The Shunting Yard Algorithm

Evaluating Postfix and Infix expressions

Have you ever wondered how the Java compiler is able to take an expression (say $1+2\times(5\times4/2)/5$) and find what it evaluates to? Take a guess as to how a programming language like java might try to solve this. Write down your first guess in a comment in the `main` method of "`App.java`".

For this project, we'll do it in two phases: (1) converting from infix (eg. 5+5) to postfix (eg. 55+), then (2) evaluating the postfix version of the equation. For the conversion we'll be using an algorithm known as the Shunting Yard Algorithm, invented by Edsger Dijkstra in 1961. We'll then use a stack to evaluate the postfix expression. Note that this project only covers working with single digit input numbers.

Problem 1: Initial Setup and `main`

First download the zip file from canvas, you'll be doing your work in "`App.java`". Then write code to do the following steps

(a) Take in an expression using a scanner

(b) Run `convertInfixToPostfix` on that expression, using the expression as the parameter

(c) Run `evaluatePostfix` on the result of `convertInfixToPostfix` and print it

Now we'll implement the shunting yard algorithm. Begin by watching this video: Link.

After that, recall the major methods of Stacks and Queues in Java.

STACKS

(1) Stacks are declared with the format `Stack<T> stackName = new Stack<T>();`. You can only access or change the top item of a stack.

(2) `stackName.peek()` allows for accessing the top item in the stack. You can't peek an empty stack.

(3) `stackName.isEmpty()` returns whether the stack is empty

(4) `stackName.pop()` removes and returns the top item in the stack (eg. `T topItem = stackName.pop();`)

(5) `stackName.push(objectName)` puts a new object on top of the stack

(6) Stacks follow LIFO (last in, first out), meaning the most recent pushed object is the first popped object

QUEUES

(1) Queues are declared with the format `Queue<T> queueName = new LinkedList<T>();`. You can only access or change the first item in a stack. Notice that Queue is techinally an interface of the `LinkedList` class

(2) `queueName.element()` allows for accessing the first item in the queue. This is the examine method. You can't examine an empty queue.

(3) `queueName.isEmpty()` returns whether the queue is empty

(4) `queueName.remove()` removes and returns the first item in the queue (eg. `T topItem = queueName.remove();`). This is the dequeue method.

(5) `queueName.add(objectName)` puts a new object in the back of the queue. This is the enqueue method.

(6) Queues follow FIFO (first in, first out), meaning the first added object is the first removed object

Now let's implement the shunting yard algorithm.

Problem 2: The Shunting Yard Algorithm

Open up "`App.java`" and go to `convertInfixToPostfix`
Use the following pseudocode, along with the comments and code in the file to write a java version of the Shunting Yard algorithm

```
iterate through expressionChars
    if it's an operator or opening parenthesis
        If the top operation of the stack is of higher precedence, then
        pop back to the beginning or last opening parenthesis
            enqueue the popped operators
        push the most recent operator to the stack
    if it's a closing parenthesis
        pop back to the last open parenthesis
            enqueue the popped operators
    otherwise
        enqueue to expression queue, only numbers should reach this step
empty out the operator stack into the expression queue, excluding parentheses
empty the expression queue into the result arraylist
```

Problem 3: `operate` method

This method is a simple one. `operate(operator, numLeft, numRight)` returns the value of the expression `numLeft operator numRight`, so for example `operate("/",5.0,2.0) = 5.0/2.0 = 2.5`.
You can use a set of `else-if` statements or another strategy to write this method.

Problem 4: `evaluatePostfix` method

Using a stack and the `Double.parseDouble(string)` method to convert strings to Doubles, write a method that evaluates a postfix expression. Consider the following broad strategy:

1. Iterate through the expression list
2. Push numbers to the stack
3. When reaching an operator, pop the top two values of the stack
4. Using `operate`, find the value of the mini expression. Consider carefully the order in which the popped numbers should be passed into `operate` (think about what LIFO means)

5. Push the result back onto the stack
6. At the end of the expression there should be one value on the stack, that is the final answer.

Explain why this strategy works inside the method.

Extensions

Try one of these extensions, in rough order of increasing challenge.

(a) Extend the program to work for non-single digit numbers

(b) Extend the program to handle negative numbers

(c) Try implementing another infix denary operator. It could be one you made up (acting as a function) or another (even binary operators could work, just treat the number as binary and use the bitwise version of the operator-you may need to convert it to an int)

(d) For extra challenge, try implementing a unary operator like factorial

(e) Even more complex, try implementing an operator like increment or decrement (++ or –)

(f) Implement a function (like *sin* or *log*) into the program