quadruple is removed and again three quadruples replace it, leaving

$$\begin{array}{|c|}(1,1,2,3)\\(1,1,3,2)\\(1,3,1,2)\\(1,1,2,3)\\(2,2,1,3)\\\hline\end{array}$$

of the top four elements is removed from the stack, it causes a disk to be moved in the following sequence: pole 1 to pole 3, pole 1 to pole 2, pole 3 to pole 2, pole 1 to pole 3. The stack now contains

$$\begin{array}{|c|}(2,2,1,3)\\\hline\end{array}$$

The quadruple is removed and three quadruples replace it, leaving

$$\begin{array}{|c|}(1,2,3,1)\\(1,2,1,3)\\(1,1,2,3)\\\hline\end{array}$$

of these elements is removed from the stack, causing disks to be moved as follows: pole 2 to pole 1, pole 2 to pole 3, pole 1 to pole 3. All three disks are now in order on pole 3 as desired. The stack has precisely captured the essence of the recursive algorithm.

Stacks and Arithmetic Expressions. The problem of evaluating an arithmetic expression arises at all levels of computation, from hand calculations to using a pocket calculator to programming large computers. It is a nontrivial task to transform the expression into a sequence of simple arithmetic operations. In this section we examine two important algorithms for such manipulations of arithmetic expressions; both algorithms center around the use of stacks. In a sense, however, this material could fall under the previous heading of "stacks and recursion," since it is possible to view these algorithms as iterative implementations of their recursive counterparts (see Exercises 6 and 8).

Before constructing algorithms for any task, we must have a clear understanding of what the task is. For the evaluation of arithmetic expressions, what we want to do is fairly obvious, except for one thing: given the arithmetic expression $A + B * C$, we want to compute $A + (B * C)$ and *not* $(A + B) * C$. Without parentheses, how do we know which one we want? The answer lies in the notion of *precedence*. It is the convention that the multiplicative operations of multiplication and division take precedence over the additive operations of addition and subtraction; in other words, when there are no parentheses to make it otherwise, the multiplicative operations are done first, before the additive operations. All right, but how do we evaluate $A - B + C$? Should it be $A - (B + C)$ or $(A - B) + C$? Of course we want $(A - B) + C$, because it is the convention that operations of equal precedence are performed from left to right. As a further example, $A/B * C$ would be evaluated as $(A/B) * C$ and not $A/(B * C)$.

Our problem is thus to evaluate arithmetic expressions that contain variables, $+$, $-$, $*$, $/$, and parentheses, with the conventions that $*$ and $/$ will be done before $+$ and $-$ (in the absence of parentheses, of course!) and that sequences of $+$s and $-$s or sequences of $*$s and $/$s will be done from left to right. This evaluation is tricky and can be done by a very clever recursive algorithm (see Exercise 6), but here we will do it by a simpler two-stage process that emphasizes the use of stacks. We will first show how to evaluate the expression, assuming it has been converted into an intermediate form, and then show how to do the conversion to that form.

The intermediate form is *Polish postfix* notation, in which the operator follows its two operands, rather than separating them as in conventional notation. For example, instead of $A + B$ we would write $AB+$. (To eliminate the ambiguities possible when multicharacter variable names are adjacent to one another, we will insist that all variable names be single characters; this restriction is easy to overcome by using separator characters, but at the expense of some clarity of presentation.) For $A + B * C$ we would have $ABC*+$, which is interpreted as follows: The two operands for the $*$ are the B and C that precede it; the two operands of the $+$ are A and the expression $BC*$. If instead we want $(A + B) * C$, we write $AB + C*$, so that the two operands for the $+$ are the A and B that precede it, while the two operands for the $*$ are the expression $AB+$ and C. The examples $ABC*+$ and $AB+C*$ illustrate the most important characteristic of Polish postfix notation: it does not need parentheses or precedence conventions to indicate the order of the computation; the order is defined completely by the relative order of the operands and the operators.

We can define the class of Polish postfix expressions recursively as follows: such an expression is either a simple variable, or consists of two Polish postfix expressions followed by an operator. The recursive definition gives us the key to the evaluation of postfix expressions. Consider, for example, the expression

$$AB + CD - E * F + *,$$

which corresponds to the expression

$$(A + B) * ((C - D) * E + F).$$

The order of evaluation is as follows:

$$\underbrace{\underbrace{A\ B\ +}_{A+B}\ \underbrace{\underbrace{\underbrace{\underbrace{C\ D\ -}_{C-D}\ E\ *}_{(C-D)*E}\ F\ +}_{(C-D)*E+F}\ *}_{}}_{(A+B)*((C-D)*E+F)}$$

The general rule used in this example is that whenever we find two operands followed by an operator, that operator is applied to those operands and the result replaces the substring consisting of the operands and operator. Thus, in the above example, we replaced the substring "$AB+$" by the value of $A + B$; then we replaced the substring "$CD-$" by the value of $C - D$; then we replaced the substring consisting of the value of $C - D$ followed by "$E*$" with the value of $(C - D) * E$, and so on, finally replacing the entire string with the value of $(A + B) * ((C - D) * E + F)$.

More precisely, the algorithm to evaluate postfix expressions operates by scanning the expression one character at a time from left to right. Operands are placed on a stack and operators are applied to the top two stack entries, which are deleted and replaced by the result of the operation. Algorithm 4.7 is a straightforward implementation of this process, assuming that the postfix expression is stored in an array $P[1], P[2], \ldots, P[n]$.

We are left with the problem of converting the usual infix expression into its equivalent Polish postfix form. To convert an expression into postfix form we must repeatedly replace an operand-operator-operand sequence by operand-oper-

```
S ← empty stack
for i = 1 to n do
    if P[i] is an operand then S ⇐ P[i]
        else
            y ⇐ S
            x ⇐ S
            S ⇐ value of the operator P[i]
                  applied to x and y
```

Algorithm 4.7
Evaluation of Polish postfix expression $P[1], P[2], \ldots, P[n]$.

and-operator. For example, the following illustrates the transformation of

$$(A + B) * ((C - D) * E + F)$$

into postfix form:

$$\underbrace{(\underbrace{A + B}_{AB+}) * (\underbrace{(\underbrace{(C - D)}_{CD-} * E}_{CD-E*} + F}_{CD-E*F+})}_{AB+CD-E*F+*}$$

To do this as we scan the expression from left to right, we use a stack as follow. When we scan an operator, then we know its left operand has already been converted to postfix and is in the output string. So, we store the operator on the stack and process its right operand. After finishing with its right operand, the operator will conveniently be at the top of the stack; we remove it and add it to the output string.

It is clear from this description of the process that operands must go directly into the output string and operators go into the stack. However, if we have just finished the second of the two operands of an operator, then that operator will be on top of the stack and we must recognize that it is time to put it into the output string. The end of the second of the operands occurs for some operator at a closing parenthesis, at another operator for which the preceding was the first operand, or at the end of the input string. The case of another operator is handled by observing that, if this incoming operator has lower or equal precedence to the one on top of the stack, then we must have completed the second operand of the operator on top of the stack; that operator can now be popped off the stack and added to the output string. This process must be repeated for the new top stack element, and so on. We now have only finished the first operand for this incoming operator, and it is added to the stack. We handle the case of the end of the input string by adding a special end-of-string-marker, a "$" that is treated as a very low precedence operator, causing the above outlined loop to dump out the stack when the "$" is encountered.

Algorithm 4.8 embodies the procedure just outlined; it converts an expression in an array E to Polish form in array P. In the algorithm the bottom of the stack is marked by the symbol "#" that is treated as an operator of even lower precedence than the "$" that marks the end of the input string. This causes the "$" to be put onto the stack, the end condition of the algorithm. The algorithm

uses the precedence function *PREC* as given in Table 4.1 to determine the relative precedences of two operators. Notice that the "(" has precedence lower than the arithmetic operations in order to keep it in the stack appropriately.

```
$S \leftarrow$ empty stack
$S \Leftarrow$ "#"          ⟦bottom of stack marker⟧
$j \leftarrow 0$            ⟦output string cursor⟧
$i \leftarrow 0$            ⟦input string cursor⟧
while $top(S) \neq$ "$" do
    ⟦"$" is the end of input string marker; its precedence is set so
    that it will cause the stack to be emptied before it is pushed on⟧
    $i \leftarrow i + 1$      ⟦process next character of input string⟧
    case
        $E[i]$ is an operand:
            ⟦transfer it to the output string⟧
            $j \leftarrow j + 1$
            $P[j] \leftarrow E[i]$
        $E[i] =$ "(": $S \Leftarrow E[i]$     ⟦stack left parenthesis⟧
        $E[i] =$ ")":
            ⟦empty stack contents down to the matching parenthesis⟧
            $x \Leftarrow S$
            while $x \neq$ "(" do
                $j \leftarrow j + 1$
                $P[j] \leftarrow x$
                $x \Leftarrow S$
        $E[i]$ is an operator:
            ⟦empty stack contents down to first operator
            with lower precedence, then stack the operator)⟧
            while $PREC(E[i]) \leq PREC(top(S))$ do
                $j \leftarrow j + 1$
                $P[j] \Leftarrow S$
            $S \Leftarrow E[i]$
```

Algorithm 4.8
Conversion of an infix expression $E[1], E[2], \ldots, E[n]$ into its equivalent Polish postfix form $P[1], P[2], \ldots, P[m]$. The *PREC* function used is defined in Tabie 4.1.

Algorithm 4.8 is really only the bare bones. To be useful, such an algorithm must check for syntactic errors (how does Algorithm 4.8 react to invalid expressions?) and allow for other operations such as exponentiation and unary minus. These issues are the subject of Exercises 9 and 10.

Character	*Precedence*
#	0
$	1
(	2
+, −	3
*, /	4

Table 4.1
The precedence function *PREC* used in Algorithm 4.8 for the conversion of infix expressions to their equivalent postfix form. The character "#" marks the bottom of the stack and "$" marks the end of the input string.

Queues. Applications of queues tend to be too intricate to allow the extraction of a concise example. For example, queues are needed in the simulation of various business systems requiring processing customers, orders, jobs, or requests in the order that they arrive. The operation of some computer systems requires that jobs be executed in their order of submission; in this case, again, a queue is mandated. Within the computer itself, queues are needed to keep track of input/output requests—since they are so time consuming relative to internal operations, they can accumulate; if their order is not carefully adhered to, a program might end up, for instance, trying to read a record that has not yet been written.

Exercises

1. In the Towers of Hanoi problem, how many moves are required to move the pile of n disks from the original pole to the final pole?
2. Modify Algorithms 4.5 and 4.6 to keep enough information so that after each move a picture can be printed of the current arrangement of the disks on the poles.
3. Modify Algorithm 4.5 so that the "basic" case is $n = 0$, not $n = 1$. Compare the efficiency of this modification to the original algorithm.
4. Suppose there are n disks and *four* poles. Design an algorithm to do the moving in this case.
5. Consider the following recursively defined sequences of integers: $T_1 =$ (1), $T_{n+1} = (T_n, n + 1, T_n)$. Thus, for example, $T_2 = (1, 2, 1)$ and $T_3 =$ (1, 2, 1, 3, 1, 2, 1). Design a nonrecursive algorithm based on a stack to generate T_n. Find a different algorithm based on divisibility by 2. What is the relation of this sequence T_n to the Towers of Hanoi problem?
6. Suppose we have arithmetic expressions over +, *, parentheses, and variable names, with each expression terminated by a "$". Such an expression