Using the Stack ADT

April 5, 2016

CMPU 102 Assignment 5:

Due: Thursday, April 7th, by midnight

In this assignment, you will write programs that use stacks, programs also known as stack clients, to solve two problems. Note: It is your responsibility to make sure your program meets all the requirements of the problem statement as it is posted on the due date, so read over the requirements carefully before you submit your final version. You can resubmit as many times as you want.

Problem 1: Matching nested bracketing symbols

Write a stack-client program called PARENTHESES that reads in a text stream from standard input and uses a stack to determine whether the parentheses, brackets, and braces are properly balanced. Ignore all other characters.

For example, your program should print "All parenthesis, braces and brackets match!" if it is true that the line is balanced and matched, as shown below.

\[
[(()){}[(()())()]]
\]

and

\[
\{ s = 2 * (a[2] + 3); x = (1 + (2)); \}
\]

If you go through the strings above carefully, you will discover that all the bracketing characters are correctly nested, with each open parenthesis matched by a close parenthesis, each open bracket matched by a close bracket, and so on.

The following strings are all unbalanced for the reasons indicated:

- \(([])\) The line is missing a close parenthesis.
- \((())\) The close parenthesis comes before the open parenthesis.
- \({}())\) The bracketing characters are improperly nested.

Your program can ignore all other non-bracket type characters in the input. It should output true with an appropriate message if the bracketing symbols are properly matched and nested and false with an appropriate message for those in which the bracketing symbols are not properly matched or nested.

In the main method, your program should tell the user what the program does and then give the user a prompt to enter a line. Single lines can be input by the user, followed by your indication of whether or not the line has properly nested bracket characters. Keep prompting for input until the user chooses to stop.

Design your program so that you read the line of text in one method, using helper methods to manage the data structure you use to implement the program. Write the algorithm you use at the top of the class containing the main method in comments. You will probably want to write your own StackEmptyException class.

You can use any class you want to implement the stack. This class should be separate from the PARENTHESES program but in the same package or in the Java API. You can also choose any input and output classes to read and write data in the program.
Problem 2: Arithmetic Expression Evaluation

Consider the computation of arithmetic expressions like this one:

\[
( 1 + ( ( 2 + 3 ) * ( 4 * 5 ) ) )
\]

Write a Java program `EVALUATE` that can read a string as input (the expression) and produce the number represented by the expression as output. An arithmetic expression, for the purposes of this problem is either a number, or a left parenthesis followed by an arithmetic expression followed by an operator followed by another arithmetic expression, followed by a right parenthesis. For simplicity, this definition is for fully parenthesized arithmetic expressions, which specify precisely which operations apply to which operands. Your program should support the binary operations `+`, `−`, `∗`, `/`, and the unary operator “sqrt”. For the input string, assume all numbers will be single digit integers. This makes the problem easier because you don’t have to read decimal points. During the execution of the program, the operand stack will need to hold decimal numbers, but that should not be as difficult as trying to read multi-digit decimal numbers from the start.

An algorithm developed in the 1960’s uses two stacks (one for operands and one for operators) to solve this problem. An expression consists of parentheses, operators, and operands (numbers). Proceeding from left to right across the input expression, we manipulate the stacks according to four possible cases, as follows:

- Push operands onto the operand stack (a stack of doubles).
- Push operators onto the operator stack (a stack of Strings).
- Ignore left parentheses.
- On encountering a right parenthesis, pop an operator, pop the required number of operands, and push onto the operand stack the result of applying that operator to those operands.

After the final right parenthesis has been processed, there should be only one value on the operand stack, which is the value of the expression. This code is a simple example of an interpreter: a program that interprets the computation specified by a given string and performs the computation to arrive at the result. Example executions of this algorithm are shown below:

```
> java Evaluate
Please enter a fully-parenthesized arithmetic expression using only operators +, −, ∗, /, and "sqrt" and only numeric operands.
( 1 + ( ( 2 + 3 ) * ( 4 * 5 ) ) )
101.0

> java Evaluate
Please enter a fully-parenthesized arithmetic expression using only operators +, −, ∗, /, and "sqrt" and only numeric operands.
( 1 + sqrt ( 5 ) ) / 2
1.618033988749895

> java Evaluate
Please enter a fully-parenthesized arithmetic expression using only operators +, −, ∗, /, and "sqrt" and only numeric operands.
( 7 + 35 )
42.0

> java Evaluate
Please enter a fully-parenthesized arithmetic expression using only operators +, −, ∗, /, and "sqrt" and only numeric operands.
42
42.0
```

If the entire input String is processed and the operator stack is not empty, throw a “StackEmptyException”.

Submit the zipped folder “cs102-assn5-yourlastname” containing the solutions to problems 1 and 2. Make sure you follow the naming convention given above for your solution or you may not receive full credit.