# **Infix to Postfix and Evaluate**

## ***Infix to Postfix***

Standard infix expressions can be converted to postfix. We develop an algorithm for this conversion in several steps. The basic idea is to move operands directly to the postfix string but to store operators temporarily on a stack. Each operator waits on the stack until its second operand is processed. In this way the operator will appear after the operands. Since full order of operations requires careful stack manipulation, we begin with only the notion of left association.

## ***2a. Algorithm using only Left Association***

Adding and subtracting, without parentheses, evaluates from left to right, so that infix expression 3+4-5+6 is converted to the postfix "3 4 + 5 – 6 +" . Trace the algorithm below:

* Initialize a blank result string for the postfix expression. String result = \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_
* Loop over the input string **3 + 4 – 5 + 6** and process each token as follows:

|  |
| --- |
|  |
|  |
|  |
|  |

* + If the token is an operand (a number), then append it to the postfix string.
  + If the token is an operator
    - If the stack is not empty, then pop the previous operator off the stack   
      and append it to the postfix string.
    - Push the current operator on the stack.

* When the loop ends pop the last operator off the stack and append it to the postfix string.

## ***2b. Precedence and Parentheses***

The algorithm above fails when given a string like "3 + 4 \* 5". The correct postfix string would be

"3 4 5 \* +", but we would get "3 4 + 5 \*" instead. The problem occurs when we pop and append a lower precedence operator, +, even though the current operator has a higher precedence, \*. In such cases we should leave the lower precedence operator on the stack to be popped and appended later.

Handling parentheses is a separate matter altogether. Do not consider left or right parentheses as either operands or operators. Instead, consider the characters between parentheses as an expression within an expression. In this sense, a left parenthesis indicates the logical bottom of a stack within a stack. Normal processing occurs otherwise and the left parenthesis remains on the stack until its matching right parenthesis is encountered.

## ***2c. Complete Algorithm for Infix to Postfix***

Initialize a blank result string for the postfix expression. Assuming the input string contains a valid infix expression, e.g., **3 \* ( 4 \* 5 – 6 + 2 )** loop over the string and process each token as follows:

* If the token is an operand, then append it to the postfix string.
* If the token is a left parenthesis, then push it on the stack.
* If the token is a right parenthesis, then continue popping operators off the stack and appending them to the postfix string until you pop a left parenthesis. Discard both the left and right parentheses.
* If the token is an operator, then we want to push it on the stack but we can only do so if one of three conditions is met:
  1. if the stack is empty
  2. if it is greater than stack.pop
  3. on top of the stack is the left parentheses
* If the conditions are not met, pop the operator off the stack and append it to the postfix string. Continue popping until any one of the three conditions is met. If you encounter a left parenthesis, leave it on the stack.
* When the infix string ends pop all the operators off the stack and append them to the postfix string.

The postfix string for our example will be, of course, **3 4 5 \* 6 - 2 + \***

## ***Assignment***

InfixPostfixEval.java (uses PostfixEval.java). The program changes infix expressions to postfix expressions, then evaluates the postfix expression. Every token is separated by spaces. At first, the only operators are add, subtract, multiply, divide, and modulus. You may call PostfixEval.isOperator and PostfixEval.eval. (Do not change those PostfixEval methods!)

**public static** String infixToPostfix(String infix)  
 { }

//returns true if c1 has strictly lower precedence than c2   
 **public static** **boolean** isLower(**String** c1, **String** c2)  
 { }

## ***Test Run***

Your program should output all three columns.

## Infix --> Postfix --> Evaluate 3 + 4 \* 5 3 4 5 \* + 23.0 3 \* 4 + 5 3 4 \* 5 + 17.0 1.3 + 2.7 + -6 \* 6 1.3 2.7 + -6 6 \* + -32.0 ( 33 + -43 ) \* ( -55 + 65 ) 33 -43 + -55 65 + \* -100.0 3 \* 4 + 5 / 2 - 5 3 4 \* 5 2 / + 5 - 9.5 8 + 1 \* 2 - 9 / 3 8 1 2 \* + 9 3 / - 7.0 3 \* ( 4 \* 5 + 6 ) 3 4 5 \* 6 + \* 78.0 3 + ( 4 - 5 - 6 \* 2 ) 3 4 5 - 6 2 \* - + -10.0 2 + 7 % 3 2 7 3 % + 3.0 ( 2 + 7 ) % 3 2 7 + 3 % 0.0

## Submit InfixPostfixEvalto Codepost. The teachers will upload PostfixEval and ParenMatch as support files.

## ***Extension (extra lab)***

Copy InfixPostfixEval into a new lab Infix\_Extension. Improve the isLower method to handle the precedence for powers and factorials (now you have four levels of precedence). Calling methods from ParenMatch, check the infix string for non-matching parentheses and non-math symbols. If it has any of those, infixToPostfix will return “ERROR” and do not evaluate the expression. All tokens are still separated by one space. Assume each number is well-formed. Use the test data below.

Infix --> Postfix --> Evaluate  
( 3.0 + -1.0 ) ^ 3.0 3.0 -1.0 + 3.0 ^ 8.0  
2 ^ 3 + 3 2 3 ^ 3 + 11.0  
3 \* 2 ^ 3 3 2 3 ^ \* 24.0  
( 1 + 3 ) ! 1 3 + ! 24.0  
1 + 3 ! 1 3 ! + 7.0  
1 \* 3 ! 1 3 ! \* 6.0  
3 ? 2 ERROR  
3 @ 2 ERROR  
( 3 + 2 ERROR  
3 + 2 ] ERROR  
( 3 + 2 ] ERROR

## Submit Infix\_Extensionto CodePost. The teachers will upload PostfixEval and ParenMatch as support files.