1. Infix, Prefix and Postfix Expressions

When you write an arithmetic expression such as B \* C, the form of the expression provides you with information so that you can interpret it correctly. In this case we know that the variable B is being multiplied by the variable C since the multiplication operator \* appears between them in the expression. This type of notation is referred to as **infix** since the operator is *in between* the two operands that it is working on.

Consider another infix example, A + B \* C. The operators + and \* still appear between the operands, but there is a problem. Which operands do they work on? Does the + work on A and B or does the \* take B and C? The expression seems ambiguous.

In fact, you have been reading and writing these types of expressions for a long time and they do not cause you any problem. The reason for this is that you know something about the operators + and \*. Each operator has a **precedence** level. Operators of higher precedence are used before operators of lower precedence. The only thing that can change that order is the presence of parentheses. The precedence order for arithmetic operators places multiplication and division above addition and subtraction. If two operators of equal precedence appear, then a left-to-right ordering or associativity is used.

Let’s interpret the troublesome expression A + B \* C using operator precedence. B and C are multiplied first, and A is then added to that result. (A + B) \* C would force the addition of A and B to be done first before the multiplication. In expression A + B + C, by precedence (via associativity), the leftmost + would be done first.

Although all this may be obvious to you, remember that computers need to know exactly what operators to perform and in what order. One way to write an expression that guarantees there will be no confusion with respect to the order of operations is to create what is called a **fully parenthesized** expression. This type of expression uses one pair of parentheses for each operator. The parentheses dictate the order of operations; there is no ambiguity. There is also no need to remember any precedence rules.

The expression A + B \* C + D can be rewritten as ((A + (B \* C)) + D) to show that the multiplication happens first, followed by the leftmost addition. A + B + C + D can be written as (((A + B) + C) + D) since the addition operations associate from left to right.

There are two other very important expression formats that may not seem obvious to you at first. Consider the infix expression A + B. What would happen if we moved the operator before the two operands? The resulting expression would be + A B. Likewise, we could move the operator to the end. We would get A B +. These look a bit strange.

These changes to the position of the operator with respect to the operands create two new expression formats, **prefix** and **postfix**. Prefix expression notation requires that all operators precede the two operands that they work on. Postfix, on the other hand, requires that its operators come after the corresponding operands. A few more examples should help to make this a bit clearer (see **Table 1**).

A + B \* C would be written as + A \* B C in prefix. The multiplication operator comes immediately before the operands B and C, denoting that \* has precedence over +. The addition operator then appears before the A and the result of the multiplication.

In postfix, the expression would be A B C \* +. Again, the order of operations is preserved since the \* appears immediately after the B and the C, denoting that \* has precedence, with + coming after. Although the operators moved and now appear either before or after their respective operands, the order of the operands stayed exactly the same relative to one another.

| **Infix Expression** | **Prefix Expression** | **Postfix Expression** |
| --- | --- | --- |
| A + B | + A B | A B + |
| A + B \* C | + A \* B C | A B C \* + |

**Table 1**

Now consider the infix expression (A + B) \* C. Recall that in this case, infix requires the parentheses to force the performance of the addition before the multiplication. However, when A + B was written in prefix, the addition operator was simply moved before the operands, + A B. The result of this operation becomes the first operand for the multiplication. The multiplication operator is moved in front of the entire expression, giving us \* + A B C. Likewise, in postfix A B + forces the addition to happen first. The multiplication can be done to that result and the remaining operand C. The proper postfix expression is then A B + C \*.

Consider these three expressions again (see **Table 2**). Something very important has happened. Where did the parentheses go? Why don’t we need them in prefix and postfix? The answer is that the operators are no longer ambiguous with respect to the operands that they work on. Only infix notation requires the additional symbols. The order of operations within prefix and postfix expressions is completely determined by the position of the operator and nothing else. In many ways, this makes infix the least desirable notation to use.

| **Infix Expression** | **Prefix Expression** | **Postfix Expression** |
| --- | --- | --- |
| (A + B) \* C | \* + A B C | A B + C \* |
| **Table 2** |  |  |

**Table 3** shows some additional examples of infix expressions and the equivalent prefix and postfix expressions. Be sure that you understand how they are equivalent in terms of the order of the operations being performed.

| **Infix Expression** | **Prefix Expression** | **Postfix Expression** |
| --- | --- | --- |
| A + B \* C + D | + + A \* B C D | A B C \* + D + |
| (A + B) \* (C + D) | \* + A B + C D | A B + C D + \* |
| A \* B + C \* D | + \* A B \* C D | A B \* C D \* + |
| A + B + C + D | + + + A B C D | A B + C + D + |
| **Table 3** |  |  |

2. Conversion of Infix Expressions to Prefix and Postfix

So far, we have used ad hoc methods to convert between infix expressions and the equivalent prefix and postfix expression notations. As you might expect, there are algorithmic ways to perform the conversion that allow any expression of any complexity to be correctly transformed.

The first technique that we will consider uses the notion of a fully parenthesized expression that was discussed earlier. Recall that A + B \* C can be written as (A + (B \* C)) to show explicitly that the multiplication has precedence over the addition. On closer observation, however, you can see that each parenthesis pair also denotes the beginning and the end of an operand pair with the corresponding operator in the middle.

Look at the right parenthesis in the subexpression (B \* C) above. If we were to move the multiplication symbol to that position and remove the matching left parenthesis, giving us B C \*, we would in effect have converted the subexpression to postfix notation. If the addition operator were also moved to its corresponding right parenthesis position and the matching left parenthesis were removed, the complete postfix expression would result.



If we do the same thing but instead of moving the symbol to the position of the right parenthesis, we move it to the left, we get prefix notation. The position of the parenthesis pair is actually a clue to the final position of the enclosed operator.



So in order to convert an expression, no matter how complex, to either prefix or postfix notation, fully parenthesize the expression using the order of operations. Then move the enclosed operator to the position of either the left or the right parenthesis depending on whether you want prefix or postfix notation.

Here is a more complex expression: (A + B) \* C - (D - E) \* (F + G). [Next](http://interactivepython.org/runestone/static/pythonds/BasicDS/InfixPrefixandPostfixExpressions.html#fig-complexmove) figure shows the conversion to postfix and prefix notations.



3. General Infix-to-Postfix Conversion

We need to develop an algorithm to convert any infix expression to a postfix expression. To do this we will look closer at the conversion process.

Consider once again the expression A + B \* C. As shown above, A B C \* + is the postfix equivalent. We have already noted that the operands A, B, and C stay in their relative positions. It is only the operators that change position. Let’s look again at the operators in the infix expression. The first operator that appears from left to right is +. However, in the postfix expression, + is at the end since the next operator, \*, has precedence over addition. The order of the operators in the original expression is reversed in the resulting postfix expression.

As we process the expression, the operators have to be saved somewhere since their corresponding right operands are not seen yet. Also, the order of these saved operators may need to be reversed due to their precedence. This is the case with the addition and the multiplication in this example. Since the addition operator comes before the multiplication operator and has lower precedence, it needs to appear after the multiplication operator is used. Because of this reversal of order, it makes sense to consider using a stack to keep the operators until they are needed.

What about (A + B) \* C? Recall that A B + C \* is the postfix equivalent. Again, processing this infix expression from left to right, we see + first. In this case, when we see \*, + has already been placed in the result expression because it has precedence over \* by virtue of the parentheses. We can now start to see how the conversion algorithm will work. When we see a left parenthesis, we will save it to denote that another operator of high precedence will be coming. That operator will need to wait until the corresponding right parenthesis appears to denote its position (recall the fully parenthesized technique). When that right parenthesis does appear, the operator can be popped from the stack.

As we scan the infix expression from left to right, we will use a stack to keep the operators. This will provide the reversal that we noted in the first example. The top of the stack will always be the most recently saved operator. Whenever we read a new operator, we will need to consider how that operator compares in precedence with the operators, if any, already on the stack.

Assume the infix expression is a string of tokens delimited by spaces. The operator tokens are \*, /, +, and -, along with the left and right parentheses, ( and ). The operand tokens are the single-character identifiers A, B, C, and so on. The following steps will produce a string of tokens in postfix order.

**Algorithm**

1. Create an empty stack called stack for keeping operators. Create an empty String for output.

2. Convert the input infix string to an array by using the string method split.

3. Scan the token list from left to right.

4. If the token is an operand, output it with space at the end.

5. Else,

…..5.1 If the precedence of the scanned operator is greater than the precedence of the operator in the stack(or the stack is empty or the stack contains a "(" ), push it.

…..5.2 Else, Pop all the operators from the stack which are greater than or equal to in precedence than that of the scanned operator. After doing that Push the scanned operator to the stack. (If you encounter parenthesis while popping then stop there and push the scanned operator in the stack.)

6. If the scanned character is an "(", push it to the stack.

7. If the scanned character is an ")", pop the stack and and output it until a "(" is encountered, and discard both the parenthesis.

8. Repeat steps 2-6 until infix expression is scanned.

9. Print the output

10. Pop and output from the stack until it is not empty.

[next](http://interactivepython.org/runestone/static/pythonds/BasicDS/InfixPrefixandPostfixExpressions.html#fig-intopost) Figure shows the conversion algorithm working on the expression A \* B + C \* D. Note that the first \* operator is removed upon seeing the + operator. Also, + stays on the stack when the second \* occurs, since multiplication has precedence over addition. At the end of the infix expression the stack is popped twice, removing both operators and placing + as the last operator in the postfix expression.



**/\***

**Algorithm**

**1. Create an empty stack called stack for keeping operators. Create an empty String for output.**

**2. Convert the input infix string to an array by using the string method split.**

**3. Scan the token list from left to right.**

**4. If the token is an operand, output it with space at the end.**

**5. Else,**

**…..5.1 If the precedence of the scanned operator is greater than the precedence of the operator in the stack(or the stack is empty or the stack contains a "(" ), push it.**

**…..5.2 Else, Pop all the operators from the stack which are greater than or equal to in precedence than that of the scanned operator. After doing that Push the scanned operator to the stack. (If you encounter parenthesis while popping then stop there and push the scanned operator in the stack.)**

**6. If the scanned character is an "(", push it to the stack.**

**7. If the scanned character is an ")", pop the stack and and output it until a "(" is encountered, and discard both the parenthesis.**

**8. Repeat steps 2-6 until infix expression is scanned.**

**9. Print the output**

**10. Pop and output from the stack until it is not empty.**

**\*/**

**import java.util.Stack;**

**import java.util.Scanner;**

**class TestEvalNum{**

 **// A utility function to return precedence of a given operator**

 **// Higher returned value means higher precedence**

 **static int prec(String str) {**

 **char ch=str.charAt(0); // because switch doesn't work with Strings**

 **switch (ch) {**

 **case '+':**

 **case '-':**

 **return 1;**

 **case '\*':**

 **case '/':**

 **return 2;**

 **case '^':**

 **return 3;**

 **}**

 **return -1;**

 **}**

 **// if val is not an operator ^\*/+- or a bracket)( - it is an operand**

 **static boolean isOperand(String val){**

 **return ("^\*/+-)(".indexOf(val)==-1);**

 **}**

 **// The main method that converts given infix expression**

 **// to postfix expression.**

 **static String infixToPostfix(String exp) {**

 **// initializing empty String for result**

 **String result ="";**

 **// initializing empty stack**

 **Stack<String> stack = new Stack<>();**

 **String arr[]= exp.split(" ");**

 **for (int i = 0; i<arr.length; i++){**

 **String curr = arr[i];**

 **//If the scanned token is an operand, add it to output**

 **//with a space at the end**

 **if (isOperand(curr))**

 **result += curr+" ";**

 **// If the scanned token is an "(", push it to the stack.**

 **else if ( curr.equals("(") )**

 **stack.push(curr);**

 **//If the scanned character is an ")" pop and output from the**

 **//stack until an "(" is encountered.**

 **else if (curr.equals(")")) {**

 **while (!stack.isEmpty() && !stack.peek().equals("("))**

 **result += stack.pop()+" ";**

 **if (!stack.isEmpty() && !stack.peek().equals("("))**

 **return "Invalid Expression"; // invalid expression**

 **else**

 **stack.pop();**

 **}**

 **else { // an operator is encountered**

 **while (!stack.isEmpty() && prec(curr) <= prec(stack.peek()))**

 **result += stack.pop()+" ";**

 **stack.push(curr);**

 **}**

 **}**

 **// pop all the operators from the stack**

 **while (!stack.isEmpty())**

 **result += stack.pop()+" ";**

 **return result;**

 **}**

 **// main method**

 **public static void main(String[] args) {**

 **String exp ="";**

 **// exp = "a + b \* ( c ^ d - e ) ^ ( f + g \* h ) - i";**

 **exp = "5 + 6 \* ( 17 ^ 25 - 8 ) ^ ( 1 + 4 \* 23 ) - 9";**

 **System.out.println("Original expression: " + exp);**

 **System.out.println("Post-fix expression: " + infixToPostfix(exp));**

 **}**

**}**

**Class work Exercises:**

I. Convert the following to Postfix

a) A + B – C \* D

b) ( A \* ( B + C ) ) – ( E + G )

c) A + B / ( C – D \* E )

II. Evaluate the following Postfix expressions

(Assume A = 7, B = 4, C = 3, D = -2)

a) A B + C / D \*

b) A B C D + / \*

c) A B – C – D -

d) A B C D + + -

III. Convert the following from Postfix to Infix

a) A B C + - D \*

b) A B + C – D E \* /

IV. Convert the problems from problem I to PREFIX

a) A + B – C \* D

b) ( A \* ( B + C ) ) – ( E + G )

c) A + B / ( C – D \* E )

**Answer Key**

I. a) A B + C D \* - b) A B C + \* E G + - c) A B C D E \* - / +

II. a) -6 b) 28 c) 2 d) 2

III. a) ( A – ( B + C ) ) \* D b) ( A + B – C ) / ( D \* E )

IV. a) - + A B \* C D b) - \* A + B C + E G c) + A / B – C \* D E

**Homework Exercices :**

Infix Expression: ( AX + ( B \* C ) ) ;

Postfix Expression:

Prefix Expression:

Infix Expression: ( ( AX + ( B \* CY ) ) / ( D E) ) ;

Postfix Expression:

Prefix Expression:

Infix Expression: ( ( A + B ) \* ( C + E ) ) ;

Postfix Expression:

Prefix Expression:

Infix Expression: ( AX \* ( BX \* ( ( ( CY + AY ) + BY ) \* CX ) ) ) ;

Postfix Expression:

Prefix Expression:

Infix Expression: ( ( H \* ( ( ( ( A + ( ( B + C ) \* D ) ) \* F ) \* G ) \* E ) ) + J ) ;

Postfix Expression:

Prefix Expression:

**Solutions**

Infix Expression: ( AX + ( B \* C ) ) ;

Postfix Expression: AX B C \* +

Prefix Expression: + AX \* B C

Infix Expression: ( ( AX + ( B \* CY ) ) / ( D E) ) ;

Postfix Expression: AX B CY \* + D E /

Prefix Expression: / + AX \* B CY D E

Infix Expression: ( ( A + B ) \* ( C + E ) ) ;

Postfix Expression: A B + C E + \*

Prefix Expression: \* + A B + C E

Infix Expression: ( AX \* ( BX \* ( ( ( CY + AY ) + BY ) \* CX ) ) ) ;

Postfix Expression: AX BX CY AY + BY + CX \* \* \*

Prefix Expression: \* AX \* BX \* + + CY AY BY CX

Infix Expression: ( ( H \* ( ( ( ( A + ( ( B + C ) \* D ) ) \* F ) \* G ) \* E ) ) + J ) ;

Postfix Expression: H A B C + D \* + F \* G \* E \* \* J +

Prefix Expression: + \* H \* \* \* + A \* + B C D F G E J