CS 445: Data Structures
Fall 2016

Assignment 2

Assigned: Monday, September 26 **Due:** Monday, October 10 11:59 PM

---

1 Description

In lecture, we discussed an stack algorithm for converting an infix expression to a postfix expression, and another for evaluating postfix expressions. We briefly described how one might pipeline these algorithms to evaluate an infix expression in a single pass using two stacks. In this assignment, you will be implementing this combined algorithm. Your completed program will evaluate an infix arithmetic expression using two stacks.

You are provided with the skeleton of `InfixExpressionEvaluator`. This class accepts input from an `InputStream` and parses it into tokens. When it detects an invalid token, it throws an `ExpressionError` to end execution. To facilitate ease of use, this class also contains a `main` method. This method instantiates an object of type `InfixExpressionEvaluator` to read from `System.in`, then evaluates whatever expression is typed.

`InfixExpressionEvaluator` uses composition to store the operator and operand stacks, and calls several private helper methods to manipulate these stacks when processing various tokens. You will need to complete these helper methods and add error checking to ensure the expression is valid.

2 Tasks

First, carefully read the code provided at `https://cs.pitt.edu/~bill/445/a/a2code.zip`, including comments.

2.1 Implement helper methods, 70 points

As tokens are processed, helper methods are called to process them. In the included code, these methods do not do anything. You will need to implement the following methods to process the various types of tokens.

Method	Points
`processOperand(double)`	10
`processOperator(char)`	18
`processOpenBracket(char)`	12
`processCloseBracket(char)`	18
`processRemainingOperators()`	12

The `processOperand` method accepts a parameter representing an operand. As operands are encountered, they should be processed according to the infix-to-postfix and postfix-eval algorithms.

The `processOperator` method accepts a parameter representing an operator. Each of the following operators must be supported. Follow standard operator precedence. Assume that - is always the binary subtraction operator.

`+`	Addition
`-`	Subtraction
`*`	Multiplication
`/`	Division
`^`	Exponentiation

The `processOpenBracket` and `processCloseBracket` methods each accept a parameter representing a bracket. You must support round brackets `()` and square brackets `[]`. These brackets can be used interchangeably, but must be nested properly—a `(` cannot be closed with a `]`—in a valid expression.

Finally, `processRemainingOperators()` should process any leftover operators that have not yet been evaluated.

2.2 Error checking, 30 points

This task requires that you modify your program to account for errors in the input expression. The provided code throws `ExpressionError` when encountering an unknown token. You should modify your program to throw this exception (with an appropriate message) whenever the expression is invalid.

This requires careful consideration of all the possible syntax errors that could occur. What if several operands are given in a row? Several operators? What if an open bracket is followed by an operator? What if the brackets do not nest properly? Each of these syntax errors should be handled with an appropriate `ExpressionError`.

2.3 Extra credit

In addition to the above tasks, **for up to 10 bonus points**, you may add additional features to your program. If you do so, include a file named `README.txt` describing these extra features.

For instance, you may consider errors beside those of syntax. Can you detect and report divide-by-zero? What about other errors in value? Can you support both the binary subtraction operator and the unary negation operator using the same symbol? Can you add support for more complex operators?

This task encourages you to brainstorm outside of the assignment brief. Try to add unique and interesting features to your evaluator.

3 Submission

Create a zip file containing java files (no class files!). The TA should be able to unzip your submission, then compile and run your program without any additional changes. Be sure

to test this procedure (unzip, compile, run) before submitting.

All programs will be tested on the command line. If you use an IDE to develop your program, you must export the java files from the IDE and test that they compile and run on the command line. Do not submit the IDE's project files.

In addition to your code, you may wish to include a `README.txt` file that describes features of your program that are not working as expected to assist the TA in grading the portions that work as expected. This file should also specify any **extra credit** tasks that you completed.

Submit your zip file according to the instructions at `https://cs.pitt.edu/~bill/445/#submission`

Your project is due at 11:59 PM on Monday, October 10. Be sure to test the submission procedure in advance of this deadline: **No late assignments will be accepted.**