Harsh Kasliwal
85 /C21.

DATE

Experiment 2

Aim: Convert an infix expression to postfix expression using stack ADT

Theory:

1)

Infix: For arithmetic expression such as x+y ~~6*3~~, we know the variable 'x' is being added to the variable 'y'. Since the In this way in which operands surround the operator is called infix notation. E.g: 6*3, x+y etc.

Postfix: Postfix notation are also known as Reverse Polish Notation. They are different from infix and prefix notations in the sense that in the postfix notation, operator comes after the operands, e.g. xy+, xyz+* etc

Prefix: Also known as Polish notation. The name only suggests operator comes before the operands.
E.g: +xy, *+xyz etc.

2) Converting the following expression from infix to postfix.

5*3^2+(2*(7/11)^9)

5*23^ + (2*(711/)^9)
5*23^ + (2711/9^*)
523^*2711/9^*+

Harsh Kasliwal

Postfix to infix:

14 3 ^ - 7 9 1 ^ 5 / * + 2 -

1(4^3) - 7(9^1)5 / * + 2 -
1 - (4^3) 7 (9^1/5) * + 2 -
1 - (4^3) 7 * (9^1/5) + 2 -
1 - (4^3) + 7 * (9^1/5) 2 -
1 - (4^3) + 7 * (9^1/5) - 2

3> Infix to Postfix (Algorithm).

```
Infix to Postfix (Exp)
{
  1. Create an Empty stack (s)
  2. Create an Empty string for storing result.
  3. for i=0 to len(Exp)-1 {
     // case-1:
  4. if (Exp[i] is operand)
  5. { result = result + E[i] }
       // case-2:
  6. Else if (E[i] is "(") {
  7.   Push (E[i]) }
        // case 3:
  8. Else if (E[i] is ")") {
  9. while ((S != Empty) && (top != "(")) {
  10.  result = result + top.
  11.  POP() }
  12.    POP() }
        // case: 4
  13. Else {
       ~~while~~
```

Harsh Kasliwal
85

```
14. while ((S! = Empty) && (top! = "(") && (top>= precedence(Exp[i]
                                                                  )))
15.     result = result + top
16.     POP()
        }
~~17. while (S! Empty)){~~
~~18~~ 17. Push (Exp [i]) }}
18. while (S! = Empty) {
19. result = result + Top
20. POP() }
```

4) Example showing each step using stack

infix - A+B*C/(E-F)

	Input string	Output stack	Operator stack
0	A+B*C/(E-F)	A	
1	A+B*C/(E-F)	A	+
2	A+B*C/(E-F)	AB	+
3	A+B*C/(E-F)	AB	+*
4	-11-	ABC	+*
5	-11-	ABC*	+/
6	-11-	ABC*	+/(
7	-11-	ABC*E	+/(
8	-11-	ABC*E	+/(-
9	-11-	ABC*EF	+/(-
10	-11-	ABC*EF-	+/
11	A+B*C/(E-F)	ABC*EF-/+	

Harsh Kasliwal
C21/85

Conclusion :-

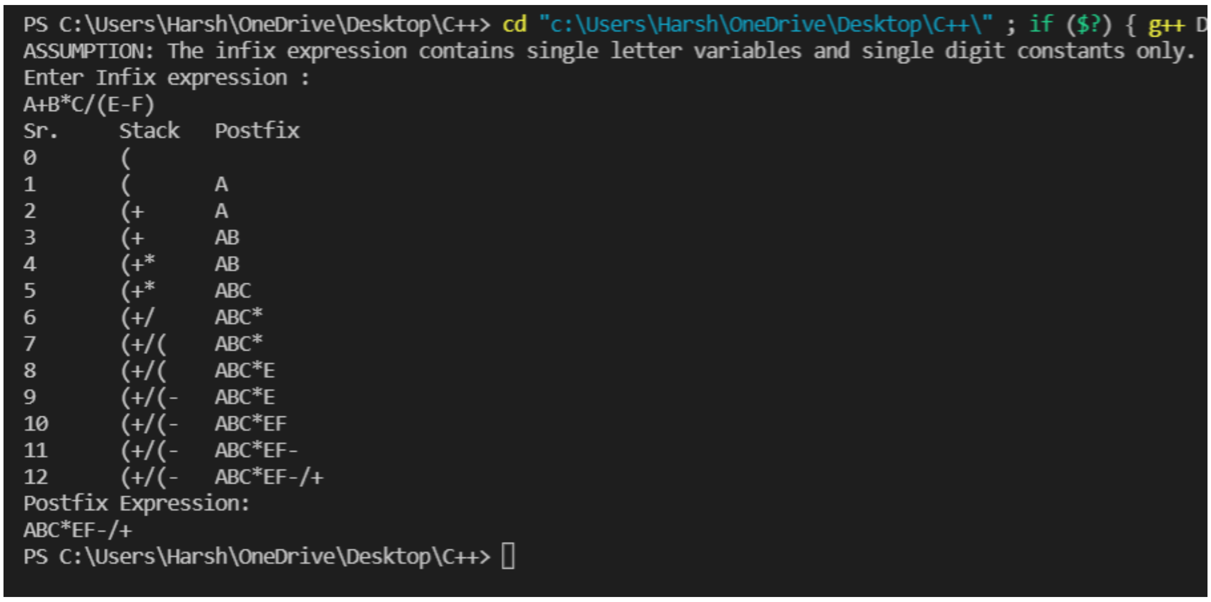
1) Infix expressions are human readable but not efficient for computer or machine reading.
2) Prefix and Postfix do not need the concept of precedence & associativity hence it becomes highly efficient to parse expressions in prefix or postfix formats.

Program:

```cpp
#include <iostream>
#include <cstring>
using namespace std;
#define SIZE 100
char stack[SIZE];
int top = -1;
void push(char item)
{
    if (top >= SIZE - 1)
    {
        cout << "Stack Overflow." << endl;
    }
    else
    {
        top = top + 1;
        stack[top] = item;
    }
}

char pop()
{
    char item;

    if (top < 0)
    {
        cout << "stack under flow: invalid infix expression" << endl;
        getchar();
        exit(1);
    }
    else
    {
        item = stack[top];
        top = top - 1;
        return (item);
    }
}

int is_operator(char symbol)
{
    if (symbol == '^' || symbol == '*' || symbol == '/' || symbol == '+' || sy
mbol == '-')
    {
        return 1;
    }
    else
    {
        return 0;
    }
}

int precedence(char symbol)
{
    if (symbol == '^')
    {
        return (3);
    }
    else if (symbol == '*' || symbol == '/')
    {
        return (2);
    }
    else if (symbol == '+' || symbol == '-')
    {
        return (1);
    }
    else
    {
        return (0);
    }
}

void InfixToPostfix(char infix_exp[], char postfix_exp[])
{
    cout<<"Sr.\tStack\tPostfix"<<endl;
    cout<<"0\t(\t"<<endl;
    int i, j;
    char item;
    char x;

    push('(');
    strcat(infix_exp, ")");

    i = 0;
    j = 0;
    item = infix_exp[i];
    int counter =1;
    while (item != '\0')
    {
        if (item == '(')
        {
            push(item);
        }
        else if (isdigit(item) || isalpha(item))
        {
            postfix_exp[j] = item;
            j++;
            postfix_exp[j] = '\0';
        }
        else if (is_operator(item) == 1)
        {
            x = pop();
            while (is_operator(x) == 1 && precedence(x) >= precedence(item))
            {
                postfix_exp[j] = x;
                j++;
                postfix_exp[j] = '\0';
                x = pop();
            }
            push(x);

            push(item);
        }
        else if (item == ')')
        {
            x = pop();
            while (x != '(')
            {
                postfix_exp[j] = x;
                j++;
                postfix_exp[j] = '\0';
                x = pop();
            }
        }
        else
        {
            cout << "Invalid infix Expression." << endl;
            getchar();
            exit(1);
        }
        i++;
        cout<<counter<<"\t"<<stack<<"\t"<<postfix_exp<<"\t"<<endl;
        counter++;
        item = infix_exp[i];
    }
    if (top > 0)
    {
        cout << "Invalid infix Expression." << endl;
        getchar();
        exit(1);
    }


    postfix_exp[j] = '\0';
}
```

```
int main()
{
    char infix[SIZE], postfix[SIZE];
    cout << "ASSUMPTION: The infix expression contains single letter variables
 and single digit constants only." << endl;
    cout << "Enter Infix expression : " << endl;
    gets(infix);

    InfixToPostfix(infix, postfix);
    cout << "Postfix Expression: " << endl;
    puts(postfix);

    return 0;
}
```

Output:



```
PS C:\Users\Harsh\OneDrive\Desktop\C++> cd "c:\Users\Harsh\OneDrive\Desktop\C++\" ; if ($?) { g++ D
ASSUMPTION: The infix expression contains single letter variables and single digit constants only.
Enter Infix expression :
A+B*C/(E-F)
Sr.     Stack   Postfix
0       (
1       (       A
2       (+      A
3       (+      AB
4       (+*     AB
5       (+*     ABC
6       (+/     ABC*
7       (+/(    ABC*
8       (+/(    ABC*E
9       (+/(-   ABC*E
10      (+/(-   ABC*EF
11      (+/(-   ABC*EF-
12      (+/(-   ABC*EF-/+
Postfix Expression:
ABC*EF-/+
PS C:\Users\Harsh\OneDrive\Desktop\C++> 
```