Evaluation of Prefix

In this article, we will discuss how to evaluate an expression written in prefix notation. The method is similar to evaluating a postfix expression. Please read Evaluation of Postfix Expression article to know how to evaluate postfix expressions

Algorithm:

```
EVALUATE_PREFIX(STRING)
Step 1: Put a pointer P at the end of the end
Step 2: If character at P is an operand push it to Stack
Step 3: If the character at P is an operator pop two
        elements from the Stack. Operate on these elements
        according to the operator, and push the result
        back to the Stack
Step 4: Decrement P by 1 and go to Step 2 as long as there
        are characters left to be scanned in the expression.
Step 5: The Result is stored at the top of the Stack,
        return it
Step 6: End
```

Example to demonstrate working of the algorithm

```
Expression: +9*26

Character | Stack       |  Explanation
```

```
Scanned   | (Front to   |
          |  Back)      |
-----------------------------------------
6           6             6 is an operand,
                            push to Stack
2           6 2           2 is an operand,
                            push to Stack
*           12 (6*2)      * is an operator,
                          pop 6 and 2, multiply
                          them and push result
                          to Stack
9           12 9          9 is an operand, push
                          to Stack
+           21 (12+9)     + is an operator, pop
                          12 and 9 add them and
                          push result to Stack

Result: 21
```

Examples:

```
Input : -+8/632
Output : 8

Input : -+7*45+20
Output : 25
```

Complexity The algorithm has linear complexity since we scan the expression once and perform at most O(N) push and pop operations which take constant time.
Implementation of the algorithm is given below.

Dash
All
Articles
Videos
Problems
Quiz

Implementation:

C++ Java

```
// Java program to evaluate
// a prefix expression.
import java.io.*;
import java.util.*;

class GFG {

    static Boolean isOperand(char c)
    {
        // If the character is a digit
        // then it must be an operand
        if (c >= 48 && c <= 57)
            return true;
        else
            return false;
    }

    static double evaluatePrefix(String exprsn)
    {
        Stack<Double> Stack = new Stack<Double>();

        for (int j = exprsn.length() - 1; j >= 0; j--) {

            // Push operand to Stack
```

```
// To convert exprsn[j] to digit subtract
// '0' from exprsn[j].
if (isOperand(exprsn.charAt(j)))
    Stack.push((double)(exprsn.charAt(j) - 48));

else {

    // Operator encountered
    // Pop two elements from Stack
    double o1 = Stack.peek();
    Stack.pop();
    double o2 = Stack.peek();
    Stack.pop();

    // Use switch case to operate on o1
    // and o2 and perform o1 O o2.
    switch (exprsn.charAt(j)) {
    case '+':
        Stack.push(o1 + o2);
        break;
    case '-':
        Stack.push(o1 - o2);
        break;
    case '*':
        Stack.push(o1 * o2);
        break;
    case '/':
        Stack.push(o1 / o2);
        break;
```

```
                }
            }
        }

        return Stack.peek();
    }

    /* Driver program to test above function */
    public static void main(String[] args)
    {
        String exprsn = "+9*26";
        System.out.println(evaluatePrefix(exprsn));
    }
}
```



```
21
```

Note:
To perform more types of operations only the switch case table needs to be modified. This implementation works only for single digit operands. Multi-digit operands can be implemented if some character-like space is used to separate the operands and operators.

Below given is the extended program which allows operands to have multiple digits.

C++ Java

```java
// Java program to evaluate a prefix expression.
import java.util.*;
public class Main
{
    static boolean isdigit(char ch)
    {
        if(ch >= 48 && ch <= 57)
        {
            return true;
        }
        return false;
    }

    static double evaluatePrefix(String exprsn)
    {
        Stack<Double> stack = new Stack<Double>();

        for (int j = exprsn.length() - 1; j >= 0; j--) {

            // if jth character is the delimiter ( which is
            // space in this case) then skip it
            if (exprsn.charAt(j) == ' ')
                continue;

            // Push operand to Stack
            // To convert exprsn[j] to digit subtract
            // '0' from exprsn[j].
            if (isdigit(exprsn.charAt(j))) {
```

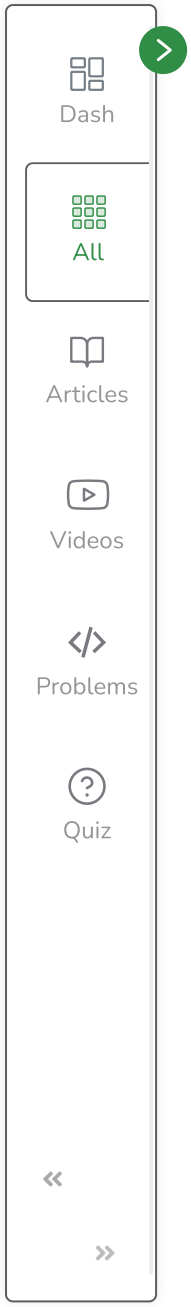
```
        // there may be more than
        // one digits in a number
        double num = 0, i = j;
        while (j < exprsn.length() && isdigit(exprsn.charAt(j)))
            j--;
        j++;

        // from [j, i] exprsn contains a number
        for (int k = j; k <= i; k++)
        {
            num = num * 10 + (double)(exprsn.charAt(k) - '0');
        }

        stack.push(num);
    }
    else {

        // Operator encountered
        // Pop two elements from Stack
        double o1 = (double)stack.peek();
        stack.pop();
        double o2 = (double)stack.peek();
        stack.pop();

        // Use switch case to operate on o1
        // and o2 and perform o1 O o2.
        switch (exprsn.charAt(j)) {
        case '+':
```

```
                    stack.push(o1 + o2);
                    break;
                case '-':
                    stack.push(o1 - o2);
                    break;
                case '*':
                    stack.push(o1 * o2);
                    break;
                case '/':
                    stack.push(o1 / o2);
                    break;
                }
            }
        }

        return stack.peek();
    }

  // Driver code
    public static void main(String[] args) {
        String exprsn = "+ 9 * 12 6";
        System.out.print((int)evaluatePrefix(exprsn));
    }
}
```

Output

```
81
```

Time Complexity: O(n)
Space Complexity: O(n)



Mark as Read

Report An Issue

If you are facing any issue on this page. Please let us know.