Infix to Postfix Expression Assignment

Overview

For this project you will write a program to convert an arithmetic expression written in infix notation to the equivalent arithmetic expression in postfix notation. The arithmetic expressions given as input to your program contain operands which are positive integers or variables. Variables contain letters and digits and must begin with a letter. The arithmetic expressions given as input to your program contain the operators: +, -, *, /. The operators have the usual precedence. The operators + and – have the same precedence. The operators * and / have the same precedence. The operators * and / have a higher precedence than the operators + and -. The arithmetic expression can contain parenthesis, (and). Parenthesis have a higher precedence than the 4 operators.

A simple infix expression: a+5
Written as a postfix expression: a 5 +

A more complex infix expression: a + 156 * b
Written as a postfix expression: a 156 b * +

Another infix expression: a * 5 + b
Written as a postfix expression: a 5 * b +

Another infix expression: at + bd * c – d
Written as a postfix expression: at bd c * + d –

Another infix expression: a – (b + c) * d
Written as a postfix expression: a b c + d * –

Algorithm

You'll use a stack to process the operators and parenthesis in the arithmetic expression. Treat the arithmetic expression as a linear sequence of tokens. A token is an operand (positive integer or variable), an operator, or a left or right parenthesis.

Create an empty stack. From left to right, read and process each token in the arithmetic expression one at a time. What is done with the token depends on the type of token.

Operand token: print the operand out right away.

Operator token:

1. If the stack is not empty and the operator at the top of the stack has a higher or equal precedence than the operator token just read, then pop the operator at the top of the stack and print the popped operator.
2. Repeat #1 until the stack is empty or the operator at the top of the stack has a lower precedence than the operator token just read, then push the operator token just read on top of the stack.

Note: Even though left parenthesis has a higher precedence than the other operators, treat it as if it has the lowest precedence than all the other operators.

Left Parenthesis token: Push the left parenthesis on top of the stack.

Right parenthesis token: Pop the operator at the top of the stack and print the popped operator until the top of the stack contains a left parenthesis. Pop the left parenthesis at the top of the stack and discard it.

After the last token in the arithmetic expression is read and processed, pop the operator at the top of the stack and print the popped operator until the stack is empty.

Design

1. Download the files Stack.java and DynamicArrayStack.java containing most of the implementation of the dynamic array stack class. You cannot add data members to or modify the data members of the dynamic array stack class. You cannot modify the dynamic array stack class constructors. You cannot add additional methods or any nested classes to the dynamic array stack class.

   To reinforce your comprehension of the stack data structure, you will write your own implementation of the 5 methods (**size**, **isEmpty**, **push**, **top**, and **pop**) of the dynamic array stack class. Because the storage for the stack is dynamic (grows and shrinks), the push method doubles the amount of storage when the stack's current storage is full. If the stack is empty when popping an element off the top of the stack or by a call to the top method to get the element at the top of the stack without removing it then have the pop and top methods return null to indicate an error occurred. Also, in the pop method, if the number of values in the stack is ¼ the amount of current storage in the stack then shrink the amount of storage in the stack to ½ of its current size. You cannot use the built in version of this data structure in the Java Collections Framework.

2. The input to your program will read from a plain text file called **project2.txt**. This is the statement you'll use to open the file:

   ```
   FileInputStream fstream = new FileInputStream("project2.txt");
   ```

   Assuming you're using Eclipse to create your project, you'll store the input file project2.txt in the parent directory of your source code (**.java** files) which happens to be the main directory of your project in Eclipse. If you're using some other development environment, you'll have to figure out where to store the input file.

   The input consists of one or more arithmetic expressions with each arithmetic expression to convert placed on a separate line. Assume that each arithmetic expression is correct, does not contain any errors. Remember, variables in the arithmetic expressions may **not** be just a single character in length. Also, operands, operators, and parenthesis in an arithmetic expression may or may **not** be separated by one or more spaces.

3. The output of your program should be the original arithmetic expression in infix notation on one line and then the equivalent arithmetic expression in postfix notation on the next line. Have a blank line between each arithmetic expression processed by your program.

4. Make the name of the driver class **Project2** and it should only contain only one method:
   `public static void main(String args[]).`
   The main method will open the file **project2.txt** and have a loop to process each arithmetic expression (line) in the file. The main method itself should be fairly short and will pass the line of text to another class to perform the conversion to postfix notation.

5. Do **NOT** use your own packages in your program. If you see the keyword **package** on the top line of any of your .java files then you created a package. Create every .java file in the **src** folder of your Eclipse project

6. Do **NOT** use any graphical user interface code in your program!

7. Make your program as modular as possible, not placing all your code in one .java file. You can create as many classes as you need in addition to the classes described above. Methods should be reasonably small following the guidance that "A function should do one thing, and do it well."

8. Document and comment your code thoroughly.

Grading Criteria

The total project is worth 20 points, broken down as follows:

4 points: Program compiles without any errors.

4 points: Following good coding standards, good comments, concise main program, no significant code duplication (i.e., good method design), and following directions as specified in this assignment. Putting all or most of the code in the main method will cause you to lose points. You must write methods and classes.

4 points: Proper implementation of the dynamic array stack class.

4 points: Ability to read the arithmetic expression from the file and process it properly.

4 points: Correct postfix notation result for each infix notation arithmetic expression.