CSC172 PROJECT 2 – Infix Calculator

SUMMARY:

This project will require you to create a Java program that will take an input file consisting of several lines of infix notation mathematical calculations, convert them to postfix notation (using the first algorithm below), and evaluate them (using the second algorithm below). The results of the calculations will then be printed to an output file. This project will build on your linked list implementations from last week's labs, using them to create your own implementations of stacks and queues for each of the algorithms.

PROGRAM:

Input file:

The input file will contain several lines of equations in infix format. Each line can use several of the available mathematical or logical operators: addition [+], subtraction [-], multiplication [*], division [/], parentheses [()], less than [<], greater than [>], equal to [=], logical AND [&], logical OR [|], and logical NOT [!]. When using the mathematical and logical operators together, let "false" be represented by 0 and "true" be represented by 1. For example, an input file could consist of the following lines:

4.7 + 3.0 * 5.5

(4 > 3) + (3 = 4) + 2

Calculator program:

For each line of input, your calculator program will begin by converting from infix notation to postfix notation. Infix notation, shown in the example input lines above, is the standard representation of mathematical equations. Postfix, or Reverse Polish notation, is a mathematical notation where the mathematical operator comes after both its operands. The two input lines in postfix notation would be:

4.7 3.0 5.5 * +

4 3 > 3 4 = + 2 +

The main parts of the calculator program are:

Converting an input line into a postfix expression:

This step will use both a stack and queue in order to convert expressions from infix to postfix notation. The stack and queue will be implemented by you, using your linked list implementation from labs (do NOT use the predefined stack/queue classes available in Java). Each value will be read from the input line, and dealt with in the following manner:

1. If the value is an operand, add it into the queue immediately.
2. If the value is a close-parenthesis [')'], pop all the stack elements and add them to the queue one by one until an open-parenthesis ['('] is found.
3. If the value is an operator, pop everything on the stack and add them to the queue one by one until you reach either an operator of lower precedence, or a right-associative operator of equal precedence (eg, the logical NOT is a right-associative operator). Add the found operator to the queue, and push the original operator onto the stack.
4. At the end of the input, pop everything that remains on the stack and add to the queue one by one.

When finished converting one statement into a queue in postfix notation, pass the queue to the next step – the postfix expression evaluator.

Evaluating the postfix expression

This step will use the queue that was the result of the infix to postfix conversion, and a stack. Again, the stack and queue will be implemented by you, using your linked list implementation

from labs (do NOT use the predefined stack/queue classes available in Java). The algorithm proceeds as follows:

1. Get the element at the front of the queue.
2. If the value is an operand, push it onto the stack.
3. If the value is an operator, pop the appropriate number of operands from the stack (eg. 2 operands for multiplication, 1 for logical NOT). Perform the operation, and push the resulting value onto the stack.

Repeat steps 1-3 until the queue is empty. When it is, there should be a single value in the stack – that value is the result of the calculation.

Output file:

For each calculation, print the result of the calculation to an output file, with one numerical result per line.

GRADING:

Report: 20%

Write up a report explaining your code, and giving the results that your program produces for the test file provided for the project.

Code Style: 10%

Make sure to add comments and implement everything as specified above.

Functionality: 70%

The program must work for all the base operations specified above – this will comprise of approximately 55% of your grade, 5% for each base operation. The remaining 15% will be spread out between dealing with the input/output files, your stack/queue implementations, and any other parts of the coded implementation.