Question 4 Trace the execution of the language-recognition algorithm described in the previous section for each of the following strings, and show the contents of the stack at each step.

a. a$a
b. ab$ab
c. ab$a
d. ab$ba

6.3 Using Stacks with Algebraic Expressions

Your use of an ADT's operations should not depend on its implementation

This section contains two more problems that you can solve neatly by using the ADT stack. Keep in mind throughout that you are using the ADT stack to solve the problems. You can use the stack operations, but you may not assume any particular implementation. You choose a specific implementation only as a last step.

Chapter 5 presented recursive grammars that specified the syntax of algebraic expressions. Recall that prefix and postfix expressions avoid the ambiguity inherent in the evaluation of infix expressions. We will now consider stack-based solutions to the problems of evaluating infix and postfix expressions. To avoid distracting programming issues, we will allow only the binary operators *, /, +, and –, and we will disallow exponentiation and unary operators.

To evaluate an infix expression, first convert it to postfix form and then evaluate the postfix expression

The strategy we shall adopt here is first to develop an algorithm for evaluating postfix expressions and then to develop an algorithm for transforming an infix expression into an equivalent postfix expression. Taken together, these two algorithms provide a way to evaluate infix expressions. This strategy eliminates the need for an algorithm that directly evaluates infix expressions, which is a somewhat more difficult problem that Programming Problem 8 considers.

6.3.1 Evaluating Postfix Expressions

As we mentioned in Chapter 5, some calculators require you to enter postfix expressions. For example, to compute the value of

2 * (3 + 4)

by using a postfix calculator, you would enter the sequence 2, 3, 4, +, and *, which corresponds to the postfix expression

2 3 4 + *

Recall that an operator in a postfix expression applies to the two operands that immediately precede it. Thus, the calculator must be able to retrieve the operands entered most recently. The ADT stack provides this capability. In fact, each time you enter an operand, the calculator pushes it onto a stack. When you enter an operator, the calculator applies it to the top two operands on the stack, pops the operands from the stack, and pushes the result of the operation onto the stack. Figure 6-4 shows the action of the calculator for the previous sequence of operands and operators. The final result, 14, is on the top of the stack.

You can formalize the action of the calculator to obtain an algorithm that evaluates a postfix expression, which is entered as a string of characters. To avoid issues that cloud the algorithm with programming details, assume that

Simplifying assumptions

- The string is a syntactically correct postfix expression
- No unary operators are present
- No exponentiation operators are present
- Operands are single lowercase letters that represent integer values

FIGURE 6-4 The effect of a postfix calculator on a stack when evaluating the expression 2 * (3 + 4)

Key entered	Calculator action		Stack (bottom to top):
2	push 2		2
3	push 3		2 3
4	push 4		2 3 4
+	operand2 = peek	(4)	2 3 4
	pop		2 3
	operand1 = peek	(3)	2 3
	pop		2
	result = operand1 + operand2	(7)	
	push result		2 7
*	operand2 = peek	(7)	2 7
	pop		2
	operand1 = peek	(2)	2
	pop		
	result = operand1 * operand2	(14)	
	push result		14

A pseudocode algorithm that evaluates postfix expressions

The pseudocode algorithm is then

```
for (each character ch in the string)
{
   if (ch is an operand)
      Push the value of the operand ch onto the stack
   else // ch is an operator named op
   {
      // Evaluate and push the result
      operand2 = top of stack
      Pop the stack

      operand1 = top of stack
      Pop the stack

      result = operand1 op operand2
      Push result onto the stack
   }
}
```

Upon termination of the algorithm, the value of the expression will be on the top of the stack. Programming Problem 5 at the end of this chapter asks you to implement this algorithm.

Question 5 Evaluate the postfix expression $a\ b - c\ +$. Assume the following values for the identifiers: $a = 7$, $b = 3$, and $c = -2$. Show the status of the stack after each step.

6.3.2 Converting Infix Expressions to Equivalent Postfix Expressions

Now that you know how to evaluate a postfix expression, you will be able to evaluate an infix expression if you first can convert it into an equivalent postfix expression. The infix expressions here are the

familiar ones, such as $(a + b) * c / d - e$. They allow parentheses, operator precedence, and left-to-right association.

Will you ever want to evaluate an infix expression? Certainly—you have written such expressions in programs. The compiler that translated your programs had to generate machine instructions to evaluate the expressions. To do so, the compiler first transformed each infix expression into postfix form. Knowing how to convert an expression from infix to postfix notation not only will lead to an algorithm to evaluate infix expressions, but also will give you some insight into the compilation process.

If you manually convert a few infix expressions to postfix form, you will discover three important facts:

Facts about converting from infix to postfix

- The operands always stay in the same order with respect to one another.
- An operator will move only "to the right" with respect to the operands; that is, if in the infix expression the operand *x* precedes the operator *op,* it is also true that in the postfix expression the operand *x* precedes the operator *op*.
- All parentheses are removed.

As a consequence of these three facts, the primary task of the conversion algorithm is determining where to place each operator.

The following pseudocode describes a first attempt at converting an infix expression to an equivalent postfix expression `postfixExp`:

First draft of an algorithm to convert an infix expression to postfix form

```
Initialize postfixExp to the empty string
for (each character ch in the infix expression)
{
   switch (ch)
   {
      case ch is an operand:
         Append ch to the end of postfixExp
         break
      case ch is an operator:
         Save ch until you know where to place it
         break
      case ch is a '(' or a ')':
         Discard ch
         break
   }
}
```

You may have guessed that you really do not want simply to discard the parentheses, as they play an important role in determining the placement of the operators. In any infix expression, a set of matching parentheses defines an isolated subexpression that consists of an operator and its two operands. Therefore, the algorithm must evaluate the subexpression independently of the rest of the expression. Regardless of what the rest of the expression looks like, the operator within the subexpression belongs with the operands in that subexpression. The parentheses tell the rest of the expression that

> *You can have the value of this subexpression after it is evaluated; simply ignore everything inside.*

Parentheses, operator precedence, and left-to-right association determine where to place operators in the postfix expression

Parentheses are thus one of the factors that determine the placement of the operators in the postfix expression. The other factors are precedence and left-to-right association.

In Chapter 5, you saw a simple way to convert a fully parenthesized infix expression to postfix form. Because each operator corresponded to a pair of parentheses, you simply moved each operator to the position marked by its close parenthesis and finally removed the parentheses.

The actual problem is more difficult, however, because the infix expression is not always fully parenthesized. Instead, the problem allows precedence and left-to-right association, and therefore

requires a more complex algorithm. The following is a high-level description of what you must do when you encounter each character as you read the infix string from left to right.

Five steps in the process to convert from infix to postfix form

1. When you encounter an operand, append it to the output string `postfixExp`.
   Justification: The order of the operands in the postfix expression is the same as the order in the infix expression, and the operands that appear to the left of an operator in the infix expression also appear to its left in the postfix expression.
2. Push each "(" onto the stack.
3. When you encounter an operator, if the stack is empty, push the operator onto the stack. However, if the stack is not empty, pop operators of greater or equal precedence from the stack and append them to `postfixExp`. You stop when you encounter either a "(" or an operator of lower precedence or when the stack becomes empty. You then push the current operator in the expression onto the stack. Thus, this step orders the operators by precedence and in accordance with left-to-right association. Notice that you continue popping from the stack until you encounter an operator of strictly lower precedence than the current operator in the infix expression. You do not stop on equality, because the left-to-right association rule says that in case of a tie in precedence, the leftmost operator is applied first—and this operator is the one that is already on the stack.
4. When you encounter a ")", pop operators off the stack and append them to the end of `postfixExp` until you encounter the matching "(".
   Justification: Within a pair of parentheses, precedence and left-to-right association determine the order of the operators, and step 3 has already ordered the operators in accordance with these rules.
5. When you reach the end of the string, you append the remaining contents of the stack to `postfixExp`.

For example, Figure 6-5 traces the action of the algorithm on the infix expression $a - (b + c * d) / e$, assuming that the stack `aStack` and the string `postfixExp` are initially empty. At the end of the algorithm, `postfixExp` contains the resulting postfix expression $a\ b\ c\ d\ *\ +\ e\ /\ -$.

You can use the previous five-step description of the algorithm to develop a fairly concise pseudocode solution, which follows. The symbol • in this algorithm means concatenate (join), so

FIGURE 6-5 A trace of the algorithm that converts the infix expression $a - (b + c * d) / e$ to postfix form

ch	aStack (bottom to top)	postfixExp	
a		a	
–	–	a	
(	– (	a	
b	– (	ab	
+	– (+	ab	
c	– (+	abc	
*	– (+ *	abc	
d	– (+ *	abcd	
)	– (+	abcd*	Move operators from stack to `postfixExp` until "("
	– (	abcd*+	
	–	abcd*+	
/	– /	abcd*+	Copy operators from stack to `postfixExp`
e	– /	abcd*+e	
		abcd*+e/–	

`postfixExp • x` means concatenate the string currently in `postfixExp` and the character `x`—that is, follow the string in `postfixExp` with the character `x`.

A pseudocode algorithm that converts an infix expression to postfix form

```
for (each character ch in the infix expression)
{
   switch (ch)
   {
      case operand:       // Append operand to end of postfix expression—step 1
         postfixExp = postfixExp • ch
         break
      case '(':           // Save '(' on stack—step 2
         aStack.push(ch)
         break
      case operator:      // Process stack operators of greater precedence—step 3
         while (!aStack.isEmpty() and aStack.peek() is not a '(' and
                precedence(ch) <= precedence(aStack.peek()))
         {
            Append aStack.peek() to the end of postfixExp
            aStack.pop()
         }
         aStack.push(ch) // Save the operator
         break
      case ')':           // Pop stack until matching '('—step 4
         while (aStack.peek() is not a '(')
         {
            Append aStack.peek() to the end of postfixExp
            aStack.pop()
         }
         aStack.pop()     // Remove the open parenthesis
         break
   }
}

// Append to postfixExp the operators remaining in the stack—step 5
while (!aStack.isEmpty())
{
   Append aStack.peek() to the end of postfixExp
   aStack.pop()
}
```

Because this algorithm assumes that the given infix expression is syntactically correct, it can ignore the return values of the stack operations. Programming Problem 7 at the end of this chapter asks you to remove this assumption.

Note: Algorithms that evaluate an infix expression or transform one to postfix form must determine which operands apply to a given operator. Doing so allows for precedence and left-to-right association so that you can omit parentheses.

Question 6 Convert the infix expression *a* / *b* * *c* to postfix form. Be sure to account for left-to-right association. Show the status of the stack after each step.

Question 7 Explain the significance of the precedence tests in the infix-to-postfix conversion algorithm. Why is a ≥ test used rather than a > test?