# 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-to-Right (equal precedence)

Ignoring precedence, or with equal precedence, we evaluate from left to right. Infix "3 + 4 - 5 + 6" converts to the postfix "\_\_\_\_\_\_\_\_\_\_\_\_\_". 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

Next, we deal with precedence. We can’t simply go left-to-right on "3 + 4 \* 5". We must compare the operator on the top of the stack to the next operator. In this example, the stack has "+" and the next operator is "\*". Do not pop; instead push the "\*" and continue processing the string. The algorithm will eventually pop and append the "\*" and then the "+", resulting in the correct order "3 4 5 \* +".

Here you have a choice, to implement **one** of these boolean methods:

boolean isStrictlyLower(String next, String top)

boolean isHigherOrEqual(String top, String next)

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

int getLevel(String operator)

In the example above, isStrictlyLower("\*", "+") returns \_\_\_\_\_\_. Don’t pop. Push \_\_\_\_

In the example above, isHigherOrEqual("+", "\*") returns \_\_\_\_\_\_. Don’t pop. Push \_\_\_\_

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

Let’s try our new precedence methods on "3 \* 4 + 5".

In the new example, isStrictlyLower("+", "\*") returns \_\_\_\_\_\_. Pop and append \_\_\_\_

In the new example, isHigherOrEqual("\*", "+") returns \_\_\_\_\_\_. Pop and append \_\_\_\_

Trace the algorithm on "8 + 1 \* 2 - 9 / 3". Answer: "8 1 2 \* + 9 3 / -"

Obviously, your precedence method needs to handle all the possible combinations of operators. Start writing them here:

## 2c. Parentheses, etc. "([{<"

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

What do you for parentheses, etc? When you encounter a left parenthesis, push it. This begins the logical bottom of a stack within a stack. Continue processing as above. The left parenthesis remains on the stack until its matching right parenthesis is encountered, after which the left is popped.

Trace this example: "2 \* ( 3 + 5 \* 2 ) / 5" becomes "2 3 5 2 \* + \* 5 /"

## 2d. 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 might push it but only if one of three conditions is met:
* If we can’t push, pop the operator off the stack and append it to the postfix string. Continue popping until any one of the three conditions above is met. Push the current operator.
* 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

First, implement the algorithm above in the method infixToPostfix. You will need to implement one of the precedence methods. You may find it useful to call your PostfixEval.isOperator

Write the infix expressions you want to convert in a list in the main.

After the conversion works, call your working PostfixEval.eval on each postfix expression. Your final program should output all three columns, as shown in the test run.

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

## Test Run

Your final 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

## Infix to Postfix and Evaluate

## Extension (optional extra lab)

There is no shell. Copy your InfixPostfixEval into a brand-new lab and call it Infix\_Extension

First, for some points, improve infixToPostfix so that it produces the correct postfix expressions for powers and factorials. Now you have four levels of precedence, +-, \*/%, ^, and !. Here are two examples for ^ and !:

1 + 3 ! 🡪 1 3 ! +

1 - 2 ^ 3 🡪 1 2 3 ^ -

Second, for more points, improve infixToPostfix so that it 1) checks for matching parentheses and 2) checks for non-algebraic characters. Presumably you already have a working ParenMatch.checkParen(). You will have to make a new method to check for non-algebraic characters.

Each check, if they fail, will throw a new Exception consisting of the offending expression and a short error message. Here is an example of each:

3 + 2 > ERROR in parentheses  
 3 ? 2 ERROR non-arithmetic symbol

Because we aren’t using try-catch blocks, you will have to modify the header to throw the Exception, like this:

public static String infixToPostfix(String infix) throws Exception

If the infix expression passes both checks, produce the postfix expression and evaluate it. You already wrote a method in a different class that evaluates a postfix expression.

In the main() method, create your own test cases.

## Submit Infix\_Extension. The teachers will upload the ParenMatch and the PostfixEval classes as support files. The autograder will run your infixToPostfix method, not your main.

## Possible Sample Run—there is no shell

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  
1 - 2 ^ 3 1 2 3 ^ - -7.0  
1 + 3 ! 1 3 ! + 7.0  
2 \* 3 ! 2 3 ! \* 12.0  
1 + 3 ! \* 2 1 3 ! 2 \* + 13.0  
2 ^ 3 + 3 2 3 ^ 3 + 11.0  
java.lang.Exception: 3 + 2 ] ERROR in parentheses  
java.lang.Exception: 3 \* ( 4 + 5 ] ERROR in parentheses  
java.lang.Exception: ( 3 + 2 ERROR in parentheses  
java.lang.Exception: 3 + 2 > ERROR in parentheses  
java.lang.Exception: ( 3 + 2 ] ERROR in parentheses  
java.lang.Exception: 3 ? 2 ERROR non-arithmetic symbol  
java.lang.Exception: 3 @ 2 ERROR non-arithmetic symbol  
java.lang.Exception: 3 + 2 & 6 ERROR non-arithmetic symbol