In this assignment, I created a program to convert infix expressions to postfix format, and then evaluate based on the postfix format. I have used NetBeans to implement my project. To do so, I created a project called *Project3CSC4101*. Then, I built a class called *Convertor.java* to implement my convertor code. First, I created two static methods called *convertToPostfix* and *evaluatePostfix*. The inputs to this method are both string data type. Also, the output data type for the first method is a string, and for the second method is a float.

To implement this project, we need to import the package java.util.Stack. The two most essential operations in stack are **push** and **pop**. Push operation adds an element to the stack, and the pop operation removes the element on the top of the stack.

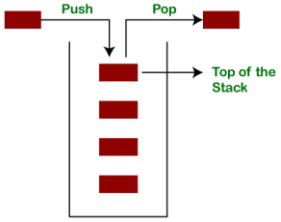

Figure 1. Push and Pop operations in stack [1].

1. ***ConvertToPostfix* method:**

First, we created a static method with string data type input and output. The input string is the infix expression, and the output of the method is the postfix. Below, we can see an example for infix and postfix:

Infix: (1 + 2) / 4 - 5 * 6 → Postfix: 1 2 + 4 / 5 6 * -

To start working on the method, we need a string data type variable called *postfix* and a *stack<character>* data type called *stack*. We also need a boolean type of variable called *isPrevDigit*, which is *false* by default, to investigate the numbers with more than one digit. Then, we need a *for* loop to go character by character for infix expression. The *for* loop length is equal to the length of the expression. *c* is a character data type variable with a character at each point. We used the *charAt* method to return the character at each specified index. Then, we use *if* function to investigate different conditions:

- If the character *c* is a digit, two possible conditions will happen based on the value of *isPrevDigit*. If the previous character is a digit, we should add the character *c* to the postfix (more than one-digit numbers); otherwise, we should first add a space to the postfix and then add the character *c*; to have space between the character *c* and the previous character. Also, since the character *c* is a digit in this scenario, we should assign a *true* value to the variable *isPrevDigit* to be considered for the following characters.
- If the character *c* is space, we only need to assign a *false* value to the *isPrevDigit* variable.
- Generally, in mathematics, parenthesis has the highest priority. If character *c* is a "(" in the infix, we should push it to the stack and assign *false* value to the *isPrevDigit*. *Stack.push()* is a method used to push an element onto the top of the stack.
- If the character *c* is a ")" sign, while the last element in the stack is not "(" and our stack is not empty, we should provide a space at each step, and pop operators from the stack and add it to the postfix variable. We also need to use *stack.pop()* one more time to remove "(" from the stack. Because we do not have "(" sign in the postfix format. In the end, we should assign a *false* value to the *isPrevDigit*.
- If the character *c* is an operator, while the stack is not empty and the precedence of character *c* is less than or equal to the precedence of the *stack.peek()*, we need to add a space in the postfix and

then pop from the stack and add it to the postfix (We will describe the precedence method in section 1.1). We use *stack.peek()* to find out the element at the top of the stack. This method does not lead to the deletion of the element from the stack. Then, character *c* will be added to the stack, and we assign a *false* value to the *isPrevDigit*.

After finishing the for loop, while the stack is not empty, we first check if the last element of the stack is "(" sign, we should return an error. After that, we add space for each element in the postfix and pop elements one by one from the stack to the end to free stack. In the end, we return the value of postfix as the output of the method.

1.1. *Prec* method:

The input to this method is a character, and the output is an integer. This function finds out the precedence of each operator. As we know, * and / have higher precedence than + and -. Therefore, for the * and /, we return the value of 2, and for + and -, we return the value of 1. Otherwise, it will return -1.

2. *evaluatePostfix* method:

What we would like to do in this step is to use the postfix expression that we returned in *convertToPostfix* (as input) to find out the value of the expression (as output). Here, the input to the static method is the postfix as string data type, and the output is a float. Again, we have a boolean type of variable called *isPrevDigit*, which is *false* by default. Also, we have a string type variable called *num* and a stack type variable called *numStk*. In this method, again, we use a *for* loop with the length of the input to go over the input character by character. We use character type variable *c* to get the value of each character using its index and *charAt* method. Then, we use *if* function to investigate different conditions:

- If the character *c* is a digit, we add it to the *num* variable and assign the *true* value to the *isPrevDigit* variable.
- If the character *c* is a space, and if the previous character is a digit, the value of the variable *num* should be converted to the float using *Float.parseFloat(num)*, and then it should be pushed to the stack *numStk*. Also, the *num* variable should change to an empty variable for future use. In addition, we assign a *false* value to the *isPrevDigit* variable.
- Otherwise (*c* is an operator), if we have two or more elements in the stack *numStk*, we should pop the last two elements *y* and then *x*. *x* and *y*, and the character *c*, which is an operator, should be input to the method *SingleOperation* to find out the value of the operation (We will describe the *SingleOperation* method in section 2.1). Then we pushed the calculated value to the stack *numStk* again.

This process will be continued until the final value is calculated. The final value is in the stack *numStk*, and we should pop and return it as method's output.

2.1. *SingleOperation* method:

The inputs to the method are *x* and *y* with float data type and then a character as an *operator*. Also, the output is a float data type value. For different cases of the character operator, which are +, -, /, and *, the operations are *x+y*, *x-y*, *x/y*, and *x*y*, respectively; for any other operator, the method should raise an error "invalid expression" and return -1.

References

[1] https://www.javatpoint.com/java-stack