the Eight Queens Problem, one has to place eight queens on a chessboard positioned so that no queen is attacking any other queen. The following is a recursive program to solve the problem. *board* is an 8×8 array that represents a chessboard.

board[*i*][*j*]==*true* if there is a queen at position [*i*][*j*], and *false* otherwise. *good*() is a method that returns *true* if no two queens on the chessboard are attacking each other and *false* otherwise. At the end of the program, the method *drawboard*() displays a solution to the problem

```
// An Applet to solve the Eight Queens Problem
import java.awt.*;
import java.applet.Applet;

public class EightQueens extends Applet {
  private static boolean[][] board = new boolean [8][8];
  private static int xPosition, yPosition;

  public void init() {
        for (int i = 0; i < 8; i++)
              for (int j = 0; j < 8; j++)
                    board[i][j] = false;

        xPosition = 20;
        yPosition = 20;
  } // end init

  // Paint the applet
  public void paint(Graphics g) {
         if(trial(0))
                drawboard(g);
  } // end paint

  public static boolean trial(int n) {
        for (int i = 0; i < 8; i++) {
              board[n][i] = true;
              if (n == 7 && good())
                    return true;
              if (n < 7 && good() && trial(n+1))
                    return true;
              board[n][i] = false;
        } // end for
        return false;
  } // end trial

      ...

} // end class EightQueens
```

The recursive method *trial* returns *true* if it is possible, given the *board* at the time it is called, to add queens in rows *n* through 7 to achieve a solution. *trial* returns *false* if there is no solution that has queens at the positions in *board* that already contain *true*. If *true* is returned, the function also adds queens in rows *n* through 7 to produce a solution.

Write the methods *good* and *drawboard* used above, and verify that the program produces a solution. The idea behind the solution is as follows: *board* represents the global situation during an attempt to find a solution. The next step toward finding a solution is chosen arbitrarily (place a queen in the next untried position in row *n*) and recursively test whether it is possible to produce a solution which includes that step. If

it is, then return. If it is not, then backtrack from the attempted next step ($board[n][i] = false$) and try another possibility. This method is called ***backtracking***.

3.3.11 A 10×10 array *maze* of 0s and 1s represents a maze in which a traveler must find a path from *maze*[0][0] to *maze*[9][9]. The traveler may move from a square into any adjacent square in the same row or column, but may not skip over any squares or move diagonally. In addition, the traveler may not move into any square that contains a 1. *maze*[0][0] and *maze*[9][9] contain 0s. Write a method that accepts such a *maze* and prints either a message that no path through the maze exists or a list of positions representing a path from [0][0] to [9][9].

3.4 SIMULATING RECURSION

In this section we examine more closely some of the mechanisms used to implement recursion so that we can simulate them using nonrecursive techniques. This activity is important for several reasons. First of all, older programming languages (e.g., FORTRAN, COBOL, and many machine languages) do not allow recursive programs. Problems like the Towers of Hanoi and prefix to postfix conversion, whose solutions can be derived and stated quite simply using recursive techniques, can be programmed in these languages by simulating the recursive solution using more elementary operations. If we know that the recursive solution is correct (and it is often fairly easy to prove such a solution correct), and we have established techniques for converting a recursive solution to a nonrecursive one, then we can create a correct solution in a nonrecursive language. Programmers are often able to state recursive solutions to problems. The ability to generate nonrecursive solutions from recursive algorithms is indispensable when using a compiler that does not support recursion.

Another reason for examining the implementation of recursion is that it will allow us to understand the implications of recursion and some of its hidden pitfalls. While these pitfalls do not occur in mathematical definitions that employ recursion, they seem to be an inevitable accompaniment of implementations in a real language on a real machine.

Finally, even in a language like Java that supports recursion, a recursive solution to a problem is often more expensive than a nonrecursive solution, in terms of both time and space. Frequently, this expense is a small price to pay for the logical simplicity and self-documentation of the recursive solution. However, in a production program (e.g., a compiler) that may be run thousands of times, the recurrent expense is a heavy burden on the system's limited resources. Thus a program may be designed to incorporate a recursive solution in order to reduce the expense of design and certification, and then carefully converted to a nonrecursive version to be put into actual day-to-day use. As we shall see, in performing such a conversion it is often possible to identify parts of the implementation of recursion that are superfluous in a particular application and thereby to significantly reduce the amount of work the program must perform.

Before examining the actions of a recursive routine, let us take a step back and examine the action of a nonrecursive routine. We will then be able to see what mechanisms

must be added to support recursion. Before proceeding we adopt the following convention. Suppose we have the statement:

```
rout(x);
```

where *rout* is defined as a method by the header:

```
void rout(a)
```

we refer to x as an ***argument*** (of the calling function) and to a as a ***parameter*** (of the called method).

What happens when a method is called? The action of invoking a method may be divided into three parts:

1. Passing arguments
2. Allocating and initializing local variables
3. Transferring control to the method.

Let us examine each of these three steps in turn.

1. *Passing Arguments.* For a parameter in Java, local storage is allocated for the parameter, and the value of the argument is copied into the parameter. Any changes to the parameter within the method are made to the local copy. The effect of this scheme is that the original input argument cannot be altered. In this method, storage for the argument is allocated in the data area of the method.
2. *Allocating and Initializing Local Variables.* After arguments have been passed, the local variables of the method are allocated. These include all the local variables declared directly in the method and any temporaries that must be created during the course of execution. For example, in evaluating the expression

   ```
   x + y + z
   ```

   a storage location must be set aside to hold the value of $x + y$ so that z can be added to it. Another storage location must be set aside to hold the value of the entire expression after it has been evaluated. Such locations are called ***temporaries*** because they are needed only temporarily during the course of execution. Similarly, in a statement such as

   ```
   x = fact(n);
   ```

   a temporary must be set aside to hold the value of *fact*(n) before it can be assigned to x.
3. *Transferring Control to the Method.* At this point control may still not be passed to the method because provision has not yet been made for saving the ***return address***. If a method is given control, it must eventually restore control to the calling routine by means of a branch. However, it cannot execute the branch unless it knows the location to which it must return. Since this location is within the calling routine and not within the method, the only way the method can know this address is to have it passed as an argument. This is exactly what happens.

Aside from the explicit arguments specified by the programmer, there are also a set of implicit arguments that contain information necessary for the method to execute and return correctly. Chief among these implicit arguments is the return address. The method stores this address in its own data area. When it is ready to return control to the calling program, the method retrieves the return address and branches to that location.

Once the arguments and the return address have been passed, control may be transferred to the method, since everything required has been done to ensure that the method can operate on the appropriate data and then safely return to the calling routine.

Return from a Method

When a method returns, three actions are performed. First, the return address is retrieved and stored in a safe location. Second, the method's data area is freed. This data area contains all the local variables (including local copies of arguments), the temporaries, and the return address. Finally, a branch is taken to the return address that was previously saved. This restores control to the calling routine at the point immediately following the instruction that initiated the call. In addition, if the method returns a value, it is placed in a secure location from which the calling program may retrieve it. Usually this location is a hardware register set aside for this purpose.

Suppose an application has called a method *b*, which has called *c*, which has, in turn, called *d*. This is illustrated in Figure 3.4.1a, where we indicate that control currently

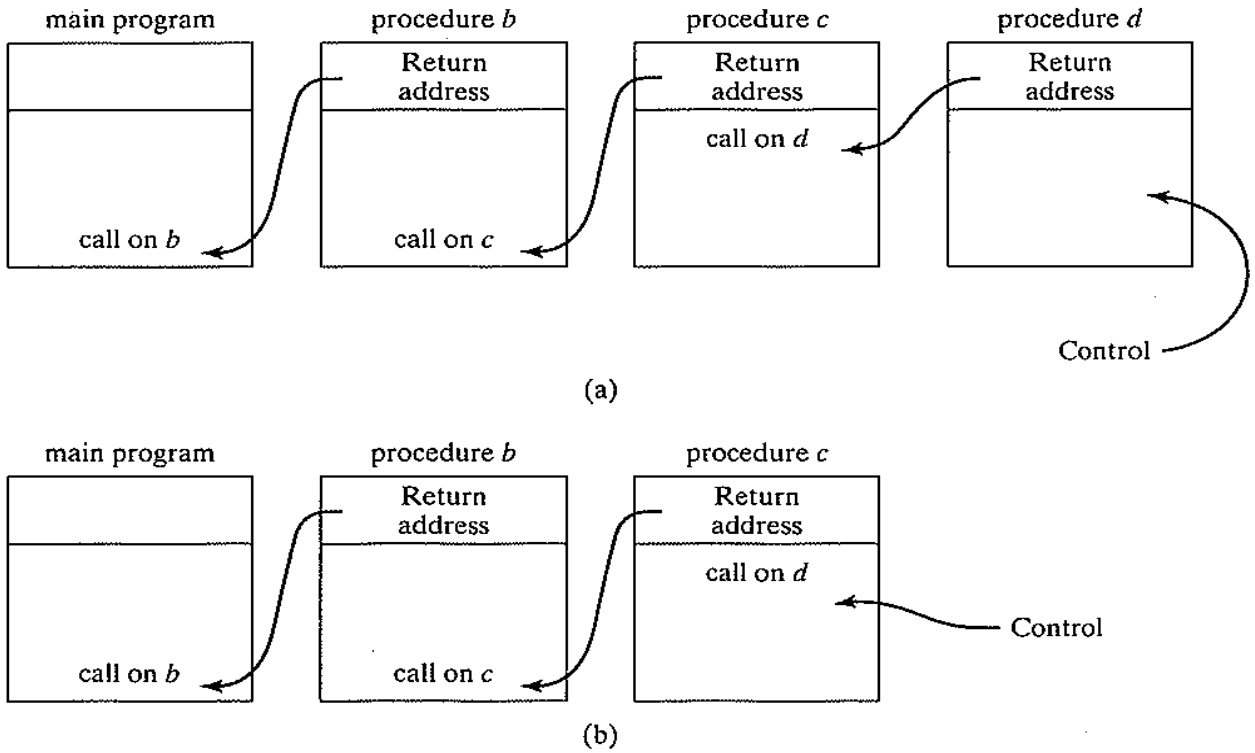

FIGURE 3.4.1 Series of procedures calling one another.

resides somewhere within *d*. A location in each function is set aside for the return address. Thus the return address area of *d* contains the address of the instruction in *c* immediately following the call to *d*. Figure 3.4.1b shows the situation immediately following *d*'s return to *c*. The return address within *d* has been retrieved and control transferred to that address.

You may have noticed that the string of return addresses forms a stack; that is, the return address most recently added to the chain is the first to be removed. At any point, we can only access the return address from within the method that is currently executing which represents the top of the stack. When the stack is popped (i.e., when the method returns), a new top is revealed within the calling routine. Invoking a method has the effect of pushing an element onto the stack, and returning pops the stack.

Implementing Recursive Methods

What must be added to this description in the case of a recursive method? The answer is, surprisingly little. Each time a recursive method calls itself, an entirely new data area must be allocated for that particular call. As before, this data area contains all the parameters, local variables, and temporaries, and a return address. The point to remember is that in recursion a data area is associated not with a method alone, but with a particular call to that method. Each call causes a new data area to be allocated, and each reference to an item in the method's data area is to the data area of the most recent call. Similarly, with each return, the current data area is freed, and the data area allocated immediately prior to it becomes current. This behavior, of course, suggests the use of a stack.

In Section 3.1.2, where we described the action of the recursive factorial method, we used a set of stacks to represent the successive allocations of each of the local variables and parameters. These stacks may be thought of as separate stacks, one for each local variable. Alternatively, and closer to reality, we may think of all of these stacks as a single large stack. Each element of this large stack is an entire data area containing subparts representing the individual local variables or parameters.

Each time the recursive routine is called, a new data area is allocated. The parameters in this data area are initialized to refer to the values of their corresponding arguments. The return address in the data area is initialized to the address following the call instruction. Any reference to local variables or parameters is via the current data area.

When the recursive routine returns, the returned value (if any) and the return address are saved, the data area is freed, and a branch to the return address is executed. The calling method retrieves the returned value (if any), resumes execution, and refers to its own data area, which is now on top of the stack.

Let us now examine how we can simulate the actions of a recursive function. We will need a stack of data areas defined by

```
Stack dataStack = new Stack();
```

dataArea is a stack containing objects representing a class containing the various items that exist in a data area. This class, which we call *DataArea*, must be defined as a class that consists of the fields required for the particular method being simulated. (We use

the Java convention of starting class names with a capital letter and objects of a class with a lowercase letter.)

Simulation of Factorial

Let us look at a specific example, the factorial method. We present the code for that method, including temporary variables explicitly and omitting the test for negative input:

```
public long fact(int n) {
  int x;
  long y;

  if (n == 0)
      return 1;
  x = n - 1;
  y = fact(x);
  return n * y;
} // end fact
```

We wish to develop an algorithm for the simulation of this method using stacks. Once an algorithm has been developed, it will be a simple matter to turn it into a method that calculates the factorial of a number without the use of explicit recursion.

How are we to define the data area for this function? It must contain the parameter n and the local variables x and y. As we shall see, no temporaries are needed. The data area must also contain a return address. In this case, there are two possible points to which we might want to return: the assignment of $fact(x)$ to y, and the main applet that called *fact*.

For the purpose of our algorithm, let us assume a new statement of the form ***goto*** *label*; that unconditionally transfers control to a statement whose label is *label*. (Note that a statement label is different from a label appearing on an applet. It identifies a particular statement in a program.) Although in general the uncontrolled use of a ***goto*** statement is frowned upon and is not in keeping with the tenets of structured programming, the concept will be useful in the initial development of algorithms for the simulation of recursion. (The Java language reserves the ***goto*** keyword, but does not implement it.)

Suppose we have two labels. Let the label *label*2 be the label of a section of code:

```
label2: y = result;
```

in the simulating method. Let the label *label*1 be the label of a statement

```
label1: return result;
```

This reflects a convention that the variable *result* contains the value to be returned by an invocation of the *fact* method. *label*2 represents a return from a recursive call, and *label*1 represents a return from a nonrecursive call. The return address will be stored as an integer i (equal to either 1 or 2).

To effect a return from a recursive call, our algorithm executes the statement:

```
switch (i) {
  case 1:        goto label1;
  case 2:        goto label2;
}
```

Thus if $i==1$, a return is executed to the applet that called *fact*, and if $i==2$, a return is simulated to the assignment of the returned value to the variable y in the previous execution of *fact*.

We define a class *DataArea* that contains the parameter, n, the local variables x and y, and the return address, i.

```
public class DataArea {
  int param, x, retAddr;
  long y;
  public DataArea(int n, int x, long y, int i) {
        this.param = n;
        this.x = x;
        this.y = y;
        this.retAddr = i;
  }
} // end DataArea class
```

Using the Java container class, *Stack*, the stack of *DataArea* objects for this example can be defined as follows:

```
import java.util.*;
  ...
Stack dataStack = new Stack();
```

The field in the data area that contains the simulated parameter is called *param* rather than n, to avoid confusion with the parameter n passed to the simulating function. Each time a new *DataArea* object is created, the class constructor intitalizes it with the appropriate parameters.

We also declare a current data area to hold the values of the variables in the simulated "current" call on the recursive function. The declaration is:

```
DataArea currArea = new DataArea(n, x, y, i);
```

In addition, we declare a single variable *result* by:

```
long result;
```

This variable is used to communicate the returned value of *fact* from one recursive call of *fact* to its caller, and from *fact* to the outside calling method.

In our algorithm, a return from *fact* is simulated by the code:

```
result = value to be returned;
i = currArea.retaddr;
currArea = (DataArea) dataStack.pop();
```

```
switch(i) {
  case 1:        goto label1;
  case 2:        goto label2;
}
```

A recursive call on *fact* is simulated by pushing the current data area on the stack, reinitializing the members of the *currArea* class, *currArea.param* and *currArea.retAddr*, respectively, to the parameter and return address of this call, and then transferring control to the start of the simulated routine. Recall that *currArea.x* holds the value of $n - 1$, which is to be the new parameter. Recall also that on a recursive call we wish to eventually return to *label2*. The code to accomplish this is:

```
dataArea.push(currArea);
DataArea currArea = new DataArea(x, 0, 0, 2);
goto start;              // start is the label of start of the simulated
                         // routine
```

When the simulation first begins, the current area must be initialized so that *currArea.param* equals *n* and *currArea.retAddr* equals 1 (indicating a return to the calling routine). A dummy data area must be pushed onto the stack so that an underflow does not occur when the *pop* method is executed in returning to the calling applet. This dummy data area must also be initialized so as not to cause an error in the *push* method. Thus the simulated algorithm of the recursive *fact* method is as follows:

```
import java.util.*;

public long simFact(long n) {
            Stack dataStack = new Stack();
            int i;
            long result;
            // initialize a dummy data area
            DataArea currArea = new DataArea(0, 0, 0, 0);
            dataStack.push(currArea);

            // set the parameter and the return address of the
            // current data area to their proper values
            DataArea currArea = new DataArea(n, 0, 0, 1);
  start:    // this is the beginning of the simulated factorial method
            if (currArea.param == 0) {
                    // simulation of return 1;
                    result = 1;
                    i = currArea.retAddr;
                    currArea = (DataArea) dataStack.pop();
                    switch(i) {
                         case 1:     goto label1;
                         case 2:     goto label2;
                    }
          } // end if
          DataArea currArea = new DataArea
                        (currArea.param, currArea.param - 1,0, i);
          dataStack.push(currArea);
```

```
            currArea.param = currArea.x;
            currArea.retAddr = 2;
            goto start;
label2:     // This is the point to which we return from the recursive call.
            // Set currArea.y to the returned value.
            CurrArea.y = result;
            // simulation of return(n * y);
            result = currArea.param * currArea.y;
            i = currArea.retAddr;
            currArea = (DataArea) dataStack.pop();
            switch(i) {
                  case 1:       goto label1;
                  case 2:       goto label2;
            }
label1:     // At this point we return to the calling applet
            return result;
} // end simFact
```

Trace through the execution of this program for $n = 5$ and be sure that you understand what the program does and how it does it.

Note that no space was reserved in the data area for temporaries, since they need not be saved for later use. The temporary location that holds the value of $n * y$ in the original recursive routine is simulated by the temporary for *currArea.param* * *currArea.y* in the simulating routine. This is generally not the case. For example, if a recursive method *funct* contained a statement such as:

```
x = a * funct(b) + c * funct(d);
```

the temporary for $a * funct(b)$ must be saved during the recursive call on $funct(d)$. However, stacking the temporary is not required in the example of the factorial function.

Improving the Simulated Routine

The foregoing discussion leads naturally to the question of whether all the local variables really need to be stacked. A variable must be saved on the stack only if its value at the point of initiation of a recursive call must be reused after return from that call. Let us examine whether the variables n, x, and y meet this requirement. Clearly n does have to be stacked. In the statement:

```
y = n * fact(x);
```

the old value of n must be used in the multiplication after return from the recursive call on *fact*. However, this is not the case for x and y. In fact, the value of y is not even defined at the point of the recursive call, so clearly it need not be stacked. Similarly, although x is defined at the point of call, it is never used again after returning, so why bother saving it?

This point is illustrated even more sharply by the realization that the routine would work just as well if x and y were not declared within the recursive method *fact*, but were declared as instance variables of the class containing the *fact* method (global variables). Thus the automatic stacking and unstacking action performed by recursion for the local variables x and y is unnecessary.

Another interesting question to consider is whether the return address is really needed on the stack. Since there is only one textual recursive call to *fact*, there is only one return address within *fact*. The other return address is to the main applet that originally called *fact*. But suppose a dummy data area had not been stacked upon initialization of the simulation. Then a data area is placed on the stack only in simulating a recursive call. When the stack is popped in returning from a recursive call, that area is removed from the stack. However, when an attempt is made to pop the stack in simulating a return to the main procedure, an underflow will occur. We can test for this underflow by catching the *EmptyStackException*, and when it is caught we can return directly to the outside calling routine rather than through a local label. This means that one of the return addresses can be eliminated. Since this leaves only a single possible return address, it need not be placed on the stack. Thus the data area has been reduced to contain the parameter alone, and the current data area is reduced to a single variable declared by

```
int currParam;
```

The algorithm is now quite compact and comprehensible.

```
import java.util.*;

public long simFact(int n) {
            Stack paramStack = new Stack();
            int currParam, x;
            long result = 1, y;
            boolean underFlow = false;
            currParam = n;
  start:    // This is the beginning of the simulated factorial method.
            if (currParam == 0) {
                    // simulation of return 1;
                    result = 1;
                    try {
                            currParam = ((Integer)
                            paramStack.pop()).intValue();
                            underFlow = false;
                    }
                    catch (EmptyStackException ese) {
                                underFlow = true;
                    }
                    if (underFlow)
                           goto label1;
                    else
                           goto label2;
                    }
            } // end if
            // currParam != 0
            x = currParam - 1;
            // Simulation of recursive call to fact.
            paramStack.push(new Integer(currParam));
            currParam = x;
            goto start;
```

```
    label2: // This is the point to which we return from the recursive call.
            // Set y to the returned value.
            y = result;
            // simulation of return n * y;
            result = currParam * y;
            try {
                    currParam = ((Integer)
                    paramStack.pop()).intValue();
                    underFlow = false;
            }
            catch (EmptyStackException ese) {
                                underFlow = true;
            }
            if (underFlow)
                    goto label1;
            else
                    goto label2;
            }
    label1: // At this point we return to the main applet.
            return result;
} // end simFact
```

Eliminating *gotos*

The Java language does not support the *goto* statement. Furthermore, if you were to look at the program without having seen its derivation, you probably would not be able to identify it as computing the factorial function. The algorithmic statements

```
goto start;
```

and

```
goto label2;
```

are especially irritating because they interrupt the flow of thought at a time when one might otherwise come to an understanding of what is happening. Let us see if we can transform this program into a still more readable version.

The use of a ***flowchart*** is often helpful in untangling the complexities of an algorithm. Figure 3.4.2 diagrammatically represents our simplified algorithm. We repeat the algorithm indicating the statements that correspond to the flowchart.

```
import java.util.*;

public long simFact(int n) {
```

```
        Stack paramStack= new Stack();
        int currParam, x;
        long result = 1, y;
        boolean underFlow = false;

        currParam = n;
```

Initializations

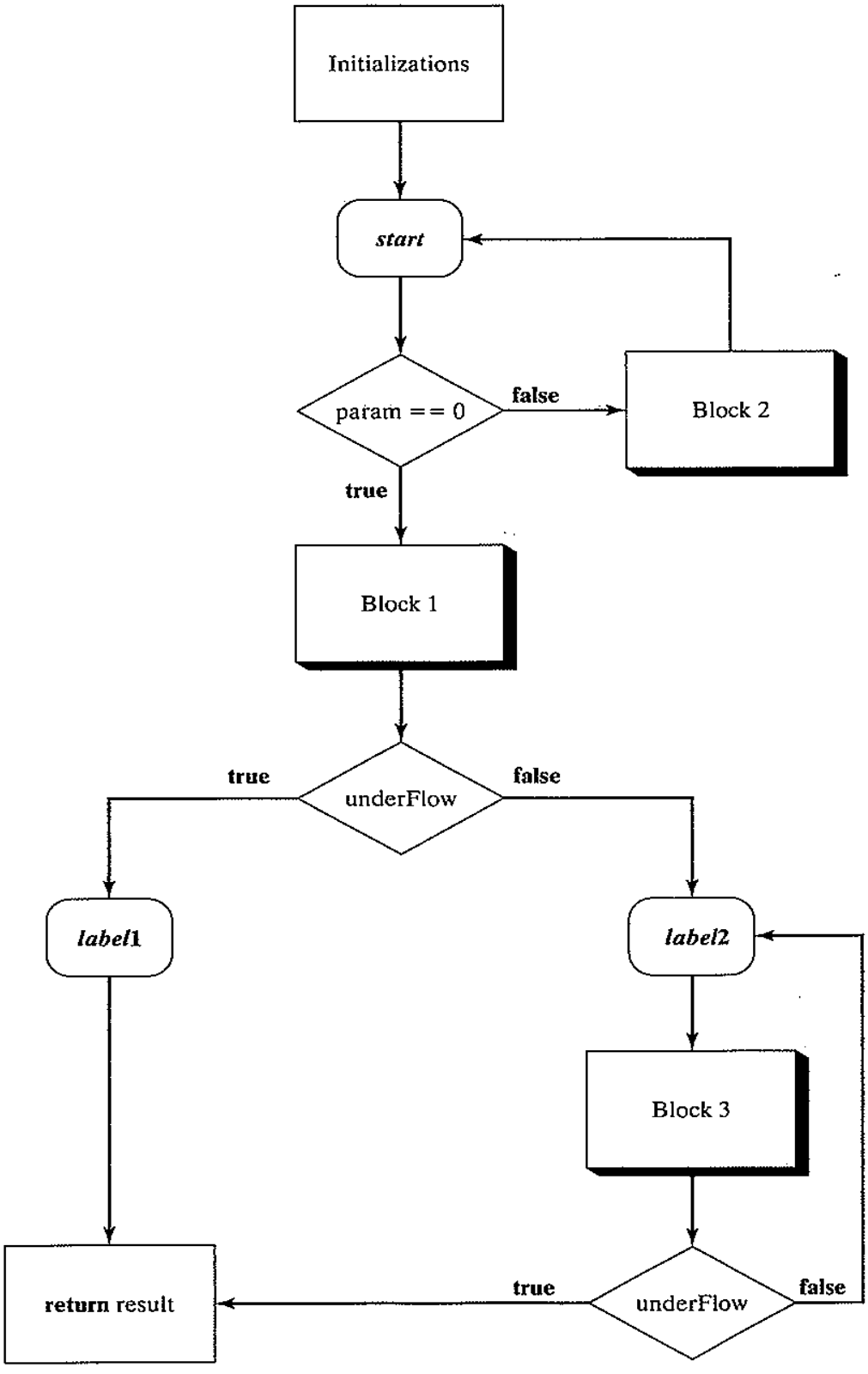

FIGURE 3.4.2 Flowchart for the factorial simulation.

```
start: // This is the beginning of the simulated factorial
       // method.
       if (currParam == 0) {

                  // simulation of return 1;
                  result = 1;
                  try {
                       currParam = ((Integer)
                                    paramStack.pop()).intValue();
                       underFlow = false;
                  }
                  catch (EmptyStackException ese) {
                            underFlow = true;
                  }
```

Block 1

```
                  if (underFlow)
                        goto label1;
                  else
                        goto label2;
       } // end if

                  // currParam != 0
                  x = currParam - 1;
                  // simulation of recursive call to fact
                  paramStack.push(new Integer(currParam));
                  currParam = x;
```

Block 2

```
                  goto start;
label2:           // This is the point to which we return from
                  // the recursive call. Set y to the returned value.

                  y = result;
                  // simulation of return n * y;
                  result = currParam * y;
                  try {
                        currParam = ((Integer)
                                     paramStack.pop()).intValue();
                        underFlow = false;
                  }
                  catch (EmptyStackException ese) {
                                    underFlow = true;
                  }
```

Block 3

```
                  if (underFlow)
                        goto label1;
                  else
                        goto label2;
```

```
    label1:   // At this point we return to the main applet.
              return result;
} // end simFact
```

Close examination of the flowchart reveals that the loop containing block 2 can be replaced by the following ***while*** loop:

```
while (currParam != 0) {
```

```
    // currParam != 0
    x = currParam - 1;
    // simulation of recursive call to fact
    paramStack.push(new Integer(currParam));
    currParam = x;
```

Block 2

```
} // end while
```

Similarly, the loop containing block 3 can be replaced by the following ***while*** loop:

```
while (!underFlow) {
```

```
    y = result;
    // simulation of return n * y;
    result = currParam * y;
    try {
          currParam = ((Integer)
          paramStack.pop()).intValue();
          underFlow = false;
    }
    catch (EmptyStackException ese) {
          underFlow = true;
    }
```

Block 3

```
} // end while
```

Substituting the revised block2 and block 3 in the algorithm obviates the need for the ***goto*** statements. Thus our *simFact* algorithm may be implemented by the following Java method:

```
import java.util.*;
public int simFact(long n) {
```

```
          Stack paramStack = new Stack();
          int currParam, x;
          long result = 1, y;
          boolean underFlow = false;

          currParam = n;
```

Initializations

```
start:  // This is the beginning of the simulated factorial method.
```

```
while (currParam != 0) {
      // currParam != 0
      x = currParam - 1;
      // Simulation of recursive call to fact.
      paramStack.push(new Integer (currParam));
      currParam = x;
} // end while
```

Block 2

```
if (currParam == 0) {
```

```
      // simulation of return 1;
      result = 1;
      try {
            currParam = ((Integer)
                  paramStack.pop()).intValue();
            underFlow = false;
      }
      catch (EmptyStackException ese) {
            underFlow = true;
      }
```

Block 1

```
} // end if
if (underFlow) {
      // At this point we return to the main applet.
      return result;
}
```

```
while (!underFlow) {
     y = result;
     // simulation of return n * y;
     result = currParam * y;
     try {
           currParam = ((Integer)
                 paramStack.pop()).intValue();
           underFlow = false;
     }
     catch (EmptyStackException ese) {
           underFlow = true;
     }
} // end while
```

Block 3

```
label1: // At this point we return to the main applet.
return result;
} // end simFact
```

Let us examine these two loops more closely. In block 1, x starts off at the value of the input parameter n and is reduced by 1 each time the subtraction loop is repeated. Each time x is set to a new value, the old value of x is saved on the stack. This continues until x is 0. Thus, after the first loop has been executed, the stack contains, from top to bottom, the integers 1 to n.

The multiplication loop merely removes each of these values from the stack and sets y to the product of the popped value and the old value of y. Since we know what the stack contains at the start of the multiplication loop, why bother popping the stack? We can use those values directly. We can eliminate the stack and the first loop entirely and replace the multiplication loop with a loop that multiplies y by each of the integers from 1 to n in turn. The resulting method is:

```
public long simFact(int n) {
  int x;
  long  y;

  for (y=x=1; x <= n; x++)
        y *= x;
  return y;
} // end simfact
```

But this program is a direct Java implementation of the iterative version of the factorial method as presented in Section 3.1. The only change is that x varies from 1 to n rather than from n to 1.

Simulating the Towers of Hanoi

We have shown that successive transformations of a nonrecursive simulation of a recursive routine may lead to a simpler program for solving a problem. Let us now look at a more complex example of recursion, the Towers of Hanoi problem presented in Section 3.3. We will simulate its recursion and attempt to simplify the simulation to produce a nonrecursive solution. We present again the recursive method from Section 3.3.

```
void towers(int n, char fromPeg, char toPeg, char auxPeg) {
  // If only one disk, make the move and return
  if (n == 1) {
    output1.append("\nmove disk 1 from peg " + fromPeg +
                                              " to peg " + toPeg);
    return;
  } // end if
  // Move top n - 1 disks from A to B, using C as auxiliary
  towers(n - 1, fromPeg, auxPeg, toPeg);
  output1.append("\nmove disk " + n + " from peg " + fromPeg +
                                              " to peg " + toPeg);
  // Move n - 1 disk from B to C using A as auxiliary
  towers(n - 1, auxPeg, toPeg, fromPeg);
} // end towers
```

Make sure that you understand the problem and the recursive solution before proceeding. If you do not, reread Section 3.3.

Earlier in this section, we developed a technique for simulating recursion using a stack. Our method involves the following steps:

1. Define a stack, *dataStack* of objects, of the *DataArea* class, representing the parameters, local variables, and return address of the recursive call. Use the class constructor to initialize an object prior to pushing it on the stack.

2. Construct an algorithm that
   a. pushes the current *DataArea* onto the stack each time a recursive call is initiated, and
   b. pops the stack, branching to the current return address, each time a recursive return is to be simulated.
3. Use a flowchart to analyze the algorithm.
4. Construct a Java method, using the techniques of structured programming in order to eliminate the ***goto*** statements.

There are four parameters in the *towers* method, each of which is subject to change in a recursive call. Therefore, the data area must contain elements representing all four. There are no local variables. There is a single temporary that is needed to hold the value of $n - 1$, but this can be represented by a similar temporary in the simulating program and does not have to be stacked. There are three possible points to which the function returns on various calls: the calling applet and the two points following the recursive calls. Therefore, four labels are necessary:

```
start:
label1:
label2:
label3:.
```

The return address is encoded as an integer (either 1, 2, or 3) in each data area.

We define a class *DataArea* which contains the parameters, *n*, representing the number of pegs; *fromPeg*, *toPeg*, and *auxPeg*, respectively representing the peg from which we are removing disks, the peg to which we will take the disks, and the auxiliary peg; and the return address, *retAddr*.

```
public class DataArea {
  int nParam;
  char fromParam, toParam, auxParam;
  int retAddr;

  public DataArea(int n, char fromPeg, char toPeg,  char auxPeg,
  int retAddr) {
        this.nParam = n;
        this.fromParam = fromPeg;
        this.toParam = toPeg;
        this.auxParam = auxPeg;
        this.retAddr = retAddr;
  }

  public DataArea(DataArea dataArea) {
        this.nParam = dataArea.nParam;
        this.fromParam = dataArea.fromParam;
        this.toParam = dataArea.toParam;
        this.auxParam = dataArea.auxParam;
```

```
            this.retAddr = dataArea.retAddr;
        }
    } // end DataArea class
```

The first constructor allows for the initialization of an object of the *DataArea* class by individually specifying each of the parameters. For example:

```
CurrArea = new DataArea(n, fromPeg, toPeg, AuxPeg, retAddr);
```

It is also convenient to have a constructor that allows us to instantiate a new object from an existing object of the same class (without having to specify each member individually). For example, assuming a ***stack*** *dataStack*, it is easier to write

```
dataStack.push(new DataArea(currArea));
```

than to write

```
dataStack.push(new DataArea(currArea.nParam, currArea.fromParam,
                            currArea.toParam, currArea.auxParam,
                            currArea.retAddr);
```

Consider the following nonrecursive algorithm for the simulation of *towers*: (The highlighted sections refer to the flowchart in Figure 3.4.3.)

```
    import java.util.*;

    public void simTowers(int n, char fromPeg, char toPeg, char
    auxPeg) {
                Stack dataStack = new Stack();
                DataArea currArea;
                char temp;
                int i;

                // Initialize a dummy data area.
                currArea = new DataArea(0, ' ', ' ', ' ', 0);
                // Push dummy data area onto stack.
                dataStack.push(new DataArea(currArea));
                // Set the parameters and the return addresses of the
                // current data area to their proper values.
                currArea = new DataArea(n, fromPeg, toPeg,
                auxPeg, 1);
```

Initializations

```
    start:      // This is the start of the simulated routine.
                if (currArea.nParam == 1) {

                    output1.append("\nmove disk 1 from peg " +
                          currArea.fromParam + " to peg " +
                          currArea.toParam);
                    i = currArea.retAddr;
                    currArea = (DataArea) dataStack.pop();
```

Block 0

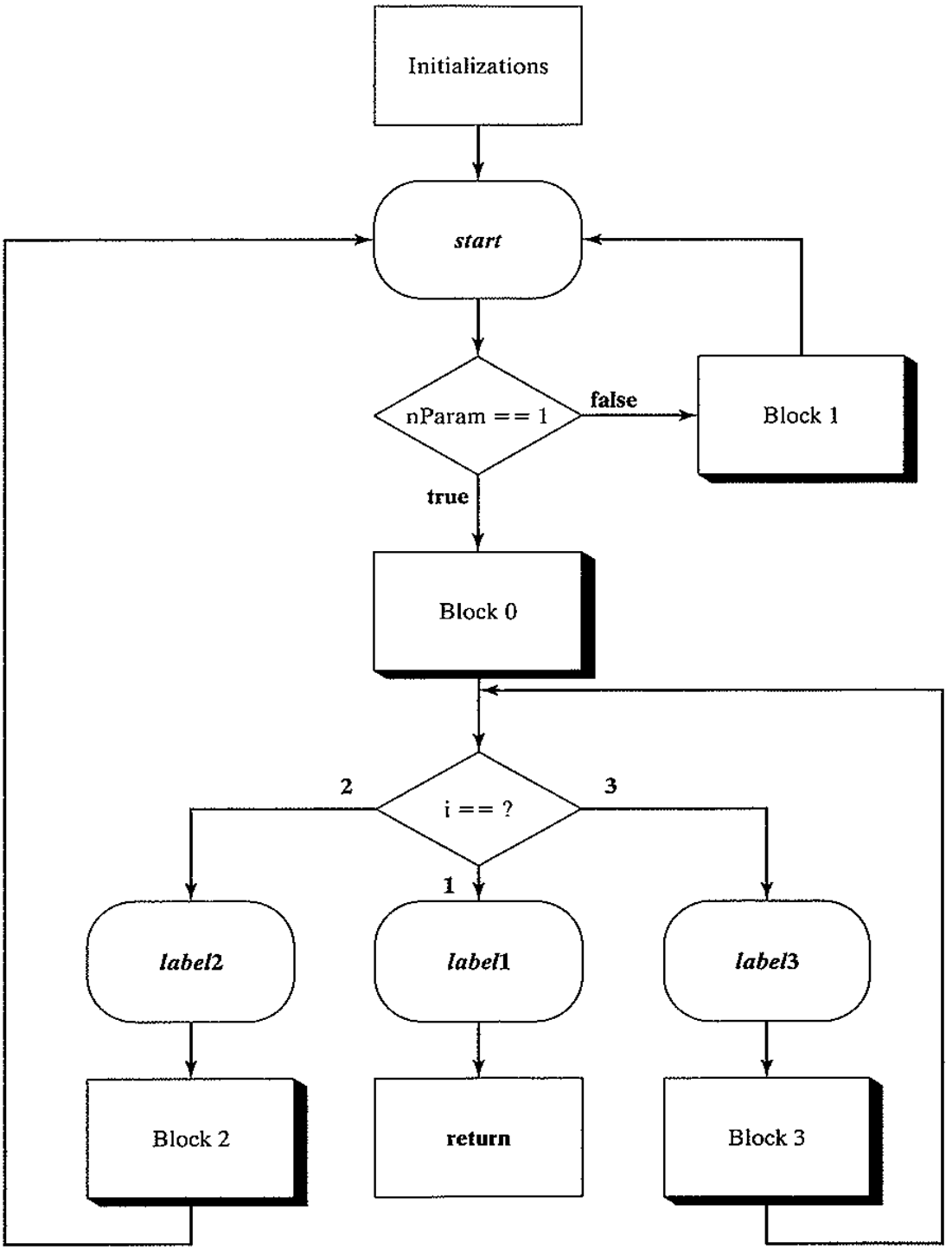

FIGURE 3.4.3 Flowchart for Towers of Hanoi simulation.

```
            switch (i) {
                  case 1:    goto label1;
                  case 2:    goto label2;
                  case 3:    goto label3;
            }
      } // end if
```

```
                    // This is the first recursive call.
                    dataStack.push(new DataArea(currArea));
                    --currArea.nParam;
                    temp = currArea.auxParam;
                    currArea.auxParam = currArea.toParam;
                    currArea.toParam = temp;
                    currArea.retAddr = 2;
```

Block 1

```
                    goto start;
label2: // We return to this point from the first recursive call.
                    output1.append("\nmove disk " + currArea.nParam +
                                   " from peg " + currArea.fromParam +
                                   " to peg " + currArea.toParam);
                  // This is the second recursive call.
                  dataStack.push(currArea);
                  --currArea.nParam;
                  temp = currArea.fromParam;
                  currArea.fromParam = currArea.auxparam;
                  currArea.auparam = temp;
                  currArea.retAddr = 3;
```

Block 2

```
                    goto start;
label3:    // Return to this point from the second recursive call.
                    i = currArea.retAddr;
                    currArea = (DataArea) dataArea.pop();
```

Block 3

```
                    switch (i) {
                        case 1:   goto label1;
                        case 2:   goto label2;
                        case 3:   goto label3;
                    }
label1:   return;
} // end simTowers
```

Close examination of the flowchart in Figure 3.4.3 reveals that the algorithm may be rewritten in outline form as:

```
initializations;
while (true) {
  while (nParam != 1)
        Block 1;
  Block 0;
  if (i == 1)
        return;
  while (i == 3)
        Block 3;
```

```
        if (i == 1)
              return;
        Block 2;
    } // end while
```

where initializations, block 0 through block 3 correspond to the lines highlighted in the original algorithm. Thus our *simTowers* algorithm may be implemented by the following Java method:

```
import java.util.*

public void simTowers(int n, char fromPeg, char toPeg, char auxPeg) {
    Stack dataStack = new Stack();
    DataArea currArea;
    char temp;
    int i;

    // Initialize a dummy data area.
    currArea = new DataArea(0, ' ', ' ', ' ', 0);
    // Push dummy data area onto stack.
    dataStack.push(currArea);
    // Set the parameters and the return addresses of the
    // current data area to their proper values.
    currArea = new DataArea(n, fromPeg, toPeg, auxPeg, 1);
```

Initializations

```
    while (true) {
        while (currArea.nParam != 1) {

            // This is the first recursive call.
            dataStack.push(new DataArea(currArea));
            --currArea.nParam;
            temp = currArea.auxParam;
            currArea.auxParam = currArea.toParam;
            currArea.toParam = temp;
            currArea.retAddr = 2;
```

Block 1

```
        } // end while

        // This is the start of the simulated routine.
        output1.append("\nmove disk 1 from peg " +
        currArea.fromParam + " to peg " + currArea.toParam);
        i = currArea.retAddr;
        currArea = (DataArea) dataArea.pop();
```

Block 0

```
        if (i == 1)
              return;
        while (i == 3) {

            // Return to this point from the second recursive
            call.
            i = currArea.retAddr;
            currArea = (DataArea) dataStack.pop();
```

Block 3

```
        } // end while
        if (i == 1)
              return;

        // We return to this point from the first recursive
        // call.
        output1.append("\nmove disk " + currArea.nParam +
                       " from peg " + currArea.fromParam +
                       " to peg " + currArea.toParam);
        // This is the second recursive call.
        dataStack.push(new DataArea(currArea));
        --currArea.nParam;
        temp = currArea.fromParam;
        currArea.fromParam = currArea.auxParam;
        currArea.auxParam = temp;
        currArea.retAddr = 3;
    } // end while
} // end simTowers
```

Block 2

Trace through the actions of this program and see how it reflects the actions of the original recursive version.

EXERCISES

3.4.1 Write a nonrecursive simulation of the functions *convert* and *find* presented in Section 3.3.

3.4.2 Write a nonrecursive simulation of the recursive binary search procedure, and transform it into an iterative procedure.

3.4.3 Write a nonrecursive simulation of *fib*. Can you transform it into an iterative method?

3.4.4 Write nonrecursive simulations of the recursive routines in Sections 3.2 and 3.3 and the exercises in those sections.

3.4.5 Show that any solution to the Towers of Hanoi problem that uses a minimum number of moves must satisfy the conditions listed below. Use these facts to develop a direct iterative algorithm for the Towers of Hanoi. Implement the algorithm as a Java applet.

1. The first move involves moving the smallest disk.
2. A minimum-move solution consists of alternately moving the smallest disk and a disk that is not the smallest.
3. At any point, there is only one possible move involving a disk that is not the smallest.
4. Define the cyclic direction from *fromPeg* to *toPeg* to *auxPeg* to *fromPeg* as clockwise, and the opposite direction (from *fromPeg* to *auxPeg* to *toPeg* to *fromPeg*) as counterclockwise. Assume that a minimum-move solution to move a k-disk tower from *fromPeg* to *toPeg* always moves the smallest disk in one direction. Show that a minimum-move solution to move a $(k + 1)$-disk tower from *fromPeg* to *toPeg* would then always move the smallest disk in the other direction. Since the solution for one disk moves the smallest disk clockwise (the single move from *fromPeg* to *toPeg*), this means that for an odd

number of disks, the smallest disk always moves clockwise, and for an even number of disks, the smallest disk always moves counterclockwise.

5. The solution is completed as soon as all the disks are on a single peg.

3.4.6 Convert the following recursive program scheme into an iterative version that does not use a stack. $f(n)$ is a method that returns *true* or *false* based on the value of n, and $g(n)$ is a function that returns a value of the same type as n (without modifying n).

```
int rec(int n) {
        if (!f(n)) {
                // any group of Java statements that
                // do not change the value of n
                rec(g(n));
        } // end if
} // end rec
```

Generalize your result to the case in which *rec* returns a value.

3.4.7 Let $f(n)$ be a method and $g(n)$ and $h(n)$ be methods that return a value of the same type as n without modifying n. Let (*stmts*) represent any group of Java statements that do not modify the value of n. Show that the recursive program scheme *rec* is equivalent to the iterative scheme *iter*:

```
void rec(int n) {
      if (!f(n)) {
              (stmts)
              rec(g(n));
              rec(h(n));

      } // end if
} // end rec

void iter(int n) {
  Stack s = new Stack();
  s.push(n);
  while (!s.empty()) {
        n = s.pop();
        if (!f(n)) {
                (stmts)
                s.push(h(n));
                s.push(g(n));
        } // end if
  } // end while
} // end iter
```

Show that the *if* statements in *iter* can be replaced by the loop:

```
while(!f(n)) {
      (stmts)
      s.push(h(n));
      n = g(n);
} // end while
```

3.5 EFFICIENCY OF RECURSION

In general, a nonrecursive version of a program will execute more efficiently in terms of time and space than a recursive version. This is because the overhead involved in entering and exiting a block is avoided in the nonrecursive version. As we have seen, it is often possible to identify a good number of local variables and temporaries that do not have to be saved and restored through the use of a stack. A nonrecursive program can eliminate this needless stacking activity. However, since the compiler in a recursive procedure is usually unable to identify such variables, they are stacked and unstacked to ensure that no problems arise.

We have also seen that a recursive solution is sometimes the most natural and logical way of solving a problem. It is doubtful whether a programmer could have developed the nonrecursive solution to the Towers of Hanoi problem directly from the problem statement. Much the same may be said about the problem of converting prefix to postfix, where the recursive solution flows directly from the definitions. A nonrecursive solution involving stacks is more difficult to develop and more prone to error.

Thus there is a conflict between machine efficiency and programmer efficiency. With the cost of programming increasing steadily, and the cost of computation decreasing, we have reached the point where in most cases it is not worth a programmer's time to laboriously construct a nonrecursive solution to a problem that is most naturally solved recursively. Of course an incompetent overly clever programmer may come up with a complicated recursive solution to a simple problem that can be solved directly by nonrecursive methods. (An example of this is the factorial function or even the binary search.) However, if a competent programmer identifies a recursive solution as the simplest and most straightforward method for solving a particular problem, it is probably not worth the time and effort to discover a more efficient method.

This is not always the case, though. If a program is to be run very frequently (often, entire computers are dedicated to continually running the same program), so that increased efficiency in execution speed significantly increases throughput, then the extra investment in programming time is worthwhile. Even in such cases, it is probably better to create a nonrecursive version by simulating and transforming the recursive solution than by attempting to create a nonrecursive solution from the problem statement.

To do this most efficiently, what is required is to first write the recursive routine and then its simulated version, including all the stacks and temporaries. Next, eliminate any superfluous stacks and variables. The final version is a refinement of the original program, and is certainly more efficient. The elimination of superfluous and redundant operations improves the efficiency of the resulting program. Nonetheless, every transformation applied to a program is another opening through which an unanticipated error may creep in.

When a stack cannot be eliminated from the nonrecursive version of a program and the recursive version does not contain any extra parameters or local variables, the recursive version can be as fast or faster than the nonrecursive version under a good compiler. The Towers of Hanoi is an example of such a recursive program. Factorial, whose nonrecursive version does not need a stack, and calculation of Fibonacci numbers, which

contains an unnecessary second recursive call (and does not need a stack either), are examples of cases where recursion should be avoided in a practical implementation. We examine another example of efficient recursion (inorder tree traversal) in Section 5.2.

Another point to remember is that explicit calls to *pop*, *push*, and *empty*, as well as tests for underflow and overflow, are quite expensive. In fact, they can often outweigh the expense of the overhead of recursion. Thus, to maximize the actual runtime efficiency of a nonrecursive translation, these calls should be replaced by inline code and the overflow/underflow tests eliminated when it is known that we are operating within the array bounds.

The ideas and transformations that we have put forward in presenting the factorial function and the Towers of Hanoi can be applied to more complex problems whose nonrecursive solutions are not readily apparent. The extent to which a recursive solution (actual or simulated) can be transformed into a direct solution depends on the specifics of the problem and the ingenuity of the programmer.

EXERCISES

3.5.1 Run the recursive and nonrecursive versions of the factorial method in Sections 3.2 and 3.4, and examine how much space and time each requires as n becomes larger.

3.5.2 Do the same as Exercise 3.5.1 above for the Towers of Hanoi problem.