# CS/COE 0445: Data Structures (Fall 2019)

Department of Computer Science, University of Pittsburgh

**Assignment #3: Infix and Postfix expression conversion and evaluation**

Released: Oct 2, 2019 **Due:** 8:00 PM, Monday, Oct 15, 2019

## Goal

Create your algorithm for evaluating infix expressions. Gain familiarity with the stack ADT.

## Introduction

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, without generating the full postfix expression in between. 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. It also catches any potential `ExpressionError`s and prints the reason for the error.

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

## Problem Description

First, carefully read the provided code.

## Implement helper methods, 70 points

As tokens are parsed, helper methods are called to handle them. In the included code, these helper methods do not do anything. You must implement the following methods to handle the various types of tokens.

- `handleOperand(double)`: Each time the evaluator encounters an operand, it passes it (as a double) to this method, which should handle it by manipulating the operand and/or operator stack according to the infix-to-postfix and postfix-evaluation algorithms.
- `handleOperator(char)`: Each time the evaluator encounters an operator, it passes it (as a char) to this method, which should handle it by manipulating the operand and/or operator stack according to the infix-to-postfix and postfix-evaluation algorithms.

  Each of the following operators must be supported. Follow standard operator precedence. You can assume that − is always the binary subtraction operator (e.g., no negative operands).

| | |
|---|---|
| + | Addition |
| − | Subtraction |
| * | Multiplication |
| / | Division |
| ^ | Exponentiation |

- `handleOpenBracket(char)`: Each time the evaluator encounters an open bracket, it passes it (as a char) to this method, which should handle it by manipulating the operand and/or operator stack according to the infix-to-postfix and postfix-evaluation algorithms.

  You *must* support both round brackets `()` and angle brackets `<>`. These brackets can be used interchangeably, but must be nested properly—a `(` cannot be closed with a `>`, and vice-versa.
- `handleCloseBracket(char)`: Each time the evaluator encounters a close bracket, it passes it (as a char) to this method, which should handle it by manipulating the operand and/or operator stack according to the infix-to-postfix and postfix-evaluation algorithms.
- `handleRemainingOperators()`: When the evaluator encounters the end of the expression, it calls this method to handle the remaining operators on the operator stack.

**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 (for instance, `&`). You must 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. What if an operand follows another operand? An operator following an open bracket? What about brackets that do not nest properly? All such syntax errors must be handled using `ExpressionError`.

**Hints:**

1. Which pairs of adjacent token types are valid vs. invalid? Create a data member that tracks the most recent token type. When handling a new token, first ensure that it is legal for the current token type to follow the previous token type.
2. Trace through the algorithm by hand using expressions with various bracket errors. When during the process can each error be detected?

**Grading**

The following points breakdown will be used to assign grades:

- **70 points** are allocated for correctly evaluating **valid** expressions. These expressions will range from very simple to very complex, but none will contain syntax errors. You should test your program extensively to ensure that it works properly even for complex expressions with multiple nested .
- **30 points** are allocated for correctly identifying the syntax errors in **invalid** expressions. These expressions will contain a wide variety of syntax errors. You should test your program with many invalid expressions to ensure that it correctly detects all possible syntax errors.

**Note:** Code that cannot be compiled and executed will be given a grade of 0.

**Extra credit**

In addition to the above tasks, **for up to 5 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? Hexadecimal operands of the form `0x7c2`?

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

## Academic Honesty

The work in this assignment is to be done *independently*. Discussions with other students on the assignment should be limited to understanding the statement of the problem. Cheating in any way, including giving your work to someone else will result in an **F** for the course and a report to the appropriate University authority.

## What to submit

Upload all your java files (preferably is a .zip) using the website. If you overwrite the provided interfaces, remember to restore them to their original versions. All programs will be tested on the command line, so if you use an IDE to develop your program, you must export the java files from the IDE and ensure that they compile and run on the command line. Do not submit the IDE's project files. Your TA should be able to compile and run your code as discussed in Lab 1. For instance, javac edu/pitt/cs/as3/InfixExpressionEvaluator.java and java edu.pitt.cs.as3.InfixExpressionEvaluator must compile and run InfixExpressionEvaluator.

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 do work as expected.

## How to submit your assignment

We will use a Web-based assignment submission interface. To submit your assignment:

- Go to the class web page `http://db.cs.pitt.edu/courses/cs0445/current.term/` and click the `Submit` button.
- Use your pittID/username as the username and your PeopleSoft ID as the password for authentication. There is a reminder service via email if you forgot your PeopleSoft ID.
- Upload your assignment file(s) to the appropriate assignment (from the drop-down list).
- Check (through the web interface) to verify what is the file size that has been uploaded and make sure it has been submitted in full. **It is your responsibility to make sure the assignment was properly submitted.**

You must submit your assignment before the due date (**8:00 PM, Tuesday, Oct 15, 2019**). There are no late submissions.