CMPSCI 187: Programming With Data Structures

Lecture #9: The Evaluator Program
26 September 2011

The Evaluator Program

- The Problem: Postfix Evaluation
- The Method: Pending Arguments on a Stack
- The Structure of L&C's Solution
- The Driver Class `Postfix`
- The Class `PostfixEvaluator`
- Complications: I/O, `StringTokenizer`, Wrapper classes, Exceptions

The Problem: Postfix Evaluation

- We have arithmetic expressions where the constants are `int` values and the operators are `+`, `-`, `*`, and `/`.
- We are more used to **infix** expressions like `(b*b)-(4*a*c)`.
- It turns out that **postfix** expressions like `b b * 4 a * c * -` are easier to evaluate with a stack -- they have no ambiguities to be resolved by operator hierarchy or parentheses.
- To translate infix to postfix, find the highest-priority operator, so that the infix expression can be read as R op S, then write “R S op”, then translate both R and S to postfix.
- We want to input the postfix expression as a string and output an int.

The Method: Pending Arguments on a Stack

- We create a `Stack<Integer>` object to store arguments still to be used.
- If our first input is “4”, we cannot process it yet so we push it.
- If our second input is “7”, we still don’t know what to do so we push it.
- If our third input is “*”, we know this applies to numbers on the stack, so we pop the top two numbers, multiply them to get 28, and push “28” back on.
- If the third argument were “2” we would have to push it as well.
- The rule: push any number inputs, when you get an operator pop the top two elements from the stack, apply the operator, and push the result.

More on the Evaluation Method

- When there are no more inputs, there ought to be exactly one number on the stack, so we should pop it and report it as the result as long as the stack is now empty.
- If there is another number below the top one, the expression was invalid.
- If our method ever pops from an empty stack, the expression was invalid.
- For `+` and `*`, the order of the two arguments to the evaluator doesn't matter, but for `-` and `/` it does because these two are not commutative. The argument that came first in the input, which is deeper in the stack, is the first argument to the operator.

The Structure of L&C's Solution

- L&C's postfix evaluator program is printed on pages 46-49.
- It has two classes, `Postfix` and `PostfixEvaluator`.
- It uses two classes from the library (from `java.util`) called `Scanner` and `StringTokenizer`.
- It also uses the `ArrayStack` class from later in the chapter, in their own package `jss2`, but their evaluator would behave the same if this were replaced by any class that implements their `StackADT` interface, such as the `Stack` class in `java.util`.
- `Postfix` interprets the input and creates a `PostfixEvaluator` object.

The Driver Class `Postfix`

- We import `java.util.scanner` and run a main method. L&C's code has a typo in the `catch` statement.

```
public static void main (String [ ]args) {
   String expression, again;
   try{Scanner in = new Scanner(System.in);
       do{
          PostfixEvaluator evaluator = newPostfixEvaluator;
          System.out.println("Enter an expression: ");
          expression = in.nextLine();
          int result = evaluator.evaluate(expression);
          System.out.println("That equals: " + result);
          System.out.print("Try again (Y/N)? ");
          again = in.nextLine();
          }while (again.equalsIgnoreCase("y"));
      }catch (IOException e)
       {System.out.println("Input exception reported");}}
```

The Class `PostfixEvaluator`

- This calls two helper methods that are pretty obvious.

```
private final char ADD = '+', SUBTRACT = '-';
private final char MULTIPLY = '*', DIVIDE = '/';
private ArrayStack<Integer> stack;
public PostfixEvaluator() {stack = new ArrayStack<Integer>();}
public int evaluate (String expr) {
   int op1, op2, result = 0;
   String token;
   StringTokenizer tokenizer = new StringTokenizer(expr);
   while (tokenizer.hasMoreTokens()) {
      token = tokenizer.nextToken();
      if (isOperator(token)){
         op2 = (stack.pop()).intValue();
         op1 = (stack.pop()).intValue();
         result = evalSingleOp (token.charAt(0), op1, op2);
         stack.push (new Integer(result));}
      else stack.push (new Integer(Integer.parseInt(token)));}
   return result;}
```

Complications: I/O, Wrappers, Exceptions

- Any time we want input or output to our programs, we need to deal with the object System. Java makes this about as easy as it can, but there are always complications. User input is unpredictable, and we always have to consider the chance of System breaking down. Thus Java requires us to check exceptions whenever we use methods from the standard I/O classes, pretty much except for System.out.print and its relatives.

- We needed the wrapper class Integer in this program in order to put int values into a generic stack object -- we couldn't have a Stack<int>. This leads to lots of "new Integer(x)" and "x.intValue( )" statements to go from int to Integer or vice versa. Java 5.0 allows free **boxing** and **unboxing** to avoid these.

- If we cared more about exception handling here, we'd do better than the general I/O message -- we'd deal with extra operators or extra input numbers. Note that L&C's code accepts several kinds of *invalid* input without throwing an exception, and an extra operator gets an `EmptyCollectionException`.