



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

# **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 COMMUNICATIONS**

**Course Code:** 18CSC304J

**Course Name:** Compiler Design

## **PORTFOLIO**

**NAME: PRIYADARSHINI**

**REG. NO.: RA1911028010072**

**SECTION: I2**

**CSE - Cloud Computing**

## **EXPERIMENT NO: 1**

**Date: 11-01-2022**

### **Implementation of Lexical analyser for a C program**

**AIM:** To write a program for lexical analyser which takes a program as the input file and converts the content as tokens.

#### **PROCEDURE:**

1. Read the C program file
2. Create functions of key, identifiers, format specifiers, operators and keywords
3. Read each line in the file, split the words in each line and compare them using the different functions.
4. If the program contains the specific characters or tokens print the output.
5. Print the identifiers/tokens which are not correct as “Invalid Identifier”.

#### **CODE –**

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

bool isValidDelimiter(char ch) {
    if (ch == ' ' || ch == '+' || ch == '-' || ch == '*' ||
        ch == '/' || ch == ',' || ch == ';' || ch == '>' ||
        ch == '<' || ch == '=' || ch == '(' || ch == ')' ||
        ch == '[' || ch == ']' || ch == '{' || ch == '}')
```

```

    return (true);

    return (false);
}

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

// Returns 'true' if the string is a VALID IDENTIFIER.

bool isValidIdentifier(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' || isValidDelimiter(str[0]) == true)
        return (false);
    return (true);
}

bool isValidKeyword(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 isValidInteger(char* str) {
    int i, len = strlen(str);
    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 detectTokens(char* str) {
    int left = 0, right = 0;

```

```

int length = strlen(str);
while (right <= length && left <= right) {
    if (isValidDelimiter(str[right]) == false)
        right++;
    if (isValidDelimiter(str[right]) == true && left == right) {
        if (isValidOperator(str[right]) == true)
            printf("Valid operator : '%c'\n", str[right]);
        right++;
        left = right;
    } else if (isValidDelimiter(str[right]) == true && left != right || (right == length && left !=
right)) {
        char* subStr = subString(str, left, right - 1);
        if (isValidKeyword(subStr) == true)
            printf("Valid keyword : '%s'\n", subStr);
        else if (isValidInteger(subStr) == true)
            printf("Valid Integer : '%s'\n", subStr);
        else if (isRealNumber(subStr) == true)
            printf("Real Number : '%s'\n", subStr);
        else if (isValidIdentifier(subStr) == true
            && isValidDelimiter(str[right - 1]) == false)
            printf("Valid Identifier : '%s'\n", subStr);
        else if (isValidIdentifier(subStr) == false
            && isValidDelimiter(str[right - 1]) == false)
            printf("Invalid Identifier : '%s'\n", subStr);
        left = right;
    }
}
return;
}

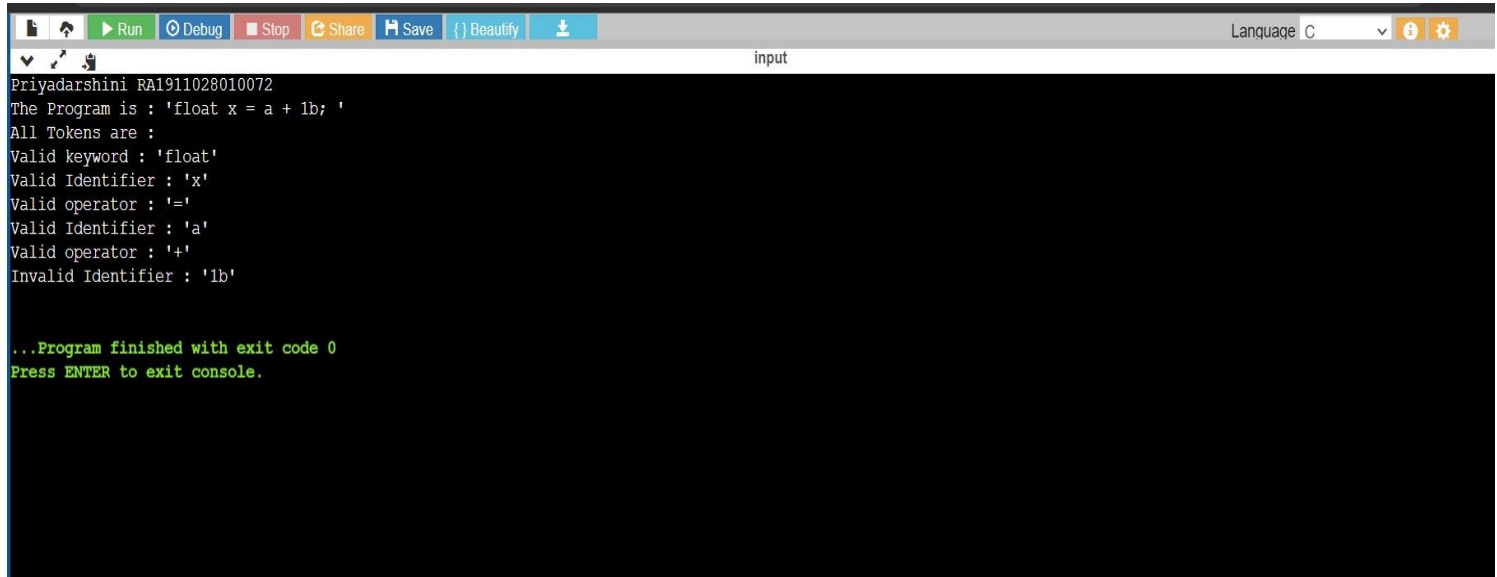
int main(){
    char str[100] = "float x = a + 1b; ";
    printf("The Program is : '%s' \n", str);
    printf("All Tokens are : \n");

```

```
detectTokens(str);
```

```
return (0);
```

## OUTPUT—



The screenshot shows a C++ IDE with a toolbar at the top containing icons for Run, Debug, Stop, Share, Save, Beautify, and a download icon. The language is set to C. The console output is as follows:

```
Priyadarshini RA1911028010072
The Program is : 'float x = a + 1b; '
All Tokens are :
Valid keyword : 'float'
Valid Identifier : 'x'
Valid operator : '='
Valid Identifier : 'a'
Valid operator : '+'
Invalid Identifier : '1b'

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

## RESULT-

The given program has been successfully executed and the tokens were identified.

**EXPERIMENT NO: 2**

**Date: 18-01-2022**

## **Regular Expression to NFA**

### **AIM :**

To convert a given regular expression into its non deterministic finite automata representation using C++

### **PROCEDURE :**

**STEP 1 :** A structure is created for each kind of regular expression that we want to create into NDFA.

**STEP 2 :** Under each structure , we define 3 variables - start state , alphabet input and n-state.

**STEP 3 :** Each state changes into another state based on the alphabet input symbol . All these are defined in the structures

**STEP 4 :** A switch case is made with 4 choices (4 different regular expressions) and based on the choice selected , an NFA diagram is shown as output.

## SOURCE CODE :

```
#include<bits/stdc++.h>
#include<stdio.h>
#include<conio.h>
using namespace std;
struct node
{
char start;
char alp;
node *nstate;
}*p,*p1,*p2,*p3,*p4,*p5,*p6,*p7,*p8;
char e='e';
void disp();
void re1()
{
p1=new(node);
p2=new(node);
p3=new(node);
p4=new(node);
p1->start='0';
p1->alp='e';
p1->nstate=p2;
p2->start='1';
p2->alp='a';
p2->nstate=p3;
p3->start='2';
p3->alp='e';
p3->nstate=p4;
p4->start='3';
p4->nstate=NULL;
disp();
getch();
}
void re2()
{
p1=new(node);
p2=new(node);
```



```

p3=new(node);
p4=new(node);
p5=new(node);
p6=new(node);
p7=new(node);
p8=new(node);
p1->start='0';
p1->alp='e';
p1->nstate=p2;
p2->start='1';
p2->alp='a';
p2->nstate=p3;
p3->start='2';
p3->alp='e';
p3->nstate=p4;
p4->start='5';
p4->alp=' ';
p4->nstate=p5;
p5->start='0';
p5->alp='e';
p5->nstate=p6;
p6->start='3';
p6->alp='b';
p6->nstate=p7;
p7->start='4';
p7->alp='e';
p7->nstate=p8;
p8->start='5';
p8->alp=' ';
p8->nstate=NULL;
disp();
getch();
}
void re3()
{
p1=new(node);
p2=new(node);

```

```

p3=new(node);
p1->start='0';
p1->alp='a';
p1->nstate=p2;
p2->start='1';
p2->alp='b';
p2->nstate=p3;
p3->start='2';
p3->alp=' ';
p3->nstate=NULL;
disp();
getch();
}
void re4()
{
p1=new(node);
p2=new(node);
p3=new(node);
p4=new(node);
p5=new(node);
p6=new(node);
p7=new(node);
p8=new(node);
p1->start='0';
p1->alp='e';
p1->nstate=p2;
p2->start='1';
p2->alp='a';
p2->nstate=p3;
p3->start='2';
p3->alp='e';
p3->nstate=p4;
p4->start='3';
p4->alp=' ';
p4->nstate=p5;
p5->start='0';
p5->alp='e';

```

```

p5->nstate=p6;
p6->start='3';
p6->alp=' ';
p6->nstate=p7;
p7->start='2';
p7->alp='e';
p7->nstate=p8;
p8->start='1';
p8->alp=' ';
p8->nstate=NULL;
disp();
getch();
}
void disp()
{
p=p1;
while(p!=NULL)
{
cout<<"\t"<<p->start<<"\t"<<p->alp;
p=p->nstate;
}
}
int main()
{
p=new(node);
int ch=1;
while(ch!=0)
{
cout<<"\nMenu"<<"\n1.a"<<"\n2.a/b"<<"\n3.ab"<<"\n4.a*";
cout<<"\n Enter the choice:";
cin>>ch;
switch(ch)
{
case 1:
{
re1();
break;

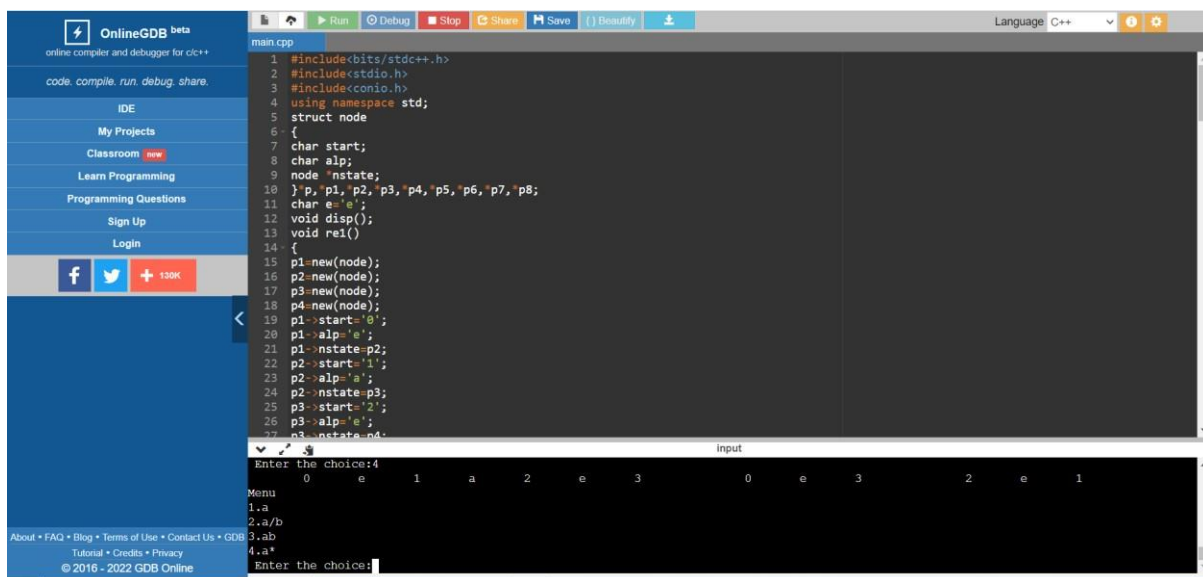
```

```

}
case 2:
{
re2();
break;
}
case 3:
{
re3();
break;
}
case 4:
{
re4();
break;
}
default:
{
exit(0);
}
}
}
return 0;
}

```

## SCREENSHOT OF OUTPUTS :



1. a

```
Enter the choice:1
0      e      1      a      2      e      3
Menu
1.a
2.a/b
3.ab
4.a*
Enter the choice:
```

2. a/b

```
Enter the choice:2
0      e      1      a      2      e      5      0      e      3      b      4      e      5
Menu
1.a
2.a/b
3.ab
4.a*
Enter the choice:
```

3. ab

```
Enter the choice:3
0      a      1      b      2
Menu
1.a
2.a/b
3.ab
4.a*
Enter the choice:
```

4. a\*

```
Enter the choice:4
0      e      1      a      2      e      3      0      e      3      2      e      1
Menu
1.a
2.a/b
3.ab
4.a*
Enter the choice:
```

## RESULT :

Thus we have successfully converted the given REs into NFA using C++.

## EXPERIMENT NO: 3

Date: 04-02-2022

### Conversion from NFA to DFA

**Aim:** To write and execute a C program to convert the given NFA to DFA

Input: NFA transition states

Output: DFA transition table

#### Procedure:

1. Construct the transition table of given NFA machine.
2. Scan the next states column in the transition table from initial state to final state.
3. If any of the next state consists more than one state on the single input alphabet. Then merge them and make it new state. Place this new constructed state in DFA transition table as present state.
4. The next state of this new constructed state on input alphabet will be the summation of each next state which parts in the NFA transition table.
5. Repeat step 2 to step 4 until all the states in NFA transition table will be scanned completely.
6. The final transition table must have a single next state at a single input alphabet.

#### Program:

```
import pandas as pd

nfa = { }
n = int(input("No. of states : "))
t = int(input("No. of transitions : "))
for i in range(n):
    state = input("state name : ")
    nfa[state] = { }
    for j in range(t):
        path = input("path : ")
        print("Enter end state from state { } travelling through path { } : ".format(state, path))
        reaching_state = [x for x in input().split()]
        nfa[state][path] = reaching_state
```

```

print("\nNFA :- \n")
print(nfa)
print("\nPrinting NFA table :- ")
nfa_table = pd.DataFrame(nfa)
print(nfa_table.transpose())

print("Enter final state of NFA : ")
nfa_final_state = [x for x in input().split()]

new_states_list = []

#.....

dfa = { }
keys_list = list(
    list(nfa.keys())[0])
path_list = list(nfa[keys_list[0]].keys())

dfa[keys_list[0]] = { }
for y in range(t):
    var = "".join(nfa[keys_list[0]][
        path_list[y]])
    dfa[keys_list[0]][path_list[y]] = var
    if var not in keys_list:
        new_states_list.append(var)
        keys_list.append(var)

while len(new_states_list) != 0:
    dfa[new_states_list[0]] = { }
    for _ in range(len(new_states_list[0])):
        for i in range(len(path_list)):
            temp = []
            for j in range(len(new_states_list[0])):
                temp += nfa[new_states_list[0]][j][path_list[i]]
            s = ""
            s = s.join(temp)
            if s not in keys_list:
                new_states_list.append(s)
                keys_list.append(s)
            dfa[new_states_list[0]][path_list[i]] = s

    new_states_list.remove(new_states_list[0])

print("\nDFA :- \n")
print(dfa)
print("\nPrinting DFA table :- ")
dfa_table = pd.DataFrame(dfa)
print(dfa_table.transpose())

dfa_states_list = list(dfa.keys())
dfa_final_states = []

```

```

for x in dfa_states_list:
    for i in x:
        if i in nfa_final_state:
            dfa_final_states.append(x)
break

print("\nFinal states of the DFA are : ", dfa_final_states)

```

### Sample output:

```

input
No. of states : 3
No. of transitions : 2
state name : A
path : 0
Enter end state from state A travelling through path 0 :
A
path : 1
Enter end state from state A travelling through path 1 :
A B
state name : B
path : 0
Enter end state from state B travelling through path 0 :
C
path : 1
Enter end state from state B travelling through path 1 :
C
state name : C
path : 0
Enter end state from state C travelling through path 0 :
path : 1
Enter end state from state C travelling through path 1 :

NFA :-

('A': {'0': ['A'], '1': ['A', 'B']}, 'B': {'0': ['C'], '1': ['C']}, 'C': {'0': [], '1': []})

Printing NFA table :-
      0      1
A  [A]  [A, B]
B  [C]  [C]
C  []   []
Enter final state of NFA :
C

DFA :-

('A': {'0': 'A', '1': 'AB'}, 'AB': {'0': 'AC', '1': 'ABC'}, 'AC': {'0': 'A', '1': 'AB'}, 'ABC': {'0': 'AC', '1': 'ABC'})

```

```

path : 1
Enter end state from state B travelling through path 1 :
C
state name : C
path : 0
Enter end state from state C travelling through path 0 :
path : 1
Enter end state from state C travelling through path 1 :

NFA :-

('A': {'0': ['A'], '1': ['A', 'B']}, 'B': {'0': ['C'], '1': ['C']}, 'C': {'0': [], '1': []})

Printing NFA table :-
      0      1
A  [A]  [A, B]
B  [C]  [C]
C  []   []
Enter final state of NFA :
C

DFA :-

('A': {'0': 'A', '1': 'AB'}, 'AB': {'0': 'AC', '1': 'ABC'}, 'AC': {'0': 'A', '1': 'AB'}, 'ABC': {'0': 'AC', '1': 'ABC'})

Printing DFA table :-
      0      1
A      A  AB
AB     AC  ABC
AC      A  AB
ABC     AC  ABC

Final states of the DFA are :  ['AC', 'ABC']

```

**Result:** Thus the C program to convert a NFA to DFA was executed and verified successfully



**EXPERIMENT NO: 4**  
**18-02-2022**

**Elimination of Left Recursion and Left Factoring**

**ELIMINATION OF LEFT RECURSION**

**AIM:** A program for Elimination of Left Recursion.

**PROCEDURE:**

1. Start the program.
2. Initialize the arrays for taking input from the user.
3. Prompt the user to input the no. of non-terminals having left recursion and no. of productions for these non-terminals.
4. Prompt the user to input the production for non-terminals.
5. Eliminate left recursion using the following rules:-  $A \rightarrow A\alpha_1 | A\alpha_2 | \dots | A\alpha_m$   $A \rightarrow \beta_1 | \beta_2 | \dots | \beta_n$  Then replace it by  $A \rightarrow \beta_i A' \quad i=1,2,3,\dots, m$   $A' \rightarrow \alpha_j \quad j=1,2,3,\dots, n$   $A' \rightarrow \epsilon$
6. After eliminating the left recursion by applying these rules, display the productions without left recursion.
7. Stop.

**CODE:**

```
#include <iostream>

#include <string>

using namespace std;

int main()
{
    int n, j, l, i, k;
    int length[10] = { };
    string d, a, b, flag;
    char c;
    cout<<"Enter Parent Non-Terminal: ";
    cin >> c;
```

```

d.push_back(c);
a += d + "'->";
d += "->";
b += d;
cout<<"Enter productions: ";
cin >> n;
for (int i = 0; i < n; i++)
{
    cout<<"Enter Production ";
    cout<<i + 1<<" :";
    cin >> flag;
    length[i] = flag.size();
    d += flag;
    if (i != n - 1)
    {
        d += "|";
    }
}
cout<<"The Production Rule is: ";
cout<<d<<endl;
for (i = 0, k = 3; i < n; i++)
{
    if (d[0] != d[k])
    {
        cout<<"Production: "<< i + 1;
        cout<<" does not have left recursion.";
        cout<<endl;
        if (d[k] == '#')
        {
            b.push_back(d[0]);
            b += "\"";

```

```

}
else

{
for (j = k; j < k + length[i]; j++)
{
b.push_back(d[j]);
}
k = j + 1;
b.push_back(d[0]);
b += "\\|";
}
}
else
{
cout<<"Production: "<< i + 1 ;
cout<< " has left recursion";
cout<< endl;
if (d[k] != '#')
{
for (l = k + 1; l < k + length[i]; l++)
{
a.push_back(d[l]);
}
k = l + 1;
a.push_back(d[0]);
a += "\\|";
}
}
}
a += "#";
cout << b << endl;

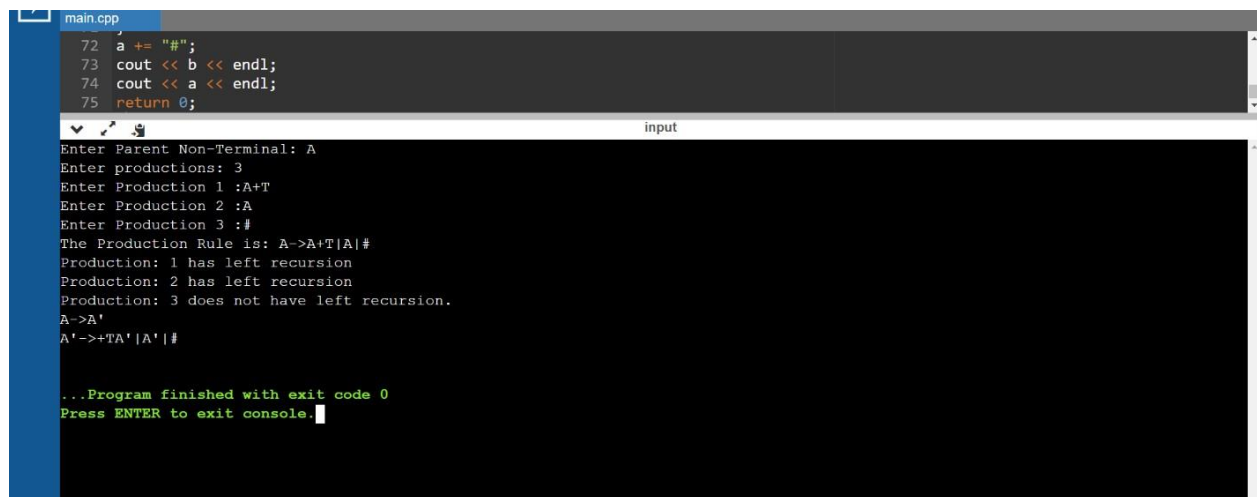
```

```
cout << a << endl;

return 0;

}
```

## OUTPUT:



```
main.cpp
72  a += "#";
73  cout << b << endl;
74  cout << a << endl;
75  return 0;

input
Enter Parent Non-Terminal: A
Enter productions: 3
Enter Production 1 :A+T
Enter Production 2 :A
Enter Production 3 :#
The Production Rule is: A->A+T|A|#
Production: 1 has left recursion
Production: 2 has left recursion
Production: 3 does not have left recursion.
A->A'
A'->+TA'|A'|#

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

## RESULT:

A program for Elimination of Left Recursion was run successfully.

## **LEFT FACTORING**

**AIM:** A program for implementation Of Left Factoring

### **PROCEDURE:**

1. Start
2. Ask the user to enter the set of productions
3. Check for common symbols in the given set of productions by comparing with:  $A \rightarrow aB1|aB2$
4. If found, replace the particular productions with:  $A \rightarrow aA' \quad A' \rightarrow B1 \mid B2 \mid \epsilon$
5. Display the output
6. Exit .

### **CODE:**

```
#include <bits/stdc++.h>
using namespace std;
int main()
{
    long long int i,j,k,l,n,m=9999999999,mini,ma=0;
    string s[100],st,ch,sc="",result,fs,maxi,rs="";
    vector<string>ss;
    vector<string>sp;
    cin>>n;

    for(i=1;i<=n;i++)
    {
        cin>>s[i];
    }
    for(i=1;i<=n;i++)
    {
        st=s[i];
        sc="";

        for(j=0;j<st.length();j++)
        {
            if(i==1)
            {
                fs=st[0];
            }
            if(st[j]==fs)
            {
                l=j;
            }
        }

        if(i==1)
```

```

{
for(k=1+1;k<st.length();k++)

{
    if(st[k]=='|')
    {
        ss.push_back(sc);
        sc="";
    }
    if(st[k]!='|')
    {
        ch=st[k];
        sc=sc+ch;
    }

}
ss.push_back(sc);
}
//cout<<sc<<endl;

}

for(k=0;k<ss.size();k++)
{
    mini=ss[k].size();
    m=min(m,mini);
    maxi=ss[k];
    //cout<<ss[k]<<endl;

}
//cout<<maxi<<endl;


for(k=0;k<m;k++)
{
    //cout<<ss[0][k]<<endl;
}

for (int i=0; i<m; i++)
{

    char current = ss[0][i];

    for (int j=1 ; j<ss.size(); j++)
    {
        if (ss[j][i] != current)
        {

```

```

        break;
    }
    result.push_back(current);
}

}

for(j=0;j<ss.size();j++)

{
    maxi=ss[j];
    //cout<<maxi<<result.length()<<endl;
    for(k=0;k<maxi.length();k++)
    {

        if(k>=result.length())
        {
            rs=rs+maxi[k];

        }

        //cout<<rs<<endl;

    }

    if(j!=ss.size()-1)
    {
        rs=rs+'|';
    }
}

cout<<fs<<"="<<result<<fs<<""<<endl;
cout<<fs<<""<<"="<<rs<<endl;

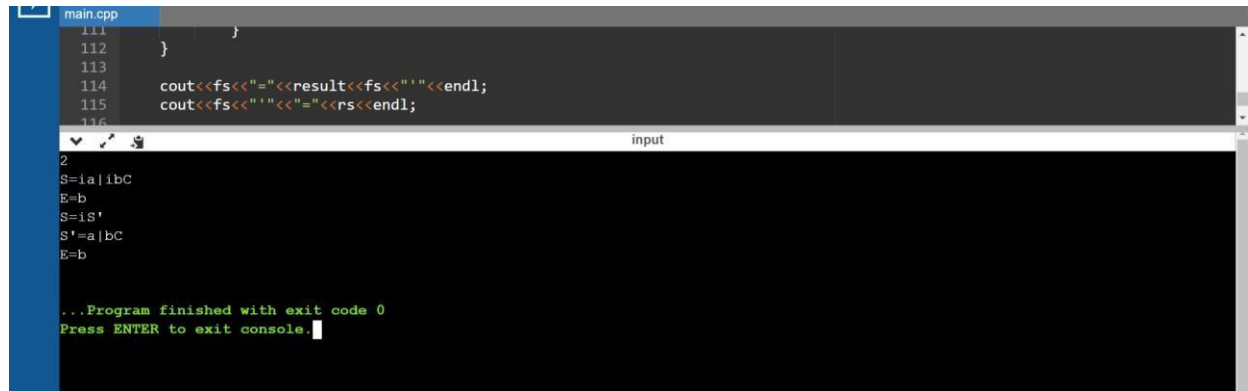
for(i=2;i<=n;i++)
{
    cout<<s[i]<<endl;
}

return 0;

}

```

## OUTPUT:



The screenshot shows a C++ IDE with two windows. The top window, titled 'main.cpp', contains the following code:

```
111  
112 }  
113  
114 cout<<fs<<"="<<result<<fs<<" "<<endl;  
115 cout<<fs<<" "<<"="<<rs<<endl;  
116
```

The bottom window, titled 'input', shows the program's execution output:

```
2  
S=ia|ibC  
E=b  
S=iS'  
S'=a|bC  
E=b  
  
...Program finished with exit code 0  
Press ENTER to exit console.
```

## RESULT:

A program for implementation Of Left Factoring was compiled and run successfully.



## EXPERIMENT NO: 5

Date: 24-02-2022

### FIRST and FOLLOW computation

#### Aim:

Finding the first and follow for the given CFG

#### Procedure:

##### FIRST-

1) To compute FIRST(X), where X is a grammar symbol, If X is a terminal, then  $\text{FIRST}(X) = \{X\}$ .

If  $X \rightarrow \epsilon$  is a production, then add  $\epsilon$  to FIRST(X).

If X is a non-terminal and  $X \rightarrow Y_1 Y_2 \cdots Y_k$  is a production, then add FIRST(Y<sub>1</sub>) to FIRST(X). If Y<sub>1</sub> derives  $\epsilon$ , then add FIRST(Y<sub>2</sub>) to FIRST(X).

##### FOLLOW-

1) For the FOLLOW(start symbol) place \$, where \$ is the input end marker.

If there is a production  $A \rightarrow \alpha B \beta$ , then everything in FIRST( $\beta$ ) except  $\epsilon$  is in FOLLOW(B).

If there is a production  $A \rightarrow \alpha B$ , or a production  $A \rightarrow \alpha B \beta$  where FIRST( $\beta$ ) contains  $\epsilon$ , then everything in FOLLOW(A) is in FOLLOW(B).

#### CODE –

##### 1) First

```
#include<stdio.h>
```

```
#include<ctype.h>
```

```
void FIRST(char[],char );
```

```
void addToResultSet(char[],char);
```

```

int numOfProductions;

char productionSet[10][10];

main()
{
    int i;

    char choice;

    char c;

    char result[20];

    printf("How many number of productions ? :");

    scanf(" %d",&numOfProductions);

    for(i=0;i<numOfProductions;i++)//read production string eg: E=E+T
    {

        printf("Enter productions Number %d : ",i+1);

        scanf(" %s",productionSet[i]);

    }

    do

    {

        printf("\n Find the FIRST of :");

        scanf(" %c",&c);

        FIRST(result,c); //Compute FIRST; Get Answer in 'result' array

        printf("\n FIRST(%c)= { ",c);

        for(i=0;result[i]!='\0';i++)

            printf(" %c ",result[i]);    //Display result

        printf("}\n");
    }
}

```

```

    printf("press 'y' to continue : ");

    scanf(" %c",&choice);}

while(choice=='y' || choice == 'Y');

}

/*

*Function FIRST:

*Compute the elements in FIRST(c) and write them

*in Result Array.

*/

void FIRST(char* Result,char c)

{

    int i,j,k;

    char subResult[20];

    int foundEpsilon;

    subResult[0]='\0';

    Result[0]='\0';

    //If X is terminal, FIRST(X) = {X}.

    if(!(isupper(c)))

    {

        addToResultSet(Result,c);

        return ;

    }

    //If X is non terminal

    //Read each production

```

```

for(i=0;i<numOfProductions;i++)

{

    //Find production with X as

    LHS

}
if(productionSet[i][0]==c)


{

//If  $X \rightarrow \epsilon$  is a production, then add  $\epsilon$  to FIRST(X).

if(productionSet[i][2]=='$') addToResultSet(Result,'$');

    //If X is a non-terminal, and  $X \rightarrow Y_1 Y_2 \dots Y_k$ 

    //is a production, then add a to FIRST(X)

    //if for some i, a is in FIRST( $Y_i$ ),

    //and  $\epsilon$  is in all of FIRST( $Y_1$ ), ..., FIRST( $Y_{i-1}$ ).

else

{

    j=2;

    while(productionSet[i][j]!='\0')

    {

        foundEpsilon=0;

        FIRST(subResult,productionSet[i][j]);

        for(k=0;subResult[k]!='\0';k++)

            addToResultSet(Result,subResult[k]);

        for(k=0;subResult[k]!='\0';k++)

            if(subResult[k]=='$')

```

```

        {
            foundEpsilon=1;

            break;
        }

//No  $\epsilon$  found, no need to check next element

if(!foundEpsilon)

break;

    j++;

    }

}

}

return ;

}

/* addToResultSet adds the computed
*element to result set.
*This code avoids multiple inclusion of elements
*/

void addToResultSet(char Result[],char val)

{

    int k;

    for(k=0 ;Result[k]!='\0';k++)

        if(Result[k]==val)

            return;

```

```
Result[k]=val;

Result[k+1]='\0';

}
```

## OUTPUT—

```
How many number of productions ? :4
Enter productions Number 1 : S=A
Enter productions Number 2 : A=aB/Ad
Enter productions Number 3 : B=b
Enter productions Number 4 : C=g

Find the FIRST of :S

FIRST(S)= { a }
press 'y' to continue : y

Find the FIRST of :A

FIRST(A)= { a }
press 'y' to continue : y

Find the FIRST of :B

FIRST(B)= { b }
press 'y' to continue : y

Find the FIRST of :C

FIRST(C)= { g }
press 'y' to continue :
n

...Program finished with exit code 0
```

## 2) Follow

```
#include<stdio.h>

#include<string.h>

#include <ctype.h>

int n,m=0,p,i=0,j=0;

char a[10][10],followResult[10];

void follow(char c);

void first(char c);

void addToResult(char);

int main()

{

    int i;

    int choice;

    char c,ch;

    printf("Enter the no.of productions: ");

    scanf("%d", &n);

    printf(" Enter %d productions\nProduction with multiple terms should be give as separate productions\n", n);

    for(i=0;i<n;i++)

        scanf("%s%c",a[i],&ch);

        // gets(a[i]);

    do

    {

        m=0;

        printf("Find FOLLOW of -->");

        scanf(" %c",&c);

        follow(c);
```

```

printf("FOLLOW(%c) = { ",c);

for(i=0;i<m;i++)

    printf("%c ",followResult[i]);

printf(" }\n");

printf("Do you want to continue(Press 1 to continue....)?");

scanf("%d%c",&choice,&ch);

}

while(choice==1);

}

void follow(char c)

{

    if(a[0][0]==c)addToResult('$');

    for(i=0;i<n;i++)

    {

        for(j=2;j<strlen(a[i]);j++)

        {

            if(a[i][j]==c)

            {

                if(a[i][j+1]!='\0')first(a[i][j+1]);

                if(a[i][j+1]=='\0'&&c!=a[i][0])

                    follow(a[i][0]);

            }

        }

    }

}

void first(char c)

{

    int k;

```



```

if(!(isupper(c)))
    //f[m++]=c;
    addToResult(c);
for(k=0;k<n;k++)
{
if(a[k][0]==c)
{
if(a[k][2]=='$') follow(a[i][0]);
else if(islower(a[k][2]))
    //f[m++]=a[k][2];
    addToResult(a[k][2]);
else first(a[k][2]);
}
}
}

```

```

void addToResult(char c)
{
    int i;
    for( i=0;i<=m;i++) if(followResult[i]==c)

        return; followResult[m++]=c;
}

```

## OUTPUT—

```
Enter the no.of productions: 4
Enter 4 productions
Production with multiple terms should be give as separate productions
S=A
A=aB/Ad
B=b
C=g
Find FOLLOW of -->S
FOLLOW(S) = { $ }
Do you want to continue(Press 1 to continue....)?1
Find FOLLOW of -->A
FOLLOW(A) = { }
Do you want to continue(Press 1 to continue....)?1
Find FOLLOW of -->B
FOLLOW(B) = { / }
Do you want to continue(Press 1 to continue....)?1
Find FOLLOW of -->C
FOLLOW(C) = { }
Do you want to continue(Press 1 to continue....)?
```

## RESULT-

Hence First and Follow was implemented and desired output was achieved.

## EXPERIMENT NO: 6

Date: 02-03-2022

### CONSTRUCTION OF PREDICTIVE PARSING TABLE

#### Aim:

To construct a predictive parsing table

#### Procedure:

##### LL(1) Parsing:

Here the 1st **L** represents that the scanning of the Input will be done from Left to Right manner and 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, means how many symbols are you going to see when you want to make a decision.

##### Construction of LL(1) Parsing Table:

To construct the Parsing table, we have two functions:

**1: 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.

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

Now, after computing the First and Follow set for each *Non-Terminal symbol* we have to construct the Parsing table. 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 First set.

#### CODE –

```
#include<stdio.h>
```

```
#include<conio.h>
```

```
#include<string.h>
```

```
char prol[7][10]={ "S","A","A","B","B","C","C"};
```

```
char pror[7][10]={ "A","Bb","Cd","aB","@","Cc","@"};
```

```
char prod[7][10]={ "S->A","A->Bb","A->Cd","B->aB","B->@","C->Cc","C->@"};
```

```
char first[7][10]={ "abcd","ab","cd","a@","@","c@","@"};
```

```
char follow[7][10]={ "$","$","$","a$","b$","c$","d$"};
```

```
char table[5][6][10];
```

```
numr(char c)
```

```
{
```

```
switch(c)
```

```
{
```

```
case 'S': return 0;
```

```
case 'A': return 1;
```

```
case 'B': return 2;
```

```
case 'C': return 3;
```

```
case 'a': return 0;
```

```
case 'b': return 1;
```

```
case 'c': return 2;
```

```
case 'd': return 3;
```

```
case '$': return 4;
```

```
}
```

```
return(2);
```

```
}
```

```
void main()
```

```
{
```

```
int i,j,k;
```

```
clrscr();
```

```
for(i=0;i<5;i++)
```

```
for(j=0;j<6;j++)
```

```
strcpy(table[i][j], " ");
```

```
printf("\nThe following is the predictive parsing table for the following grammar:\n");
```

```
for(i=0;i<7;i++)
```

```
printf("%s\n",prod[i]);
```

```
printf("\nPredictive parsing table is\n");
```

```
fflush(stdin);
```

```
for(i=0;i<7;i++)
```

```
{
```

```
k=strlen(first[i]);
```

```
for(j=0;j<10;j++)
```

```
if(first[i][j]!='@')
```

```
strcpy(table[numr(prol[i][0])+1][numr(first[i][j])+1],prod[i]);
```

```
}
```

```
for(i=0;i<7;i++)
```

```
{
```

```
if(strlen(pror[i])==1)
```

```
{
```

```
if(pror[i][0]=='@')
```

```
{
```

```
k=strlen(follow[i]);
```

```
for(j=0;j<k;j++)
```

```
strcpy(table[numr(prol[i][0])+1][numr(follow[i][j])+1],prod[i]);
```

```
}
```

```
}
```

```
}
```

```
strcpy(table[0][0], " ");
```

```
strcpy(table[0][1], "a");
```

```
strcpy(table[0][2], "b");
```

```
strcpy(table[0][3], "c");
```

```
strcpy(table[0][4], "d");
```

```
strcpy(table[0][5], "$");
```

```
strcpy(table[1][0], "S");
```

```
strcpy(table[2][0], "A");
```

```
strcpy(table[3][0], "B");
```

```
strcpy(table[4][0], "C");
```

```
printf("\n.....\n");
```



```
for(i=0;i<5;i++)
```

```
for(j=0;j<6;j++)
```

```
{
```

```
printf("%-10s",table[i][j]);
```

```
if(j==5)
```

```
printf("\n.....\n");
```

```
}
```

```
getch();
```

```
}
```

## OUTPUT

Running Turbo C Project



The following is the predictive parsing table for the following grammar:

S → A  
A → Bb  
A → Cd  
B → aB  
B → ε  
C → Cc  
C → ε

Predictive parsing table is

	a	b	c	d	\$
S	S → A	S → A	S → A	S → A	
A	A → Bb	A → Bb	A → Cd	A → Cd	
B	B → aB	B → ε	B → ε		B → ε
C			C → ε	C → ε	C → ε

## RESULT-

Hence, predictive parsing table code was implemented and desired output was achieved.

## EXPERIMENT NO: 7

Date: 03-03-2022

# CONSTRUCTION OF SHIFT-REDUCE PARSER

### Aim:

To construct a Shift-Reduce Parser

### PROCEDURE-

**Shift Reduce parser** attempts for the construction of parse in a similar manner as done in bottom up parsing i.e. the parse tree is constructed from leaves(bottom) to the root(up). A more general form of shift reduce parser is LR parser.

This parser requires some data structures i.e.

- A input buffer for storing the input string.
- A stack for storing and accessing the production rules.

### **Basic Operations –**

- **Shift:** This involves moving of symbols from input buffer onto the stack.
- **Reduce:** If the handle appears on top of the stack then, its reduction by using appropriate production rule is done i.e. RHS of production rule is popped out of stack and LHS of production rule is pushed onto the stack.
- **Accept:** If only start symbol is present in the stack and the input buffer is empty then, the parsing action is called accept. When accept action is obtained, it is means successful parsing is done.
- **Error:** This is the situation in which the parser can neither perform shift action nor reduce action and not even accept action.

### CODE –

```
#include<stdio.h>
```

```
#include<stdlib.h>
```

```
#include<conio.h>
```

```
#include<string.h>
```

```
char ip_sym[15],stack[15];
```

```
int ip_ptr=0,st_ptr=0,len,i;
```

```
char temp[2],temp2[2];
```

```
char act[15];
```

```
void check();
```

```
void main()
```

```
{
```

```
clrscr();
```

```
printf("\n\t\t SHIFT REDUCE PARSER\n");
```

```
printf("\n GRAMMER\n");
```

```
printf("\n E->E+E\n E->E/E");
```

```
printf("\n E->E*E\n E->(E)");
```

```
printf("\n E->a/b");
```

```
printf("\n enter the input symbol:\t");
```

```
gets(ip_sym);
```

```
printf("\n\t stack implementation table");
```

```
printf("\n stack \t\t input symbol\t\t action");
```

```
printf("\n_____ \t\t_____ \t\t_____ \n");
```

```
printf("\n $\t\t%s$\t\tt--",ip_sym);
```

```
strcpy(act,"shift");
```

```
temp[0]=ip_sym[ip_ptr];
```

```
temp[1]='\0';
```

```
strcat(act,temp);
```

```
len=strlen(ip_sym);
```

```
for(i=0;i<=len-1;i++)  
  
{  
  
stack[st_ptr]=ip_sym[ip_ptr];  
  
stack[st_ptr+1]='\0';  
  
ip_sym[ip_ptr]=' '  
  
ip_ptr++;  
  
printf("\n $%s\t\t%s$\t\t\t%s",stack,ip_sym,act);  
  
strcpy(act,"shift");  
  
temp[0]=ip_sym[ip_ptr];  
  
temp[1]='\0';  
  
strcat(act,temp);  
  
check();  
  
st_ptr++;
```

```
}  
st_ptr++;
```

```
check();
```

```
}
```

```
void check()
```

```
{
```

```
int flag=0;
```

```
temp2[0]=stack[st_ptr];
```

```
temp2[1]='\0';
```

```
if((!strcmpi(temp2,"a"))||(!strcmpi(temp2,"b")))
```

```
{
```

```
stack[st_ptr]='E';
```

```
if(!strcmpi(temp2,"a"))
```

```
printf("\n $%s\t\t%s$\t\ttE->a",stack,ip_sym);
```

```
else
```

```
printf("\n $%s\t\t%s$\t\ttE->b",stack,ip_sym);
```

```
flag=1;
```

```
}
```

```
if((!strcmpi(temp2,"+"))||(strcmpi(temp2,"*"))||(strcmpi(temp2,"/"))||(strcmpi(temp2,"("))||(strcmpi(temp2,")")))
```

```
{
```

```
flag=1;
```

```
}
```

```
if((!strcmpi(stack,"E+E"))||(strcmpi(stack,"E\E"))||(strcmpi(stack,"E*E"))||(strcmpi(stack,"(E)")))
```

```
{
```

```
strcpy(stack,"E");
```



```

st_ptr=0;

if(!strcmpi(stack,"E+E"))

printf("\n $%s\t\t%s$\t\tE->E+E",stack,ip_sym);


else


if(!strcmpi(stack,"E\E"))

printf("\n $%s\t\t%s$\t\tE->E\E",stack,ip_sym);


else


if(!strcmpi(stack,"E*E"))

printf("\n $%s\t\t%s$\t\tE->E*E",stack,ip_sym);


else


if(!strcmpi(stack,"E/E"))

printf("\n $%s\t\t%s$\t\tE->E/E",stack,ip_sym);

```

else

```
printf("\n $%s\t\t%s$\t\tE->E+E",stack,ip_sym);
```

```
flag=1;
```

```
}
```

```
if(!strcmpi(stack,"E")&&ip_ptr==len)
```

```
{
```

```
printf("\n $%s\t\t%s$\t\tACCEPT",stack,ip_sym);
```

```
getch();
```

```
exit(0);
```

```
}
```

```
if(flag==0)
```

```
{
```

```
printf("\n%s\t\t%s\t\treject",stack,ip_sym);
```

```
exit(0);
```

```
}
```

```
return;
```

## OUTPUT—

Running Turbo C Project

SHIFT REDUCE PARSER

GRAMMER

E→E+E

E→E/E

E→E∗E

E→(E)

E→a/b

enter the input symbol: (a)

stack

stack implementation table

input symbol

action

\$

(a)\$

—

\$(

a)\$

shift(

\$(a

)\$

shifta

\$(E

)\$

E→a

\$(E)

\$

shift)

\$E

\$

E→E+E

\$E

\$

ACCEPT

SHIFT REDUCE PARSER

GRAMMER

E→E+E

E→E/E

E→E∗E

E→(E)

E→a/b

enter the input symbol: a+b

stack

stack implementation table

input symbol

action

\$

a\$

—

\$a

\$

shifta

\$E

\$

E→a

\$E

\$

ACCEPT

## RESULT-

The given program has been successfully executed.

## EXPERIMENT NO: 8

Date: 19-03-2022

# LEADING AND TRAILING

**AIM :** A program to implement Leading and Trailing

### PROCEDURE :

1. For Leading, check for the first non-terminal.
2. If found, print it.
3. Look for next production for the same non-terminal.
4. If not found, recursively call the procedure for the single non-terminal present before the comma or End Of Production String.
5. Include it's results in the result of this non-terminal.
6. For trailing, we compute same as leading but we start from the end of the production to the beginning.
7. Stop

### CODE :

```
#include<iostream.h>
#include<string.h>
#include<conio.h>
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]);
}
}

void main()
{
int i,s,k,j,n;
char pr[30][30],b,c;
clrscr();
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";
}
getch();
}

```

## OUTPUT :

```

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

```

## RESULT :

Leading and Trailing were successfully calculated

## EXPERIMENT NO: 9

Date: 19-03-2022

### Computation of LR(0) Items

**Aim:** A program to implement LR(0) items

**Procedure: -**

1. Start.
2. Create structure for production with LHS and RHS.
3. Open file and read input from file.
4. Build state 0 from extra grammar Law  $S' \rightarrow S \$$  that is all start symbol of grammar and one Dot ( . ) before S symbol.
5. If Dot symbol is before a non-terminal, add grammar laws that this non-terminal is in Left Hand Side of that Law and set Dot in before of first part of Right Hand Side.
6. If state exists (a state with this Laws and same Dot position), use that instead.
7. Now find set of terminals and non-terminals in which Dot exist in before.
8. If step 7 Set is non-empty go to 9, else go to 10.
9. For each terminal/non-terminal in set step 7 create new state by using all grammar law that Dot position is before of that terminal/non-terminal in reference state by increasing Dot point to next part in Right Hand Side of that laws.
10. Go to step 5.
11. End of state building.
12. Display the output.
13. End.

**Program:**

```
#include<iostream.h>
#include<conio.h>
#include<string.h>
char prod[20][20],listofvar[26]="ABCDEFGHJKLMNOPQR";
int novar=1,i=0,j=0,k=0,n=0,m=0,arr[30];
```

```

int noitem=0;
struct Grammar
{
    char lhs;
    char rhs[8];
}g[20],item[20],clos[20][10];

int isvariable(char variable)
{
    for(int i=0;i<novar;i++)
        if(g[i].lhs==variable)
            return i+1;
    return 0;
}

void findclosure(int z, char a)
{
    int n=0,i=0,j=0,k=0,l=0;
    for(i=0;i<arr[z];i++)
    {
        for(j=0;j<strlen(clos[z][i].rhs);j++)
        {
            if(clos[z][i].rhs[j]=='.' && clos[z][i].rhs[j+1]==a)
            {
                clos[noitem][n].lhs=clos[z][i].lhs;
                strcpy(clos[noitem][n].rhs,clos[z][i].rhs);
                char temp=clos[noitem][n].rhs[j];
                clos[noitem][n].rhs[j]=clos[noitem][n].rhs[j+1];
                clos[noitem][n].rhs[j+1]=temp;
                n=n+1;
            }
        }
    }
}

```

```

    }
}

for(i=0;i<n;i++)
{
    for(j=0;j<strlen(clos[noitem][i].rhs);j++)
    {
        if(clos[noitem][i].rhs[j]=='.' && isvariable(clos[noitem][i].rhs[j+1])>0)
        {
            for(k=0;k<noitem;k++)
            {
                if(clos[noitem][i].rhs[j+1]==clos[0][k].lhs)
                {
                    for(l=0;l<n;l++)
                        if(clos[noitem][l].lhs==clos[0][k].lhs    &&
strcmp(clos[noitem][l].rhs,clos[0][k].rhs)==0)

                                break;

                    if(l==n)
                    {
                        clos[noitem][n].lhs=clos[0][k].lhs;
                        strcpy(clos[noitem][n].rhs,clos[0][k].rhs);
                        n=n+1;
                    }
                }
            }
        }
    }
}

arr[noitem]=n;
int flag=0;
for(i=0;i<noitem;i++)
{

```

```

        if(arr[i]==n)
        {
            for(j=0;j<arr[i];j++)
            {
                int c=0;
                for(k=0;k<arr[i];k++)
                    if(clos[noitem][k].lhs==clos[i][k].lhs      &&
strcmp(clos[noitem][k].rhs,clos[i][k].rhs)==0)
                        c=c+1;
                if(c==arr[i])
                {
                    flag=1;
                    goto exit;
                }
            }
        }
    }
    exit;;
    if(flag==0)
        arr[noitem++]=n;
}

```

```

void main()
{
    clrscr();
    cout<<"ENTER THE PRODUCTIONS OF THE GRAMMAR(0 TO END) :\n";
    do
    {
        cin>>prod[i++];
    }while(strcmp(prod[i-1],"0")!=0);
    for(n=0;n<i-1;n++)
    {

```

```

        m=0;
        j=novar;
        g[novar++].lhs=prod[n][0];
        for(k=3;k<strlen(prod[n]);k++)
        {
            if(prod[n][k] != '|')
                g[j].rhs[m++]=prod[n][k];
            if(prod[n][k]=='|')
            {
                g[j].rhs[m]='\0';
                m=0;
                j=novar;
                g[novar++].lhs=prod[n][0];
            }
        }
    }
    for(i=0;i<26;i++)
        if(!isvariable(listofvar[i]))
            break;
    g[0].lhs=listofvar[i];
    char temp[2]={g[1].lhs,'\0'};
    strcat(g[0].rhs,temp);
    cout<<"\n\n augmented grammar \n";
    for(i=0;i<novar;i++)
        cout<<endl<<g[i].lhs<<"->"<<g[i].rhs<<" ";
    getch();
    for(i=0;i<novar;i++)
    {
        clos[noitem][i].lhs=g[i].lhs;
        strcpy(clos[noitem][i].rhs,g[i].rhs);
    }

```

```

        strcpy(clos[noitem][i].rhs,".");
    else
    {
        for(int j=strlen(clos[noitem][i].rhs)+1;j>=0;j--)
            clos[noitem][i].rhs[j]=clos[noitem][i].rhs[j-1];
        clos[noitem][i].rhs[0]='.';
    }
}
arr[noitem++]=novar;
for(int z=0;z<noitem;z++)
{
    char list[10];
    int l=0;
    for(j=0;j<arr[z];j++)
    {
        for(k=0;k<strlen(clos[z][j].rhs)-1;k++)
        {
            if(clos[z][j].rhs[k]=='.')
            {
                for(m=0;m<l;m++)
                    if(list[m]==clos[z][j].rhs[k+1])
                        break;
                if(m==l)
                    list[l++]=clos[z][j].rhs[k+1];
            }
        }
    }
    for(int x=0;x<l;x++)
        findclosure(z,list[x]);
}
cout<<"\n THE SET OF ITEMS ARE \n\n";

```

```

        for(z=0;z<noitem;z++)
        {
            cout<<"\n I"<<z<<"\n\n";
            for(j=0;j<arr[z];j++)
                cout<<clos[z][j].lhs<<"->"<<clos[z][j].rhs<<"\n";
            getch();
        }
    getch();
}

```

## Output:-

```

ENTER THE PRODUCTIONS OF THE GRAMMAR(0 TO END) :
E->E+T
E->T
T->T*F
T->F
F->(E)
F->I
0

```

augmented grammar

```

A->E
E->E+T
E->T
T->T*F
T->F
F->(E)
F->I

```

THE SET OF ITEMS ARE

I0

```

A-> .E
E-> .E+T
E-> .T
T-> .T*F
T-> .F
F-> .(E)
F-> .I

```

I1

```

A->E.
E->E.+T

```

I2

```

E->T.
T->T.*F

```

I3

```

T->F.

```

I4

```

F->( .E)
E-> .E+T
E-> .T
T-> .T*F
T-> .F
F-> .(E)
F-> .I

```

I5

```

F->I.

```

I6

```

E->E+.T

```



```
I6
E->E+.T
T->*.T*F
T->*.F
F->*. (E)
F->*.I

I7
T->T*.F
F->*. (E)
F->*.I

I8
F-> (E.* )
E->E.*T

I9
E->E+T.
T->T.*F

I10
T->T*F.

I11
F-> (E) .
```

### Result:-

LR(0) Items are computed successfully

## **EXPERIMENT NO: 10**

Date: 08-04-2022

### **Intermediate code generation – Postfix, Prefix**

**Aim:** A program to implement Intermediate code generation – Postfix, Prefix.

**Procedure:-**

1. Declare set of operators.
2. Initialize an empty stack.
3. To convert INFIX to POSTFIX follow the following steps
4. Scan the infix expression from left to right.
5. If the scanned character is an operand, output it.
6. Else, 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.
7. 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.
8. If the scanned character is an '(', push it to the stack.
9. If the scanned character is an ')', pop the stack and output it until a '(' is encountered, and discard both the parenthesis.
10. Pop and output from the stack until it is not empty.
11. To convert INFIX to PREFIX follow the following steps
12. First, reverse the infix expression given in the problem.
13. Scan the expression from left to right.
14. Whenever the operands arrive, print them.
15. If the operator arrives and the stack is found to be empty, then simply push the operator into the stack.
16. Repeat steps 6 to 9 until the stack is empty

**Program:**

```
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:-



```
64 t = 1
65
66 for i in post
input
INPUT THE EXPRESSION: A+ (B-C) *D
PREFIX: +A*-BCD
POSTFIX: ABC-D*+
### THREE ADDRESS CODE GENERATION ###
t1 := B - C
t2 := t1 * D
t3 := A + t2

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

### Result:-

The program was successfully compiled and run.

## EXPERIMENT NO: 11

Date: 08-04-2022

### Intermediate code generation – Quadruple, Triple, Indirect triple

**Aim:** Intermediate code generation – Quadruple, Triple, Indirect triple

**Procedure:-**

The Procedure takes a sequence of three-address statements as input. For each three address statements of the form  $a := b \text{ op } c$  perform the various actions. These are as follows: 1. Invoke a function getreg to find out the location L where the result of computation  $b \text{ op } c$  should be stored.

2. Consult the address description for y to determine y'. If the value of y currently in memory and register both then prefer the register y' . If the value of y is not already in L then generate the instruction  $\text{MOV } y', L$  to place a copy of y in L.
3. Generate the instruction  $\text{OP } z', L$  where z' is used to show the current location of z. if z is in both then prefer a register to a memory location. Update the address descriptor of x to indicate that x is in location L. If x is in L then update its descriptor and remove x from all other descriptors.
4. If the current value of y or z have no next uses or not live on exit from the block or in register then alter the register descriptor to indicate that after execution of  $x := y \text{ op } z$  those register will no longer contain y or z.

**Program:**

```
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
INPUT THE EXPRESSION: A+(B-C)*D
PREFIX: +A*-BCD
POSTFIX: ABC-D*+
### THREE ADDRESS CODE GENERATION ###
t1 := B - C
t2 := t1 * D
t3 := A + t2
The quadruple for the expression
OP | ARG 1 | ARG 2 | RESULT
- | C | (-) | t(1)
+ | B | t(1) | t(2)
* | t(2) | D | t(3)
+ | A | t(3) | t(4)
The triple for given expression
OP | ARG 1 | ARG 2
- | C | (-)
+ | B | (0)
* | (1) | D
+ | A | (2)
```

## Result:-

The program was successfully compiled and run.

## EXPERIMENT NO: 12

Date: 08-04-2022

# Implementation of DAG

**Aim:** A program to implement DAG

**Procedure:**

1. The leaves of a graph are labeled by a unique identifier and that identifier can be variable names or constants.
2. Interior nodes of the graph are labeled by an operator symbol.
3. Nodes are also given a sequence of identifiers for labels to store the computed value.
4. If y operand is undefined then create node(y).
5. If z operand is undefined then for case(i) create node(z).
6. For case(i), create node(OP) whose right child is node(z) and left child is node(y).
7. For case(ii), check whether there is node(OP) with one child node(y).
8. For case(iii), node n will be node(y).
9. For node(x) delete x from the list of identifiers. Append x to attached identifiers list for the node n found in step 2. Finally set node(x) to n.

**Program:**

```
#include<iostream>
#include<string>
#include<unordered_map>
using namespace std;
class DAG
{ public:
    char label;
    char data;
```

```

DAG* left;
DAG* right;

DAG(char x){
    label='_';
    data=x;
    left=NULL;
    right=NULL;
}

DAG(char lb, char x, DAG* lt, DAG* rt){
    label=lb;
    data=x;
    left=lt;
    right=rt;
}

};

int main(){
    int n;
    n=3;
    string st[n];
    st[0]="A=x+y";
    st[1]="B=A*z";
    st[2]="C=B/x";
    unordered_map<char, DAG*> labelDAGNode;

    for(int i=0;i<3;i++){
        string stTemp=st[i];
        for(int j=0;j<5;j++){
            char tempLabel = stTemp[0];
            char tempLeft = stTemp[2];
            char tempData = stTemp[3];
            char tempRight = stTemp[4];
            DAG* leftPtr;

```

```

        leftPtr = new DAG(tempLeft);
    }
    else{
        leftPtr = labelDAGNode[tempLeft];
    }
    if(labelDAGNode.count(tempRight) == 0){
        rightPtr = new DAG(tempRight);
    }
    else{
        rightPtr = labelDAGNode[tempRight];
    }
    DAG* nn = new DAG(tempLabel,tempData,leftPtr,rightPtr);
    labelDAGNode.insert(make_pair(tempLabel,nn));
}
}
cout<<"Label   ptr   leftPtr   rightPtr"<<endl;
for(int i=0;i<n;i++){
    DAG* x=labelDAGNode[st[i][0]];
    cout<<st[i][0]<<"      "<<x->data<<"      ";
    if(x->left->label=='_')cout<<x->left->data;
    else cout<<x->left->label;
    cout<<"      ";
    if(x->right->label=='_')cout<<x->right->data;
    else cout<<x->right->label;
    cout<<endl;
}
return 0;

```



}

**Output :**

```
29 string st[10];
30 st[0]="A=x+y";
31 st[1]="B=A*z";
32 st[2]="C=B/x";
33 unordered_map<char, DAG*> labelDAGNode;
34
35 for(int i=0;i<3;i++){
36     string stTemp=st[i];
    input
    Label ptr leftPtr rightPtr
A      +      x      y
B      *      A      z
C      /      B      x
```

**Result:** The program was successfully compiled and run.

