Project #1 (The URCalculator)

CSC 172 (Data Structures and Algorithms) Fall 2023,
University of Rochester
Due Date: Sunday October 15th 2022 end of day

Objectives

- Practice using `Stacks` and `Queues` based on `Lists`.
- (You must implement your own `Lists`, `Stacks` and `Queues` classes. You are not allowed to use the `Java` library classes for those functionalities.)
- Learn how expression parsing and evaluation (Shunting-yard algorithm) works using `Stacks` and `Queues`. (`https://en.wikipedia.org/wiki/Shunting_yard_algorithm`). This is the beginning of understanding how computers process formal languages.

The URCalculator

For this project, you need to implement a simple calculator application. The calculator should be able to perform all of the basic operations like

- Addition (+)
- Subtraction (-)
- Multiplication (*)
- Division (/)
- Exponentiation (^)
- Equal to (=)
- Less than (<)
- Greater than (>)
- Logical AND (&)
- Logical OR (|)
- Logical NOT (!)
- Parentheses (())

For extra credit,you may add additional operations like

- Modulo (%)
- Sin (`sin`)
- Cin (`cos`)
- Tin (`tan`)

What To Do

For this project, you will need to write a Java program that reads in a series of infix expressions from a plain text file, converts the expressions to postfix notation, evaluates the postfix expressions, and saves the resulting answers to a new text file. You may assume that the input file will contain valid infix expressions that can be safely evaluated to numbers (e.g. they won't include division by zero). Extra credit will be awarded if your program can safely handle invalid input by printing a relevant error message in the output file.

The file "`infix_expr_short.txt`" provided with this prompt contains a subset of the expressions the TAs will use when grading your program. The output of your program should exactly match "`postfix_eval_short.txt`" in order to receive full credit. You can check if two files are equivalent by using the "`diff`" command on OS X or Linux, or "`FC`" on Windows. It is strongly recommended that you write your own (additional) test cases and submit them with your source code to demonstrate your program's capabilities.

The locations of the input and output files will be supplied to your program via the command line. The first command line argument will be the location of the input file (containing infix expressions), and the second argument will be the location where your postfix evaluations should be stored. For example, if your main method were in a class called InfixCalculator, your program should be run as:

```
java URCalculator infix_expr_short.txt my_eval.txt
```

You can create as many classes as required. But you need to name the main class as `URCalculator.java`

InFix_2_PostFix

The shunting-yard algorithm works by considering each "token" (operand or operator) from an infix expression and taking the appropriate action:

1. If the token is an operand, enqueue it.
2. If the token is a close-parenthesis [')'], pop all the stack elements and enqueue them one by one until an open-parenthesis ['('] is found.
3. If the token is an operator, pop every token on the stack and enqueue them one by one until you reach either an operator of lower precedence, or a right-associative operator of equal precedence (e.g. the logical NOT and the exponential are right-associative operators). Enqueue the last operator found, and push the original operator onto the stack.
4. At the end of the input, pop every token that remains on the stack and add them to the queue one by one.

The queue now holds the converted postfix expression and can be passed onto the postfix calculator for evaluation.

PostFix Evaluation

With your postfix expression stored in the queue, the next step is to evaluate it. The algorithm for evaluating a postfix expression proceeds as follows:

1. Get the token at the front of the queue.
2. If the token is an operand, push it onto the stack.

3. If the token is an operator, pop the appropriate number of operands from the stack (e.g. 2 operands for multiplication, 1 for logical NOT). Perform the operation on the popped operands, 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 expression.

Hand In

Hand in the source code from this lab at the appropriate location on the Blackboard system at my.rochester.edu. You should hand in a single compressed/archived (i.e. "zipped" file that contains the following.)

1. A plain text file named README that includes your contact information, a detailed synopsis of how your code works and any notable obstacles you overcame, and a list of all files included in the submission. If you went above and beyond in your implementation and feel that you deserve extra credit for a feature in your program, be sure to explicitly state what that feature is and why it deserves extra credit.

2. Source code files (you may decide how many you need) representing the work accomplished in this project. All source code files should contain author identification in the comments at the top of the file.

3. A plain text file named OUTPUT that includes author information at the beginning and shows the compile and run steps of your code. The best way to generate this file is to cut and paste from the command line.

4. Any additional files containing extra test cases and your program's corresponding output. Theses cases should be described in your README so the grader knows what you were testing and what the expected results were.

Grading

This is an individual project. Total points: 100

- 70% Functionality
  - 10% driver program that handles File I/O
  - 30% infix to postfix conversion according to the shunting-yard algorithm
  - 30% postfix evaluation
- 20% Testing
  - 15% program passes the short tests
  - 5% program passes the extended tests
- 10% README

Extra Credit

- Maximum Total Extra Credit on all items cannot exceed 30%
- up to 10% For supporting modulo, unary minus, sine, cosine, and/or tangent operators (2% each)
- 10% For gracefully handling invalid expressions and/or expressions that cannot be evaluated
- 10% ForGUI implementation

- 3% for not having to require spaces between tokens.

Note:

- **You must implement your own `Lists`, `Stacks` and `Queues` classes. You are not allowed to use the `Java` library classes for those functionalities.**
- No group projects. You may discuss ideas with your classmates, but do not exchange files containing code.
- Your README must mention any of your classmates that you collaborated with.
- If you use any source from the Internet you must provide a reference to it in your comments and in your README.
- No GUI for this project is required. You may add additional features to your calculator (for example, trigonometric functions, exponential, power of a number, log, etc) for extra credit. You may implement a GUI for extra credit. **But, you must have a system that runs from the command line with filie I/O.**
- You can create as many classes as required. But you must name the main class as `URCalculator.java`
- Expressions will contain spaces. Extra credit for not needing spaces.
- You can assume the datatype for all the numeric values is **double**.
- It is ok to use a special characterlike tilde "~" for unary minus to distinguish it from binary subtraction.