**EXP- 10**

**Intermediate code generation-Postfix and Prefix**

**RA1911026010045 K1**

**Postfix Expression**

**Aim:** To generate postfix expression of a given expression.

**Algorithm:**

* Create a stack.
* For each character c in the input stream:

If c is an operand

{

Output c

}

Else if c is a right parentheses

{

Pop and output tokens until a left parentheses is popped

}

Else

{ // c is an operator or left parentheses

Pop and output tokens until one of the lower priorities than c

are encountered, or a left parentheses is encountered, or the stack is empty.

Push c

}

* Pop and output tokens until the stack is empty.

**CODE:**

#include<bits/stdc++.h>

using namespace std;

//precedence of operators

int precedence(char ch)

{

if(ch == '^')

return 3;

else if(ch == '/' || ch =='\*')

return 2;

else if(ch == '+' || ch == '-')

return 1;

else

return -1;

}

string infixToPostfix(string s)

{

stack<char> st;

string postfix\_exp;

for(int i = 0; i < s.length(); i++)

{

char ch = s[i];

// If the input character is

an operand, add it to the postfix output string.

if((ch >= 'a' && ch <= 'z') || (ch >= 'A' && ch <= 'Z') || (ch >= '0' && ch <= '9'))

postfix\_exp += ch;

// If the input character is an

'(', push it to the stack.

else if(ch == '(')

st.push('(');

// If the input character is an ')',

pop and output string from the stack

until an '(' is encountered.

else if(ch == ')') {

while(st.top() != '(')

{

postfix\_exp += st.top();

st.pop();

}

st.pop();

}

//If the character is an operator

else

{

while(!st.empty() && precedence(s[i]) <= precedence(st.top()))

{

postfix\_exp += st.top();

st.pop();

}

st.push(ch);

}

}

// Pop all the remaining elements from the stack

while(!st.empty())

{

postfix\_exp += st.top();

st.pop();

}

return postfix\_exp;

}

int main()

{

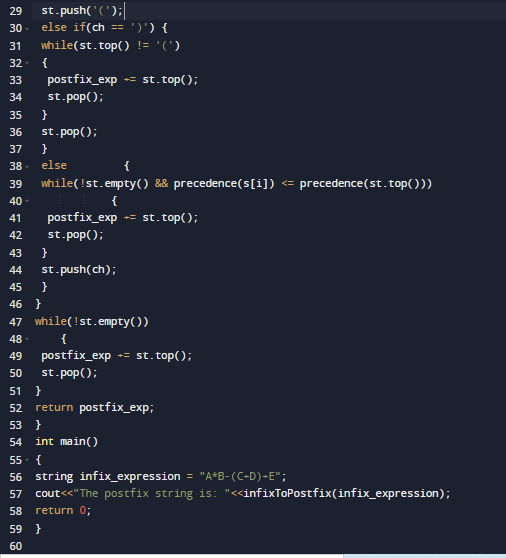
string infix\_expression = "A\*B-(C+D)+E";

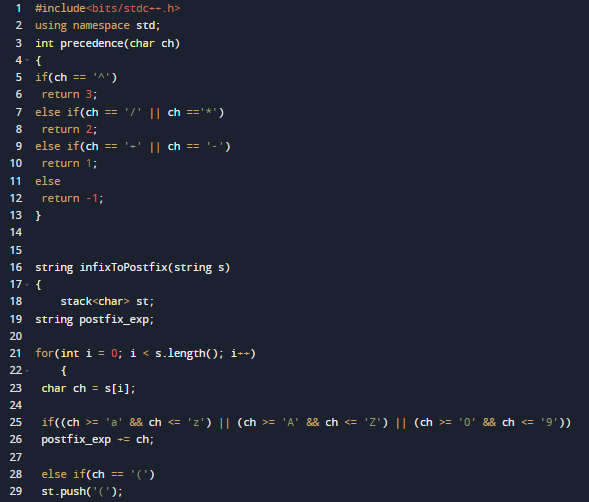
cout<<"The postfix string is: "<<infixToPostfix(infix\_expression);

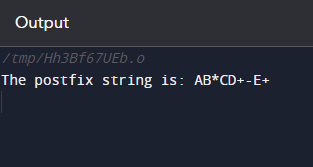
return 0;

}

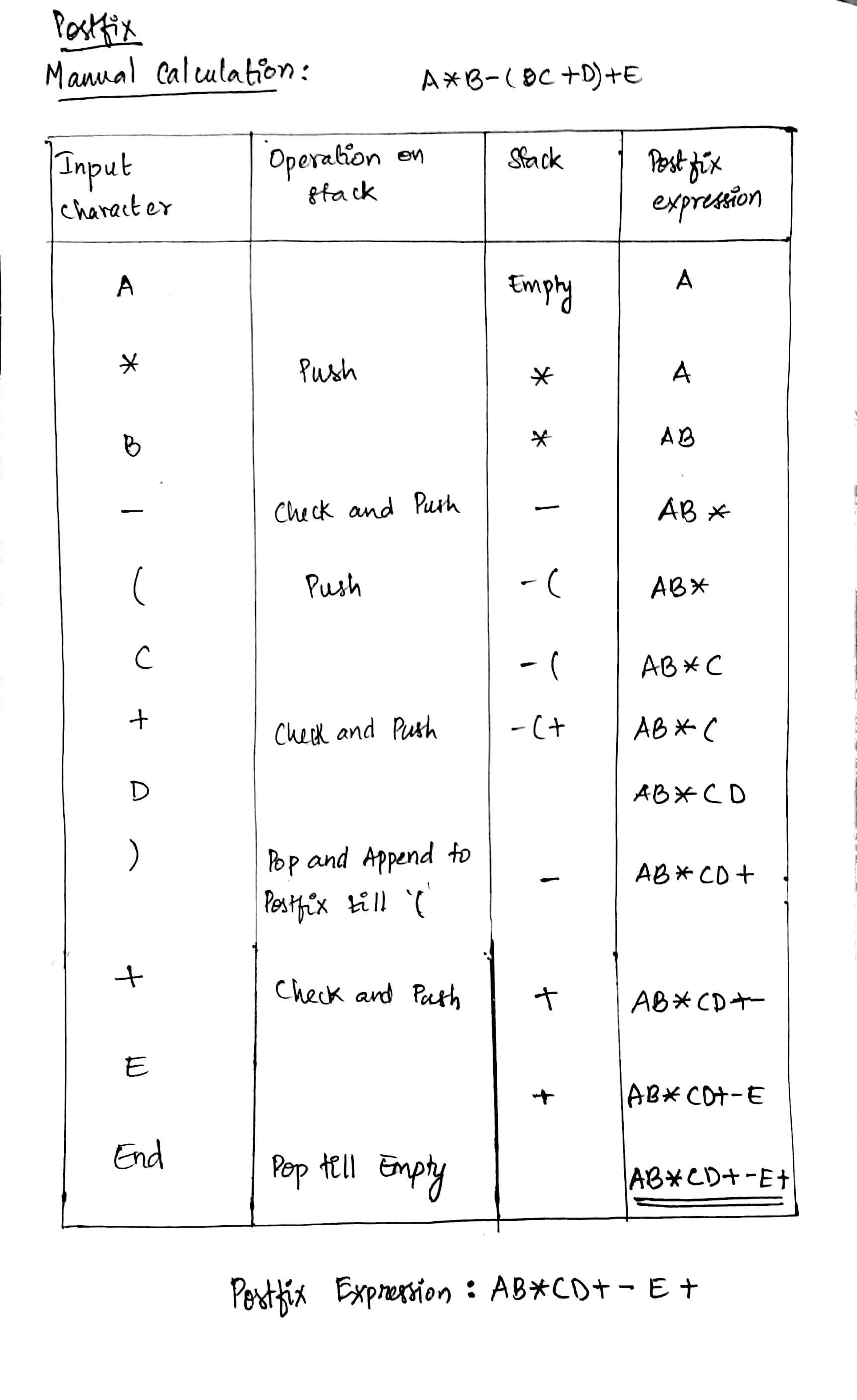
**CODE SCREENSHORT:**







**Manual Calculation:**

****

**Prefix Expression**

**Aim:** To generate prefix expression of a given expression.

**Algorithm:**

* Initially reverse the expression given for the infix.
* One by one scan of characters.
* Is if character is an operand, then copy it to the output of the prefix notation.
* When the character is a parenthesis closing then push it to the stack.
* If the character is an opening parenthesis, pop the elements in the stack till we find the closing parenthesis that corresponds.
* If a character scanned is operator.
* If the operator has greater or equal precedence than the top of the stack, push the operator to the stack.
* If the operator has less precedence than the top of the stack, pop the operator and output it to the output of the prefix notation, then check the above condition with the new top of the stack again.
* After scanning all the characters, reverse the notation output for the prefix.

**CODE:**

#include <iostream>

#include <limits.h>

#include <stdio.h>

#include <stdlib.h>

#include <string.h>

using namespace std;

//definition of functions

struct Stack \*create (int max);

int stackFull (struct Stack \*stack);

int stackEmpty (struct Stack \*stack);

void pushElement (struct Stack \*stack, int item);

int popElement (struct Stack \*stack);

int peekElement (struct Stack \*stack);

int checkOperand (char ch);

int precedence (char ch);

int postfix (char \*expression);

void reverse (char \*exp);

void brackets (char \*exp);

void conversionInfixToPrefix (char \*exp);

// A structure to represent a stack

struct Stack

{

int top;

int maxSize;

int \*array;

};

int main ()

{

int n = 10;

cout << "The infix expression is: \n";

char expression[] = "(P+(Q\*R)/(S-T))";

cout << expression << "\n";

conversionInfixToPrefix (expression);

cout << "The prefix expression is: \n";

cout << expression;

return 0;

}

//stack implementation

struct Stack \* create (int max)

{

struct Stack \*stack = (struct Stack \*) malloc (sizeof (struct Stack));

stack->maxSize = max;

stack->top = -1;

stack->array = (int \*) malloc (stack->maxSize \* sizeof (int));

return stack;

}

// Checking with this function is stack is full or not

int stackFull (struct Stack \*stack)

{

if (stack->top == stack->maxSize - 1)

{

cout << "Will not be able to push maxSize reached\n";

}

// We know array index from 0 and maxSize starts from 1

return stack->top == stack->maxSize - 1;

}

// if Stack is empty when top is equal to -1 and return true

int stackEmpty (struct Stack \*stack)

{

return stack->top == -1;

}

// Push function it inserts value in stack and increments stack top by 1

void pushElement (struct Stack \*stack, int item)

{

if (stackFull (stack))

return;

stack->array[++stack->top] = item;

}

// pop Function it remove an item from stack and decreases top by 1

int popElement (struct Stack \*stack)

{

if (stackEmpty (stack))

return INT\_MIN;

return stack->array[stack->top--];

}

// Function to return the top from stack without removing it

int peekElement (struct Stack \*stack)

{

if (stackEmpty (stack))

return INT\_MIN;

return stack->array[stack->top];

}

// A function check the given character is operand

int checkOperand (char ch)

{

return (ch >= 'a' && ch <= 'z') || (ch >= 'A' && ch <= 'Z');

}

// Fucntion to compare precedence if return larger value means higher precedence

int precedence (char ch)

{

switch (ch)

{

case '+':

case '-':

return 1;

case '\*':

case '/':

return 2;

case '^':

return 3;

}

return -1;

}

// The function for infix to postfix conversion

int postfix (char \*expression)

{

int i, j;

struct Stack \*stack = create (strlen (expression));

if (!stack)

return -1;

for (i = 0, j = -1; expression[i]; ++i)

{

// checking the character we scanned is operand or not

if (checkOperand (expression[i]))

expression[++j] = expression[i];

// if we scan character push it to the stack

else if (expression[i] == '(')

pushElement (stack, expression[i]);

//if we scan character we need to pop and print from the stack

else if (expression[i] == ')')

{

while (!stackEmpty (stack) && peekElement (stack) != '(')

expression[++j] = popElement (stack);

if (!stackEmpty (stack) && peekElement (stack) != '(')

return -1; // invalid expression

else

popElement (stack);

}

else // if an operator

{

while (!stackEmpty (stack)

&& precedence (expression[i]) <=

precedence (peekElement (stack)))

expression[++j] = popElement (stack);

pushElement (stack, expression[i]);

}

}

// if all first expression characters are scanned

// adding all left elements from stack to expression

while (!stackEmpty (stack))

expression[++j] = popElement (stack);

expression[++j] = '\0';

}

void reverse (char \*exp)

{ //reverse function for expression

int size = strlen (exp);

int j = size, i = 0;

char temp[size];

temp[j--] = '\0';

while (exp[i] != '\0')

{

temp[j] = exp[i];

j--;

i++;

}

strcpy (exp, temp);

}

void brackets (char \*exp)

{

int i = 0;

while (exp[i] != '\0')

{

if (exp[i] == '(')

exp[i] = ')';

else if (exp[i] == ')')

exp[i] = '(';

i++;

}

}

void conversionInfixToPrefix (char \*exp)

{

int size = strlen (exp);

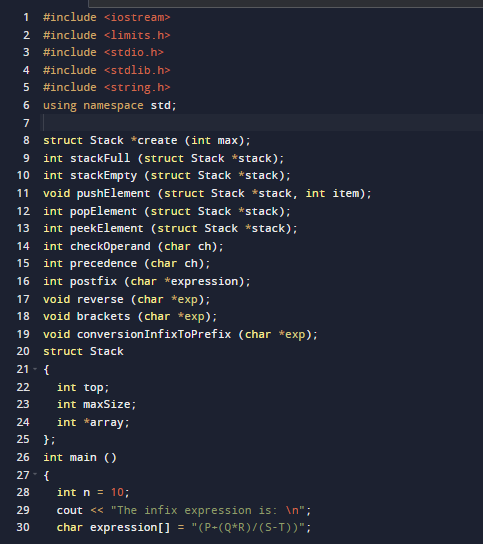
reverse (exp);

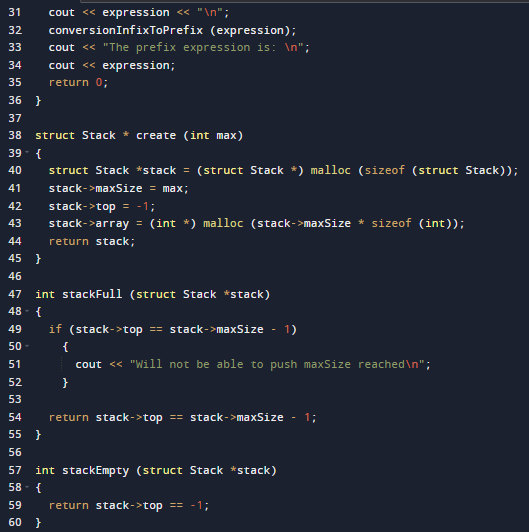
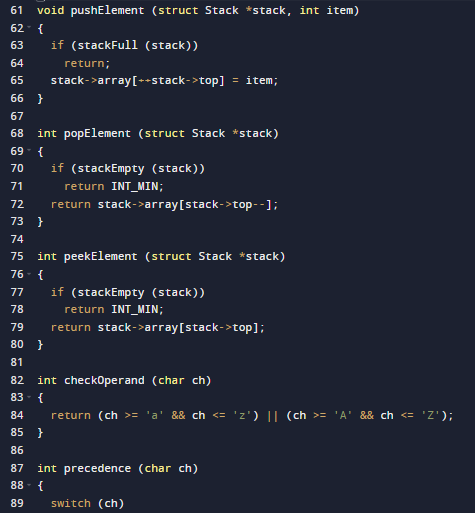
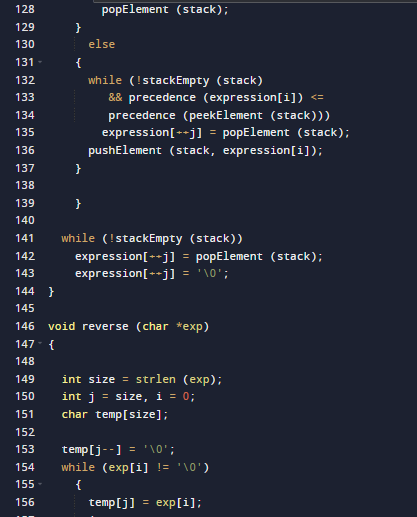
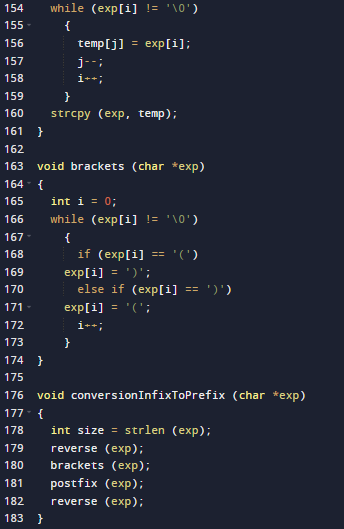
brackets (exp);

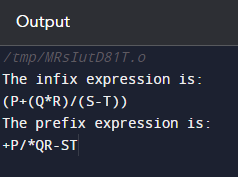
postfix (exp);

reverse (exp);

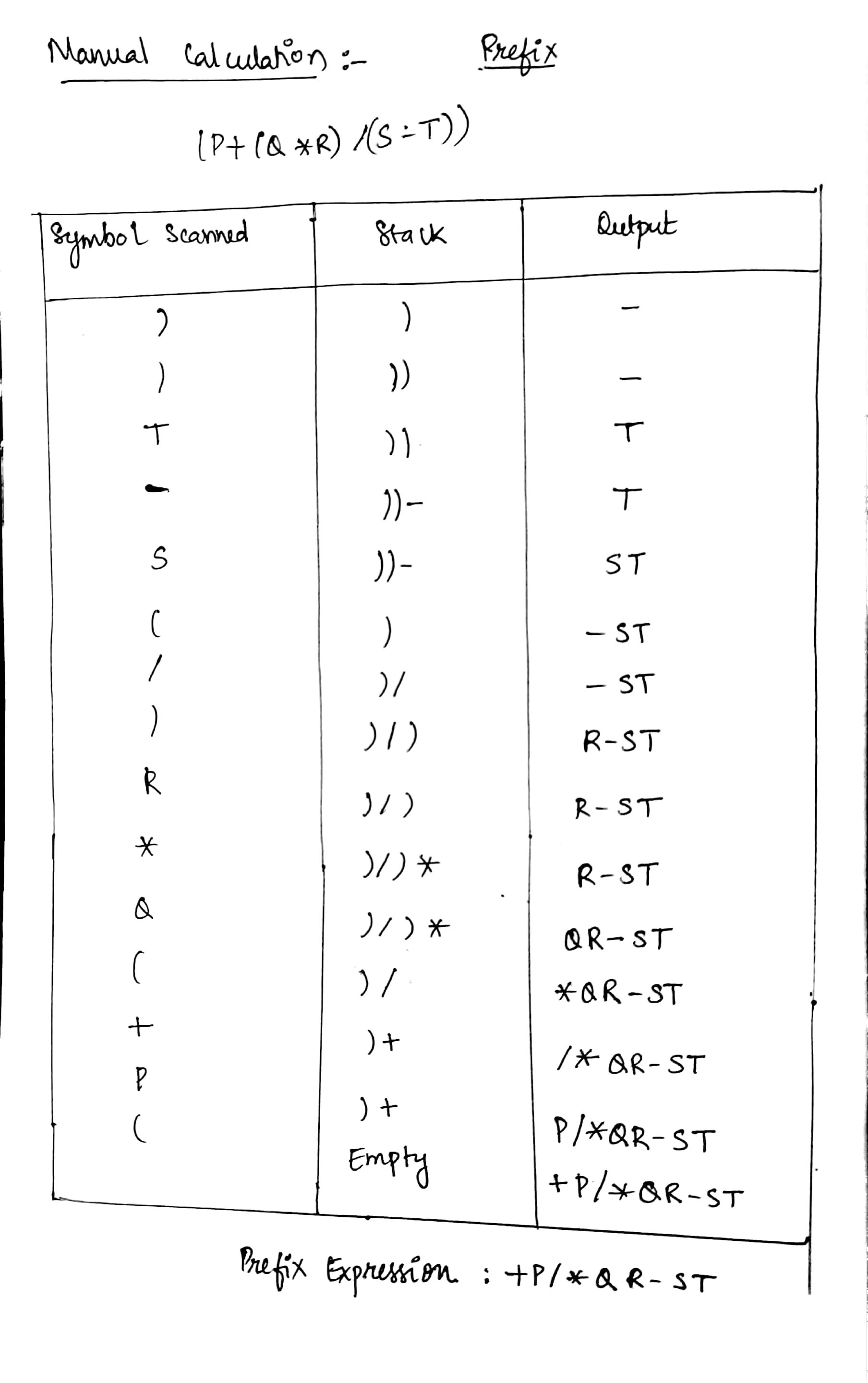
}

**CODE SCREENSHORT:**



**Manual Calculation:**

****

**RESULT:** Thus the implementation of intermediate code generation-Postfix , Prefix was executed successfully and was verified using manual calculation.