Lab Goals

- Practice working with the Java Stack class
- Learn the difference between infix and postfix notation
- Answer a few questions to test your knowledge of course material

Assignment Details

In recent classes, we began discussing the Stack data structure. Stacks can store information of a variety of data types, with two restrictions: (1) only the top item in the stack is accessible, and (2) the first data item entered into the stack must be the last data item removed from the stack (also thought of as the last data item entered into the stack must be the first data item removed from the stack).

In this lab, we will implement a method to turn an input string representing a calculation from infix notation into postfix notation.

Part 1: Definitions and Practice (15 points)

Infix notation is the representation of mathematical formulas that you are used to seeing. In infix notation, operators are placed between the operands. For example, the formula “a + b” is written in infix notation, with the “plus” operator located between the “a” and “b” operands.

In contrast, operators are placed after their associated operands in postfix notation. The equivalent formula to the above is “ab+” in postfix notation, with the “plus” operator located after the “a” and “b” operands.

Postfix notation is typically implemented in compilers, because parentheses can make the direct conversion from infix to machine language difficult. Adding postfix notation as a step midway simplifies compilation. Parentheses are not needed, and hence not used, in postfix notation. This is because postfix notation structures strings so that the operator computes two operands that precede it.

For example, consider the string “a * b + c” in infix notation. The “star” operator is applied to the “a” and “b” operands, so it should follow those operands. Hence, we can start our postfix notation string with “ab*.” After the system computes this string, it is effectively a single operand.
(think “\(d = a \times b\)”, so that “\(d\)” operand added to operand “\(c\)” is handled next. Hence, the final postfix string will be “\(ab \times c+\)”.

This gets more complicated when we start to consider order of operations. The infix string “\(a + b \times c\)” is mathematically evaluated in reverse, with “\(b \times c\)” evaluated before “\(a\)” is added to get the final result. To handle this case, we can introduce the idea of stacks to the infix-to-postfix conversion. As we read the infix string from left to right, we start by leaving the “\(a\)” alone, moving it directly to the postfix string. When reading the “+,” we will push it to a stack. We will read the “\(b\)” character and add it to the postfix string. Now we read the “\(\times\)” character. Because the “\(\times\)” operation supercedes the “+” operation in the order of operations, we will push the “\(\times\)” to the stack. Finally, we read the “\(c\)” character, and add it to the postfix string. Since we have reached the end of the infix string, we must now pop the operators off the the stack. First the “\(\times\)” is popped, followed by the “+” operator. This yields a final postfix string of “\(abc+\)” in which the “\(\times\)” operator is applied to the “\(b\)” and “\(c\)” operands, then the “+” operator is applied to that result and the “\(a\)” operand.

We can add an additional layer of complexity when adding parentheses to the infix string input, the whole reason for making this conversion. Given the input string “\((a + b) \times c\)” we want the “\(a + b\)” component to be evaluated before the “\(\times c\)” is applied. Hence, our final postfix string should be “\(ab + c\times\)” We can handle this by pushing the opening parenthesis character to the same stack as the operators. Reading across the infix string, we read “(” and push it onto the stack. We read the “\(a\)” and add it to the postfix string, read the “+” and push it to the stack, and then read the “\(b\)” and add it to the postfix string. Now, when we encounter the closing parenthesis character, we will pop everything from the stack until the first “)” is found on the stack, which adds the “+” character to the postfix string. The closing parenthesis character is discarded, and the rest of the postfix string generation continues as before.

The final case that we should consider is if our input string includes two or more consecutive operators on the same level of order of operations precedence. For example, consider the infix string “\(a + b - c - d\)” Naturally, these operators are evaluated in order – the “+” is applied to the “\(a\)” and “\(b\)” the first “-” is applied to the “\(a + b\)” result and “\(c\)” and the second “-” is applied to the “\(a + b - c\)” result and “\(d\)” Hence, our postfix string should be “\(ab + c - d -\)” We can generate this string with a comparison step similar to the comparison between operator levels in the “\(a + b \times c\)” example. When we reach the first “-” operator, we already have “\(ab\)” in the postfix string and “+” on the top of the stack. We compare the “-” in the infix string to the “+” at the top of the stack. Because they are on the same level, we will pop the “+” from the stack and add it to the postfix string, and then push the “-” onto the stack. Similarly, when we reach the second “-” character, we already have “\(ab + c\)” in the postfix string and “-” on the top of the stack. We compare the “-” in the infix string to the “-” at the top of the stack. Because they are on the same level (and the same operator), we will pop the “-” from the stack and add it to the postfix string, then push the new “-” onto the stack.

We can summarize the actions that we should take in converting a string from infix to postfix notation with the table shown on the next page.
Using these rules, please convert the following infix strings into postfix notation:

1. \( a \ast b \)
2. \( a + b/c \)
3. \( (a + b)/c + d \)
4. \( (a + b)/(c + d) \)
5. \( ((a \ast b) - c)/d \)

**Part 2: Implementing an Infix-to-Postfix Converter (25 points)**

Given the rules from the table at the top of the page and the expertise that you have gained from practicing in the previous section, you will now write a Java program to convert an input string from infix notation into postfix notation.

First, you should check the string for matching parentheses using the function that we wrote in class this morning. This will prevent both of the error conditions noted in the table above. If that test is passed, you can then pass the infix string into the converter function.
The converter function should take the infix string as an input parameter, and should return the postfix string at the end of the computation. You should use a Stack to store the operators as described in the previous section. You can assume that each of the operands will be a single character in length.

Partial credit will be given for (in order of implementation complexity):

- Converting strings with only one operator (e.g. “a + b” or “a * b”)
- Converting strings with only chains of operators all at the same level of precedence (e.g. “a + b + c + d” or “a * b / c * d”)
- Converting strings without parentheses (e.g. “a + b * c” or “a * b + c - d”)

Part 3: Additional Questions (10 points)

Please answer the following questions thoroughly:

1. Provide a table similar to the one on Slide #6 of Lecture #18 for the following sequence of Stack commands: isEmpty(), push(3), push(1), push(4), peek(), size(), pop(), push(1), push(5), peek(), size(), pop(), pop(), pop(), empty()

2. Suppose an initially empty stack S has performed a total of 35 push operations, 12 top operations, and 10 pop operations, 3 of which returned null to indicate an empty stack. What is the current size of S?

3. Write a short method to copy the contents of a given stack S into a new stack T. When complete, the contents in T should have the same ordering that they had in S. You can either give pseudocode or Java code for this question.

Part 4: Extra Credit (5 points)

Up to 5 points of extra credit will be awarded for adding exponentiation support with the “∧” character into your infix-to-postfix converter. Remember that, in the order of operations, exponentiation has precedence over multiplication, division, addition, and subtraction, but not over parentheses. You will need to determine how the system should behave when a “∧” character is reached in the infix string – what should get added to the stack, what should get removed from the stack, and so on.
Submission Details

For this assignment, your submission (to both your BitBucket repository and by hardcopy) should include the following:

1. (Print and Upload) Your answers to the infix-to-postfix conversions in Part One.
2. (Upload) Your source code for your converted in Part Two.
3. (Print and Upload) Sample output showing how your converter works on several different input strings.
4. (Print and Upload) The answers to the questions from Part Three
5. (Print and Upload) An Assignment Information Sheet for your code submission

Before you turn in this assignment, you also must ensure that the course instructor has read access to your BitBucket repository that is named according to the convention cs112s2016-<your user name>.