SRM INSTITUTE OF SCIENCE AND TECHNOLOGY

FACULTY OF ENGINEERING & TECHNOLOGY

(Formerly SRM University, Under section 3 of UGC Act, 1956)

S.R.M. NAGAR, KATTANKULATHUR –603 203,
KANCHEEPURAM DISTRICT

SCHOOL OF COMPUTING
DEPARTMENT OF NETWORKING AND COMMUNICATION

Course Code: 18CSC304J

Course Name: Compiler Design

SUBMITTED BY : - NAMAN CHITKARA
RA1911029010046
CSE-CN
P1

SUBMITTED TO :-

SRM
INSTITUTE OF SCIENCE & TECHNOLOGY
(Deemed to be University u/s 3 of UGC Act, 1956)

COLLEGE OF ENGINEERING & TECHNOLOGY
SRM INSTITUTE OF SCIENCE & TECHNOLOGY
S.R.M. NAGAR, KATTANKULATHUR - 603203
Chengalpattu District

BONAFIDE CERTIFICATE

Register No RA1911029010046

Certified to be the bonafide record of work done by NAMAN CHITKARA of CSE - CN B.Tech Degree course in the Practical COMPILER DESIGN LAB (18CSC304J) in **SRM INSTITUTE OF SCIENCE & TECHNOLOGY,** Kattankulathur during the academic year 2022.

FACULTY INCHARGE

DATE:

HEAD OF THE DEPARTMENT

Submitted for University Examination held in ____________________,

in ____

SRM INSTITUTE OF SCIENCE & TECHNOLOGY, Kattankulathur.

EXAMINER - I

EXAMINER - II

Exp 1
Lexical Analysis

Naman CHITKARA
RA1911029010046

AIM:

To write a program to implement a lexical analyser.

ALGORITHM:

1. Start.
2. Get the input program from the file prog.txt.
3. Read the program line by line and check if each word in a line is a keyword, identifier,
constant or an operator.
4. If the word read is an identifier, assign a number to the identifier and make an entry into
the symbol table stored in sybol.txt.
5. For each lexeme read, generate a token as follows:
a. If the lexeme is an identifier, then the token generated is of the form <id, number>
b. If the lexeme is an operator, then the token generated is <op, operator>.
c. If the lexeme is a constant, then the token generated is <const, value>.
d. If the lexeme is a keyword, then the token is the keyword itself.
6. The stream of tokens generated are displayed in the console output.
7. Stop.

PROGRAM:

```
      #include <stdbool.h>

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

bool isDelimiter(char ch)
{
    if (ch == ' ' || ch == '+' || ch == '-' || ch == '*' ||
        ch == '/' || ch == ',' || ch == ';' || ch == '>' ||
        ch == '<' || ch == '=' || ch == '(' || ch == ')' ||
        ch == '[' || ch == ']' || ch == '{' || ch == '}')
        return (true);
    return (false);
}

bool isSpecialCharacter(char ch)
{
```

```
    if (ch == ',' || ch == ';' || ch == '>' ||
        ch == '<' || ch == '=' || ch == '(' || ch == ')' ||
        ch == '[' || ch == ']' || ch == '{' || ch == '}')
        return (true);
    return (false);
}

bool isOperator(char ch)
{
    if (ch == '+' || ch == '-' || ch == '*' ||
        ch == '/' || ch == '>' || ch == '<' ||
        ch == '=')
        return (true);
    return (false);
}


bool validIdentifier(char* str)
{
    if (str[0] == '0' || str[0] == '1' || str[0] == '2' ||
        str[0] == '3' || str[0] == '4' || str[0] == '5' ||
        str[0] == '6' || str[0] == '7' || str[0] == '8' ||
        str[0] == '9' || isDelimiter(str[0]) == true)
        return (false);
    return (true);
}

bool isKeyword(char* str)
{
    if (!strcmp(str, "if") || !strcmp(str, "else") ||
        !strcmp(str, "while") || !strcmp(str, "do") ||
        !strcmp(str, "break") ||
        !strcmp(str, "continue") || !strcmp(str, "int")
        || !strcmp(str, "double") || !strcmp(str, "float")
        || !strcmp(str, "return") || !strcmp(str, "char")
        || !strcmp(str, "case") || !strcmp(str, "char")
        || !strcmp(str, "sizeof") || !strcmp(str, "long")
        || !strcmp(str, "short") || !strcmp(str, "typedef")
        || !strcmp(str, "switch") || !strcmp(str, "unsigned")
        || !strcmp(str, "void") || !strcmp(str, "static")
        || !strcmp(str, "struct") || !strcmp(str, "goto"))
        return (true);
    return (false);
}

bool isInteger(char* str)
{
    int i, len = strlen(str);
```

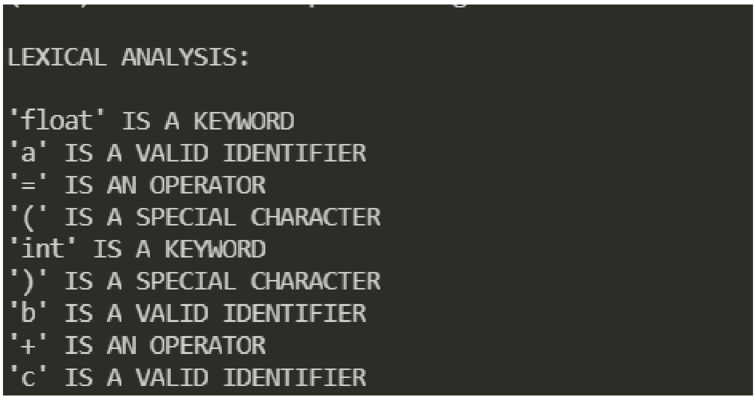
```c
    if (len == 0)
        return (false);
    for (i = 0; i < len; i++) {
        if (str[i] != '0' && str[i] != '1' && str[i] != '2'
            && str[i] != '3' && str[i] != '4' && str[i] != '5'
            && str[i] != '6' && str[i] != '7' && str[i] != '8'
            && str[i] != '9' || (str[i] == '-' && i > 0))
            return (false);
    }
    return (true);
}

bool isRealNumber(char* str)
{
    int i, len = strlen(str);
    bool hasDecimal = false;

    if (len == 0)
        return (false);
    for (i = 0; i < len; i++) {
        if (str[i] != '0' && str[i] != '1' && str[i] != '2'
            && str[i] != '3' && str[i] != '4' && str[i] != '5'
            && str[i] != '6' && str[i] != '7' && str[i] != '8'
            && str[i] != '9' && str[i] != '.' ||
            (str[i] == '-' && i > 0))
            return (false);
        if (str[i] == '.')
            hasDecimal = true;
    }
    return (hasDecimal);
}


char* subString(char* str, int left, int right)
{
    int i;
    char* subStr = (char*)malloc(
                  sizeof(char) * (right - left + 2));

    for (i = left; i <= right; i++)
        subStr[i - left] = str[i];
    subStr[right - left + 1] = '\0';
    return (subStr);
}


void parse(char* str)
{
    int left = 0, right = 0;
```

```c
    int len = strlen(str);

    while (right <= len && left <= right) {
        if (isDelimiter(str[right]) == false)
            right++;

        if (isDelimiter(str[right]) == true && left == right) {
            if (isOperator(str[right]) == true)
                printf("'%c' IS AN OPERATOR\n", str[right]);
            else if (isSpecialCharacter(str[right]) == true)
                printf("'%c' IS A SPECIAL CHARACTER\n", str[right]);
            right++;
            left = right;
        }

        else if (isDelimiter(str[right]) == true && left != right
                || (right == len && left != right)) {
            char* subStr = subString(str, left, right - 1);

            if (isKeyword(subStr) == true)
                printf("'%s' IS A KEYWORD\n", subStr);

            else if (isInteger(subStr) == true)
                printf("'%s' IS AN INTEGER\n", subStr);

            else if (isRealNumber(subStr) == true)
                printf("'%s' IS A REAL NUMBER\n", subStr);

            else if (validIdentifier(subStr) == true
                    && isDelimiter(str[right - 1]) == false)
                printf("'%s' IS A VALID IDENTIFIER\n", subStr);

            else if (validIdentifier(subStr) == false
                    && isDelimiter(str[right - 1]) == false)
                printf("'%s' IS NOT A VALID IDENTIFIER\n", subStr);
            left = right;
        }
    }
    return;
}


int main()
{
    char str[100] = "float a = (int)b + c)";
    printf("\nLEXICAL ANALYSIS:\n\n");
    parse(str);
    return (0);
}
```

Result:

Input - “float a = (int)b + c”;

Output –



```
LEXICAL ANALYSIS:

'float' IS A KEYWORD
'a' IS A VALID IDENTIFIER
'=' IS AN OPERATOR
'(' IS A SPECIAL CHARACTER
'int' IS A KEYWORD
')' IS A SPECIAL CHARACTER
'b' IS A VALID IDENTIFIER
'+' IS AN OPERATOR
'c' IS A VALID IDENTIFIER
```

EXP 2

CONVERSION FROM REGULAR EXPRESSION TO NFA

RA1911029010046

AIM:

To write a program for converting Regular Expression to NFA.

ALGORITHM:

1. Start
2. Get the input from the user
3. Initialize separate variables and functions for Postfix , Display and NFA
4. Create separate methods for different operators like +,*, .
5. By using Switch case Initialize different cases for the input
6. For ' . ' operator Initialize a separate method by using various stack functions do the same for the other operators like ' * ' and ' + '.
7. Regular expression is in the form like a.b (or) a+b
8. Display the output
9. Stop

PROGRAM:

```
transition_table = [ [0]*3 for _ in range(20) ]
re = input("Enter the regular expression : ")
re += " "

i = 0
j = 1
while(i<len(re)):
    if re[i] == 'a':
        try:
            if re[i+1] != '|' and re[i+1] !='*':
                transition_table[j][0] = j+1
                j += 1
            elif re[i+1] == '|' and re[i+2] =='b':
                transition_table[j][2]=((j+1)*10)+(j+3)
                j+=1
                transition_table[j][0]=j+1
                j+=1
                transition_table[j][2]=j+3
                j+=1
                transition_table[j][1]=j+1
                j+=1
                transition_table[j][2]=j+1
                j+=1
                i=i+2
            elif re[i+1]=='*':
                transition_table[j][2]=((j+1)*10)+(j+3)
```

```
                j+=1
                transition_table[j][0]=j+1
                j+=1
                transition_table[j][2]=((j+1)*10)+(j-1)
                j+=1
        except:
            transition_table[j][0] = j+1

    elif re[i] == 'b':
        try:
            if re[i+1] != '|' and re[i+1] !='*':
                transition_table[j][1] = j+1
                j += 1
            elif re[i+1]=='|' and re[i+2]=='a':
                transition_table[j][2]=((j+1)*10)+(j+3)
                j+=1
                transition_table[j][1]=j+1
                j+=1
                transition_table[j][2]=j+3
                j+=1
                transition_table[j][0]=j+1
                j+=1
                transition_table[j][2]=j+1
                j+=1
                i=i+2
            elif re[i+1]=='*':
                transition_table[j][2]=((j+1)*10)+(j+3)
                j+=1
                transition_table[j][1]=j+1
                j+=1
                transition_table[j][2]=((j+1)*10)+(j-1)
                j+=1
        except:
            transition_table[j][1] = j+1

    elif re[i]=='e' and re[i+1]!='|'and re[i+1]!='*':
        transition_table[j][2]=j+1
        j+=1

    elif re[i]==')' and re[i+1]=='*':

        transition_table[0][2]=((j+1)*10)+1
        transition_table[j][2]=((j+1)*10)+1
        j+=1

    i +=1

print ("Transition function:")
```

```python
print("s  a  b  e\n")
for i in range(j):
    if(transition_table[i][0]!=0):
        print("q[{0},a]-->{1}".format(i,transition_table[i][0]))
    if(transition_table[i][1]!=0):
        print("q[{0},b]-->{1}".format(i,transition_table[i][1]))
    if(transition_table[i][2]!=0):
        if(transition_table[i][2]<10):
            print("q[{0},e]-->{1}".format(i,transition_table[i][2]))
        else:
            print("q[{0},e]-->{1} & {2}".format(i,int(transition_table[i][2]/10),transition_table[i][2]%10))
```

RESULT:

```
Enter the regular expression : (a|b)*a

Transition function:
q[0,e]-->7 & 1
q[1,e]-->2 & 4
q[2,a]-->3
q[3,e]-->6
q[4,b]-->5
q[5,e]-->6
q[6,e]-->7 & 1
q[7,a]-->8
```

Exp 3 : Elimination Of Left Recursion

Naman CHITKARA
RA1911029010046

CODE:

```
#include<stdio.h>
 #include<string.h>
 #define SIZE 10
 int main () {
      char non_terminal;
      char beta,alpha;
      int num;
      char production[10][SIZE];
      int index=3; /* starting of the string following "->" */
      printf("Enter Number of Production : ");
     scanf("%d",&num);
      printf("Enter the grammar :\n");
      for(int i=0;i<num;i++){
           scanf("%s",production[i]);
      }
      for(int i=0;i<num;i++){
           printf("\nGRAMMAR : : : %s",production[i]);
           non_terminal=production[i][0];
           if(non_terminal==production[i][index]) {
                alpha=production[i][index+1];
                printf(" is left recursive.\n");
                while(production[i][index]!=0 && production[i][index]!='|')
                     index++;
                if(production[i][index]!=0) {
                     beta=production[i][index+1];
                     printf("Grammar without left recursion:\n");
                     printf("%c->%c%c\'",non_terminal,beta,non_terminal);
                     printf("\n%c\'->%c%c\'|E\n",non_terminal,alpha,non_terminal);
                }
                else
```

```
                printf(" can't be reduced\n");
            }
            else
                printf(" is not left recursive.\n");
            index=3;
        }
}
```

OUTPUT:

```
Enter Number of Production : 4
Enter the grammar :
E->EA|A
A->AT|a
T->a
E->i

GRAMMAR : : : E->EA|A is left recursive.
Grammar without left recursion:
E->AE'
E'->AE'|E

GRAMMAR : : : A->AT|a is left recursive.
Grammar without left recursion:
A->aA'
A'->TA'|E

GRAMMAR : : : T->a is not left recursive.

GRAMMAR : : : E->i is not left recursive.


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

Left Factoring

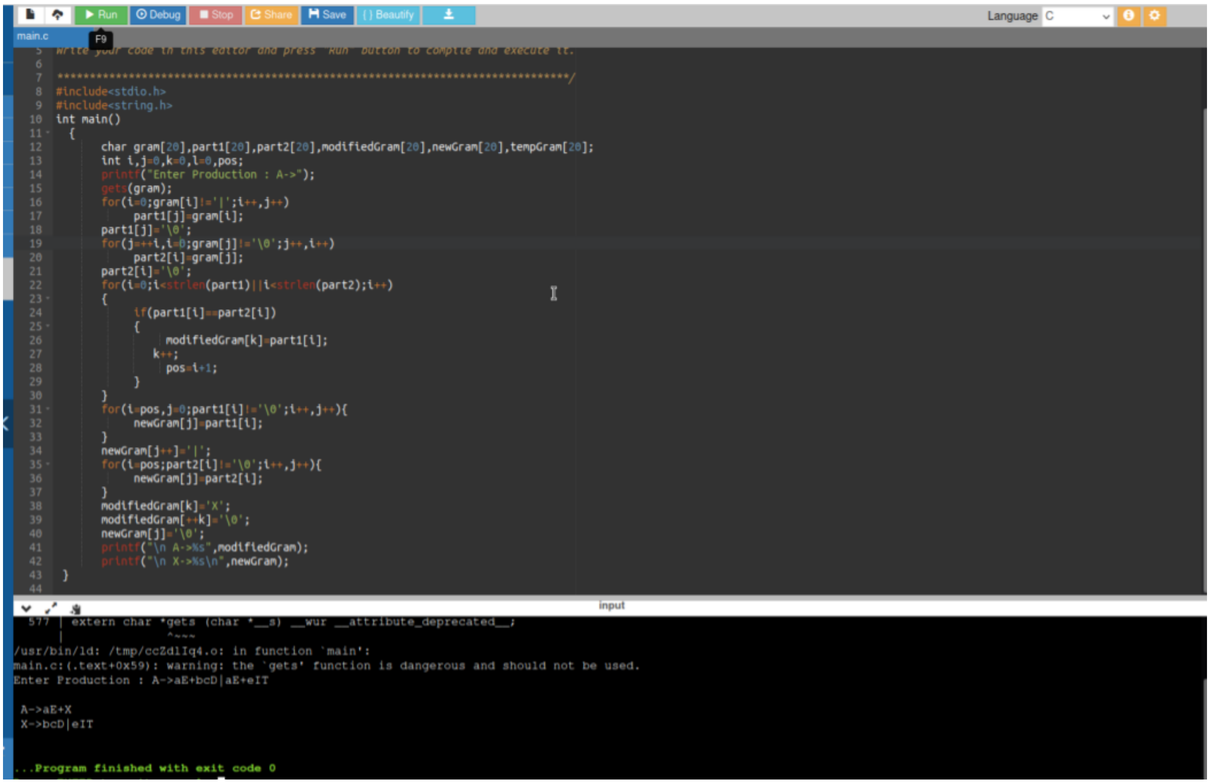
Naman CHITKARA
RA1911029010046

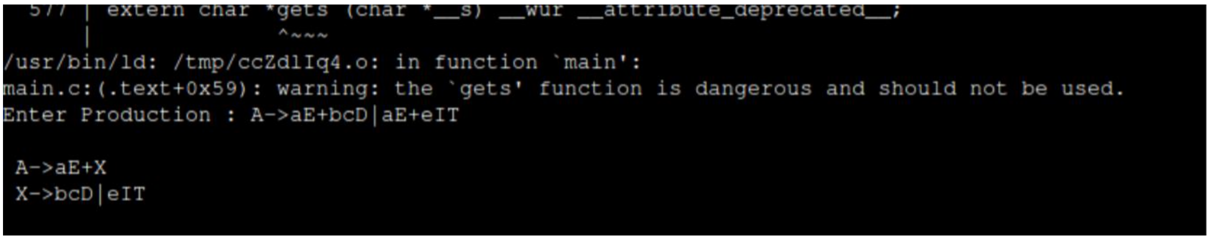
CODE:

```
#include<stdio.h>

#include<string.h>

int main()

{


char gram[20],part1[20],part2[20],modifiedGram[20],newGram[20],tempGram[20];

int i,j=0,k=0,l=0,pos;

printf("Enter Production : A->");

gets(gram);

for(i=0;gram[i]!='|';i++,j++)

part1[j]=gram[i];

part1[j]='\0';

for(j=++i,i=0;gram[j]!='\0';j++,i++)

part2[i]=gram[j];

part2[i]='\0';

for(i=0;i<strlen(part1)||i<strlen(part2);i++)

{

if(part1[i]==part2[i])

{

modifiedGram[k]=part1[i];

k++;

pos=i+1;

}

}

for(i=pos,j=0;part1[i]!='\0';i++,j++){
```

```
newGram[j]=part1[i];
}
newGram[j++]='|';
for(i=pos;part2[i]!='\0';i++,j++){
newGram[j]=part2[i];
}
modifiedGram[k]='X';
modifiedGram[++k]='\0';
newGram[j]='\0';
printf("\n A->%s",modifiedGram);
printf("\n X->%s\n",newGram);
}
```

OUTPUT:





Experiment-5

FIRST & FOLLOW COMPUTATION

NAME: Naman CHITKARA
REG NO: RA1911029010046

CODE: C++

```
#include<bits/stdc++.h>
using namespace std;

set<char> ss;
bool dfs(char i, char org, char last, map<char,vector<vector<char>>> &mp){
    bool rtake = false;
    for(auto r : mp[i]){
        bool take = true;
        for(auto s : r){
            if(s == i) break;
            if(!take) break;
            if(!(s>='A'&&s<='Z')&&s!='e'){
                ss.insert(s);
                break;
            }
            else if(s == 'e'){
                if(org == i||i == last)
                ss.insert(s);
                rtake = true;
                break;
            }
            else{
                take = dfs(s,org,r[r.size()-1],mp);
                rtake |= take;
            }
        }
    }
    return rtake;
}

int main(){
    int i,j;
    string num;
    vector<int> fs;
    vector<vector<int>> a;
    map<char,vector<vector<char>>> mp;
    char start;
    bool flag = 0;
    int n;
    cout<<"Enter the number of grammar :" << endl;
```

```
cin >> n;
while(n--){
   cin >> num;
   if(flag == 0) start = num[0],flag = 1;
   vector<char> temp;
   char s = num[0];
   for(i=3;i<num.size();i++){
      if(num[i] == '|'){
         mp[s].push_back(temp);
         temp.clear();
      }
      else temp.push_back(num[i]);
   }
   mp[s].push_back(temp);
}
map<char,set<char>> fmp;
for(auto q : mp){
   ss.clear();
   dfs(q.first,q.first,q.first,mp);
   for(auto g : ss) fmp[q.first].insert(g);
}

cout<<'\n';
cout<<"FIRST: "<<'\n';
for(auto q : fmp){
   string ans = "";
   ans += q.first;
   ans += " = {";
   for(char r : q.second){
      ans += r;
      ans += ',';
   }
   ans.pop_back();
   ans+="}";
   cout<<ans<<'\n';
}

map<char,set<char>> gmp;
gmp[start].insert('$');
int count = 10;
while(count--){
   for(auto q : mp){
      for(auto r : q.second){
         for(i=0;i<r.size()-1;i++){
            if(r[i]>='A'&&r[i]<='Z'){
               if(!(r[i+1]>='A'&&r[i+1]<='Z')) gmp[r[i]].insert(r[i+1]);
               else {
                  char temp = r[i+1];
                  int j = i+1;
                  while(temp>='A'&&temp<='Z'){
                     if(*fmp[temp].begin()=='e'){
                        for(auto g : fmp[temp]){
                           if(g=='e') continue;
```

```
                        gmp[r[i]].insert(g);
                    }
                    j++;
                    if(j<r.size()){
                        temp = r[j];
                        if(!(temp>='A'&&temp<='Z')){
                            gmp[r[i]].insert(temp);
                            break;
                        }
                    }
                    else{
                        for(auto g : gmp[q.first]) gmp[r[i]].insert(g);
                        break;
                    }
                }
                else{
                    for(auto g : fmp[temp]){
                        gmp[r[i]].insert(g);
                    }
                    break;
                }
            }
        }
    }
}
if(r[r.size()-1]>='A'&&r[r.size()-1]<='Z'){
    for(auto g : gmp[q.first]) gmp[r[i]].insert(g);
}
}
}
}

cout<<'\n';
cout<<"FOLLOW: "<<'\n';
for(auto q : gmp){
    string ans = "";
    ans += q.first;
    ans += " = {";
    for(char r : q.second){
        ans += r;
        ans += ',';
    }
    ans.pop_back();
    ans+="}";
    cout<<ans<<'\n';
}
return 0;
}
```

OUTPUT:

```
Output

Enter the number of grammar :4
S->ACB|CbB|Ba
A->da|BC
B->g|e
C->h|e
S->ACB|CbB|Ba

A->da|BC

B->g|e

C->h|e

FIRST:
A = {d,e,g,h}
B = {e,g}
C = {e,h}
S = {a,b,d,e,g,h}

FOLLOW:
A = {$,g,h}
B = {$,a,g,h}
C = {$,b,g,h}
S = {$}
```

Experiment-6

SHIFT REDUCE PARSING

NAMAN CHITKARA
RA1911029010046

CODE:

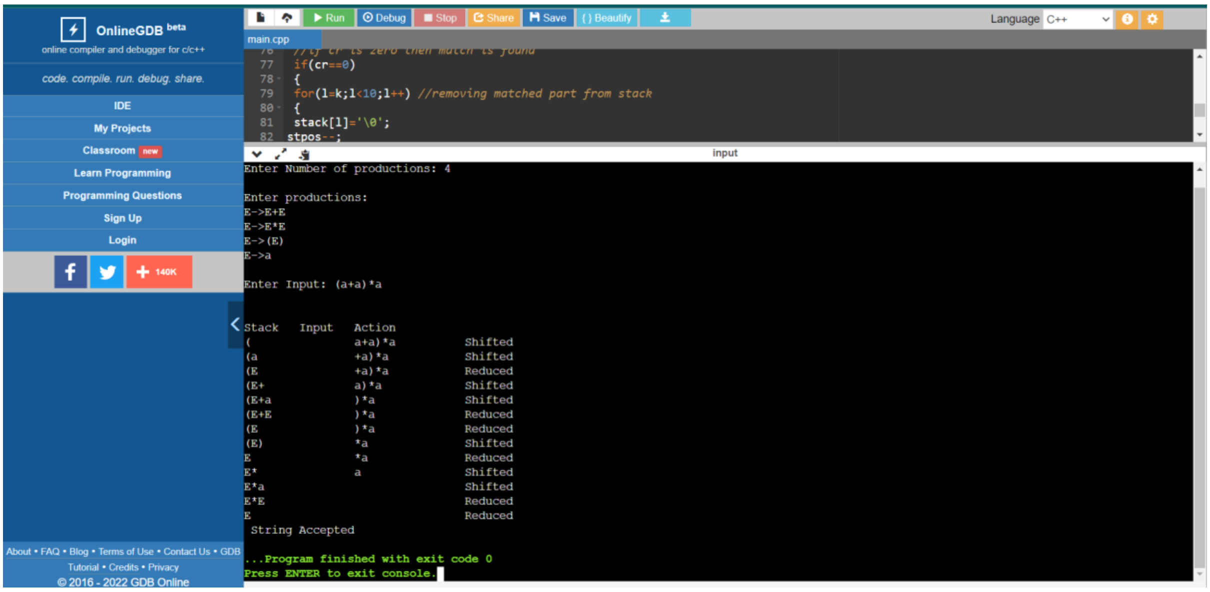
```
#include <iostream>
#include <cstring>
using namespace std;
struct grammer{
 char p[20];
 char prod[20];
}g[10];
int main()
{
 int i,stpos,j,k,l,m,o,p,f,r;
 int np,tspos,cr;
cout<<"\nEnter Number of productions: ";
cin>>np;
 char sc,ts[10];
cout<<"\nEnter productions: \n";
 for(i=0;i<np;i++)
 {
cin>>ts;
strncpy(g[i].p,ts,1);
strcpy(g[i].prod,&ts[3]);
 }
 char ip[10];
cout<<"\nEnter Input: ";
cin>>ip;
```

```
 int lip=strlen(ip);
 char stack[10];
stpos=0;
i=0;
 //moving input
sc=ip[i];
 stack[stpos]=sc;
i++;stpos++;
cout<<"\n\nStack\tInput\tAction";
 do
 {
 r=1;
 while(r!=0)
 {
cout<<"\n";
 for(p=0;p<stpos;p++)
 {
cout<<stack[p];
 }
cout<<"\t\t";
 for(p=i;p<lip;p++)
 {
cout<<ip[p];
 }
 if(r==2)
 {
cout<<"\t\tReduced";
 }
 else
```

```
{
cout<<"\t\tShifted";
}
r=0;
//try reducing
for(k=0;k<stpos;k++)
{
f=0;
for(l=0;l<10;l++)
{
ts[l]='\0';
}
tspos=0;
for(l=k;l<stpos;l++) //removing first caharcter
{
ts[tspos]=stack[l];
tspos++;
}
//now compare each possibility with production
for(m=0;m<np;m++)
{
cr = strcmp(ts,g[m].prod);
//if cr is zero then match is found
if(cr==0)
{
for(l=k;l<10;l++) //removing matched part from stack
{
stack[l]='\0';
stpos--;
```

```
}
stpos=k;
//concatinate the string
strcat(stack,g[m].p);
stpos++;
r=2;
}
}
}
}
//moving input
sc=ip[i];
stack[stpos]=sc;
i++;stpos++;
}while(strlen(stack)!=1 &&stpos!=lip);
if(strlen(stack)==1)
{
cout<<"\n String Accepted";
}
return 0;
}
```

OUTPUT:



OnlineGDB beta
online compiler and debugger for c/c++
code. compile. run. debug. share.
IDE
My Projects
Classroom new
Learn Programming
Programming Questions
Sign Up
Login
About • FAQ • Blog • Terms of Use • Contact Us • GDB
Tutorial • Credits • Privacy
© 2016 - 2022 GDB Online
Run
Debug
Stop
Share
Save
{ } Beautify
Language C++
main.cpp
if(cr==0)
{
for(l=k;l<10;l++) //removing matched part from stack
{
stack[l]='\0';
stpos--;
input
Enter Number of productions: 4
Enter productions:
E->E+E
E->E*E
E->(E)
E->a
Enter Input: (a+a)*a
Stack Input Action
(a+a)*a Shifted
(a +a)*a Shifted
(E +a)*a Reduced
(E+ a)*a Shifted
(E+a)*a Shifted
(E+E)*a Reduced
(E)*a Reduced
(E) *a Shifted
E *a Reduced
E* a Shifted
E*a Shifted
E*E Reduced
E Reduced
String Accepted
...Program finished with exit code 0
Press ENTER to exit console.

EXPERIMENT 7

LEADING AND TRAILING

NAMAN CHITKARA
RA1911029010046

Aim: Study and implement Leading and Trailing in CPP

Code:

```
#include<iostream>s
#include<cstring>
using namespace std;
int nt,t,top=0;
char s[50],NT[10],T[10],st[50],l[10][10],tr[50][50];

int searchnt(char a){
    int count=-1,i;
    for(i=0;i<nt;i++){
        if(NT[i]==a)
        return i;}
    return count;
}
int searchter(char a){
    int count=-1,i;
    for(i=0;i<t;i++){
        if(T[i]==a)
        return i;
    }
   return count;
}

void push(char a){
    s[top]=a;
    top++;
}
char pop(){
    top--;
    return s[top];
}
void installl(int a,int b){
    if(l[a][b]=='f'){
       l[a][b]='t';
       push(T[b]);
       push(NT[a]);
    }
}
void installt(int a,int b){
    if(tr[a][b]=='f'){
        tr[a][b]='t';
        push(T[b]);
        push(NT[a]);
    }
```

```
}

int main(){
    int i,s,k,j,n;
    char pr[30][30],b,c;
    cout<<"Enter the no of productions:";
    cin>>n;
    cout<<"Enter the productions one by one\n";
    for(i=0;i<n;i++)
        cin>>pr[i];
    nt=0;
    t=0;
    for(i=0;i<n;i++){
        if((searchnt(pr[i][0]))==-1)
            NT[nt++]=pr[i][0];
    }
    for(i=0;i<n;i++){
        for(j=3;j<strlen(pr[i]);j++){
            if(searchnt(pr[i][j])==-1){
                if(searchter(pr[i][j])==-1)
                    T[t++]=pr[i][j];
            }
        }
    }
    for(i=0;i<nt;i++){
        for(j=0;j<t;j++)
            l[i][j]='f';

    }
    for(i=0;i<nt;i++){
        for(j=0;j<t;j++)
            tr[i][j]='f';

    }
    for(i=0;i<nt;i++){
        for(j=0;j<n;j++){
            if(NT[(searchnt(pr[j][0]))]==NT[i]){
                if(searchter(pr[j][3])!=-1)
                    installl(searchnt(pr[j][0]),searchter(pr[j][3]));
                else{
                    for(k=3;k<strlen(pr[j]);k++){
                        if(searchnt(pr[j][k])==-1){
                            installl(searchnt(pr[j]
[0]),searchter(pr[j][k]));
                            break;

                        }
                    }
                }
            }
        }
    }
    while(top!=0){
```

```
        b=pop();
        c=pop();
        for(s=0;s<n;s++){
            if(pr[s][3]==b)
               installl(searchnt(pr[s][0]),searchter(c));
        }
    }
    for(i=0;i<nt;i++){
        cout<<"Leading["<<NT[i]<<"]"<<"\t{";
        for(j=0;j<t;j++){
            if(l[i][j]=='t')
               cout<<T[j]<<",";

        }
        cout<<"}\n";
    }

    top=0;
    for(i=0;i<nt;i++){
        for(j=0;j<n;j++){
            if(NT[searchnt(pr[j][0])]==NT[i]){
                if(searchter(pr[j][strlen(pr[j])-1])!=-1)
                    installt(searchnt(pr[j][0]),searchter(pr[j]
[strlen(pr[j])-1]));
                else{
                    for(k=(strlen(pr[j])-1);k>=3;k--){
                        if(searchnt(pr[j][k])==-1){
                            installt(searchnt(pr[j]
[0]),searchter(pr[j][k]));
                            break;
                        }
                    }
                }
            }
        }
    }
    while(top!=0){
        b=pop();
        c=pop();
        for(s=0;s<n;s++){
            if(pr[s][3]==b)
                installt(searchnt(pr[s][0]),searchter(c));
        }
    }
    for(i=0;i<nt;i++){
        cout<<"Trailing["<<NT[i]<<"]"<<"\t{";
        for(j=0;j<t;j++){
            if(tr[i][j]=='t')
                cout<<T[j]<<","; }
        cout<<"}\n";
    }
    return 0;
```

Output:

```
input
Enter the no of productions:6
Enter the productions one by one
E->E+E
E->T
T->T*F
F->(E)
T->F
F->i
Leading[E]       {+,*,(,i,}
Leading[T]       {*,(,i,}
Leading[F]       {(,i,}
Trailing[E]      {+,*,),i,}
Trailing[T]      {*,),i,}
Trailing[F]      {),i,}


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

Result:

Leading and Trailing has been studied, coded and successfully implemented in CPP.

Name : Naman CHITKARA
Reg. No.: RA1911029010046

18CSC304J-Systems Compiler Design
Predictive parsing Table

Aim:

A program to Construct a Predictive Parsing Table for the given Grammar.

LL(1) Parsing:

Here the 1st L represents that the scanning of the Input will be done from Left to Right manner and the second L shows that in this parsing technique we are going to use Left most Derivation Tree. And finally, the 1 represents the number of look-ahead, which means how many symbols are you going to see when you want to make a decision.

Algorithm to construct LL(1) Parsing Table:

Step 1: First check for left recursion in the grammar, if there is left recursion in the grammar remove that and go to step 2.

Step 2: Calculate First() and Follow() for all non-terminals.

First(): If there is a variable, and from that variable, if we try to drive all the strings then the beginning Terminal Symbol is called the First.

Follow(): What is the Terminal Symbol which follows a variable in the process of derivation.

Step 3: For each production A –> α. (A tends to alpha)

Find First(α) and for each terminal in First(α), make entry A –> α in the table.

If First(α) contains ε (epsilon) as terminal than, find the Follow(A) and for each terminal in Follow(A), make entry A –> α in the table.

If the First(α) contains ε and Follow(A) contains $ as terminal, then make entry A –> α in the table for the $.

To construct the parsing table, we have two functions:

In the table, rows will contain the Non-Terminals and the column will contain the Terminal Symbols. All the Null Productions of the Grammars will go under the Follow elements and the remaining productions will lie under the elements of the First set.

Code:

```
import re
import pandas as pd
def parse(user_input, start_symbol, parsingTable):
    flag = 0
    user_input = user_input + "$"
    stack = []
    stack.append("$")
    stack.append(start_symbol)
    input_len = len(user_input)
    index = 0
    while len(stack) > 0:
        top = stack[len(stack) - 1]
        print("Top =>", top)
        current_input = user_input[index]
        print("Current_Input => ", current_input)
        if top == current_input:
            stack.pop()
            index = index + 1
        else:
            key = top, current_input
            print(key)
            if key not in parsingTable:
                flag = 1
                break
            value = parsingTable[key]
            if value != '@':
```

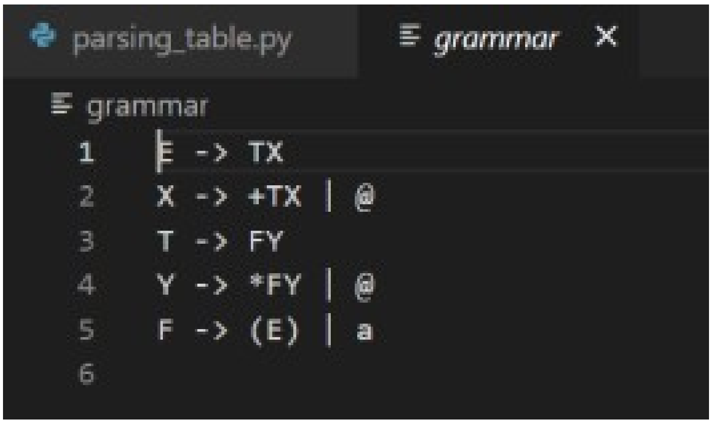
```
                value = value[::-1]
                value = list(value)
                stack.pop()
                for element in value:
                    stack.append(element)
            else:
                stack.pop()
    if flag == 0:
        print("String accepted!")
    else:
        print("String not accepted!")
def ll1(follow, productions):
    print("\nParsing Table\n")
    table = {}
    for key in productions:
        for value in productions[key]:
            if value != '@':
                for element in first(value, productions):
                    table[key, element] = value
            else:
                for element in follow[key]:
                    table[key, element] = value
    for key, val in table.items():
        print(key, "=>", val)
    new_table = {}
    for pair in table:
        new_table[pair[1]] = {}
    for pair in table: new_table[pair[1]]
        [pair[0]] = table[pair]
    print("\n\nTable\n")
    print(pd.DataFrame(new_table).fillna('-'))
    print("\n")
    return table
def follow(s, productions, ans):
```

```
    if len(s) != 1:
        return {}
    for key in productions:
        for value in productions[key]:
            f = value.find(s)
            if f != -1:
                if f == (len(value) - 1):
                    if key != s:
                        if key in ans:
                            temp = ans[key]
                        else:
                            ans = follow(key, productions, ans)
                            temp = ans[key]
                        ans[s] = ans[s].union(temp)
                else:
                    first_of_next = first(value[f + 1:], productions)
                    if '@' in first_of_next:
                        if key != s:
                            if key in ans:
                                temp = ans[key]
                            else:
                                ans = follow(key, productions, ans)
                                temp = ans[key]
                            ans[s] = ans[s].union(temp)
                            ans[s] = ans[s].union(first_of_next) - {'@'}
                    else:
                        ans[s] = ans[s].union(first_of_next)
    return ans
def first(s, productions):
    c = s[0]
    ans = set()
    if c.isupper():
        for st in productions[c]:
            if st == '@':
```

```
                if len(s) != 1:
                    ans = ans.union(first(s[1:], productions))
                else:
                    ans = ans.union('@')
            else:
                f = first(st, productions)
                ans = ans.union(x for x in f)
    else:
        ans = ans.union(c)
    return ans
if__name__== "__main__":
    productions = dict()
    grammar = open("grammar", "r")
    first_dict = dict()
    follow_dict = dict()
    flag = 1
    start = ""
    for line in grammar:
        l = re.split("( |->|\n|\|)*", line)
        line = line.replace(" ", "").replace("\n", "")
        l = line.split("->")
        lhs = l[0]
        rhs = set(l[1:-1]) - {''}
        rhs = l[1].split("|")
        if flag:
            flag = 0
            start = lhs
        productions[lhs] = rhs print('\
    nFirst\n')
    for lhs in productions:
        first_dict[lhs] = first(lhs, productions)
    for f in first_dict:
        print(str(f) + " : " + str(first_dict[f]))
    print("")
```

```
print('\nFollow\n')
for lhs in productions:
    follow_dict[lhs] = set()
follow_dict[start] = follow_dict[start].union('$')
for lhs in productions:
    follow_dict = follow(lhs, productions,
follow_dict) for lhs in productions:
    follow_dict = follow(lhs, productions,
follow_dict) for f in follow_dict:
    print(str(f) + " : " + str(follow_dict[f]))
ll1Table = ll1(follow_dict, productions)
```

Output:



```
First

E : {'a', '('}
X : {'+', '@'}
T : {'a', '('}
Y : {'@', '*'}
F : {'a', '('}


Follow

E : {'$', ')'}
X : {'$', ')'}
T : {'+', ')', '$'}
Y : {'+', ')', '$'}
F : {'+', ')', '*', '$'}

Parsing Table

('E', 'a') => TX
('E', '(') => TX
('X', '+') => +TX
('X', '$') => @
('X', ')') => @
('T', 'a') => FY
('T', '(') => FY
('Y', '*') => *FY
('Y', '+') => @
('Y', ')') => @
('Y', '$') => @
('F', '(') => (E)
('F', 'a') => a


Table

    a    (    +   $   )    *
E  TX   TX    -   -   -    -
T  FY   FY    -   -   -    -
F   a  (E)    -   -   -    -
X   -    -  +TX   @   @    -
Y   -    -    @   @   @  *FY
```

<u>Result:</u> The Predictive Parsing Table was successfully constructed for the given grammar.

Naman CHITKARA
RA1911029010046

Exp 10 - Intermediate code generation – Postfix, Prefix

Aim - To implement intermediate code generation using postfix and prefix

Code -

```
OPERATORS = set(['+', '-', '*', '/', '(', ')'])
PRI = {'+':1, '-':1, '*':2, '/':2}

### INFIX ===> POSTFIX ###
def infix_to_postfix(formula):
  stack = [] # only pop when the coming op has priority
  output = ''
  for ch in formula:
    if ch not in OPERATORS:
      output += ch
    elif ch == '(':
      stack.append('(')
    elif ch == ')':
      while stack and stack[-1] != '(':
        output += stack.pop()
      stack.pop() # pop '('
    else:
      while stack and stack[-1] != '(' and PRI[ch] <= PRI[stack[-1]]:
        output += stack.pop()
      stack.append(ch)
  # leftover
  while stack:
      output += stack.pop()
  print(f'POSTFIX: {output}')
  return output

### INFIX ===> PREFIX ###
def infix_to_prefix(formula):
  op_stack = []
  exp_stack = []
  for ch in formula:
    if not ch in OPERATORS:
```

```
        exp_stack.append(ch)
      elif ch == '(':
        op_stack.append(ch)
      elif ch == ')':
        while op_stack[-1] != '(':
          op = op_stack.pop()
          a = exp_stack.pop()
          b = exp_stack.pop()
          exp_stack.append( op+b+a )
        op_stack.pop() # pop '('
      else:
        while op_stack and op_stack[-1] != '(' and PRI[ch] <= PRI[op_stack[-1]]:
          op = op_stack.pop()
          a = exp_stack.pop()
          b = exp_stack.pop()
          exp_stack.append( op+b+a )
        op_stack.append(ch)

    # leftover
    while op_stack:
      op = op_stack.pop()
      a = exp_stack.pop()
      b = exp_stack.pop()
      exp_stack.append( op+b+a )
    print(f'PREFIX: {exp_stack[-1]}')
    return exp_stack[-1]

### THREE ADDRESS CODE GENERATION ###
def generate3AC(pos):
	print("### THREE ADDRESS CODE GENERATION ###")
	exp_stack = []
	t = 1

	for i in pos:
		if i not in OPERATORS:
			exp_stack.append(i)
		else:
			print(f't{t} := {exp_stack[-2]} {i} {exp_stack[-1]}')
			exp_stack=exp_stack[:-2]
			exp_stack.append(f't{t}')
			t+=1

expres = input("INPUT THE EXPRESSION: ")
pre = infix_to_prefix(expres)
```

```
pos = infix_to_postfix(expres)
generate3AC(pos)
```

Output -

```
INPUT THE EXPRESSION: a*b+c-d
PREFIX: -+*abcd
POSTFIX: ab*c+d-
### THREE ADDRESS CODE GENERATION ###
t1 := a * b
t2 := t1 + c
t3 := t2 - d
```

Result - Intermediate code generation using prefix and postfix implemented

Name : Naman CHITKARA

Reg. No.:RA1911029010046

Compiler Design

Exp-11

Aim:

To generate intermediate code- Quadruples, Triples form from the given expression.

Algorithm:

There are 3 representations of three address code namely

- Quadruple
- Triples
- Indirect Triples

1. Quadruple –

It is a structure with 4 fields namely op, arg1, arg2 and result. op denotes the operator and arg1 and arg2 denotes the two operands and result is used to store the result of the expression.

2. Triples –

This representation doesn't make use of extra temporary variables to represent a single operation; instead , when a reference to another triple's value is needed, a pointer to that triple is used. So, it consists of only three fields namely op, arg1 and arg2.

3. Indirect Triples –

This representation makes use of a pointer to the listing of all references to computations which are made separately and stored. It is similar in utility as compared to quadruple representation but requires less space than it. Temporaries are implicit and easier to rearrange code.

Code:

```
OPERATORS = set(['+', '-', '*', '/', '(', ')'])
PRI = {'+':1, '-':1, '*':2, '/':2}
```

```
### INFIX ===> POSTFIX ###
def infix_to_postfix(formula):
    stack = [] # only pop when the coming op has priority
    output = ''
    for ch in formula:
        if ch not in OPERATORS:
            output += ch
        elif ch == '(':
            stack.append('(')
        elif ch == ')':
            while stack and stack[-1] != '(':
                output += stack.pop()
            stack.pop() # pop '('
        else:
            while stack and stack[-1] != '(' and PRI[ch] <= PRI[stack[-1]]:
                output += stack.pop()
            stack.append(ch)
    # leftover
    while stack:
            output += stack.pop()
    print(f'POSTFIX: {output}')
    return output

### INFIX ===> PREFIX ###
def infix_to_prefix(formula):
    op_stack = []
    exp_stack = []
    for ch in formula:
        if not ch in OPERATORS:
            exp_stack.append(ch)
        elif ch == '(':
```

```
            op_stack.append(ch)
        elif ch == ')':
            while op_stack[-1] != '(':
                op = op_stack.pop()
                a = exp_stack.pop()
                b = exp_stack.pop()
                exp_stack.append( op+b+a )
            op_stack.pop() # pop '('
        else:
            while op_stack and op_stack[-1] != '(' and PRI[ch] <= PRI[op_stack[-1]]:
                op = op_stack.pop()
                a = exp_stack.pop()
                b = exp_stack.pop()
                exp_stack.append( op+b+a )
            op_stack.append(ch)

    # leftover
    while op_stack:
        op = op_stack.pop()
        a = exp_stack.pop()
        b = exp_stack.pop()
        exp_stack.append( op+b+a )
    print(f'PREFIX: {exp_stack[-1]}')
    return exp_stack[-1]

### THREE ADDRESS CODE GENERATION ###
def generate3AC(pos):
        print("### THREE ADDRESS CODE GENERATION ###")
        exp_stack = []
        t = 1
```

```
        for i in pos:
                if i not in OPERATORS:
                        exp_stack.append(i)
                else:
                        print(f't{t} := {exp_stack[-2]} {i} {exp_stack[-1]}')
                        exp_stack=exp_stack[:-2]
                        exp_stack.append(f't{t}')
                        t+=1

expres = input("INPUT THE EXPRESSION: ")
pre = infix_to_prefix(expres)
pos = infix_to_postfix(expres)
generate3AC(pos)
def Quadruple(pos):
  stack = []
  op = []
  x = 1
  for i in pos:
    if i not in OPERATORS:
      stack.append(i)
    elif i == '-':
        op1 = stack.pop()
        stack.append("t(%s)" %x)
        print("{0:^4s} | {1:^4s} | {2:^4s}|{3:4s}".format(i,op1,"(-)"," t(%s)" %x))
        x = x+1
        if stack != []:
          op2 = stack.pop()
          op1 = stack.pop()
          print("{0:^4s} | {1:^4s} | {2:^4s}|{3:4s}".format("+",op1,op2," t(%s)" %x))
          stack.append("t(%s)" %x)
          x = x+1
```

```
  elif i == '=':
   op2 = stack.pop()
   op1 = stack.pop()
   print("{0:^4s} | {1:^4s} | {2:^4s}|{3:4s}".format(i,op2,"(-)",op1))
  else:
   op1 = stack.pop()
   op2 = stack.pop()
   print("{0:^4s} | {1:^4s} | {2:^4s}|{3:4s}".format(i,op2,op1," t(%s)" %x))
   stack.append("t(%s)" %x)
   x = x+1
print("The quadruple for the expression ")
print(" OP | ARG 1 |ARG 2 |RESULT  ")
Quadruple(pos)

def Triple(pos):
    stack = []
    op = []
    x = 0
    for i in pos:
     if i not in OPERATORS:
      stack.append(i)
     elif i == '-':
      op1 = stack.pop()
      stack.append("(%s)" %x)
      print("{0:^4s} | {1:^4s} | {2:^4s}".format(i,op1,"(-)"))
      x = x+1
      if stack != []:
       op2 = stack.pop()
       op1 = stack.pop()
       print("{0:^4s} | {1:^4s} | {2:^4s}".format("+",op1,op2))
       stack.append("(%s)" %x)
```

```
            x = x+1
        elif i == '=':
          op2 = stack.pop()
          op1 = stack.pop()
          print("{0:^4s} | {1:^4s} | {2:^4s}".format(i,op1,op2))
        else:
          op1 = stack.pop()
          if stack != []:
            op2 = stack.pop()
            print("{0:^4s} | {1:^4s} | {2:^4s}".format(i,op2,op1))
            stack.append("(%s)" %x)
            x = x+1
print("The triple for given expression")
print("  OP | ARG 1 |ARG 2  ")
Triple(pos)
```

Output:

```
INPUT THE EXPRESSION: a+b*c
PREFIX: +a*bc
POSTFIX: abc*+
### THREE ADDRESS CODE GENERATION ###
t1 := b * c
t2 := a + t1
The quadruple for the expression
 OP | ARG 1 |ARG 2 |RESULT
 *    |  b    |   c   | t(1)
 +    |  a    | t(1)| t(2)
The triple for given expression
  OP | ARG 1 |ARG 2
 *    |  b    |   c
 +    |  a    | (0)


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

Result:

Successfully generated intermediate code- Quadruples, Triples form from the given expression.