CSCI 204 - Introduction to Computer Science II

Lab 9- Stacks

1 Objectives

- Use a stack ADT

Stacks have many uses. For example, they make it easy to convert from infix to postfix notation. (Formatting the addition operator as `(3 + 4)` which is infix or `(3 4 +)` which is postfix). In this lab, you will take such a program and extend it so that it will handle expressions with parentheses.

2 Lab Details

Create a new project for today's lab and then import the files that you will need from

`~csci204/2011-spring/student/labs/lab09`

Commit your files to your lab repository.

This stack ADT is implemented as a linked list instead of an array. If you have imported your files correctly you should have the following files in Eclipse:

```
Infix2Postfix.java
Infix2PostfixTester.java
LinkedList.java
ListIterator.java
ParensMismatchException.java
Stack.java
```

This is a complete program for converting from infix to postfix notation. The input to the program is given as string arguments to the `main()` method of the `Infix2PostfixTester` class. Try the following examples to see how it works.

```
"a+b"
"a+b*c"
"a*b+c"
"a+b*c-d"
```

You have to enter command line arguments using the Run → Run Configurations → Arguments → Program Arguments and setting the argument to your desired input. In this case, you can put as many expressions as you would like as arguments as long as they are separated by spaces (e.g., `"a+b"␣"a+b*c"␣"a*b+c"`).

The program converts a line of infix-style math to the equivalent postfix-style math. Save the output for later. You will be editing this program to make it convert a more complex form of math from infix to postfix. You will not be editing the implementation of the Stack ADT.

Here is how the program works. It examines each character in the input string. If it's a variable (a character between `'a'` and `'z'`), it just prints it. Otherwise, the character must be an operator. In this case, the program will

remove and print operators that are on the stack if they have greater or equal precedence. Note that the algorithm never pushes anything other than operators onto the stack, so we can be sure that the characters we are popping are indeed operators.

2.1 Allow Parentheses

For this lab, you are to change the program so that it will handle input strings with parentheses. You will need to make several changes to the `Infix2Postfix` class which are outlined below.

The program examines each character in the expression. If the character is a variable it does one thing, and if the character is an operator, it does something else. You will need to add other possibilities for the characters the program sees. You will add processing for a left parenthesis and processing for a right parenthesis.

2.1.1 Left Parenthesis

If the character being processed is a left parenthesis, just push it onto the stack. Note that the stack may now contain left parentheses in addition to operators. You will need to make other changes because of this.

2.1.2 Right Parenthesis

When your program sees a right parenthesis, pop operators off the stack and print them until you reach a left parenthesis or the stack is empty. If you reach a left parenthesis, remove it from the stack but don't print it. If there is no left parenthesis on the stack, you should throw a `ParensMismatchException`. This case can only arise when there are too many right parentheses in your expression. Put the right parenthesis processing code in a separate method and give it a meaningful name.

2.1.3 Operator Processing

The processing for an operator will be similar to what was done before. Previously, the program would pop and print operators until the stack became empty or it reached an operator of higher or equal precedence. Now there is an additional condition that will stop this loop. It must also end the loop if it sees a left parenthesis on top of the stack. However, you should never remove a left parenthesis when seen in this context.

It is actually possible to use the same basic code for processing left parentheses and operators. This can be accomplished by giving a left parenthesis the lowest precedence (lower than addition and subtraction) when it is being pushed onto the stack and the highest precedence (higher than multiplication and division) when it is already on the stack. In this way, it will not cause any operators to be popped off of the stack when added and will never be popped off of the stack itself when an operator or left parenthesis is stacked on top of it. This requires two simple tests to be added in the `comparePrecedence()` method of `Infix2Postfix`.

2.1.4 Emptying the Stack

After the main `for` loop, the program removes any remaining operators from the stack and prints them. You still want to remove all the operators and print them, but there will no longer be any left parentheses on the stack. This is because whenever you encounter a right parenthesis, you remove the matching left parenthesis. If there are any left parentheses left on the stack when it is being emptied, you should throw a `ParensMismatchException`. This means there were unclosed left parentheses.

Try to use this fact to your advantage. Your goal is to empty the stack.

2.2 Parentheses Mismatch

If these is a parentheses mismatch that is found during translating, `Infix2Postfix` will throw a `ParensMismatchException`. You should added code to `Infix2PostfixTester` to catch this exception and print a message similar to the one it prints when it encounters an `IllegalArgumentException`.

2.3 Expected Results

The following samples are the correct output for a completed program. They should help you to check if your revised program works.

Infix	**Postfix**
`"a+(b-c)/d"`	`abc-d/+`
`"(a+b)*c/d"`	`ab+c*d/`
`"a+b*c/d"`	`abc*d/+`

3 Upon Completion

Make sure the original four examples in Lab Details still work and then test the following examples.

```
"a*(b-c)*d"
"(a+b)*(c+d)"
"a/(b*(c-d))"
"(((a)))"
"((a)"
"(a))"
```

Commit all of your files to your repository.