# CSC521 Programming Project 3

# A Stack-based Calculator With Postfix Expression Using Java

## Purpose

- To make a simple integer calculator for 8-bit signed numbers (i.e. accept 12*4*(5-3) = and produce an answer).
- To see how the stack can be used to support arithmetic operations.
- Note: Groups of at most three students.

## Program

1. The calculator accepts an arithmetic expression in *infix notation* from user input, parses it, and prints an answer for the expression. Possible operators are *,/,+,- and parentheses are used to specify the order of operations. The priority of * and / are higher than that of + and - operators. That is, the answer for 3+4*5= is 23 and not 35. Among operators of equal priority, such as + and - or * and /, we assume association is from the left, so that 4*a*c means (4*a)*c, 12 – 4 – 3 = 5, not 11.

2. *Postfix expressions* can be used to specify arithmetic expressions using a parenthesis-free notation. An infix expression from user input is translated to a postfix expression first. Translating infix expressions into their corresponding postfix expressions is a part of this project. Below are some examples.

| **Infix** | **Postfix** |
|---|---|
| (a + b) | ab+ |
| (x-y-z) | xy-z- |
| ((x-y-z)/u+v) | xy-z-u/v+ |
| ((x-y-z)/u+v*w) | xy-z-u/vw*+ |
| ((a-b-c)/(d+e)) | ab-c-de+/ |
| a*b+c*(d-e) | ab*cde-*+ |
| (-a + b) | ^ab+ |
| (x-(-y)-z) | x^y-z- |
| a | a |
| -b | ^b |

Treat the symbol "^" as negative sign as oppose to the subtraction operation, and available only in the postfix notation. Moreover, treat it as part of a number rather than an operation.

3. Your task is to use stacks to evaluate postfix expressions. To evaluate a postfix expression, P, you scan P from left-to-right. When you encounter an operand, X, you push it onto an evaluation stack, S. Repeat this when you have more than one operand. When you encounter an operator, ß, while scanning P, you pop the topmost operand stacked on S into D2 (which denotes the right operand), then you pop another topmost operand stacked on S into D3 (which denotes the left operand). Finally, you perform the operation ß on D2 and D3, getting the value of the expression (D3 ß D2), and you push the value back onto the stack S. When you are finished scanning P, the value of P is the only item remaining on the stack S.

4. Assume that infix expression input is always correct; that is, no syntax error checking is necessary.
5. For the addition and subtraction, check overflow.

6. For the division, roundup the remainder. Remember this is an 8-bit dividend and 8-bit divisor.

7. A few assumptions concerning valid input and the handling of certain input:

   For the unary minus, you assume that the unary minus will either be the first character of the expression or will immediately follow a left parentheses. Under this assumption expressions such as 1--2 and 1+-2 are not allowed. However, you allow the unary minus to be used outside of any parenthetical enclosure such as -(1+2) or -(-(-(1))). Another assumption, make the empty parentheses not allowed, because one may not sure what value they would evaluate to.

## Checklist

- Well-commented code
- Your program is valid for all possible operands and operations. (Operands can be integers between -128 and +127).

## Test

- After executing your program, it should ask for the use input expression. It first prints out its postfix notation, and then print the final evaluated value. Again, we assume the user input is always correct.
- Please submit a zip file titled "CSC521_StudentLastName1_LastName2_LastName3_proj3.zip", which includes a readme specifying how to execute your program, the source Java programs, and 5-8 screen shots showing if your program executes correctly.

## The cases of Overflow

1. For the addition and subtraction, please use two's complement, referred as below link

https://www.doc.ic.ac.uk/~eedwards/compsys/arithmetic/index.html

2. For multiplication and division, two's complement is a bit more complex. One easiest way is to simply find the magnitude of the two multiplicands (or dividend and divisor), multiply (or divide) them, and then use the original sign bits to determine the sign of the result. For example, for multiplication, if the multiplicands had the same sign (both positive or both negative), the result must be positive; if the they had different signs, the result is negative. Multiplication by zero is a special case, which needs to be dealt with.

3. To make it simpler, when the result of the user-input expression causes overflow, your program just indicates

a. overflow occurs, and

b. the result of your evaluation.

For example, if the user input is 80+80, your program should print out a message "overflow occurs!", with the output value -96.

4. Among others, a few examples I will test are:

a. 40+50
b. 45+(-50)
c. 80+80 (overflow)
d. -36+107
e. -50-122 (overflow)
f. -33*3
g. 101*61 (overflow)
h. -101*61 (overflow)
i. -70/3
j. -120/(-34)

Above examples only consider one operator in an expression. Your program should certainly consider the case of multiple operators with parentheses, convert to postfix, and evaluate using stack.

5. To help me to test your cases, please indicate in the readme file if any of above or extra examples have been successfully tested, and give a set of your own screenshots.