CMPSCI 187: Programming With Data Structures

Lecture #14: Evaluating Postfix Expressions
David Mix Barrington
5 October 2012

Evaluating Postfix Expressions

- Review of Postfix Expressions
- The Stack-Based Evaluation Algorithm
- Error Conditions
- The `PostFixEvaluator` Class
- The `PFixConsole` Class
- Toward an Infix Evaluator

Review of Postfix Expressions

- A **postfix expression** is a sequence of **operators** and **operands**. We'll restrict our attention here to **binary operators**, ones that take two operands. An operator acts on the two latest values before it in the sequence, where a value can be either an operand or the result of an operator.

- In the postfix expression "3 4 5 + 2 * 3 4 * - +" the first + acts on the 5 and the 4 to make 9, the first * acts on the 2 and the result to make 18, the second * acts on the 3 and the 4 to make 12, the - acts on the two *-results to make 18 - 12 = 6, and the second + acts on the first 3 and the result of the - to make 9. We care about the order that the - takes its arguments -- it subtracts the later one from the earlier one.

- We saw in Discussion #4 that *valid* postfix expressions always have k operators and k+1 operands for some number k.

The Stack-Based Evaluation Algorithm

- We can evaluate postfix expressions easily using a stack whose entries each hold one of our numerical values (an operand or the result of an operator).
- The basic idea is that (1) when we see an operand, we push it onto the stack, and (2) when we see an operator, we pop two values off of the stack (remembering them!), apply the operator to them, and push the result back onto the stack. At the end, if we have exactly one value on the stack, that is our answer.
- If we ever empty the stack, or if we finish with more than one value on it, the postfix expression was not valid. Thus our algorithm also tests a candidate postfix expression for validity.
- In the example “3 4 5 + 2 * 3 4 * - +” we push 3, push 4, push 5, pop 4 and 5 to make 9, push 2, pop 9 and 2 to make 18, push 3, push 4, pop 3 and 4 to make 12, pop 18 and 12 to make 6, and finally pop 3 and 6 to make 9.

Error Conditions

- Just like our parenthesized expressions, postfix expressions can go wrong in two ways, by having too many operators too quickly or by having not enough operators when the expression ends. In the first case the stack will be emptied, and in the second case more than one value will be left on the stack at the end. DJW use a bounded stack, which adds the additional risk of stack overflow.

- We'll define a new class of exceptions called `PostFixException` to represent both of these conditions, counting on our error messages to tell the user what went wrong.

- We can choose to guard against any stack exceptions, or to enclose them in try blocks and catch them. DJW choose the latter. We could also generate an ArithmeticException by dividing by 0, but that isn't our business.

(Most of) the `PostFixEvaluator` Class

```
public class PostFixEvaluator {
  • public static int evaluate(String expression) {
      BSI<Integer> stack = new ArrayStack<Integer>(50);
      int value, operand1, operand2, result = 0; String op;
      Scanner tokenizer = new Scanner(expression);
      while (tokenizer.hasNext( )) {
         if (tokenizer.hasNextInt( )) {
            value = tokenizer.nextInt( );
            if (stack.isFull( )) throw new PFE("stack overflow");
            stack.push(value);}
         else {op = tokenizer.next( );
            if (stack.isEmpty( )) throw new PFE("stack underflow");
            operand2 = stack.top( ); stack.pop( );
            if (stack.isEmpty( )) throw new PFE("stack underflow");
            operand1 = stack.top( ); stack.pop( );
            // result = result of op on operand1 and operand2
            stack.push(result);}
      if (stack.isEmpty( )) throw new PFE ("stack underflow");
      result = stack.top( ); stack.pop( );
      if (!stack.isEmpty( )) thrown new PFE ("too many operands");
      return result;}}
```

The `PFixConsole` Class

- This class' main method takes and evaluates expressions from the console as long as the user wants to give more. We catch `PostFixExceptions` and handle them with messages without crashing our main method.

```
public class PFixConsole {
   public static void main (String [ ] args) {
      Scanner conIn = new Scanner(System.in);
      String line = null, more = null; int result;
      do {System.out.println("Enter a PFE to be evaluated: ");
         line = conIn.nextLine( );
         try{
            result = PostFixEvaluator.evaluate(line);
            System.out.println("Result = " + result);
         catch(PostFixException error) {
            System.out.println
               ("Error in PFE: " + error.getMessage());}
         System.out.println("Evaluate another? (Y = yes): ");
         more = conIn.nextLine( );}
      while (more.equalsIgnoreCase("y");
      System.out.println ("Goodbye.");}}
```

Toward an Infix Evaluator

- Building an evaluator for ordinary infix expressions would require us to deal with both parentheses and the hierarchy of operations. The usual method is to actually translate the infix expression into the correct postfix expression, and use a stack-based evaluator to handle the result.

- This translation (see, for example, http://scriptasylum.com/tutorials/infix_postfix/algorithms/infix-postfix), uses a second stack to hold operators that are not yet ready to appear in the postfix translation. (Operands may be output as soon as they are seen.) Any operator must be held until its second argument has been output, but some must be held longer due to precedence.

- If we see an operator with lower precedence than the one on the top of the stack, we output it. Otherwise we push it. We empty the stack to the output after we have read all the characters.