DATA STRUCTURES LAB EXPERIMENTS

SANSKRUTI SAMANT

USN: 2GI17EC113

DIV: B

# **EXPERIMENT 2 b:**

# Conversion of Infix Expression to Postfix Expression..

* **PROBLEM STATEMENT**: Write a C program to evaluate a postfix expression by converting an expression from infix to postfix.
* **ALGORITHM:**

Let, X be an arithmetic expression written in infix notation. This algorithm finds the equivalent postfix expression Y.

1. Push ‘(‘onto Stack, and add ‘)’ to the end of X.

2. Scan X from left to right and repeat Step 3 to 6 for each element of

X until the Stack is empty.

3. If an operand is encountered, add it to Y.

4. If a left parenthesis is encountered, push it onto Stack.

5. If an operator is encountered, then:

* + Repeatedly pop from Stack and add to Y each operator (on the top of Stack) which has the same precedence as or higher precedence than operator.
  + Add operator to Stack.

[End of If]

6. If a right parenthesis is encountered, then:

-- Repeatedly pop from Stack and add to Y each operator (on the top

of Stack) until a left parenthesis is encountered.

-- Remove the left Parenthesis.

[End of If]  
 [End of If]

7. END.

# **PROGRAM CODE:**

#include<stdio.h>

#include<stdlib.h>

#include<stdbool.h>

#include<string.h>

#define SIZE 100

char stack[SIZE];

int top = -1;

int main()

{

char infix[SIZE], postfix[SIZE];

printf("\n The limitation is infix exp must hv single letter");

printf("\nEnter the valid infix expression: ");

gets(infix);

infixToPostfix(infix, postfix);

printf("\nThe equivalent postfix expression: ");

puts(postfix);

return 0;

}

void push(char item)

{

if(top>=SIZE-1)

printf("\nStack is full!!");

else

{

top++;

stack[top]=item;

}

return;

}

char pop(void)

{

char item;

if(top<0)

{

printf("\nStack is empty!!");

exit(1);

}

else

{

item = stack[top];

top--;

return(item);

}

}

bool isOperator(char symbol)//bool tells true or false

{

if(symbol=='^' || symbol=='\*' || symbol=='/' || symbol=='+' || symbol=='-')

{return true;

}

else

{

return false;

}

}

int priority(char symbol)

{

if(symbol=='^')//exponent operator--highest priority

{return(3);

}

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

{return(2);

}

else if(symbol=='+' || symbol=='-')//lowest priority

{return(1);

}

else

{return(0);

}

}

void infixToPostfix(char infix[], char postfix[])

{

char token; //scanned element

char stackTop;

int i=0, j=0;

push('(');

strcat(infix,")");

token=infix[i];

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

{

if(token=='(')

{push(token);

}

if (isdigit(token) || isalpha(token))

{

postfix[j]=token;//add operand to postfix

j++;

}

else if(isOperator(token))

{

stackTop=pop();

while(isOperator(stackTop) && priority(stackTop)>=priority(token))

{

postfix[j]=stackTop;

j++;

stackTop = pop();

}

push(stackTop);

push(token);

}

else if(token==')')//if current symbol is ')'

{

stackTop=pop();

while(stackTop!='(')

{

postfix[j]=stackTop;

j++;

stackTop=pop();

}

}

i++;

token=infix[i];

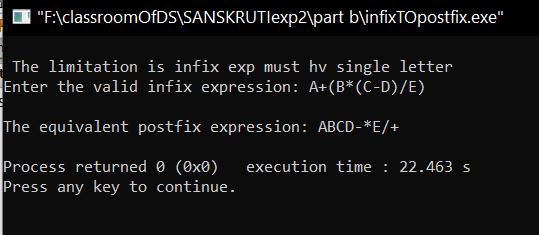
}

postfix[j]='\0';

return;

}

* **PROGRAM OUTPUT:**

****

* **ANALYSIS (LIMITATIONS):**

There are following limitations of above implementation.

1. It supports only 4 binary operators ‘+’, ‘\*’, ‘-‘and ‘/’. It can be extended for more operators by adding more switch cases.
2. The allowed operands are only single digit operands. The program can be extended for multiple digits by adding a separator like space between all elements (operators and operands) of given expression.