

DATA STRUCTURES

AND ALGORITHMS

Challenge 1

GROUP 11508289 | ARITHMETIC EXPRESSION CALCULATION | 01/11/2020

# **INTRODUCTION**

## Group ID: **11508289**

## Option: **Input** **expressions contain both integers and floating-point numbers. The group has done the calculate and transfer the expression.**

|  |  |
| --- | --- |
| Name | ID |
| La Ngọc Hồng Phúc | 19127511 |
| Hồ Lâm Bảo Khuyên | 19127289 |
| Nguyễn Tất Trường | 19127082 |
| Trần Quốc Tuấn | 19127650 |

# Research

Overall

The way to write expression is known as notation. There are three ways to demonstrate  
an arithmetic expression:

1. Infix notation
2. Prefix – Polish notation
3. Postfix – Reverse Polish notation

Three of these have differences in syntax. Because of the operator’s precedence, the infix notation is not suitable for computer programming, so we have to find a way to convert mathematical problems to the others.

**Infix notation:** This expression is easy to understand. The operator is between two operands.  
*e.g.* A+B, C^D.

**Prefix notation:** In this expression, the operator is laid on the left side of two operands.  
*e.g.* +AB, ^CD

**Postfix notation:** This is a reverse model of the prefix notation; the operator is written after the operands.  
*e.g.* AB+, CD^

**Precedence level of operators:** This is a descending list.

|  |  |
| --- | --- |
| Operator description | operator |
| Curly brackets | {} |
| Square brackets | [] |
| Round brackets | () |
| Power | ^ |
| Multiplication, division, power | \*, / |
| Addition, subtraction | + , - |

## Converting an infix expression to a prefix expression

(A + B) \* (C + D) => \* + A B + C D

### Illustrate the algorithm:

Scan the reverse input string from right to left until the stack is empty.  
We will use a stack variable to store operators and a string to store the output.  
Put the close bracket at the top of the stack and open bracket at the end of the input.  
If we get an operand, it will be added to the output.  
If we encounter an operator, then:  
We will repeatedly pop from the stack variable the operator which has higher or the same precedence level and add it to the output   
Add the scanned operator to the stack.  
If we encounter the open operator, then:  
We pop elements on the top of the stack until we encounter the close bracket and remove the scanned element.

e.g.: convert (E-F) / (B\*A) to prefix expression:  
Reverse input: )A\*B(/)F-E((

|  |  |  |
| --- | --- | --- |
| Scanned | Stack | Output |
|  | ) |  |
| ) | )) |  |
| A | )) | A |
| \* | ))\* | A |
| B | ))\* | A B |
| ( | ) | A B \* |
| / | ) / | A B \* |
| ) | ) / ) | A B \* |
| F | ) / ) | A B \* F |
| - | ) / ) - | A B \* F |
| E | ) / ) - | A B \* F E |
| ( | ) / | A B \* F E - |
| ( | ) / | A B \* F E - |
| empty | empty | end |

Reverse again the output, we get: / - E F \* A B

// illustrate the code fragment

## Converting an infix expression to a postfix expression

(A + B) \* (C + D) => A B + C D + \*

### Illustrate the algorithm:

Scan the string from left to right to catch operators and operands until the stack is empty.  
We will use a stack variable to store operators and a string to store operands  
Put the open bracket at the top of the stack and the close bracket at the end of the input string to check if we have scanned correctly.  
If we encounter a left parenthesis, add it to the stack variable.  
If we catch an operand, it will be added to the output string.  
If we catch an operator, then the precedence level will be checked:  
If on the top of the stack, there is a higher or the same level operator then we pop that operator from the stack and add it to output string.  
Else add the caught operator to the stack.  
If we got a right parenthesis:  
we will pop the stack until the open bracket is caught and remove it. The operator got from stack is added to the output. After checking the stack, we delete the scanned bracket.  
Finish the loop, we got the postfix expression.

e.g. This is an example: convert ( A + B ) \* ( C – D ) to postfix expression

|  |  |  |
| --- | --- | --- |
| Scanned | Stack | Output |
|  | ( |  |
| ( | (( |  |
| A | (( | A |
| + | ((+ | A |
| B | ((+ | A B |
| ) | ( | A B + |
| \* | (\* | A B + |
| ( | (\*( | A B + |
| C | (\*( | A B + C |
| - | (\*(- | A B + C |
| D | (\*(- | A B + C D |
| ) | (\* | A B + C D - \* |
| empty | empty | end |

The result: A B + C D -\*

the illustration for the code fragment

string infixToPostfix(string str)

{

str = '(' + str + ')';

stack<char> char\_stack;

string output;

for (int i = 0; i < str.size();; i++) {

if str[i] is operand, add str[i] to output

// If the scanned character is an

// ‘(‘, push it to the stack.

else if str[i] is open bracket, push to char\_stack

// If the scanned character is an

// ‘)’, pop and output from the stack

// until an ‘(‘ is encountered.

else if str[i] is close bracket {

while (char\_stack.top() != '(') {

output += char\_stack.top();

char\_stack.pop();

}

// Remove '(' from the stack

char\_stack.pop();

}

// Operator found

else {

if (isOperator(char\_stack.top())) {

while (getPriority(str[i])

<= getPriority(char\_stack.top())) {

output += char\_stack.top();

char\_stack.pop();

}

// Push current Operator on stack

char\_stack.push(str[i]);

}

}

}

return output;

}

## Converting a prefix expression to a postfix expression

\* + A B + C D => A B + C D + \*

Illustrate the algorithm:

The input from right to left until the input is empty.  
We will use a stack variable and a temporary string for this convert.  
If we get an operand from scanning, it will be pushed to the stack.  
If we encounter an operator, then:  
Pop from stack two operands called op1 and op2 correspondingly.  
Using a temporary string to connect the scanned operator and two operands as structure: op1 + op2 + operator.  
Then push that string to the stack, it will become a new operand.  
Finally, when the input is empty, we initialize a new string and pop everything from the stack so that we get a postfix expression.

*e.g.:* convert \*-A/EF+\*CDG

|  |  |  |
| --- | --- | --- |
| Scanned | stack | description |
| G | G |  |
| D | GD |  |
| C | GDC |  |
| \* | G CD\* | String: C D \* and push back to the stack |
| + | CD\* G + | String: CD\* G + |
| F | CD\* G + F |  |
| E | CD\* G + F E |  |
| / | CD\* G + E F / | String: E F / |
| A | CD\* G + E F / A |  |
| - | CD\* G + A E F / - | String: A E F / - |
| \* | A E F / - CD\* G + \* | String: A E F / - CD\* G + \* |
| Empty | A E F / - CD\* G + \* |  |

Push everything from to stack to a new string we got the result: A E F / - CD\* G + \*

The code illustration:

*Reference*

[*https://scanftree.com/Data\_Structure/infix-to-prefix*](https://scanftree.com/Data_Structure/infix-to-prefix)

[*https://www.includehelp.com/c/infix-to-postfix-conversion-using-stack-with-c-program.aspx*](https://www.includehelp.com/c/infix-to-postfix-conversion-using-stack-with-c-program.aspx)

[*https://www.geeksforgeeks.org/prefix-postfix-conversion/*](https://www.geeksforgeeks.org/prefix-postfix-conversion/)