15.6.2 Evaluating Reverse Polish Expressions

Use a stack to evaluate expressions in reverse Polish notation.

Consider how you write arithmetic expressions, such as (3 + 4) × 5. The parentheses are needed so that 3 and 4 are added before multiplying the result by 5.

However, you can eliminate the parentheses if you write the operators *after* the numbers, like this: 3 4 + 5 × (see Random Fact 15.2 on page 701). To evaluate this expression, apply + to 3 and 4, yielding 7, and then simplify 7 5 × to 35. It gets trickier for complex expressions. For example, 3 4 5 + × means to compute 4 5 + (that is, 9), and then evaluate 3 9 ×. If we evaluate this expression left-to-right, we need to leave the 3 somewhere while we work on 4 5 +. Where? We put it on a stack. The algorithm for evaluating reverse Polish expressions is simple:

```
If you read a number
   Push it on the stack.
Else if you read an operand
   Pop two values off the stack.
   Combine the values with the operand.
   Push the result back onto the stack.
Else if there is no more input
   Pop and display the result.
```

Here is a walkthrough of evaluating the expression 3 4 5 + ×:

Stack	Unread expression	Comments
Empty	3 4 5 + x	
3	4 5 + x	Numbers are pushed on the stack
3 4	5 + x	
3 4 5	+ x	
3 9	x	Pop 4 and 5, push 4 5 +
27	No more input	Pop 3 and 9, push 3 9 x
Empty		Pop and display the result, 27

The following program simulates a reverse Polish calculator:

section_6_2/Calculator.java

```
1  import java.util.Scanner;
2  import java.util.Stack;
3
4  /**
5     This calculator uses the reverse Polish notation.
6  */
7  public class Calculator
8  {
9     public static void main(String[] args)
10    {
11       Scanner in = new Scanner(System.in);
12       Stack<Integer> results = new Stack<Integer>();
13       System.out.println("Enter one number or operator per line, Q to quit. ");
14       boolean done = false;
```

```
15          while (!done)
16          {
17             String input = in.nextLine();
18
19             // If the command is an operator, pop the arguments and push the result
20
21             if (input.equals("+"))
22             {
23                results.push(results.pop() + results.pop());
24             }
25             else if (input.equals("-"))
26             {
27                Integer arg2 = results.pop();
28                results.push(results.pop() - arg2);
29             }
30             else if (input.equals("*") || input.equals("x"))
31             {
32                results.push(results.pop() * results.pop());
33             }
34             else if (input.equals("/"))
35             {
36                Integer arg2 = results.pop();
37                results.push(results.pop() / arg2);
38             }
39             else if (input.equals("Q") || input.equals("q"))
40             {
41                done = true;
42             }
43             else
44             {
45                // Not an operator--push the input value
46
47                results.push(Integer.parseInt(input));
48             }
49             System.out.println(results);
50          }
51       }
52    }
```

15.6.3 Evaluating Algebraic Expressions

Using two stacks, you can evaluate expressions in standard algebraic notation.

In the preceding section, you saw how to evaluate expressions in reverse Polish notation, using a single stack. If you haven't found that notation attractive, you will be glad to know that one can evaluate an expression in the standard algebraic notation using two stacks—one for numbers and one for operators.

Use two stacks to evaluate algebraic expressions.

First, consider a simple example, the expression 3 + 4. We push the numbers on the number stack and the operators on the operator stack. Then we pop both numbers and the operator, combine the numbers with the operator, and push the result.

	Number stack Empty	Operator stack Empty	Unprocessed input 3 + 4	Comments
1	3		+ 4	
2	3	+	4	
3	4 3	+	No more input	Evaluate the top.
4	7			The result is 7.

This operation is fundamental to the algorithm. We call it "evaluating the top".

In algebraic notation, each operator has a *precedence*. The + and - operators have the lowest precedence, * and / have a higher (and equal) precedence.

Consider the expression 3 × 4 + 5. Here are the first processing steps:

	Number stack Empty	Operator stack Empty	Unprocessed input 3 × 4 + 5	Comments
1	3		× 4 + 5	
2	3	×	4 + 5	
3	4 3	×	+ 5	Evaluate × before +.

Because × has a higher precedence than +, we are ready to evaluate the top:

	Number stack	Operator stack		Comments
4	12	+	5	
5	5 12	+	No more input	Evaluate the top.
6	17			That is the result.

With the expression, 3 + 4 × 5, we add × to the operator stack because we must first read the next number; then we can evaluate × and then the +:

	Number stack Empty	Operator stack Empty	Unprocessed input 3 + 4 × 5	Comments
1	3		+ 4 × 5	
2	3	+	4 + 5	

3	4 3	+	× 5	Don't evaluate + yet.
4	4 3	× +	5	

In other words, we keep operators on the stack until they are ready to be evaluated. Here is the remainder of the computation:

	Number stack	Operator stack		Comments
5	5 4 3	× +	No more input	Evaluate the top.
6	20 3	+		Evaluate top again.
7	23			That is the result.

To see how parentheses are handled, consider the expression 3 × (4 + 5). A (is pushed on the operator stack. The + is pushed as well. When we encounter the), we know that we are ready to evaluate the top until the matching (reappears:

	Number stack Empty	Operator stack Empty	Unprocessed input 3 × (4 + 5)	Comments
1	3		× (4 + 5)	
2	3	×	(4 + 5)	
3	3	(×	4 + 5)	Don't evaluate × yet.
4	4 3	(×	+ 5)	
5	4 3	+ (×	5)	
6	5 4 3	+ (×	)	Evaluate the top.
7	9 3	(×	No more input	Pop (.
8	9 3	×		Evaluate top again.
9	27			That is the result.

Here is the algorithm:

```
If you read a number
   Push it on the number stack.
Else if you read a (
   Push it on the operator stack.
Else if you read an operator op
   While the top of the stack has a higher precedence than op
      Evaluate the top.
   Push op on the operator stack.
Else if you read a )
   While the top of the stack is not a (
      Evaluate the top.
   Pop the (.
Else if there is no more input
   While the operator stack is not empty
      Evaluate the top.
```

At the end, the remaining value on the number stack is the value of the expression.

The algorithm makes use of this helper method that evaluates the topmost operator with the topmost numbers:

```
Evaluate the top:
Pop two numbers off the number stack.
Pop an operator off the operator stack.
Combine the numbers with that operator.
Push the result on the number stack.
```

ONLINE EXAMPLE

The complete code for the expression calculator.

15.6.4 Backtracking

Use a stack to remember choices you haven't yet made so that you can backtrack to them.

Suppose you are inside a maze. You need to find the exit. What should you do when you come to an intersection? You can continue exploring one of the paths, but you will want to remember the other ones. If your chosen path didn't work, you can go back to one of the other choices and try again.

A stack can be used to track positions in a maze.

Of course, as you go along one path, you may reach further intersections, and you need to remember your choice again. Simply use a stack to remember the paths that still need to be tried. The process of returning to a choice point and trying another choice is called *backtracking*. By using a stack, you return to your more recent choices before you explore the earlier ones.

Figure 11 shows an example. We start at a point in the maze, at position (3, 4). There are four possible paths. We push them all on a stack 1. We pop off the topmost one, traveling north from (3, 4). Following this path leads to position (1, 4). We now push two choices on the stack, going west or east 2. Both of them lead to dead ends 3 4.

Now we pop off the path from (3,4) going east. That too is a dead end 5. Next is the path from (3, 4) going south. At (5, 4), it comes to an intersection. Both choices are pushed on the stack 6. They both lead to dead ends 7 8.

Finally, the path from (3, 4) going west leads to an exit 9.

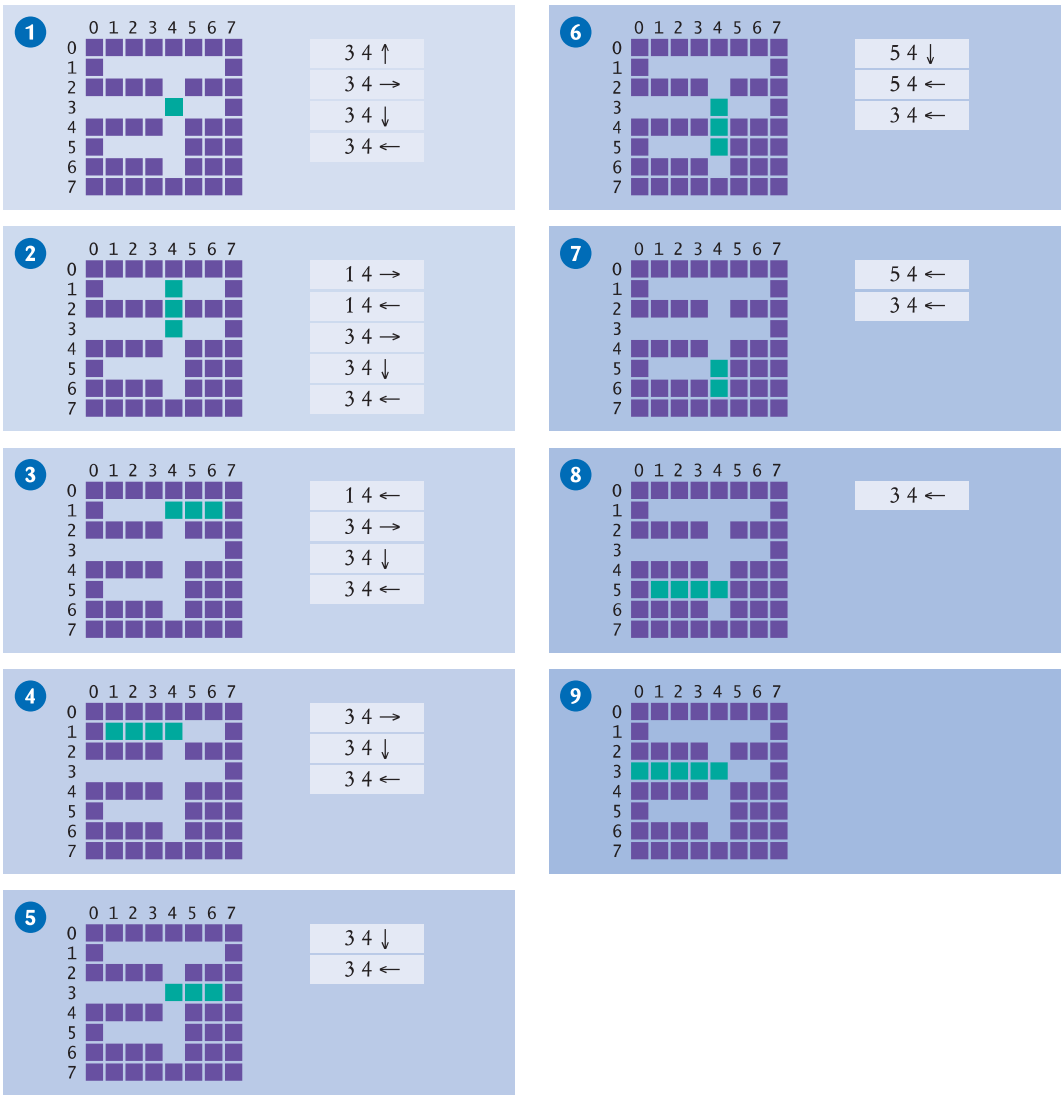

Figure 11 Backtracking Through a Maze

Using a stack, we have found a path out of the maze. Here is the pseudocode for our maze-finding algorithm:

```
Push all paths from the point on which you are standing on a stack.
While the stack is not empty
   Pop a path from the stack.
   Follow the path until you reach an exit, intersection, or dead end.
   If you found an exit
      Congratulations!
   Else if you found an intersection
      Push all paths meeting at the intersection, except the current one, onto the stack.
```

This algorithm will find an exit from the maze, provided that the maze has no *cycles*. If it is possible that you can make a circle and return to a previously visited intersection along a different sequence of paths, then you need to work harder—see Exercise P15.25.