



**Green University of Bangladesh**  
**Department of Computer Science and Engineering (CSE)**  
**Faculty of Sciences and Engineering**  
**Semester: (Spring, Year:2024), B.Sc. in CSE (Day)**

**Lab Report # 04**  
**Course Title: Compiler Lab**

**Course Code: CSE 306      Section: 222 D2**

**Lab Experiment Name:** To check whether a mathematical statement solvable or not.

**Student Details**

| Name |                   | ID        |
|------|-------------------|-----------|
|      | Md. Moshir Rahman | 221902324 |

**Submission Date** : 22/03/2024  
**Course Teacher's Name** : Tasnim Tayiba Zannat

|                                 |                         |
|---------------------------------|-------------------------|
| <b><u>Lab Report Status</u></b> |                         |
| <b>Marks:</b> .....             | <b>Signature:</b> ..... |
| <b>Comments:</b> .....          | <b>Date:</b> .....      |

## 1. EXPERIMENT NAME:

- A) Write a program to convert  $a+(b-c)$  to **prefix** operation.
- B) Write a program to convert  $a+(b-c)$  to **postfix** operation.

## 2. OBJECTIVES

- ★ To understand the concepts of prefix (Polish) and postfix (Reverse Polish) notation for mathematical expressions.
- ★ To develop algorithms for converting infix expressions to their equivalent prefix and postfix forms.

## 3. INTRODUCTION

In mathematics and computer science, infix notation is the standard way of writing mathematical expressions, where operators are placed between operands (e.g.,  $a + b$ ). Prefix notation and postfix notation (Reverse Polish notation) are alternative notations where the operator precedes or follows the operands, respectively. Converting expressions from infix to prefix/postfix notation is an important step in compiler design and evaluation of mathematical expressions by computers. It helps in unambiguous interpretation and efficient evaluation of expressions.

## 4. ALGORITHM

### A)

1. Create a stack to store operators and an empty string to store the prefix expression.
2. Traverse the infix expression from right to left.
3. If the current character is an operand (alphabet or number), append it to the prefix expression string.
4. If the current character is an opening parenthesis '(', push it onto the stack. If the current character is a closing parenthesis ')', pop operators from the stack and append them to the prefix expression string until an opening parenthesis '(' is encountered. Discard the opening and closing parentheses.
5. If the current character is an operator, pop operators from the stack with higher or equal precedence and append them to the prefix expression string. Then, push the current operator onto the stack.
6. After traversing the entire infix expression, pop any remaining operators from the stack and append them to the prefix expression string. Reverse the prefix expression string to get the final prefix notation.

## B)

1. Create a stack to store operators and an empty string to store the **postfix** expression.
2. Traverse the infix expression from left to right.
3. If the current character is an operand (alphabet or number), append it to the postfix expression string.
4. If the current character is an opening parenthesis '(', push it onto the stack.
5. If the current character is a closing parenthesis ')', pop operators from the stack and append them to the postfix expression string until an opening parenthesis '(' is encountered. Discard the opening and closing parentheses.
6. If the current character is an operator, pop operators from the stack with higher or equal precedence and append them to the postfix expression string. Then, push the current operator onto the stack.
7. After traversing the entire infix expression, pop any remaining operators from the stack and append them to the postfix expression string.

## 5. PROCEDURE

### Implementation:

#### A) convert a+(b-c) to prefix operation

```
#include <stdio.h>
```

```
#include <stdlib.h>
```

```
#include <string.h>
```

```
#define MAX_SIZE 100
```

```
struct Stack {  
    int top;  
    unsigned capacity;  
    char* array;  
};
```

```
struct Stack* createStack(unsigned capacity) {  
    struct Stack* stack = (struct Stack*)malloc(sizeof(struct Stack));  
    stack->capacity = capacity;  
    stack->top = -1;  
    stack->array = (char*)malloc(stack->capacity * sizeof(char));  
    return stack;  
}
```

```
int isFull(struct Stack* stack) {
    return stack->top == stack->capacity - 1;
}
```

```
int isEmpty(struct Stack* stack) {
    return stack->top == -1;
}
```

```
void push(struct Stack* stack, char item) {
    if (isFull(stack))
        return;
    stack->array[++stack->top] = item;
}
```

```
char pop(struct Stack* stack) {
    if (isEmpty(stack))
        return '\0';
    return stack->array[stack->top--];
}
```

```
int precedence(char op) {
    switch (op) {
        case '+':
        case '-':
            return 1;
        case '*':
        case '/':
            return 2;
    }
    return -1;
}
```

```
void reverseString(char* str) {
    int length = strlen(str);
    int i, j;
```

```

    for (i = 0, j = length - 1; i < j; i++, j--) {
        char temp = str[i];
        str[i] = str[j];
        str[j] = temp;
    }
}

```

```

void infixToPrefix(char* infix, char* prefix) {
    struct Stack* stack = createStack(strlen(infix) + 1);
    int i, j = 0;

    reverseString(infix);
    for (i = 0; infix[i]; ++i) {
        if ((infix[i] >= 'a' && infix[i] <= 'z') || (infix[i] >= 'A' && infix[i] <= 'Z')) {
            prefix[j++] = infix[i];
        }
        else if (infix[i] == ')') {
            push(stack, infix[i]);
        }
        else if (infix[i] == '+' || infix[i] == '-' || infix[i] == '*' || infix[i] == '/') {
            while (!isEmpty(stack) && precedence(stack->array[stack->top]) >
precedence(infix[i])) {
                prefix[j++] = pop(stack);
            }
            push(stack, infix[i]);
        }
        else if (infix[i] == '(') {
            while (!isEmpty(stack) && stack->array[stack->top] != ')') {
                prefix[j++] = pop(stack);
            }
            if (!isEmpty(stack) && stack->array[stack->top] == ')') {
                pop(stack);
            }
        }
        while (!isEmpty(stack)) {
            prefix[j++] = pop(stack);
        }
        prefix[j] = '\0';
        reverseString(prefix);
    }
}

```

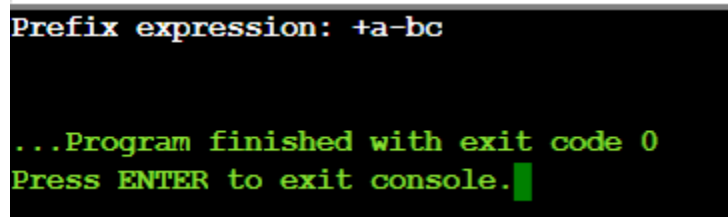
```

}

int main() {
    char infix[MAX_SIZE] = "a+(b-c)";
    char prefix[MAX_SIZE];
    infixToPrefix(infix, prefix);
    printf("Prefix expression: %s\n", prefix);
    return 0;
}

```

### Output:



```

Prefix expression: +a-bc

...Program finished with exit code 0
Press ENTER to exit console.

```

### **B) Convert a+(b-c) to postfix operation.**

#### **Implementation in C:**

```

#include <stdio.h>
#include <stdlib.h>
#include <string.h>

#define MAX_SIZE 100

struct Stack {
    int top;
    unsigned capacity;
    char* array;
};

struct Stack* createStack(unsigned capacity) {

```

```

    struct Stack* stack = (struct Stack*)malloc(sizeof(struct Stack));
    stack->capacity = capacity;
    stack->top = -1;
    stack->array = (char*)malloc(stack->capacity * sizeof(char));
    return stack;
}

int isFull(struct Stack* stack) {
    return stack->top == stack->capacity - 1;
}

int isEmpty(struct Stack* stack) {
    return stack->top == -1;
}

void push(struct Stack* stack, char item) {
    if (isFull(stack))
        return;
    stack->array[++stack->top] = item;
}

char pop(struct Stack* stack) {
    if (isEmpty(stack))
        return '\0';
    return stack->array[stack->top--];
}

int precedence(char op) {
    switch (op) {
        case '+':
        case '-':
            return 1;
        case '*':
        case '/':
            return 2;
    }
    return -1;
}

```

```

void infixToPostfix(char* infix, char* postfix) {
    struct Stack* stack = createStack(strlen(infix) + 1);
    int i, j = 0;

    for (i = 0; infix[i]; ++i) {

        if ((infix[i] >= 'a' && infix[i] <= 'z') || (infix[i] >= 'A' && infix[i] <= 'Z')) {
            postfix[j++] = infix[i];
        }
        else if (infix[i] == '(') {
            push(stack, infix[i]);
        }

        else if (infix[i] == ')') {

            while (!isEmpty(stack) && stack->array[stack->top] != '(') {
                postfix[j++] = pop(stack);
            }

            if (!isEmpty(stack) && stack->array[stack->top] == '(') {
                pop(stack);
            }

        }

        else {
            while (!isEmpty(stack) && precedence(stack->array[stack->top]) >=
precedence(infix[i])) {
                postfix[j++] = pop(stack);
            }
            push(stack, infix[i]);
        }

        while (!isEmpty(stack)) {
            postfix[j++] = pop(stack);
        }
        postfix[j] = '\0';
    }
}

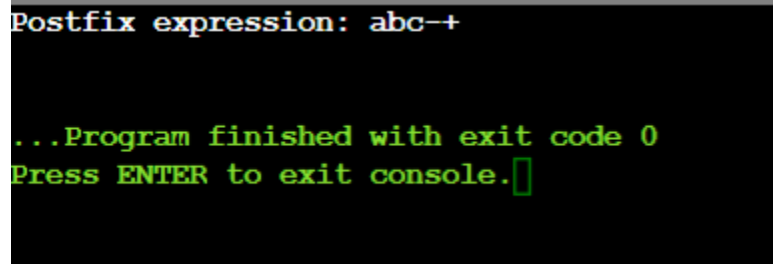
int main() {
    char infix[MAX_SIZE] = "a+(b-c)";

```



```
char postfix[MAX_SIZE];  
infixToPostfix(infix, postfix);  
printf("Postfix expression: %s\n", postfix);  
return 0;  
}
```

**Output:**



```
Postfix expression: abc-+  
  
...Program finished with exit code 0  
Press ENTER to exit console.█
```

## 6. ANALYSIS AND DISCUSSION

The algorithms follow systematic approaches to convert infix expressions to prefix and postfix notations, respectively. They use a stack to keep track of operators and their precedence. By traversing the expression in the specified order and applying the given rules, the operators and operands are rearranged to obtain the desired notation.

## 7. SUMMARY

These algorithms and programs demonstrate the conversion of infix expressions to prefix and postfix notations, which are fundamental concepts in compiler design and expression evaluation. They lay the groundwork for more advanced topics in compilers and programming language theory.