ASSIGNMENT 9

AIM:-

Implement stack for expression conversion (infix to postfix)

OBJECTIVE:-

1) Understand the Stack Data Structure and its basic operators.

2) Understand the method of defining stack ADT and implement the basic operators. 3) Learn how to create objects from an ADT and invoke member functions.

THEORY:-In high level programming languages, we use arithmetic expression in its infix form. An expression in infix form contains operators in between operands on which it operates. Parentheses also appear in infix expressions to specify the order of evaluation. During compilation, the compiler converts the infix expression to postfix for easy evaluation, since a postfix expression does not contain any parenthesis. Also, a postfix expression can be evaluated easily by using a stack. Postfix notation has the following virtues: No parenthesis. The priority of the operations is no longer relevant. Enables easy evaluation (evaluated by making a left to right scan, stacking the operands.)

ALGORITHM:-

1. Scan the infix expression from left to right.

2. If the scanned character is an operand, output it.

3. Else,

…..3.1 If the precedence of the scanned operator is greater than the precedence of the operator in the stack(or the stack is empty or the stack contains a ‘(‘ ), push it.

…..3.2 Else, Pop all the operators from the stack which are greater than or equal to in precedence than that of the scanned operator. After doing that Push the scanned operator to the stack. (If you encounter parenthesis while popping then stop there and push the scanned operator in the stack.)

4. If the scanned character is an ‘(‘, push it to the stack.

5. If the scanned character is an ‘)’, pop the stack and and output it until a ‘(‘ is encountered, and discard both the parenthesis.

6. Repeat steps 2-6 until infix expression is scanned.

7. Print the output

8. Pop and output from the stack until it is not empty.

SOURCE CODE:-

#include<iostream>

#include<stack>

#include<string>

using namespace std;

// Function to convert Infix expression to postfix

string InfixToPostfix(string expression);

// Function to verify whether an operator has higher precedence over other

int HasHigherPrecedence(char operator1, char operator2);

// Function to verify whether a character is operator symbol or not.

bool IsOperator(char C);

// Function to verify whether a character is alphanumeric chanaracter (letter or numeric digit) or not.

bool IsOperand(char C);

int main()

{

string expression;

cout<<"Enter Infix Expression \n";

getline(cin,expression);

string postfix = InfixToPostfix(expression);

cout<<"Output = "<<postfix<<"\n";

}

// Function to evaluate Postfix expression and return output

string InfixToPostfix(string expression)

{

// Declaring a Stack from Standard template library in C++.

stack<char> S;

string postfix = ""; // Initialize postfix as empty string.

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

// Scanning each character from left.

// If character is a delimitter, move on.

if(expression[i] == ' ' || expression[i] == ',') continue;

// If character is operator, pop two elements from stack, perform operation and push the result back.

else if(IsOperator(expression[i]))

{

while(!S.empty() && S.top() != '(' && HasHigherPrecedence(S.top(),expression[i]))

{

postfix+= S.top();

S.pop();

}

S.push(expression[i]);

}

// Else if character is an operand

else if(IsOperand(expression[i]))

{

postfix +=expression[i];

}

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

{

S.push(expression[i]);

}

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

{

while(!S.empty() && S.top() != '(') {

postfix += S.top();

S.pop();

}

S.pop();

}

}

while(!S.empty()) {

postfix += S.top();

S.pop();

}

return postfix;

}

// Function to verify whether a character is english letter or numeric digit.

// We are assuming in this solution that operand will be a single character

bool IsOperand(char C)

{

if(C >= '0' && C <= '9') return true;

if(C >= 'a' && C <= 'z') return true;

if(C >= 'A' && C <= 'Z') return true;

return false;

}

// Function to verify whether a character is operator symbol or not.

bool IsOperator(char C)

{

if(C == '+' || C == '-' || C == '\*' || C == '/' || C== '$')

return true;

return false;

}

// Function to verify whether an operator is right associative or not.

int IsRightAssociative(char op)

{

if(op == '$') return true;

return false;

}

// Function to get weight of an operator. An operator with higher weight will have higher precedence.

int GetOperatorWeight(char op)

{

int weight = -1;

switch(op)

{

case '+':

case '-':

weight = 1;

case '\*':

case '/':

weight = 2;

case '$':

weight = 3;

}

return weight;

}

// Function to perform an operation and return output.

int HasHigherPrecedence(char op1, char op2)

{

int op1Weight = GetOperatorWeight(op1);

int op2Weight = GetOperatorWeight(op2);

// If operators have equal precedence, return true if they are left associative.

// return false, if right associative.

// if operator is left-associative, left one should be given priority.

if(op1Weight == op2Weight)

{

if(IsRightAssociative(op1)) return false;

else return true;

}

return op1Weight > op2Weight ? true: false;

}

OUTPUT:-

Enter Infix Expression

(a+b)-(c+d)

Output = ab+cd+-

CONCLUSION:-To convert infix expression to postfix expression,we will use the stack data structure.By scanning the infix expression from left to right ,when we will get any operand ,simply add them to postfix form,and for the operator and parenthesis ,add them in the stack maintaining the precedence of them.