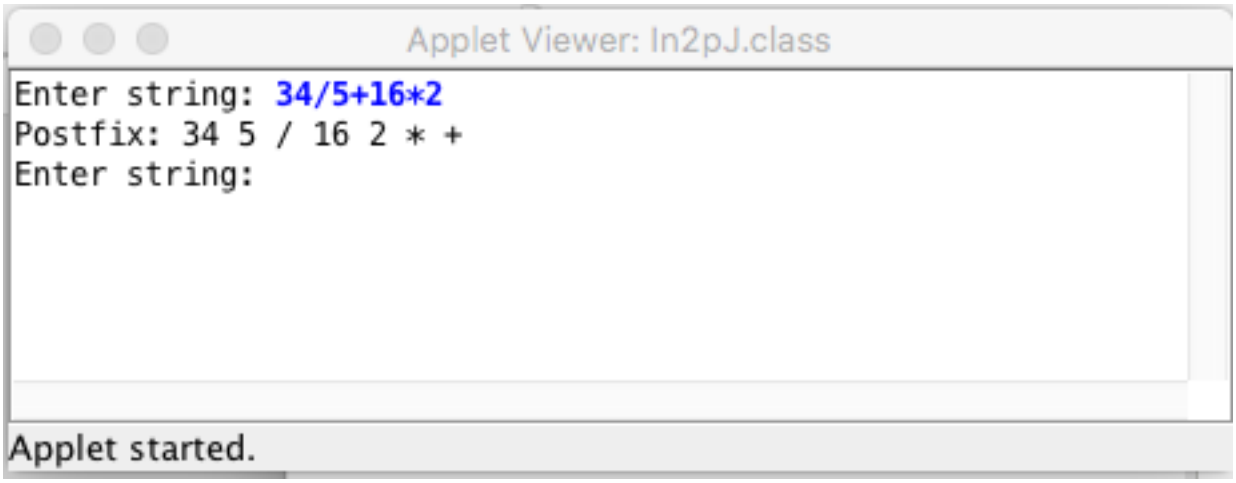
Department of Electrical and Computer Engineering
ECSE 202 – Introduction to Software Development
Assignment 3
Application of Stacks and Queues: Expression Parsing

Problem Description

Write a program, In2pJ, which reads in an infix expression from the console and outputs a postfix expression that can subsequently be processed by a simple interpreter. This output will be used in the next assignment to create a simple 4-function calculator, as outlined in the attached file. You are not being asked to do a full implementation of the calculator, only the first stage which converts an algebraic expression of the form 34 / 5 + 16 * 2 (infix) to postfix notation, which for this example is 34 5 / 16 2 * +.

Your program should operate as follows:



Note that in the above example, the program is run in an infinite loop – you do not need to do this for the assignment. Expressions will be limited to operands and binary operators only (bonus points awarded for also handling unary operators and parentheses).

Method

You are to use the Shunting Yard algorithm, the simple version of which is described in the attached file, and uses 3 data structures: 2 queues for holding input and output and a stack for holding operators. You have already written these classes for Assignment 2.

One of the difficulties in writing this program is in parsing the input string and separating it into operators and operands. Fortunately Java provides the stringTokenizer class that makes this easy. You can look up how this works online. Here is a simple test program you might want to try.

```
import java.util.StringTokenizer;
import acm.program.ConsoleProgram;

public class TestParse extends ConsoleProgram {
  public void run() {
          String str = readLine("Enter string: ");
          StringTokenizer st = new StringTokenizer(str,"+-*/",true);
          while (st.hasMoreTokens()) {
                  println("-->"+st.nextToken());
          }
  }
}
```

Here's what happens when you run the program:

```
Enter string: 34/5+16*2
-->34
-->/
-->5
-->+
-->16
-->*
-->2
```

You need to devise an appropriate loop (hint, look at the while loop in the above example) that enqueues this data in the input queue. From here, implementation of the Shunting Yard program follows the recipe outlined in the notes (there also a lot of information available online). Once your program is running, run through each of the following test cases, saving the results to a file.

Test Cases:

Enter string: 34/5+16*2
Postfix: 34 5 / 16 2 * +
Enter string: 5+9.27/1.4*3 + 2/3
Postfix: 5 9.27 1.4 / 3 * + 2 3 / +
Enter string: 1.1-2.2*3.4/5.6
Postfix: 1.1 2.2 3.4 * 5.6 / -
Enter string: 6/7+3/4+1/2
Postfix: 6 7 / 3 4 / + 1 2 / +
Enter string: 9.8+3*6.7/2-4
Postfix: 9.8 3 6.7 * 2 / + 4 –

Instructions

Write the In2pJ program as described in the preceding sections. It should operate interactively and be able to replicate the output from the test cases shown above. To obtain full marks your code must be fully documented and correctly replicate the test cases.

Your submission should consist of 5 files:

1. Stack.java - stack class
2. Queue.java - queue class
3. listNode.java - node object
4. In2pJ.java - program
5. In2pJ.txt - output

Upload your files to myCourses as indicated.

fpf/February 19, 2018

A simple calculator

- Stacks and queues allow us to design a simple expression evaluator
- Prefix, infix, postfix notation: operator before, between, and after operands, respectively

Infix	Prefix	Postfix
A + B	+ A B	A B +
A * B - C	- * A B C	A B * C -
(A + B) * (C - D)	* + A B - C D	A B + C D - *

- Infix more natural to write, postfix easier to evaluate

from MIT Open Courseware, 6.087 Practical Programming in "C"
IAP 2010

Infix to postfix

- "Shunting yard algorithm" - Dijkstra (1961): input and output in queues, separate stack for holding operators
- Simplest version (operands and binary operators only):
  1. dequeue token from input
  2. if operand (number), add to output queue
  3. if operator, then pop operators off stack and add to output queue as long as
     - top operator on stack has higher precedence, or
     - top operator on stack has same precedence and is left-associative

     and push new operator onto stack
  4. return to step 1 as long as tokens remain in input
  5. pop remaining operators from stack and add to output queue

from MIT Open Courseware, 6.087 Practical Programming in "C"
IAP 2010

Infix to postfix example

- Infix expression: A + B * C - D

Token	Output queue	Operator stack
A	A	
+	A	+
B	A B	+
*	A B	+ *
C	A B C	+ *
-	A B C * +	-
D	A B C * + D	-
(end)	A B C * + D -	

- Postfix expression: A B C * + D -
- What if expression includes parentheses?

from MIT Open Courseware, 6.087 Practical Programming in "C"
IAP 2010

Example with parentheses

- Infix expression: (A + B) * (C - D)

Token	Output queue	Operator stack
(		(
A	A	(
+	A	(+
B	A B	(+
)	A B +	
*	A B +	*
(	A B +	* (
C	A B + C	* (
-	A B + C	* (-
D	A B + C D	* (-
)	A B + C D -	*
(end)	A B + C D - *	

- Postfix expression: A B + C D - *

Evaluating postfix

- Postfix evaluation very easy with a stack:
  1. dequeue a token from the postfix queue
  2. if token is an operand, push onto stack
  3. if token is an operator, pop operands off stack (2 for binary operator); push result onto stack
  4. repeat until queue is empty
  5. item remaining in stack is final result

from MIT Open Courseware, 6.087 Practical Programming in "C"
IAP 2010

Postfix evaluation example

- Postfix expression: 3 4 + 5 1 - *

Token	Stack
3	3
4	3 4
+	7
5	7 5
1	7 5 1
-	7 4
*	28
(end)	answer $= 28$

- Extends to expressions with functions, unary operators
- Performs evaluation in one pass, unlike with prefix notation

from MIT Open Courseware, 6.087 Practical Programming in "C"
IAP 2010