COP3530 Data Structures
Project#2 (Due on June 9nd by 11:59 pm)
25 points
Summer 2019

**Project Description**

In this project, we are going to use the Stack ADT to convert an expression in infix notation to postfix notation. The Stack ADT will be implemented with an array. The Stack ADT implementation for this project contains an additional function that you will write to print the contents of the stack as characters instead of integers. If you manually convert a few infix expressions to postfix form, you will discover three important facts:

- The operands always stay in the same order with respect to one another.
- An operator will only move "to the right" with respect to the operands; that is, if , in the infix expression, the operand precedes the operator, it is also true that in the postfix expression, the operand precedes the operator.
- All parentheses are removed.

As a consequence of these three facts, the primary task of the conversion algorithm is determining where to place each operator. The following pseudocode describes a first attempt at converting an infix expression to an equivalent postfix expression postfixExp:

```
Initialize postfixExp to the null string
for (each character ch in the infix expression) {
  switch (ch) {
  case ch is an operand:
    Append ch to the end of postfixExp
    break;
  case ch is an operator:
    Store ch until you know where to place it
    break;
  case ch is '(' or ')':
    Discard ch
    break;
  }
}
```

You may have guessed that you really do not want to simply discard the parentheses, as they play an important role in determining the placement of the operators. In any infix expression, a set of matching parentheses defines an isolated subexpression that consists of an operator and its two operands. Therefore, the algorithm must evaluate the subexpression independently of the rest of the expression. Regardless of what the rest of the expression looks like, the operator within the subexpression belongs with the operands in that subexpression. Parentheses are thus one of the factors that determine the placement of the operators in the postfix expression. The other factors are precedence and left-to-right association.

The following is a high level description of what you must do when you encounter each character as you read the infix string from left to right.

- When you encounter an operand, append it to the output string postfixExp. Justification: The order of the operands in the postfix expression is the same as the order in the infix expression, and the operands that appear to the left of an operator in the infix expression also appear to its left in the postfix expression.
- Push each "(" onto the stack.
- When you encounter an operator, if the stack is empty, push the operator onto the stack. However, if the stack is not empty, pop operators of greater or equal precedence from the stack and append them to postfixExp. You stop when you encounter either a "(" or an operator of lower precedence

or when the stack becomes empty. You then push the new operator onto the stack. Thus, this step orders the operators by precedence and in accordance with left-to-right association. Notice that you continue popping from the stack until you encounter an operator of strictly lower precedence than the current operator in the infix expression. You do not stop on equality, because the left-to-right association rule says that in case of a tie in precedence, the leftmost operator is applied first--and this operator is the one that is already on the stack.

- When you encounter a ")", pop operators off the stack and append them to the end of postfixExp until you encounter the matching "(". Justification: Within a pair of parentheses, precedence and left-to-right association determine the order of the operators, and Step 3 has already ordered the operators in accordance with these rules.
- When you reach the end of the string, you append the remaining contents of the stack to postfixExp.

For example, the figure below traces the action of the algorithm on the infix expression 1+(2+3*4)/5, assuming that the stack and the string postfixExp are initially empty. At the end of the algorithm, postfixExp contains the resulting postfix expression 1234*+5/+.

| Operator/ Operand/ Parenthesis read | Postfix Expression | Stack (bottom to top) |
|---|---|---|
| 1 | 1 | < > |
| + | 1 | < + > |
| ( | 1 | < + ( > |
| 2 | 12 | < + ( > |
| + | 12 | < + ( + > |
| 3 | 123 | < + ( + > |
| * | 123 | < + ( + * > |
| 4 | 1234 | < + ( + * > |
| ) | 1234* | < + ( + > |
| | 1234*+ | < + ( > |
| | 1234*+ | < + > |
| / | 1234*+ | < + / > |
| 5 | 1234*+5 | < + / > |
| | 1234*+5/ | < + > |
| | 1234*+5/+ | < > |

The first step in this project is to complete the following implementation of the Stack ADT. Notice that there is a new function in this implementation: printCharStack(). This function will be used to print the contents of our stack as characters instead of integers. This function will be very useful when we print our trace.

**Stack.java**

```
class Stack {

  private int count; //number of elements in the stack

  private int top;   //top element of the stack.
                     //top == -1 if the stack is empty

  private int MAXSIZE = 1000;  //Physical size of the stack
```

```
  private int [ ] array;

  //constructor
  Stack( ) {

  }

  //inspectors
  public boolean stackEmpty() {
  //Returns true if the stack is empty. Otherwise returns false.

  }

  public int stackTop() {
  //Returns the top element of the stack

  }

  public int stackCount() {
  //Returns the number of elements in the stack

  }

  public String toString() {
  //Returns the elements of the stack with the following format:
  //< E1 E2 E3 ... En> where E1 is at the bottom of the stack
  //and En is at the top of the stack.

  }

  public String printCharStack() {
  //Returns the elements of the stack with the following format:
  //< E1 E2 E3 ... En> where E1 is at the bottom of the stack
  //and En is at the top of the stack.
  //The elements are printed to the returning string as chars.

  }

  //modifiers
  public void stackPush(int element) {
  //Pushes element to the top of the stack.
  //Assumes that we will not run out of space in the stack.

  }

  public boolean stackPop() {
  //Pops out the top element of the stack.
  //Returns true if the operation was successful.
  //Returns false otherwise.


  }

}
```

COP3530 Data Structures
Project#2 (Due on June 9nd by 11:59 pm)
25 points
Summer 2019

The second step in this project is to complete the following program which reads an infix expression entered by the user and then converts this expression to postfix notation using the algorithm described above.

To simplify our work, we will assume that the infix expression that we read has no spaces at all between the operands, operators, and parenthesis, that all the operands are integers in the range 0..9, and that the only operators allowed are +, *, and /.

See the execution examples given at the bottom.

Note that in each execution example, a trace of the conversion procedure has been printed. Basically, for each operand/operator/parenthesis that we read, we print this operator/operand/parenthesis along with the postfix expression and the contents of the stack after processing this operator/operand/parenthesis. You are required to print this trace as well.

**Note:** When I grade your projects, I will test your program with other infix expressions, not just the ones shown in the execution examples below. Therefore, make sure that you test you program with lots of other infix expressions.

**Project2.java**

```
import java.util.Scanner;

class Project2 {

  public static void main(String args[]) {

    String infixExp; //infix expression to be read
    String postfixExp; //postfix expression to be computed
    Stack S = new Stack(); //stack for converting the infix expression
                           //to postfix notation
    Scanner scan = new Scanner(System.in);
}
```

**Sample Execution**

Example 1

```
Enter an infix expression: 3+4

3:      3       < >
+:      3       < + >
4:      34      < + >
        34+     < >

The expression in postfix notation is 34+
```

Example 2

```
Enter an infix expression: 2*(3+4)

2:      2       < >
*:      2       < * >
(:      2       < * ( >
3:      23      < * ( >
+:      23      < * ( + >
```

```
4:      234     < * ( + >
):      234+    < * ( >
        234+    < * >
        234+*   < >

The expression in postfix notation is 234+*
```

Example 3

```
Enter an infix expression: 6/2+9/3+8/4

6:      6       < >
/:      6       < / >
2:      62      < / >
+:      62/     < >
        62/     < + >
9:      62/9    < + >
/:      62/9    < + / >
3:      62/93   < + / >
+:      62/93/  < + >
        62/93/+ < >
        62/93/+ < + >
8:      62/93/+8        < + >
/:      62/93/+8        < + / >
4:      62/93/+84       < + / >
        62/93/+84/      < + >
        62/93/+84/+     < >

The expression in postfix notation is 62/93/+84/+
```

Example 4

```
Enter an infix expression: 4/2+3*(5+1)

4:      4       < >
/:      4       < / >
2:      42      < / >
+:      42/     < >
        42/     < + >
3:      42/3    < + >
*:      42/3    < + * >
(:      42/3    < + * ( >
5:      42/35   < + * ( >
+:      42/35   < + * ( + >
1:      42/351  < + * ( + >
):      42/351+ < + * ( >
        42/351+ < + * >
        42/351+*        < + >
        42/351+*+       < >

The expression in postfix notation is 42/351+*+
```

Example 5

```
Enter an infix expression: (3+4)/(5+8)

(:              < ( >
3:      3       < ( >
+:      3       < ( + >
```

```
4:      34      < ( + >
):      34+     < ( >
        34+     < >
/:      34+     < / >
(:      34+     < / ( >
5:      34+5    < / ( >
+:      34+5    < / ( + >
8:      34+58   < / ( + >
):      34+58+  < / ( >
        34+58+  < / >
        34+58+/ < >

The expression in postfix notation is 34+58+/
```

Example 6

```
Enter an infix expression: ((3+1)/4+((1+1)*(2+1)))

(:              < ( >
(:              < ( ( >
3:      3       < ( ( >
+:      3       < ( ( + >
1:      31      < ( ( + >
):      31+     < ( ( >
        31+     < ( >
/:      31+     < ( / >
4:      31+4    < ( / >
+:      31+4/   < ( >
        31+4/   < ( + >
(:      31+4/   < ( + ( >
(:      31+4/   < ( + ( ( >
1:      31+4/1  < ( + ( ( >
+:      31+4/1  < ( + ( ( + >
1:      31+4/11 < ( + ( ( + >
):      31+4/11+        < ( + ( ( >
        31+4/11+        < ( + ( >
*:      31+4/11+        < ( + ( * >
(:      31+4/11+        < ( + ( * ( >
2:      31+4/11+2       < ( + ( * ( >
+:      31+4/11+2       < ( + ( * ( + >
1:      31+4/11+21      < ( + ( * ( + >
):      31+4/11+21+     < ( + ( * ( >
        31+4/11+21+     < ( + ( * >
):      31+4/11+21+*    < ( + ( >
        31+4/11+21+*    < ( + >
):      31+4/11+21+*+   < ( >
        31+4/11+21+*+   < >

The expression in postfix notation is 31+4/11+21+*+
```

Example 7

```
Enter an infix expression: 1+(2+3*4)/5

1:      1       < >
+:      1       < + >
(:      1       < + ( >
2:      12      < + ( >
```

COP3530 Data Structures
Project#2 (Due on June 9nd by 11:59 pm)
25 points
Summer 2019

```
+:      12      < + ( + >
3:      123     < + ( + >
*:      123     < + ( + * >
4:      1234    < + ( + * >
):      1234*   < + ( + >
        1234*+  < + ( >
        1234*+  < + >
/:      1234*+  < + / >
5:      1234*+5 < + / >
        1234*+5/        < + >
        1234*+5/+       < >

The expression in postfix notation is 1234*+5/+
```

**Deliverables:**

Make sure your project folder (i.e., Lastname_P1.zip) contains:

1. Source code (i.e., .java files NOT .class files)
    - Stack.java
    - Project2.java
2. A readme file (i.e., readme.txt) explaining how to compile and run your project
3. Screenshots of your output.
4. Any input files related to this project

Finally, compress your project folder and submit it on Canvas.