

SRM INSTITUTE OF SCIENCE AND TECHNOLOGY

**FACULTY OF ENGINEERING & TECHNOLOGY**

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

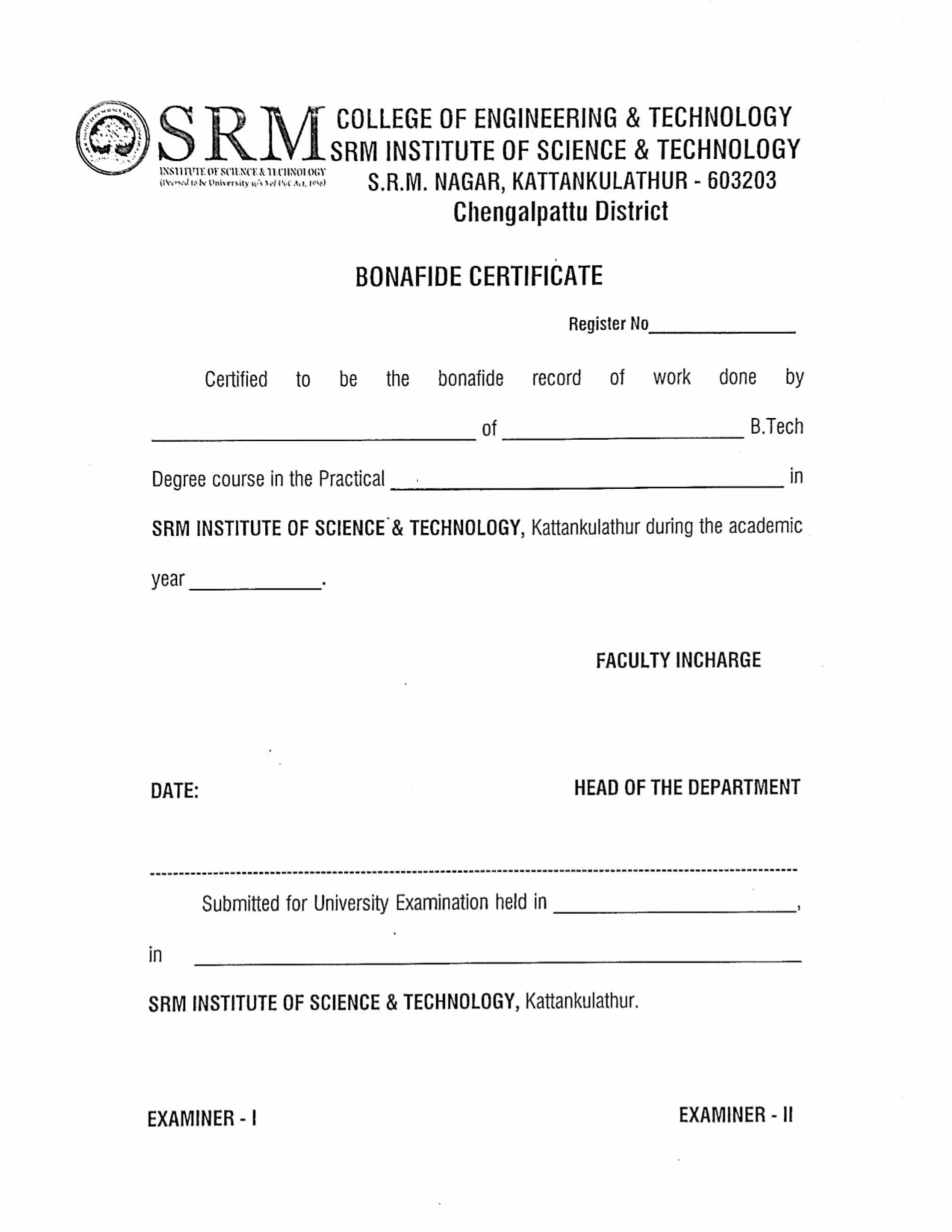
S.R.M. NAGAR, KATTANKULATHUR –603 203, KANCHEEPURAM DISTRICT

**SCHOOL OF COMPUTING DEPARTMENT OF COMPUTER SCIENCE**

**18CSC304J - COMPILER DESIGN LAB MANUAL**

Sai Krishna Movva

Reg No: RA1911003010751



Compiler Design Lab-18CSC304J

Exercise\_1 : Lexical Analyser

Sai Krishna Movva

RA1911003010751

**Aim:** To convert the given input program into tokens.

**Code:**

#include <stdbool.h>

#include <stdio.h>

#include <string.h>

#include <stdlib.h>

// Returns 'true' if the character is a DELIMITER.

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);

}

// Returns 'true' if the character is an OPERATOR.

bool isOperator(char ch)

{

if (ch == '+' || ch == '-' || ch == '\*' ||

ch == '/' || ch == '>' || ch == '<' ||

ch == '=')

return (true);

return (false);

}

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

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);

}

// Returns 'true' if the string is a KEYWORD.

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);

}

// Returns 'true' if the string is an INTEGER.

bool isInteger(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);

}

// Returns 'true' if the string is a REAL NUMBER.

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);

}

// Extracts the SUBSTRING.

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);

}

// Parsing the input STRING.

void parse(char\* str)

{

int left = 0, right = 0;

int len = strlen(str);

char keywords[10][100];

char integers[10][100];

char realNum[10][100];

char validIdent[10][100];

char invalidIden[10][100];

char operator[10];

int k = 0, integ = 0, realN = 0, validId = 0, invalidId = 0, oper = 0;

while (right <= len && left <= right) {

if (isDelimiter(str[right]) == false)

right++;

if (isDelimiter(str[right]) == true && left == right) {

if (isOperator(str[right]) == true){

operator[oper] = str[right];

oper++;

}

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){

strcpy(keywords[k], subStr);

k++;

}

else if (isInteger(subStr) == true){

strcpy(integers[integ], subStr);

integ++;

}

else if (isRealNumber(subStr) == true){

strcpy(realNum[realN++], subStr);

realN++;

}

else if (validIdentifier(subStr) == true

&& isDelimiter(str[right - 1]) == false){

strcpy(validIdent[validId], subStr);

validId++;

}

else if (validIdentifier(subStr) == false

&& isDelimiter(str[right - 1]) == false){

strcpy(invalidIden[invalidId], subStr);

invalidId++;

}

left = right;

}

}

printf("Keywords : ");

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

printf("%s ", keywords[i]);

}

printf("\n");

printf("Operators : ");

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

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

}

printf("\n");

printf("IDENTIFIER : ");

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

printf("%s ", validIdent[i]);

}

printf("\n");

printf("Numerical values : ");

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

printf("%s ", realNum[i]);

}

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

printf("%s ", integers[i]);

}

printf("\n");

printf("Others : ");

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

printf("%s ", invalidIden[i]);

}

return;

}

// DRIVER FUNCTION

int main()

{

// maximum length of string is 100 here

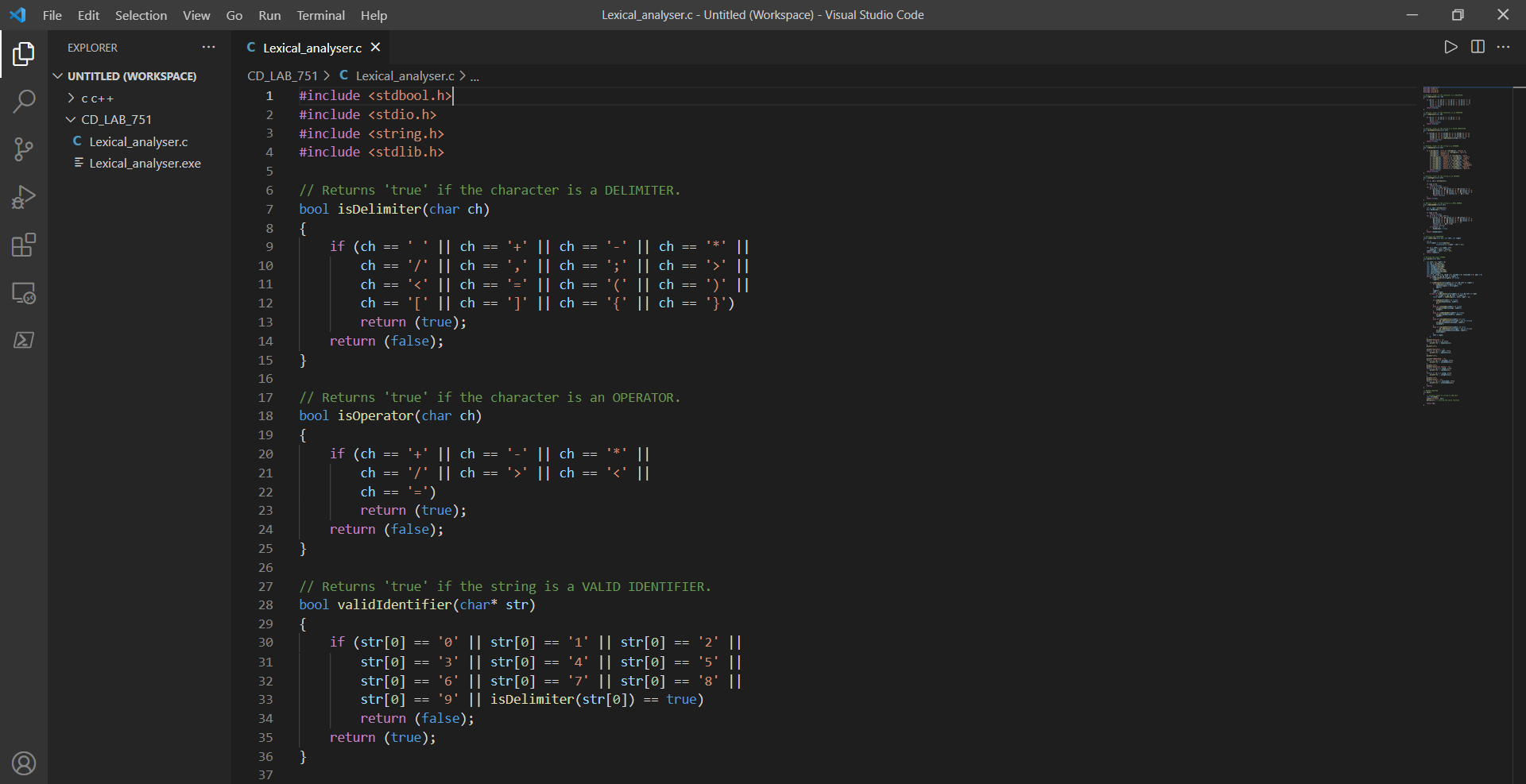
char str[5000];

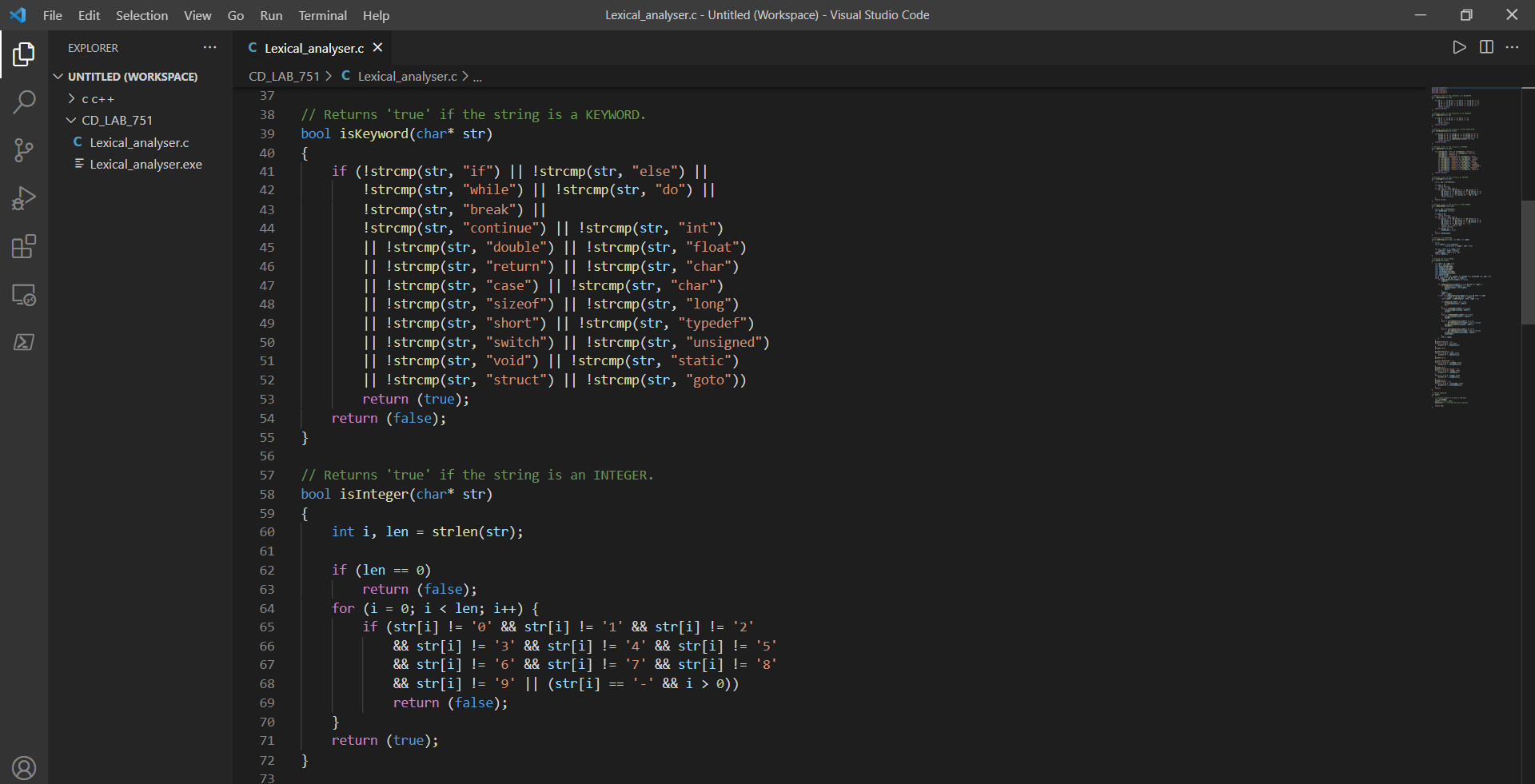
scanf("%[^\n]s", str);

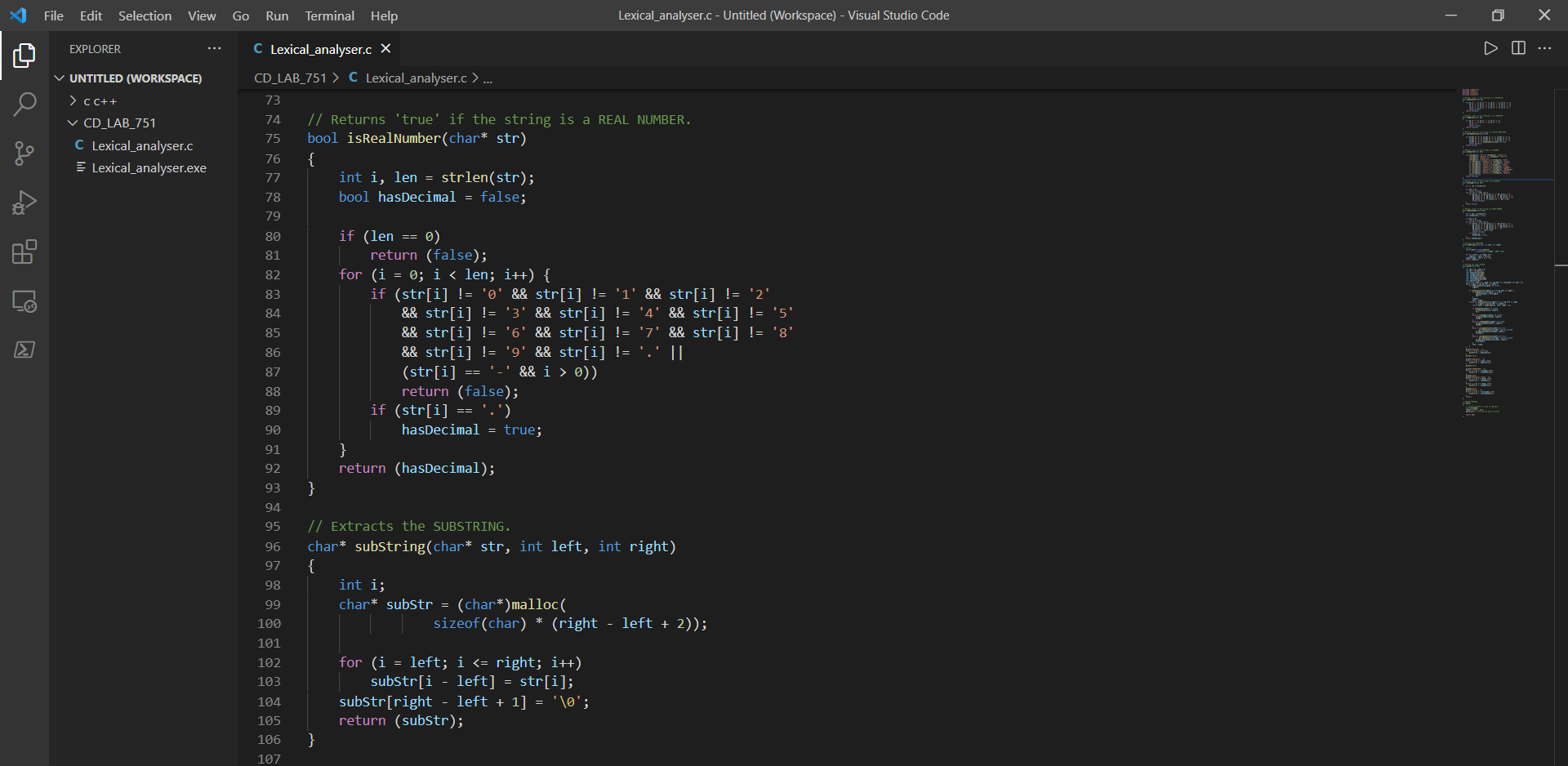
parse(str); // calling the parse function

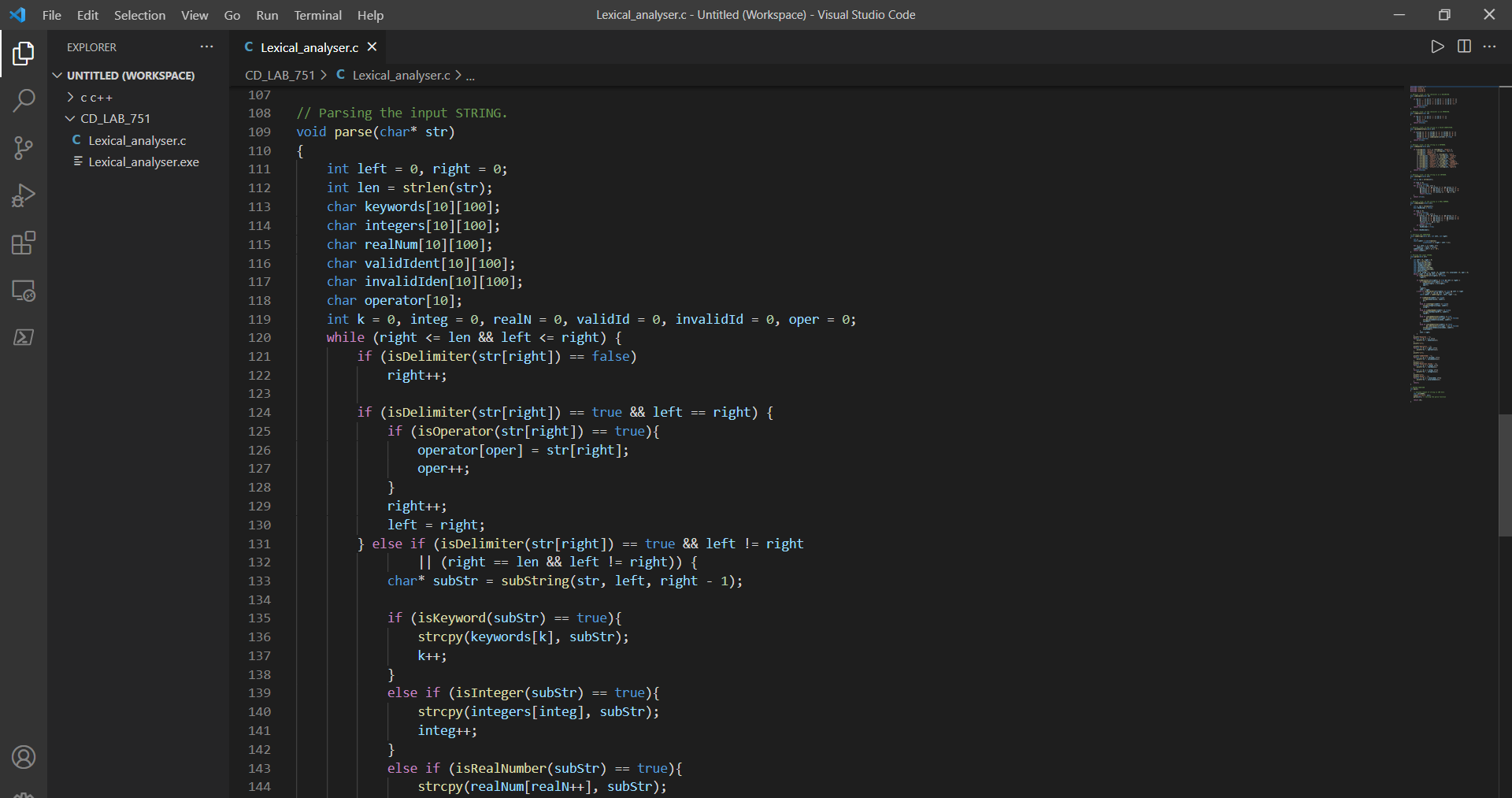
return (0);

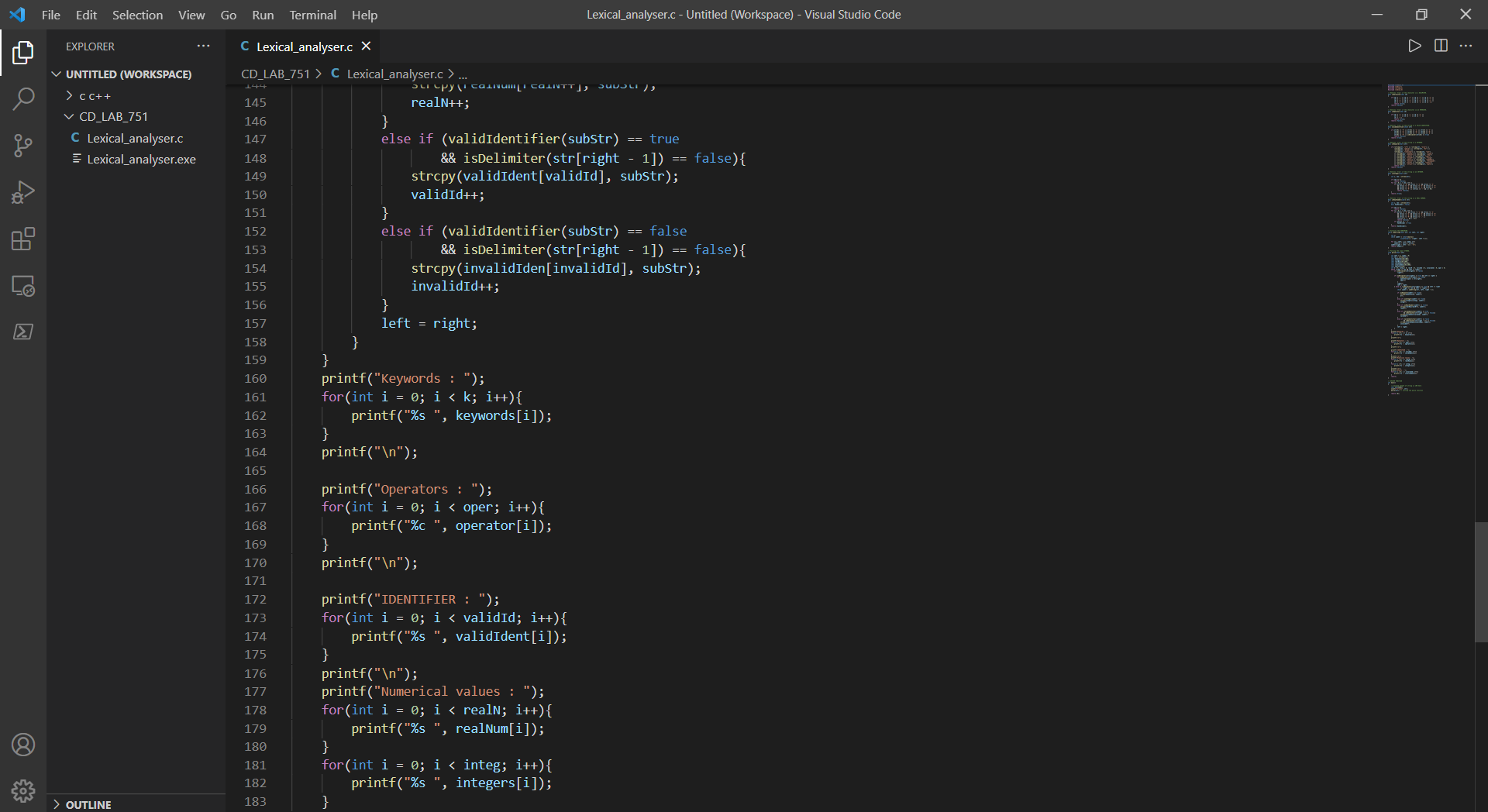
}

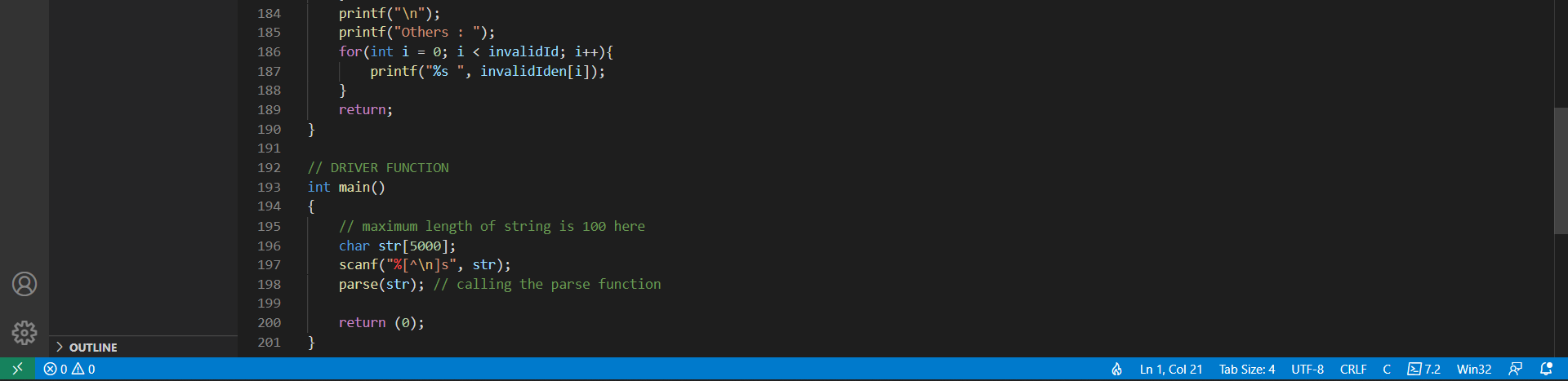




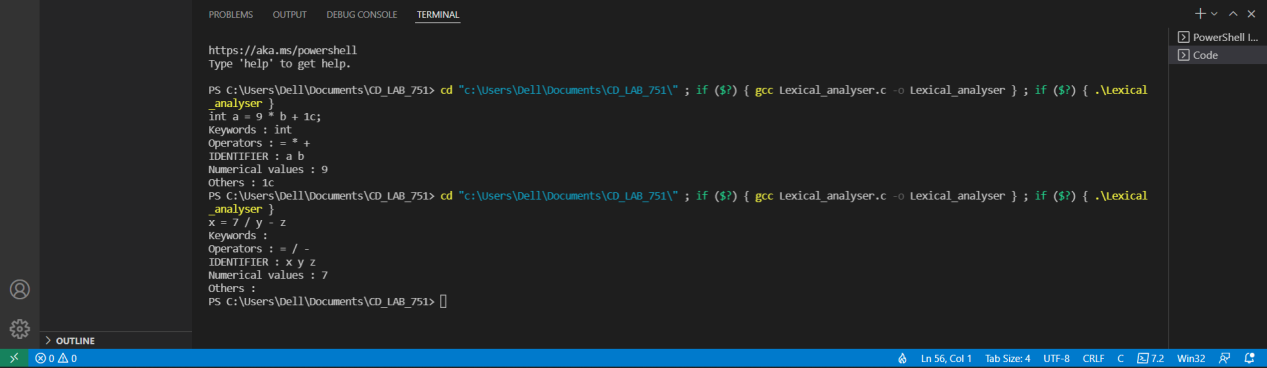








**Output:**

****

**Result:** Hence the given input is converted into tokens successfully.

Compiler Design Lab-18CSC304J

Exercise\_2 : Regular expression to NFA

Sai Krishna Movva

RA1911003010751

**Aim:** To convert the given Regular Expression to NFA.

**Code:**

#include<iostream>

#include<string.h>

int main()

{

printf("Enter the regular expression: ");

char reg[20];

int q[20][3],i,j,len,a,b;

for(a=0;a<20;a++)

{

for(b=0;b<3;b++)

{

q[a][b]=0;

}

}

scanf("%s",reg);

len=strlen(reg);

i=0;

j=1;

while(i<len)

{

if(reg[i]=='a'&&reg[i+1]!='|'&&reg[i+1]!='\*')

{

q[j][0]=j+1;

j++;

}

if(reg[i]=='b'&&reg[i+1]!='|'&&reg[i+1]!='\*')

{

q[j][1]=j+1;

j++;

}

if(reg[i]=='e'&&reg[i+1]!='|'&&reg[i+1]!='\*')

{

q[j][2]=j+1;

j++;

}

if(reg[i]=='a'&&reg[i+1]=='|'&&reg[i+2]=='b')

{

q[j][2]=((j+1)\*10)+(j+3);

j++;

q[j][0]=j+1;

j++;

q[j][2]=j+3;

j++;

q[j][1]=j+1;

j++;

q[j][2]=j+1;

j++;

i=i+2;

}

if(reg[i]=='b'&&reg[i+1]=='|'&&reg[i+2]=='a')

{

q[j][2]=((j+1)\*10)+(j+3);

j++;

q[j][1]=j+1;

j++;

q[j][2]=j+3;

j++;

q[j][0]=j+1;

j++;

q[j][2]=j+1;

j++;

i=i+2;

}

if(reg[i]=='a'&&reg[i+1]=='\*')

{

q[j][2]=((j+1)\*10)+(j+3);

j++;

q[j][0]=j+1;

j++;

q[j][2]=((j+1)\*10)+(j-1);

j++;

}

if(reg[i]=='b'&&reg[i+1]=='\*')

{

q[j][2]=((j+1)\*10)+(j+3);

j++;

q[j][1]=j+1;

j++;

q[j][2]=((j+1)\*10)+(j-1);

j++;

}

if(reg[i]==')'&&reg[i+1]=='\*')

{

q[0][2]=((j+1)\*10)+1;

q[j][2]=((j+1)\*10)+1;

j++;

}

i++;

}

printf("Transition function \n");

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

{

if(q[i][0]!=0)

printf("\n q[%d,a]-->%d",i,q[i][0]);

if(q[i][1]!=0)

printf("\n q[%d,b]-->%d",i,q[i][1]);

if(q[i][2]!=0)

{

if(q[i][2]<10)

printf("\n q[%d,e]-->%d",i,q[i][2]);

else

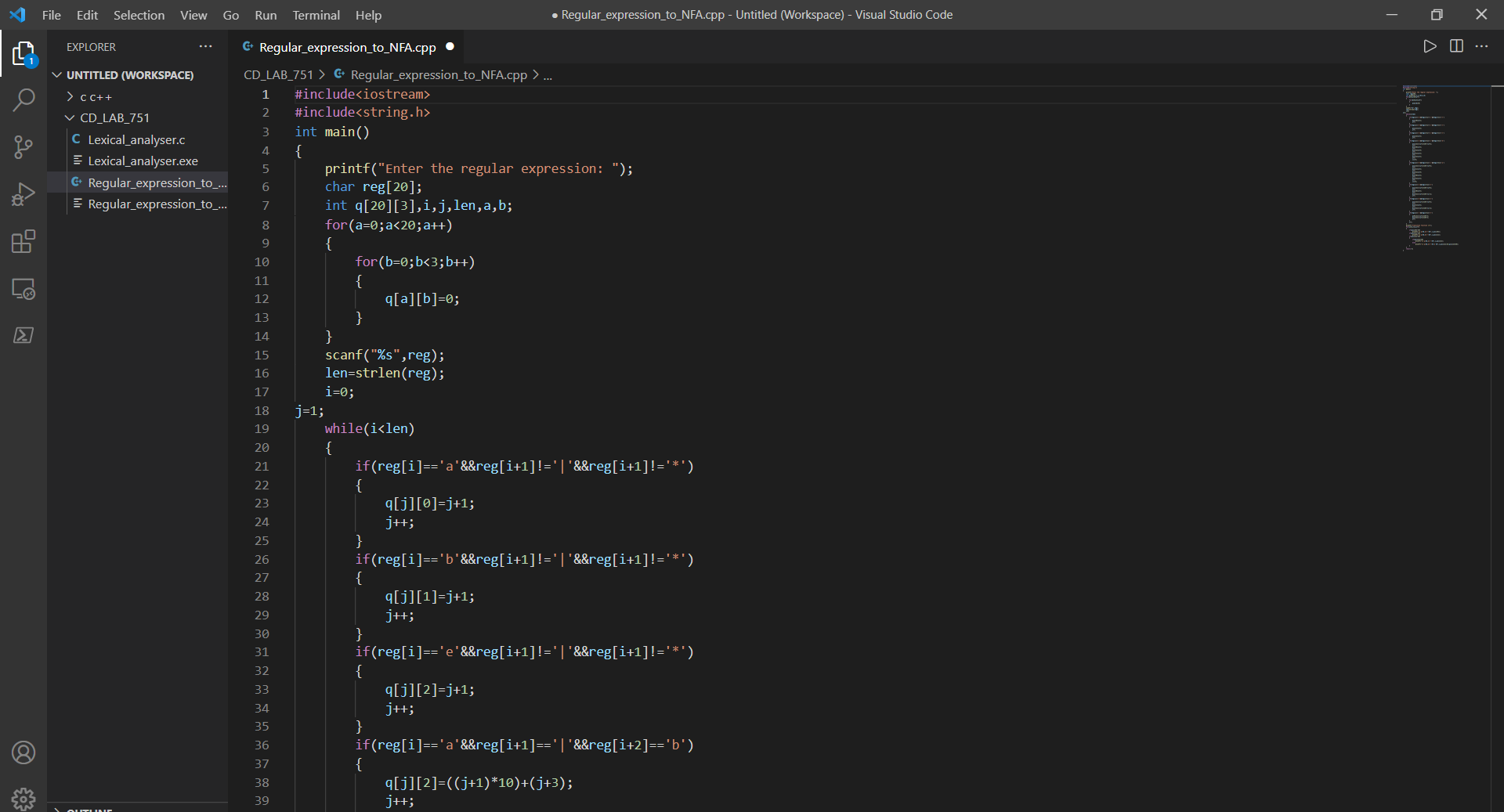
printf("\n q[%d,e]-->%d & %d",i,q[i][2]/10,q[i][2]%10);

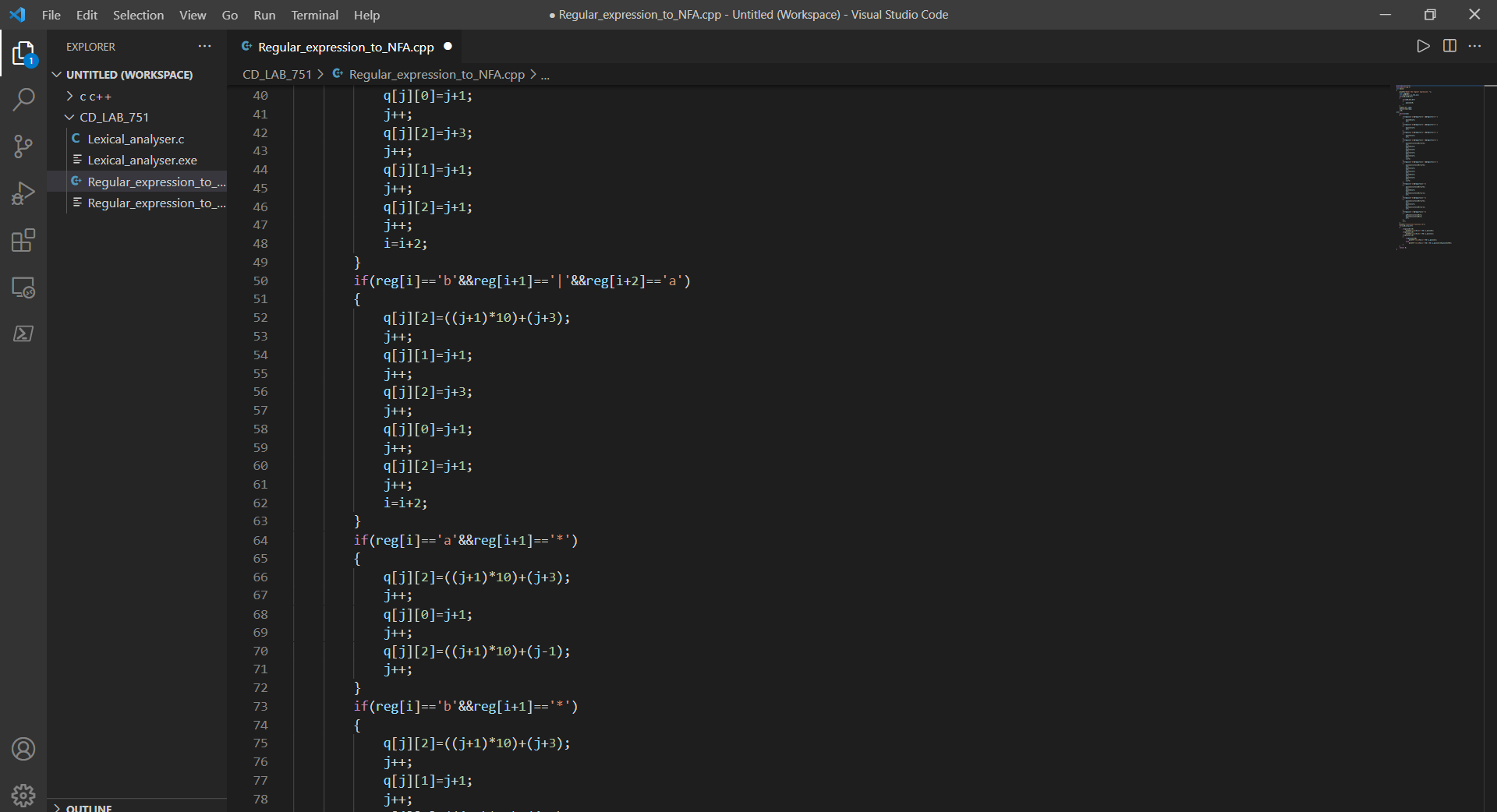
}

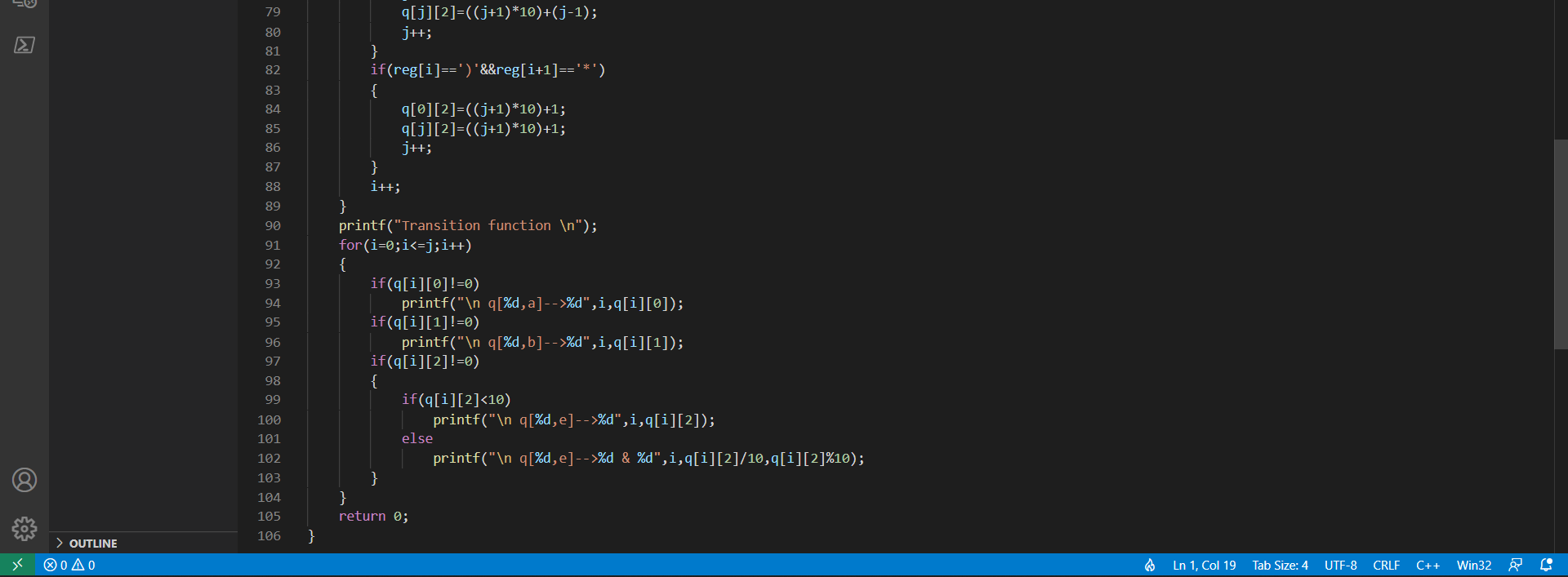
}

return 0;

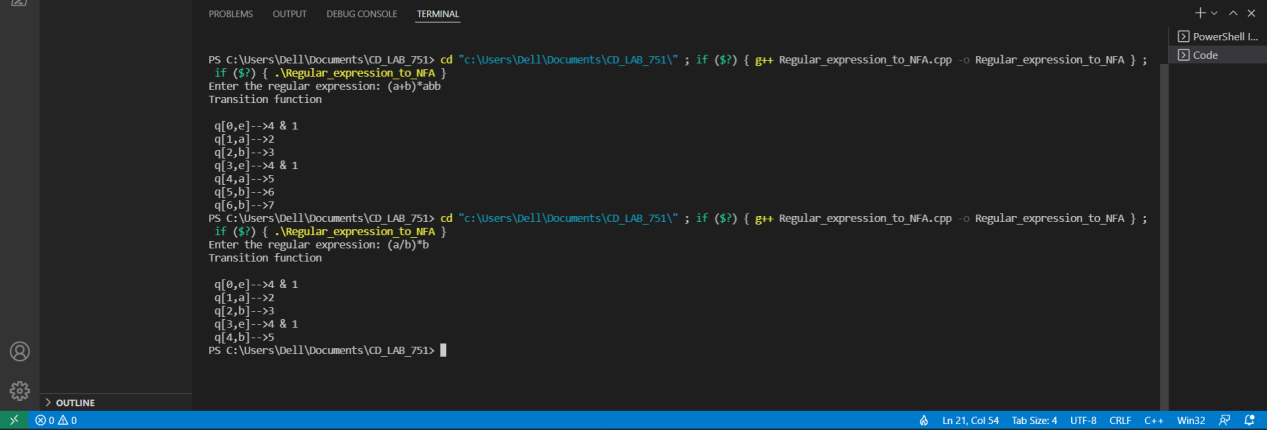
}







**Output:**

****

**Result:** Hence, the given Regular expression is converted into NFA.

Compiler Design Lab-18CSC304J

Exercise : Left Factoring

Sai Krishna Movva

RA1911003010751

**Aim:** To develop a program to eliminate left Factoring.

**Code:**

#include<string.h>

#include<stdio.h>

#include<stdlib.h>

#include<conio.h>

void main()

{

char ch,lhs[20][20],rhs[20][20][20],temp[20],temp1[20];

int n,n1,count[20],x,y,i,j,k,c[20];

printf("\nEnter the no. of productions : ");

scanf("%d",&n);

n1=n;

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

{

printf("\nProduction %d \nEnter the no. of productions : ",i+1);

scanf("%d",&c[i]);

printf("\nEnter LHS : ");

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

for(j=0;j<c[i];j++)

{

printf("%s->",lhs[i]);

scanf("%s",rhs[i][j]);

}

}

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

{

count[i]=1;

while(memcmp(rhs[i][0],rhs[i][1],count[i])==0)

count[i]++;

}

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

{

count[i]--;

if(count[i]>0)

{

strcpy(lhs[n1],lhs[i]);

strcat(lhs[i],"'");

for(k=0;k<count[i];k++)

temp1[k] = rhs[i][0][k];

temp1[k++] = '\0';

for(j=0;j<c[i];j++)

{

for(k=count[i],x=0;k<strlen(rhs[i][j]);x++,k++)

temp[x] = rhs[i][j][k];

temp[x++] = '\0';

if(strlen(rhs[i][j])==1)

strcpy(rhs[n1][1],rhs[i][j]);

strcpy(rhs[i][j],temp);

}

c[n1]=2;

strcpy(rhs[n1][0],temp1);

strcat(rhs[n1][0],lhs[n1]);

strcat(rhs[n1][0],"'");

n1++;

}

}

printf("\n\nThe resulting productions are : \n");

for(i=0;i<n1;i++)

{

if(i==0)

printf("\n %s -> %c|",lhs[i],(char)238);

else

printf("\n %s -> ",lhs[i]);

for(j=0;j<c[i];j++)

{

printf(" %s ",rhs[i][j]);

if((j+1)!=c[i])

printf("|");

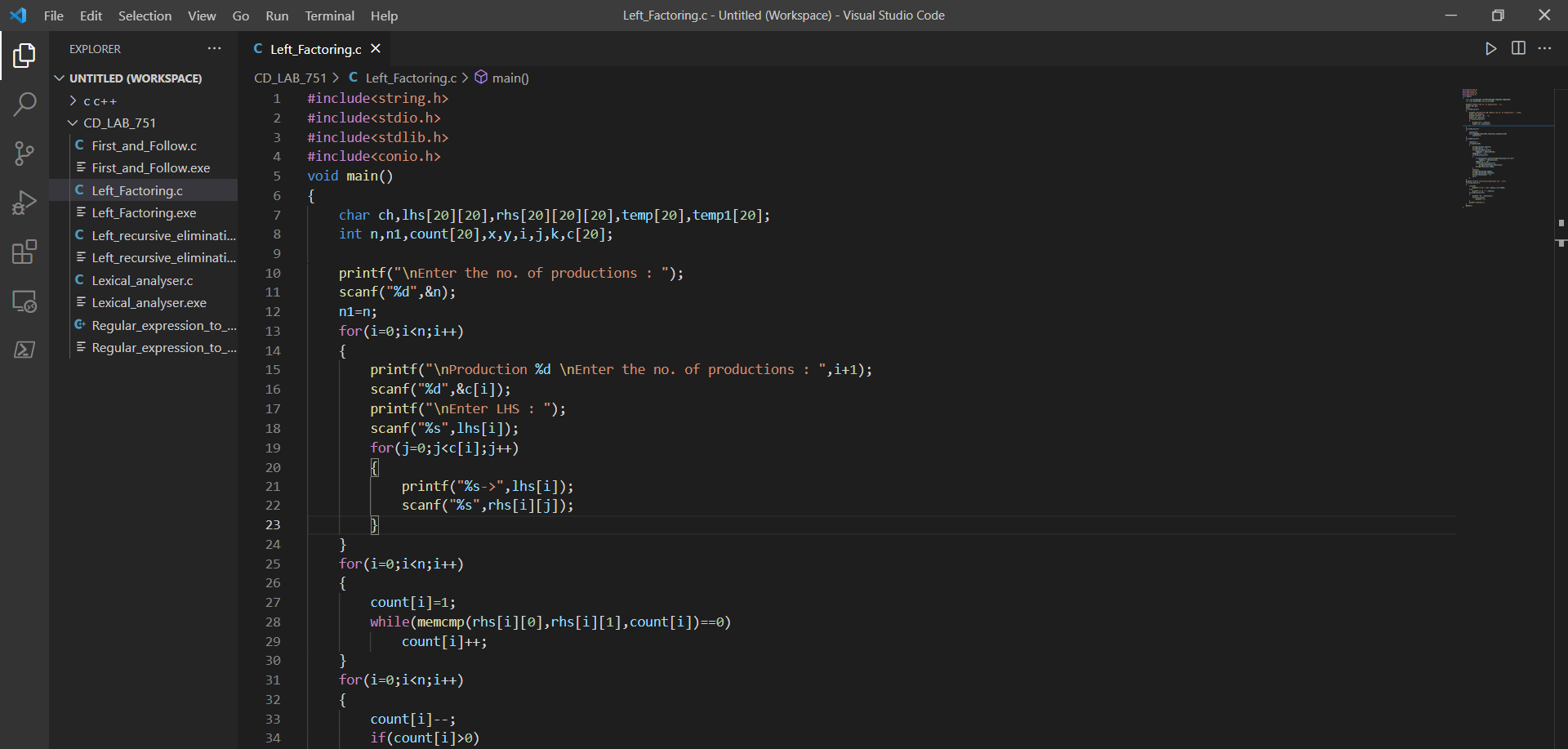
}

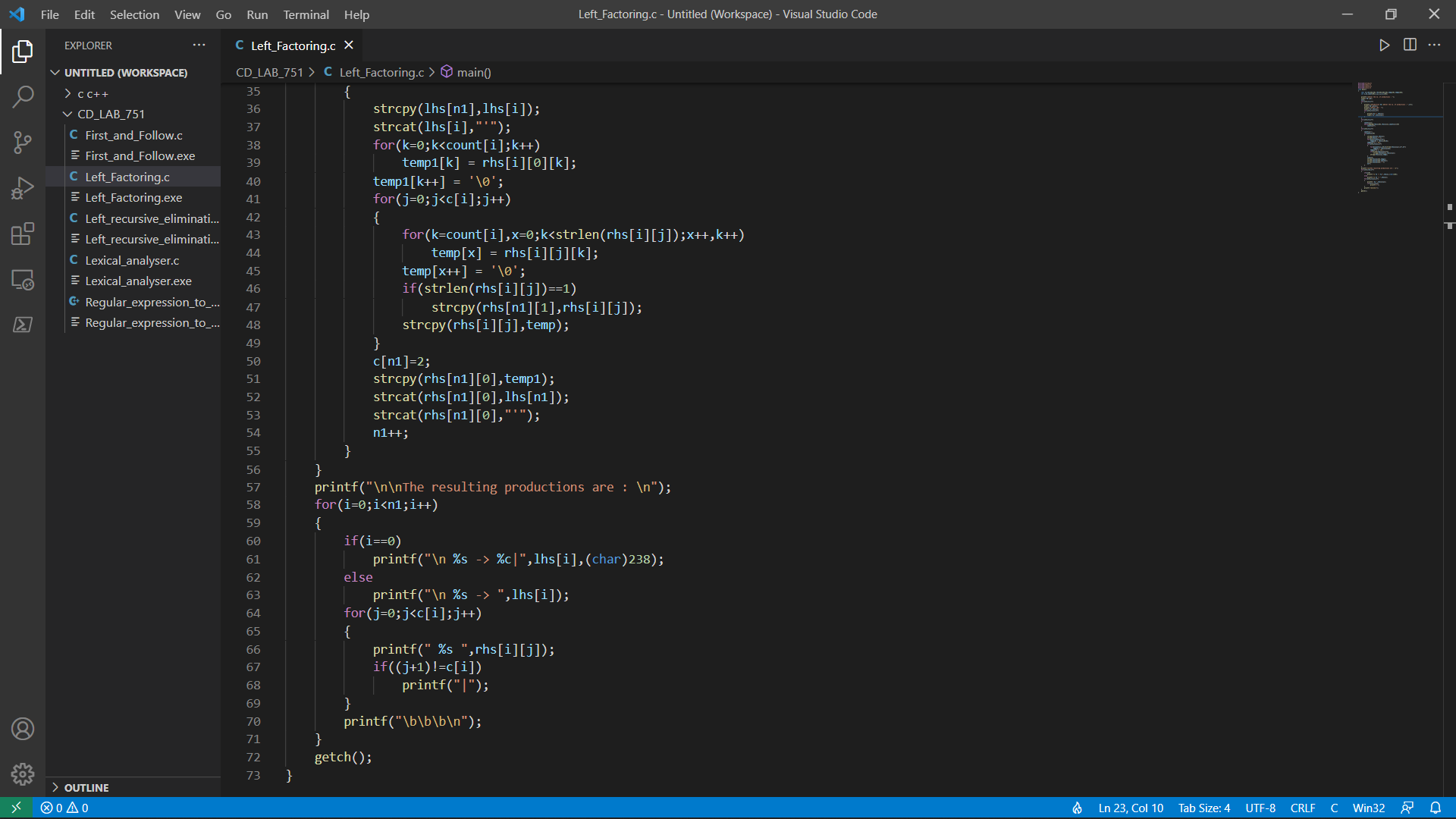
printf("\b\b\b\n");

}

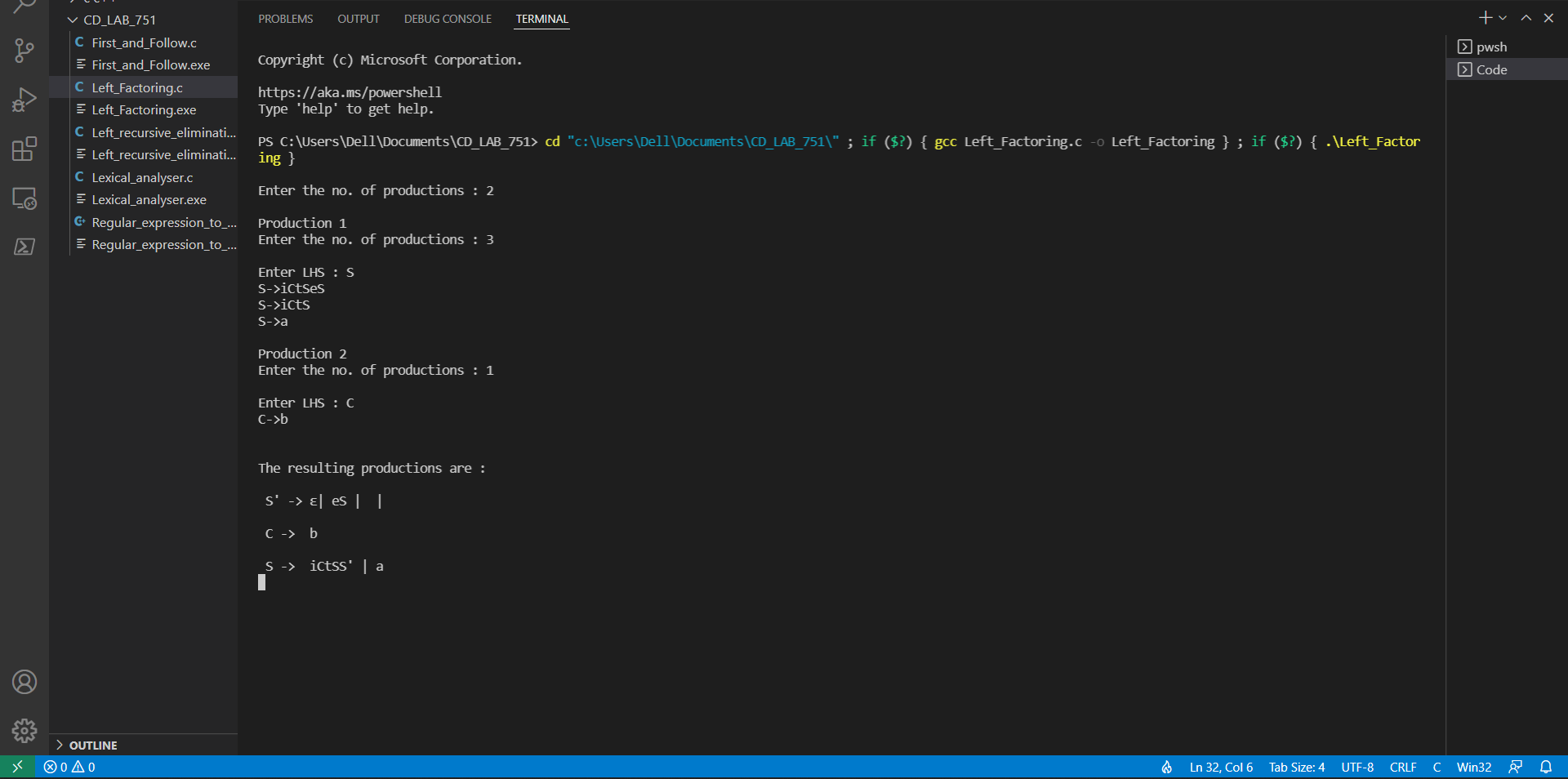
getch();

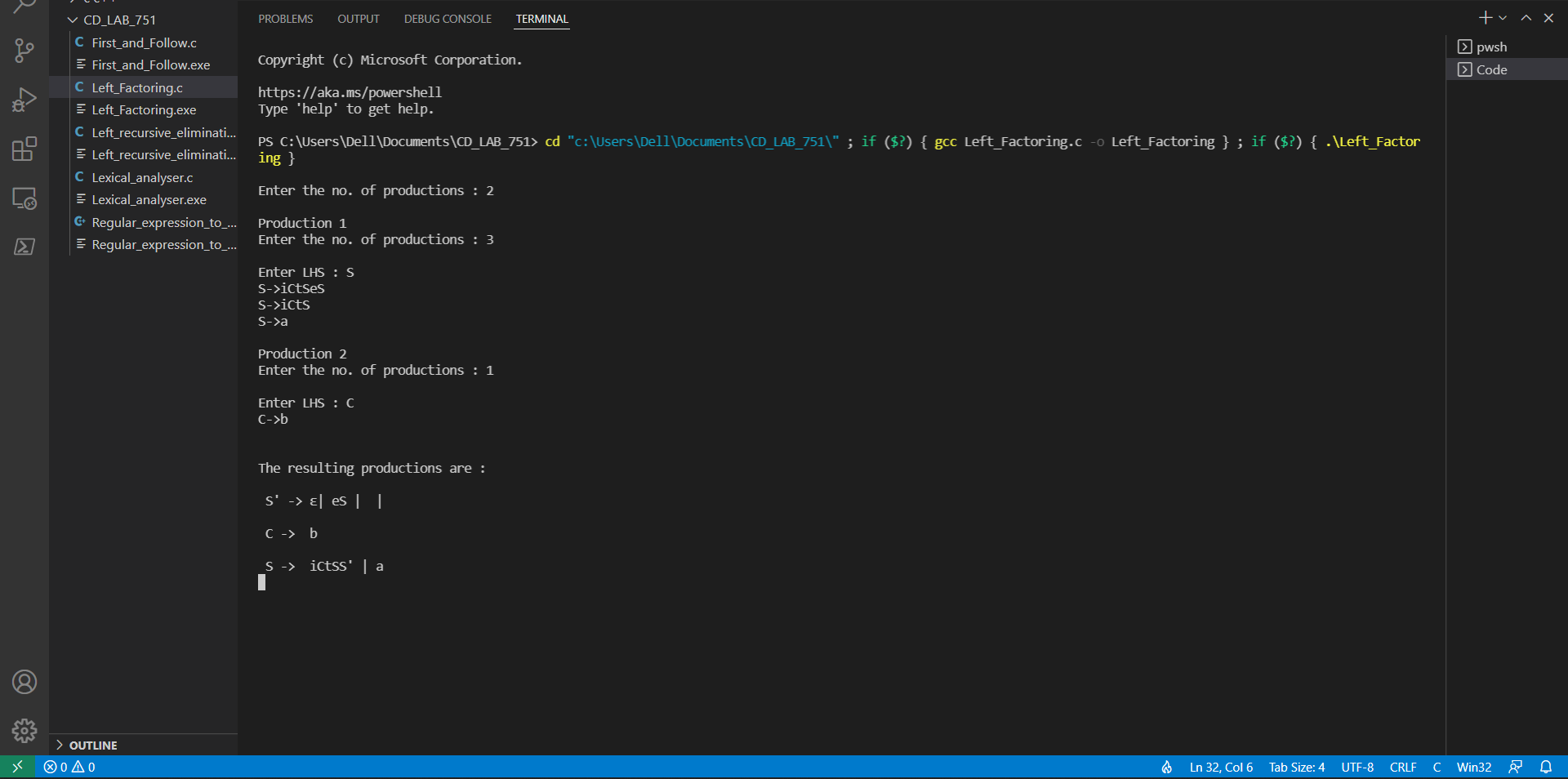
}





**Output:**

****

****

**Result:** Hence, a code is developed to remove left factoring.

Compiler Design Lab-18CSC304J

Exercise: Left Recursive Elimination

Sai Krishna Movva

RA1911003010751

**Aim:** To develop a program to eliminate left recursion.

**Code:**

#include<string.h>

#include<stdio.h>

#include<stdlib.h>

#include<conio.h>

void main()

{

char a[10],b[50][10]={""},d[50][10]={""},ch;

int i,n,c[10]={0},j,k,t,n1;

printf("\nEnter the left production(s) (NON TERMINALS) : ");

scanf("%s",a);

n=strlen(a);

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

{

printf("\nEnter the number of productions for %c : ",a[i]);

scanf("%d",&c[i]);

}

t=0;

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

{

printf("\nEnter the right productions for %c",a[i]);

k=t;

for(j=0;j<c[i];j++)

{

printf("\n%c->",a[i]);

do

{

scanf("%s",b[k]);

k++;

}while(k<j);

}

t=t+10;

}

t=0;

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

{

if(a[i]==b[t][0])

{

n1=strlen(b[t]);

for(k=1;k<n1;k++)

{

d[t][k-1]=b[t][k];

}

}

t=t+10;

}

t=0;

printf("\n\nThe resulting productions after eliminating Left Recursion are : \n");

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

{

if(a[i]==b[t][0])

{

for(j=1;j<c[i];j++)

{

printf("\n%c -> %s%c'",a[i],b[t+j],a[i]);

}

}

t=t+10;

}

t=0;

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

{

if(a[i]==b[t][0])

printf("\n%c' -> %s%c'|\u03B5",a[i],d[t],a[i]);

else

for(j=0;j<c[i];j++)

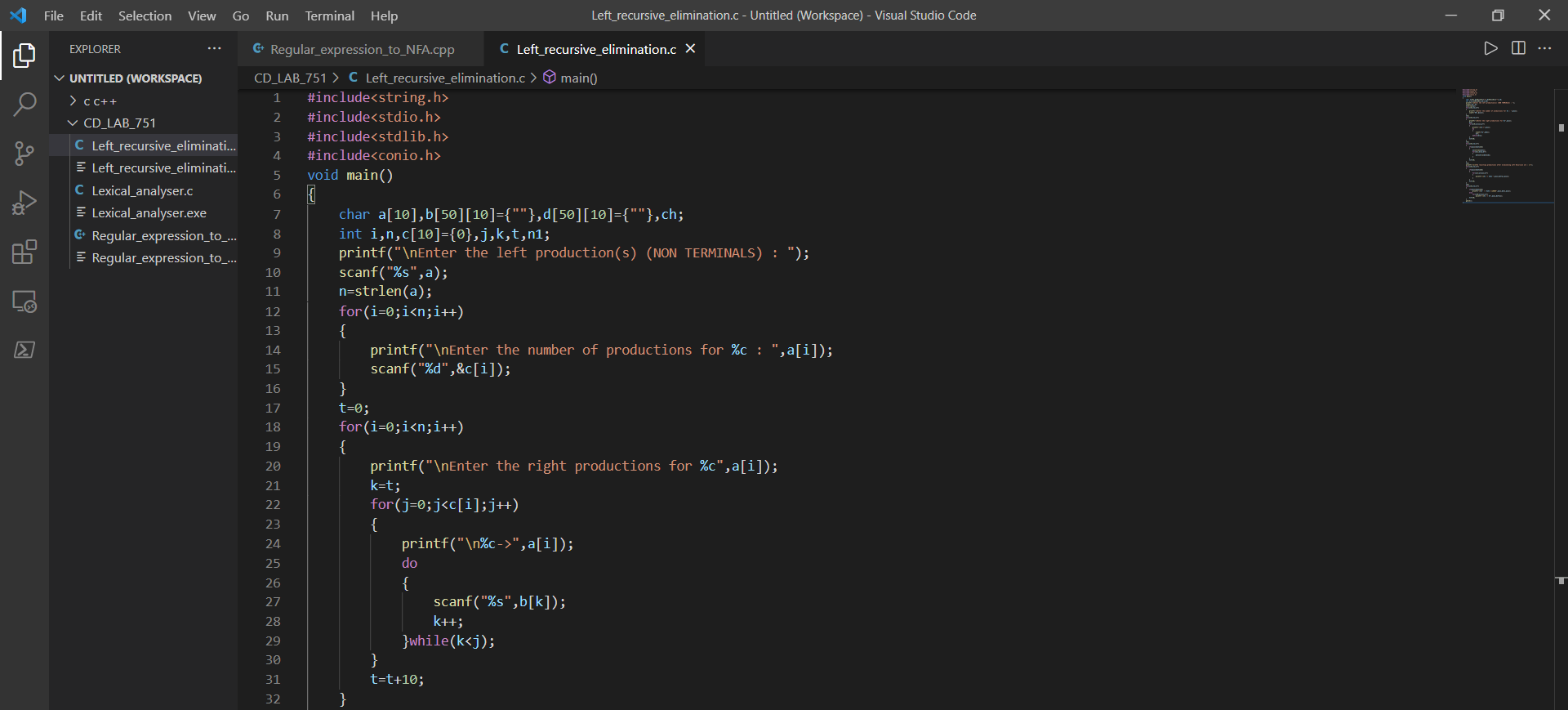
printf("\n%c -> %s",a[i],b[t+j]);

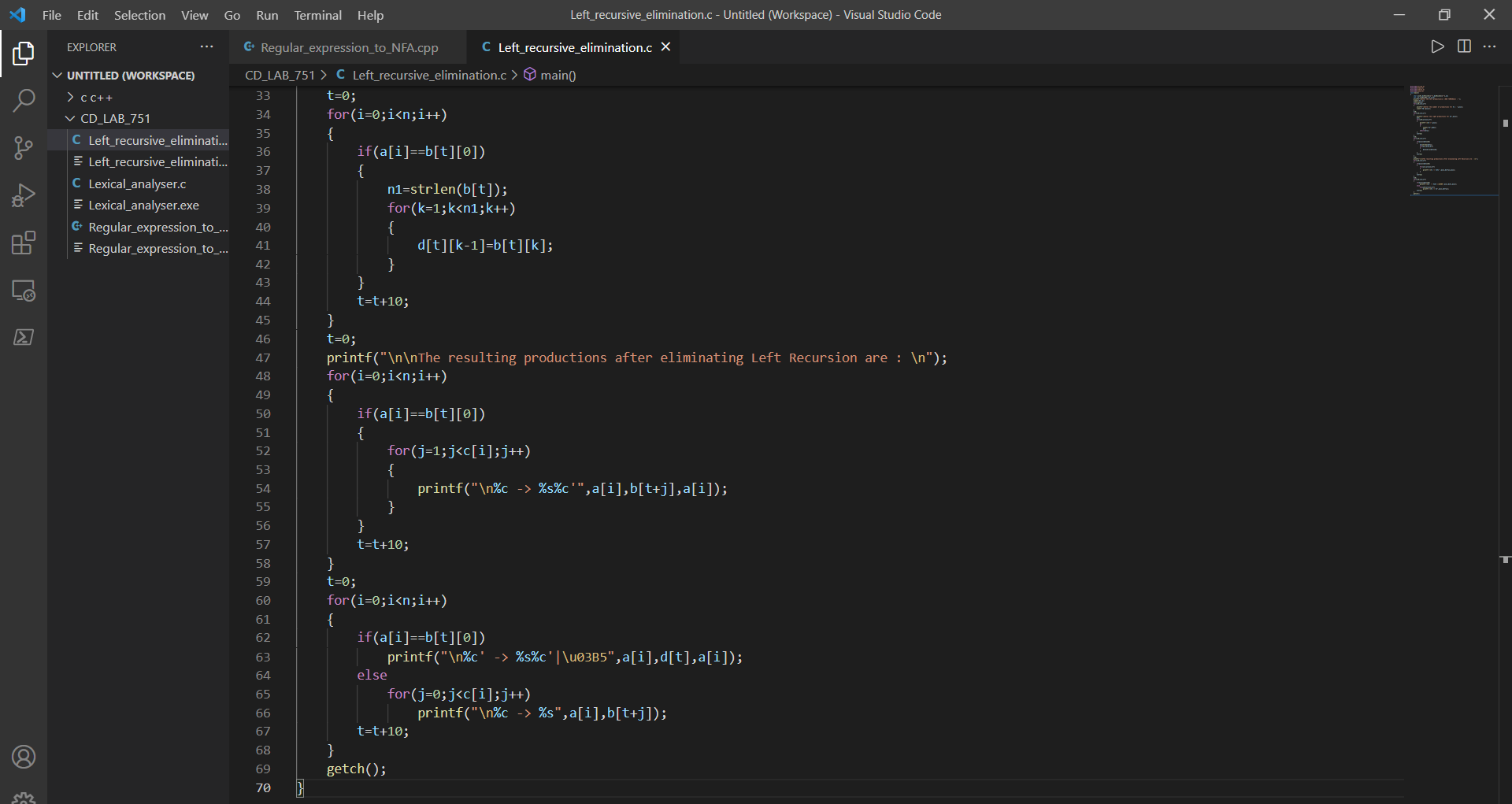
t=t+10;

}

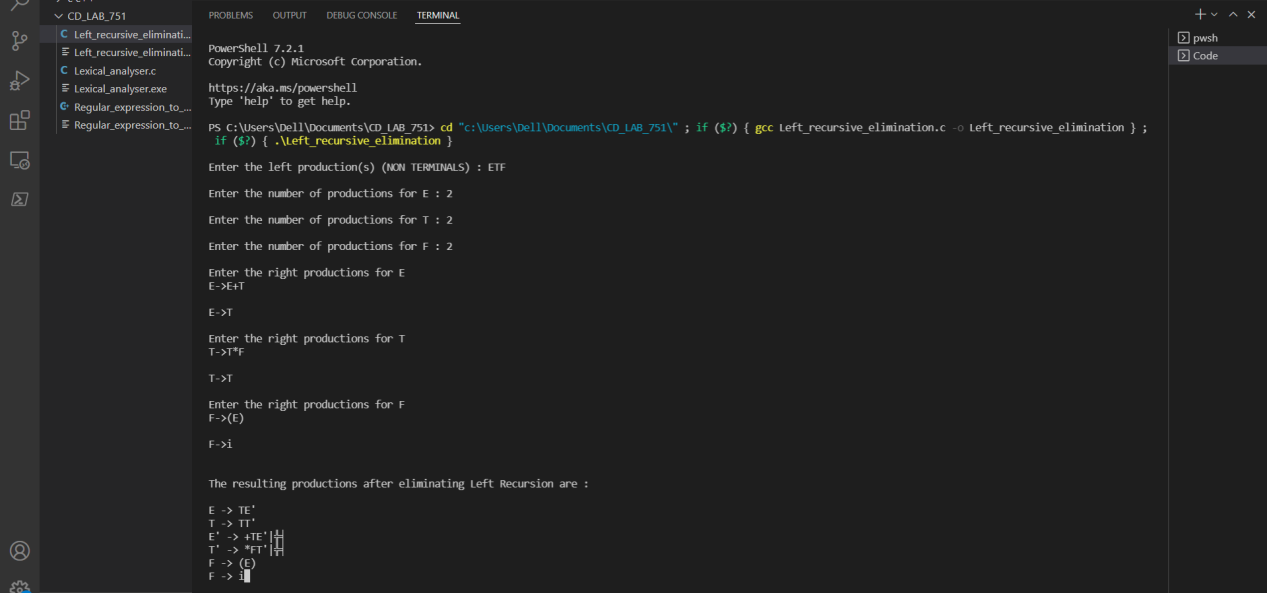
getch();

}





**Output:**

****

**Result:** Hence, a code is developed to remove left recursion.

Compiler Design Lab-18CSC304J

Exercise : First and follow computation

Sai Krishna Movva

RA1911003010751

**Aim:** To design and implementation of First and follow computation.

**Code:**

#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: ";

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();

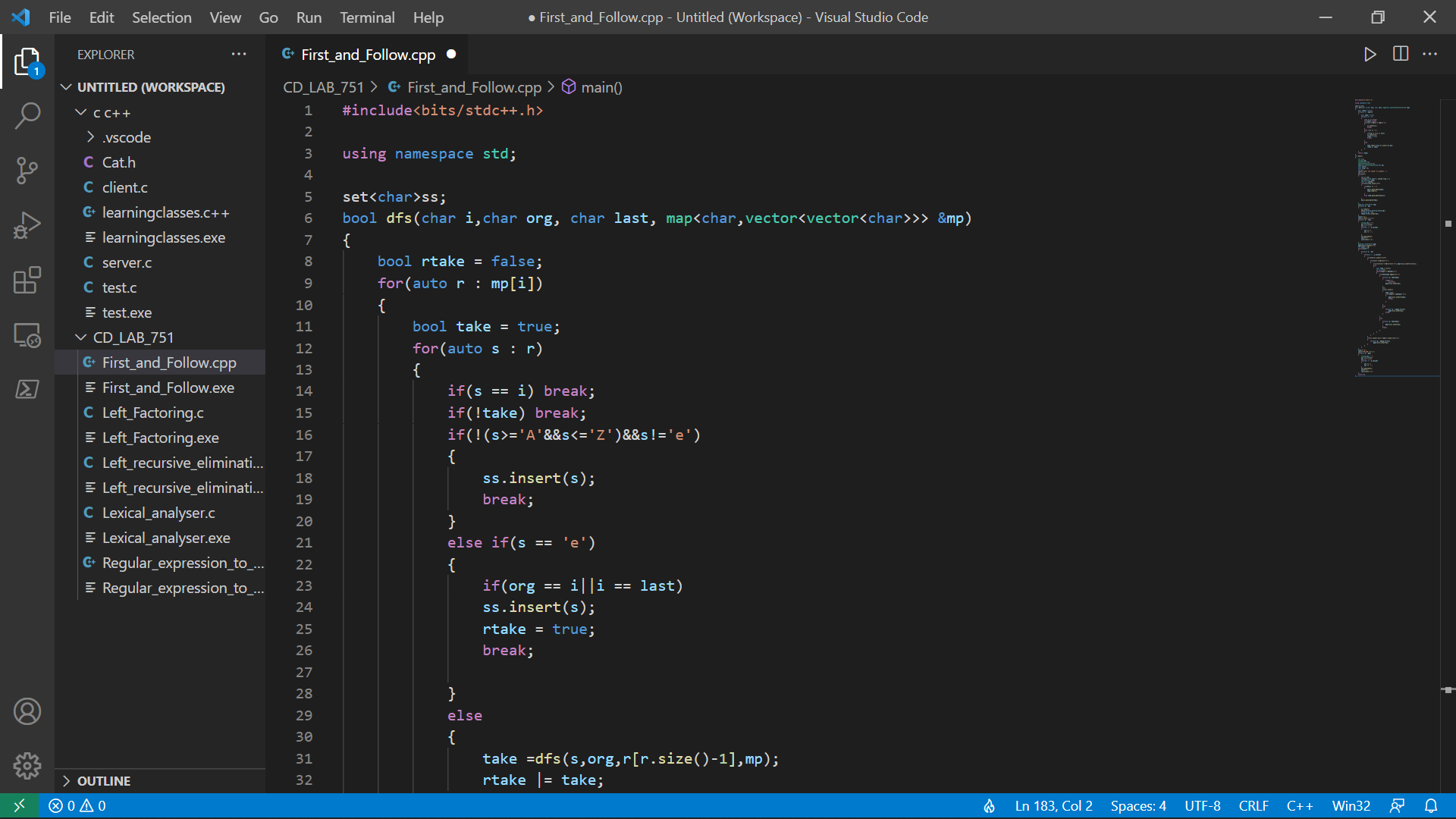
ans+="}";

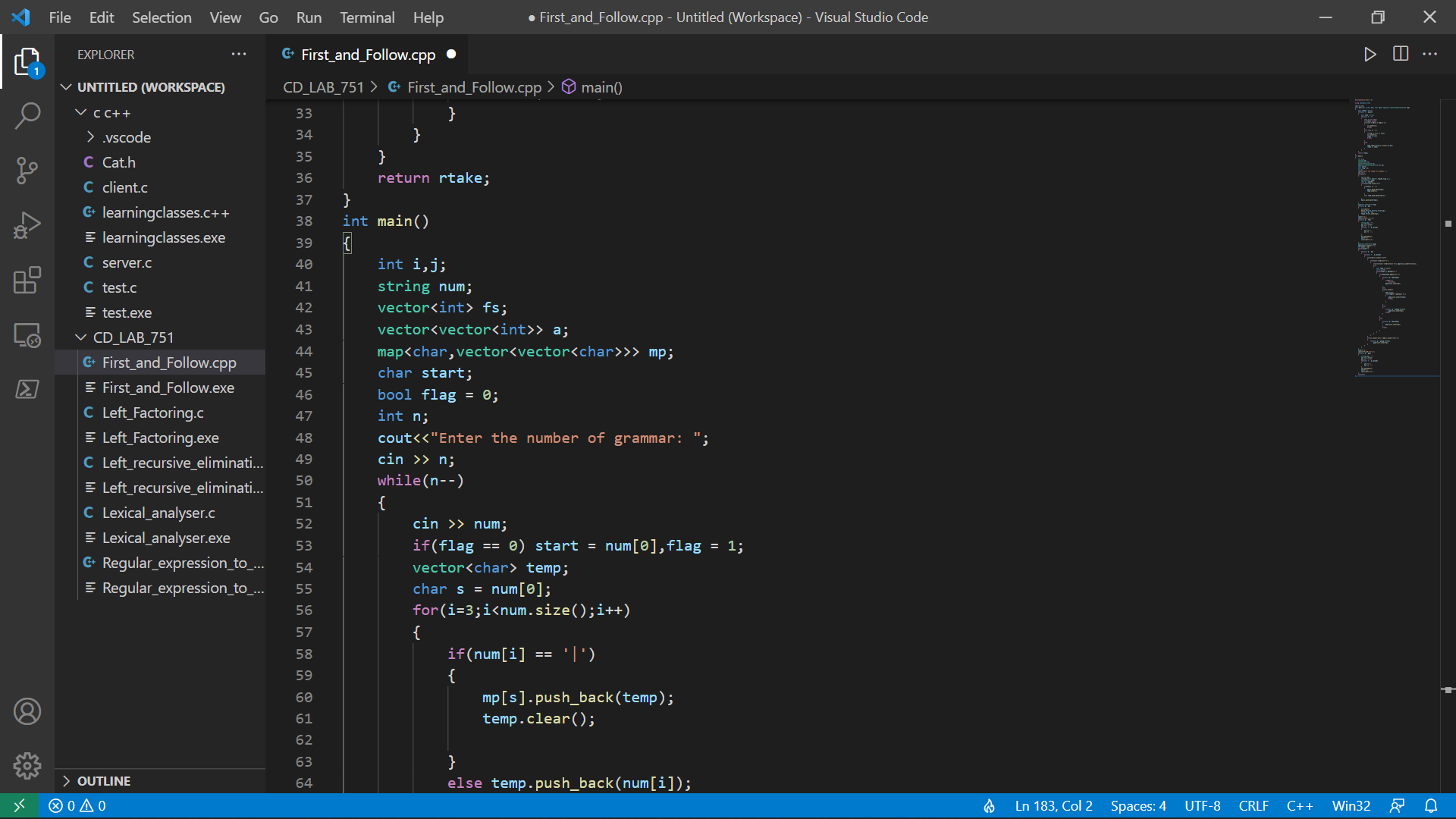
cout<<ans<<'\n';

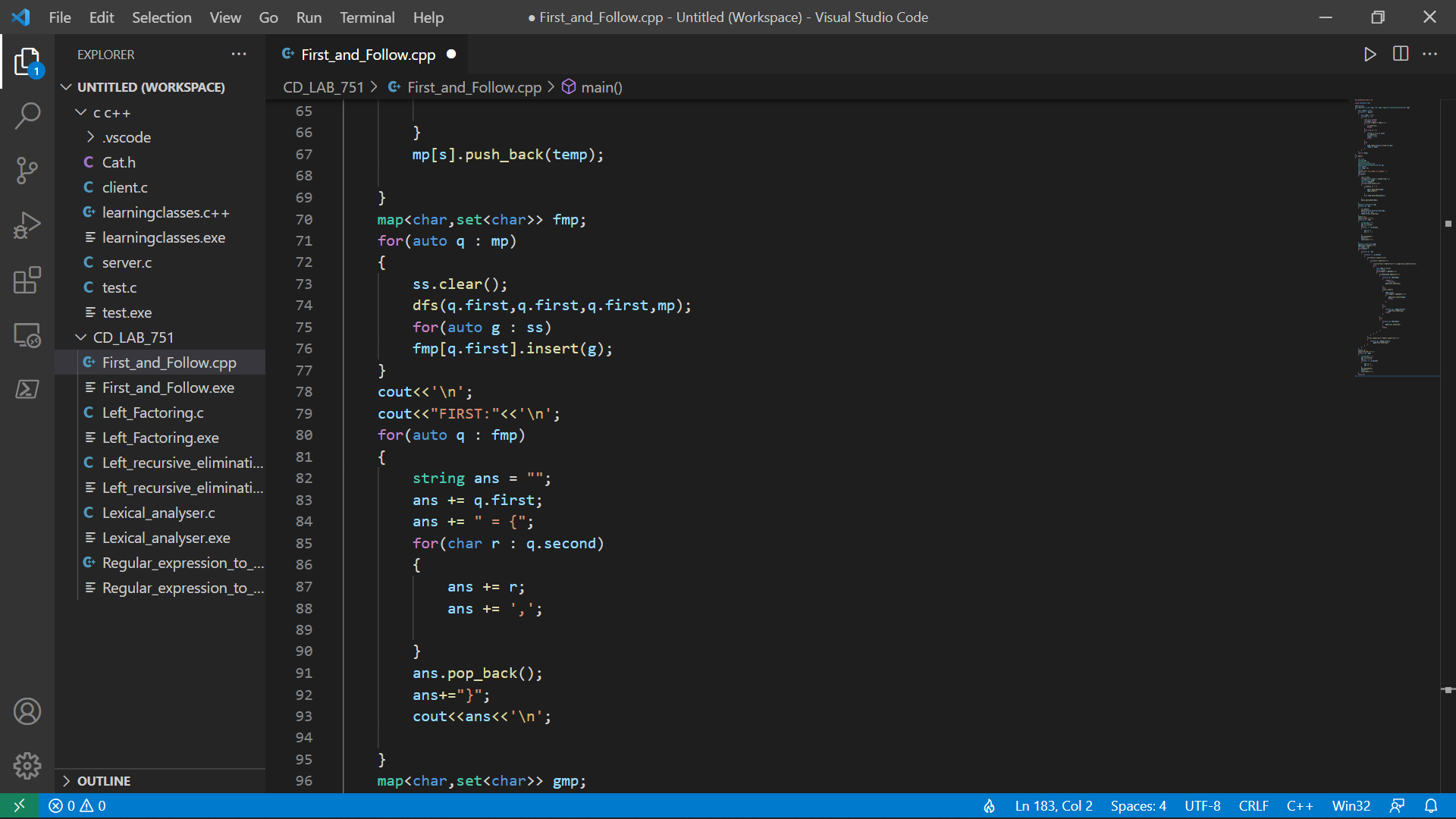
}

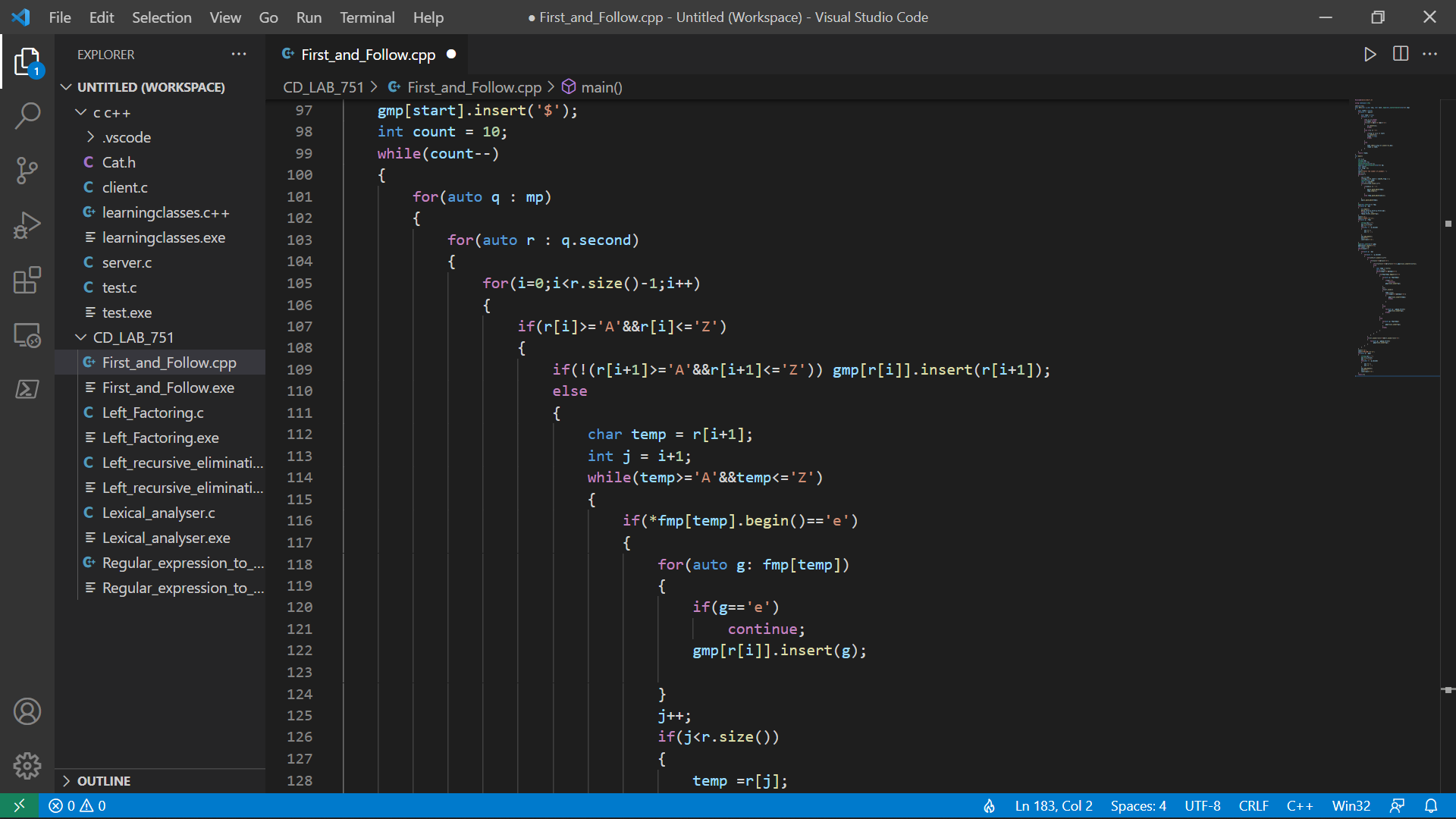
return 0;

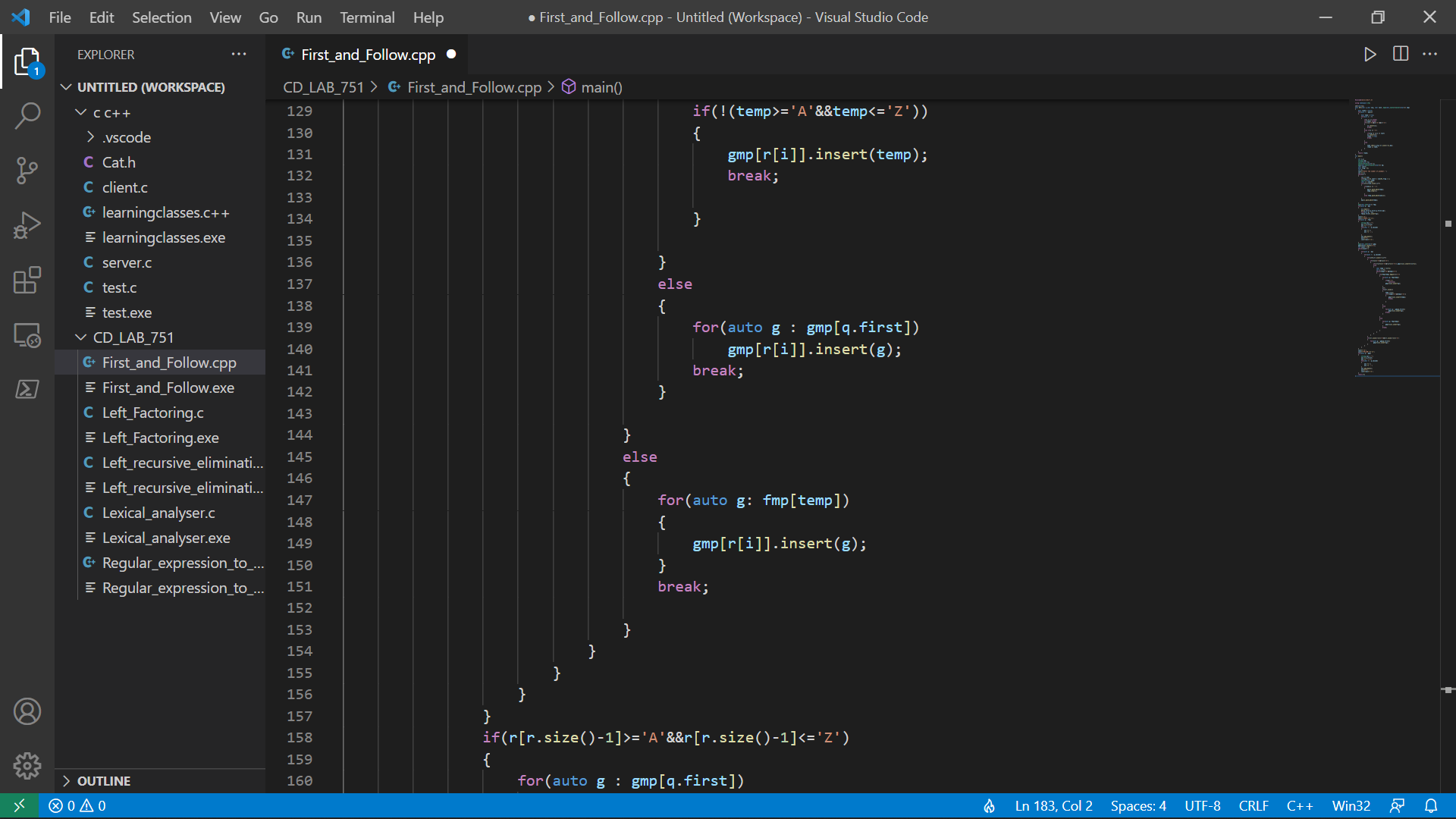
}

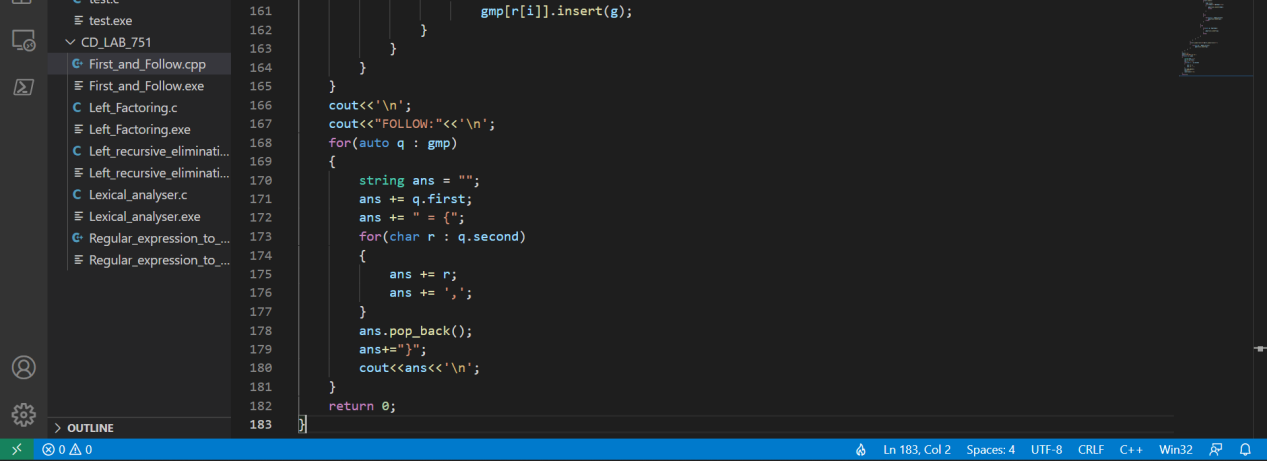




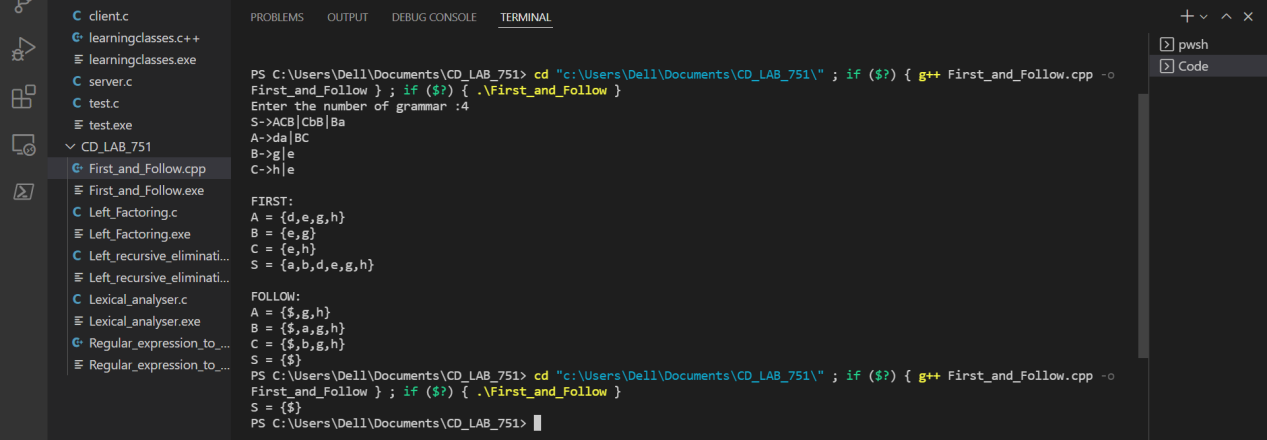








**Output:**

****

**Result:** Hence, the first and follow is computed in the given production rules.

Compiler Design Lab-18CSC304J

Exercise : Shift reduce Parsing.

Sai Krishna Movva

RA1911003010751

**Aim:** To design and implementation of Shift reduce Parsing.

**Code:**

#include<stdio.h>

#include<string.h>

#include<conio.h>

#include<dos.h>

int novar=0,sttop=1,intop=1,j=0,i=0,handlelength=0;

char ipstr1[100],ipstr[100],popped,var;

char prod[20][20],handle[100],stack[100]="#",input[100]="#";

struct grammar

{

char lhs,rhs[20][20];

int noprod;

}g[20];

int checkhandle()

{

int i,m,k;

char temp[2]={ ' ' , '\0'};

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

{

strcpy(handle,"");

for(m=i;m<=sttop-1;temp[0]=stack[m],strcat(handle,temp),m++);

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

{

for(k=0;k<g[m].noprod && strcmp(handle,g[m].rhs[k])!=0;k++);

if(k!=g[m].noprod)

{

var=g[m].lhs;

return strlen(handle);

}

}

}

return 0;

}

void print(char \*text,int textlen)

{

int i;

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

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

printf("\t\t\t");

}

void printi(char \*text,int textlen)

{

int i;

for(i=textlen-1;i>=0;i--)

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

printf("\t\t\t");

}

void main()

{

int n,m,k,len,j=0,v;

clrscr();

printf("\n Enter the productions of the grammar(END to end):\n");

do

{

scanf("%s",prod[i++]);

}while(strcmp(prod[i-1],"END")!=0);

for(n=0;n<i-1;n++)

{

m=0,k=0;

for(j=0;j<novar;j++)

if(g[j].lhs==prod[n][0])

break;

if(j==novar)

g[novar++].lhs=prod[n][0];

for(k=3;k<strlen(prod[n])+1;k++)

{

if(prod[n][k]!='|' && prod[n][k]!='\0')

g[j].rhs[g[j].noprod][m++]=prod[n][k];

if(prod[n][k]=='|' || prod[n][k]=='\0')

{

g[j].rhs[g[j].noprod++][m]='\0';

m=0;

}

}

}

printf("\nENTER THE INPUT STRING:");

scanf("%s",ipstr);

printf("\n\n\n\n");

for(i=strlen(ipstr)-1;i>=0;i--)

input[intop++]=ipstr[i];

printf("-------------------------------------------------------------\n");

printf(" STACK\t\t\tINPUT\t\t\tACTION\n");

printf("-------------------------------------------------------------\n");

print(stack,sttop);

printi(input,intop);

while(1)

{

int count=0;

while((handlelength=checkhandle())>0)

{

if(input[intop-1]=='\*' && count++==2)

break;

else if(input[intop-1]=='=' && count++==3)

break;

else if(input[intop-1]=='e' && count++==1)

break;

for(i=0;i<handlelength;i++,--sttop);

stack[sttop++]=var;

printf("REDUCE BY %c -> %s\n",var,handle);

print(stack,sttop);

printi(input,intop);

}

popped=input[--intop];

if(popped!='#')

stack[sttop++]=popped;

handlelength=checkhandle();

if(popped=='#' && (handlelength=checkhandle())==0)

break;

printf("SHIFT ' %c '\n",popped);

print(stack,sttop);

printi(input,intop);

}

if(sttop==2 && stack[1]==g[0].lhs)

printf("ACCEPT\n");

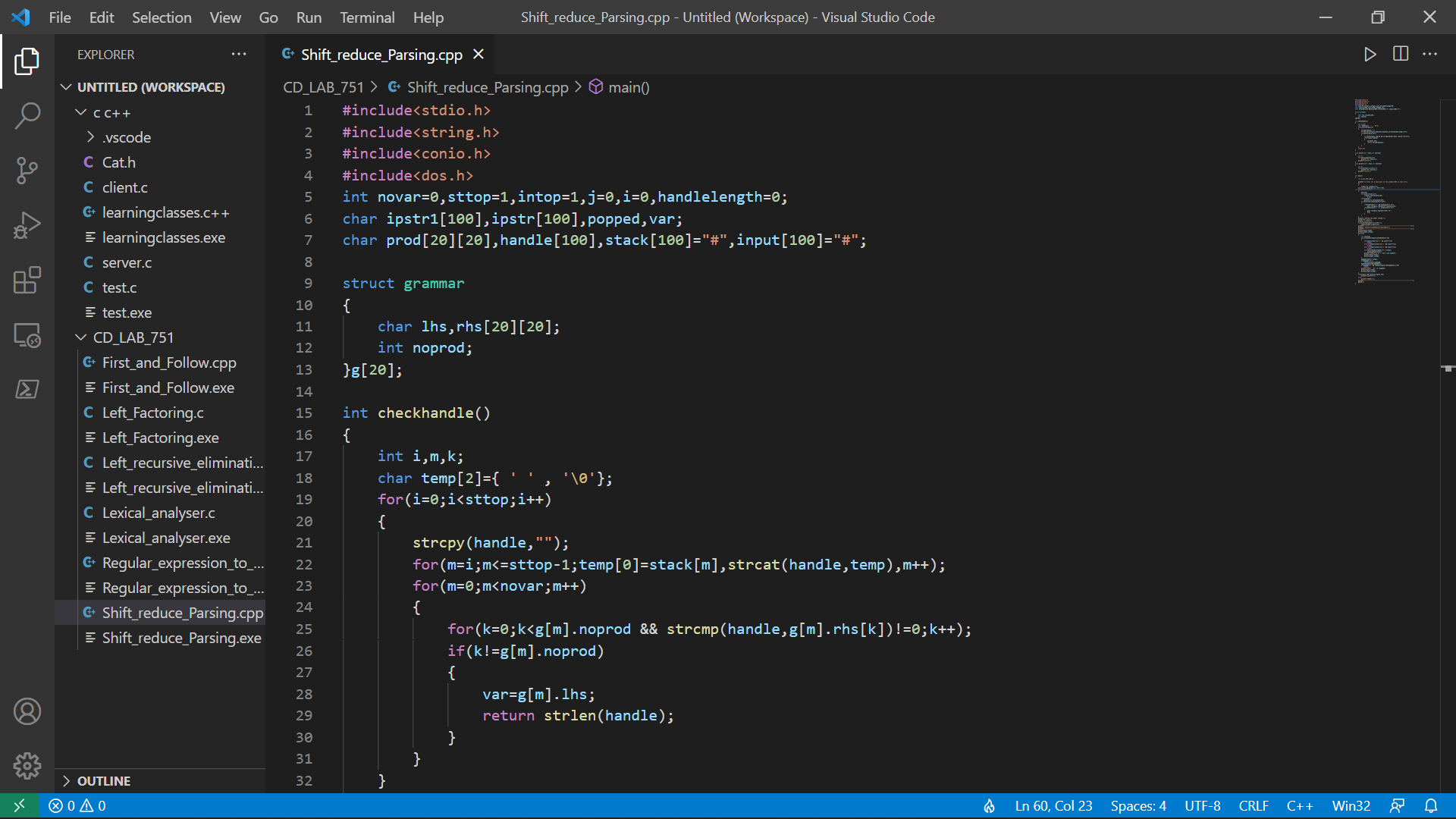
else

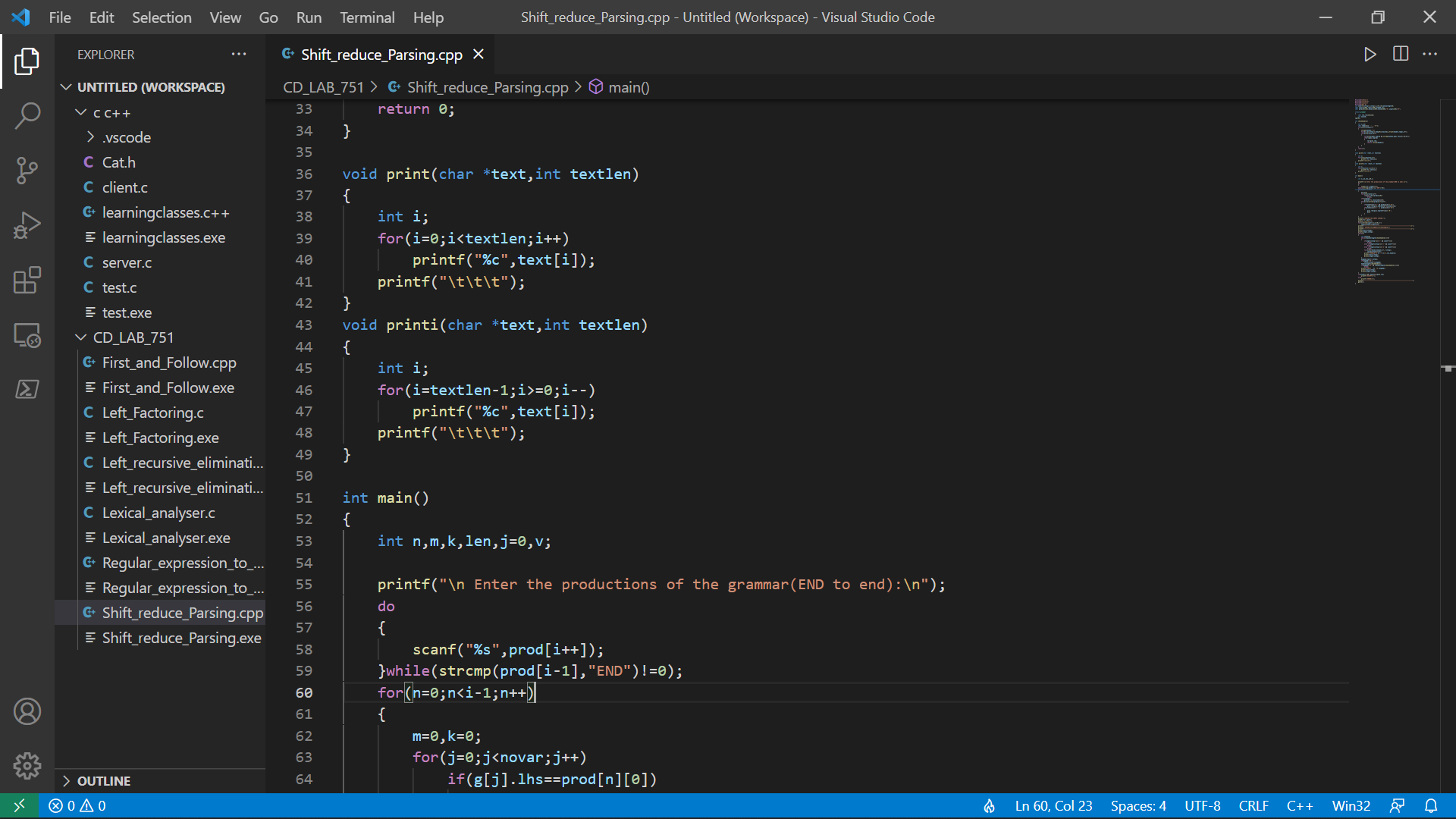
printf("ERROR\n");

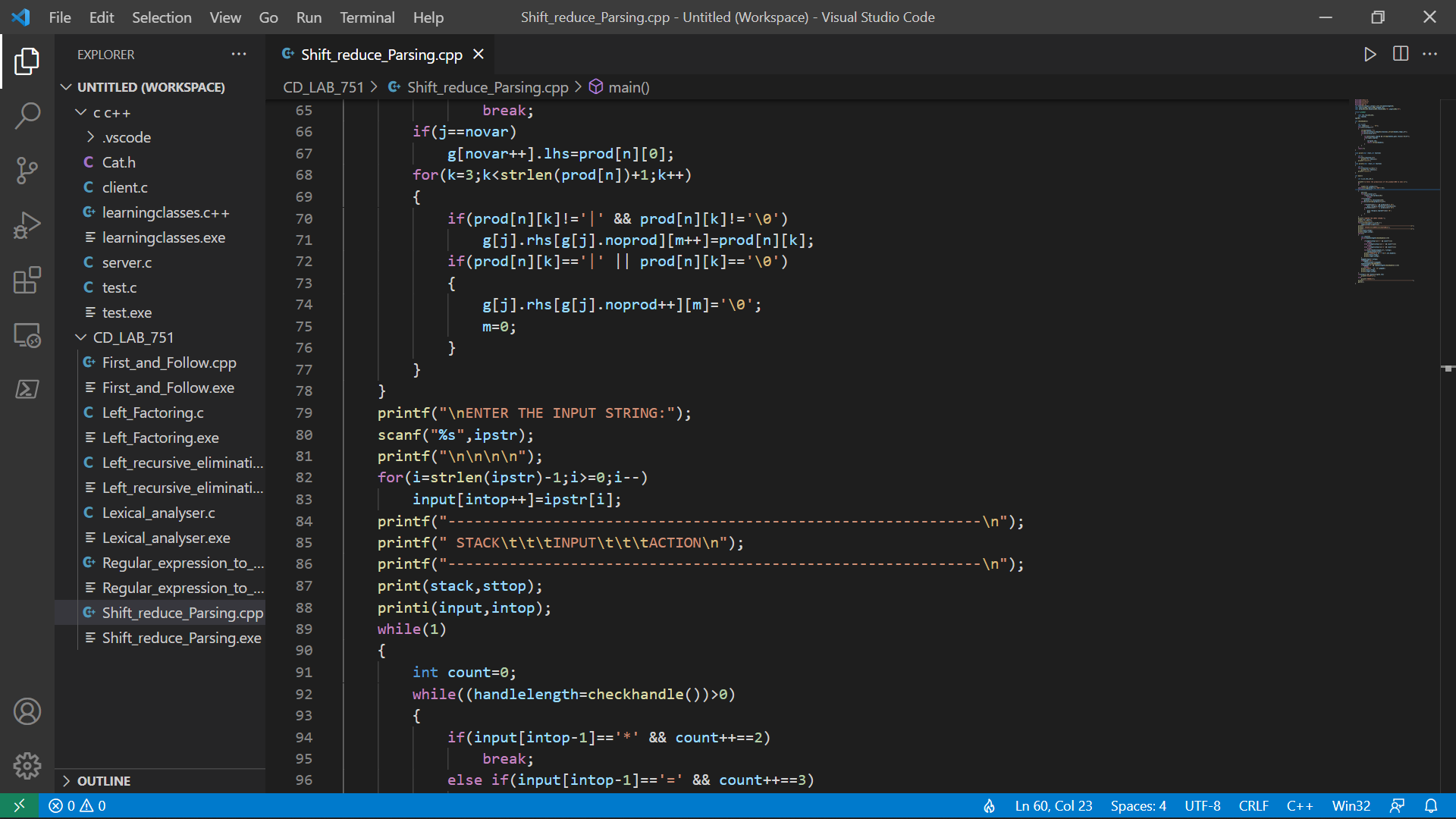
printf("---------------------------------------------------------------");

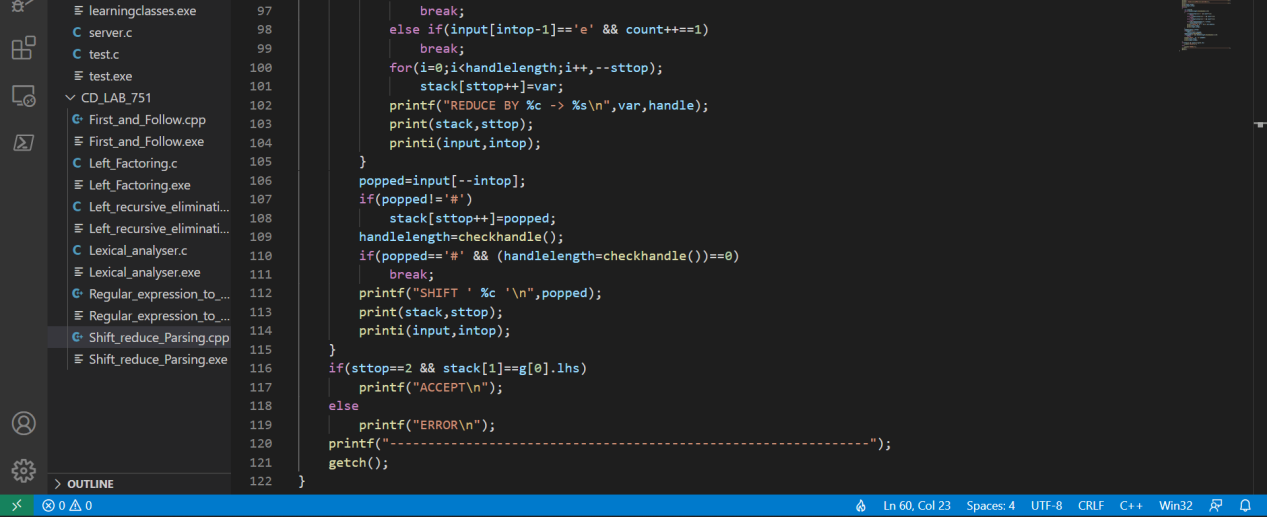
getch();

}

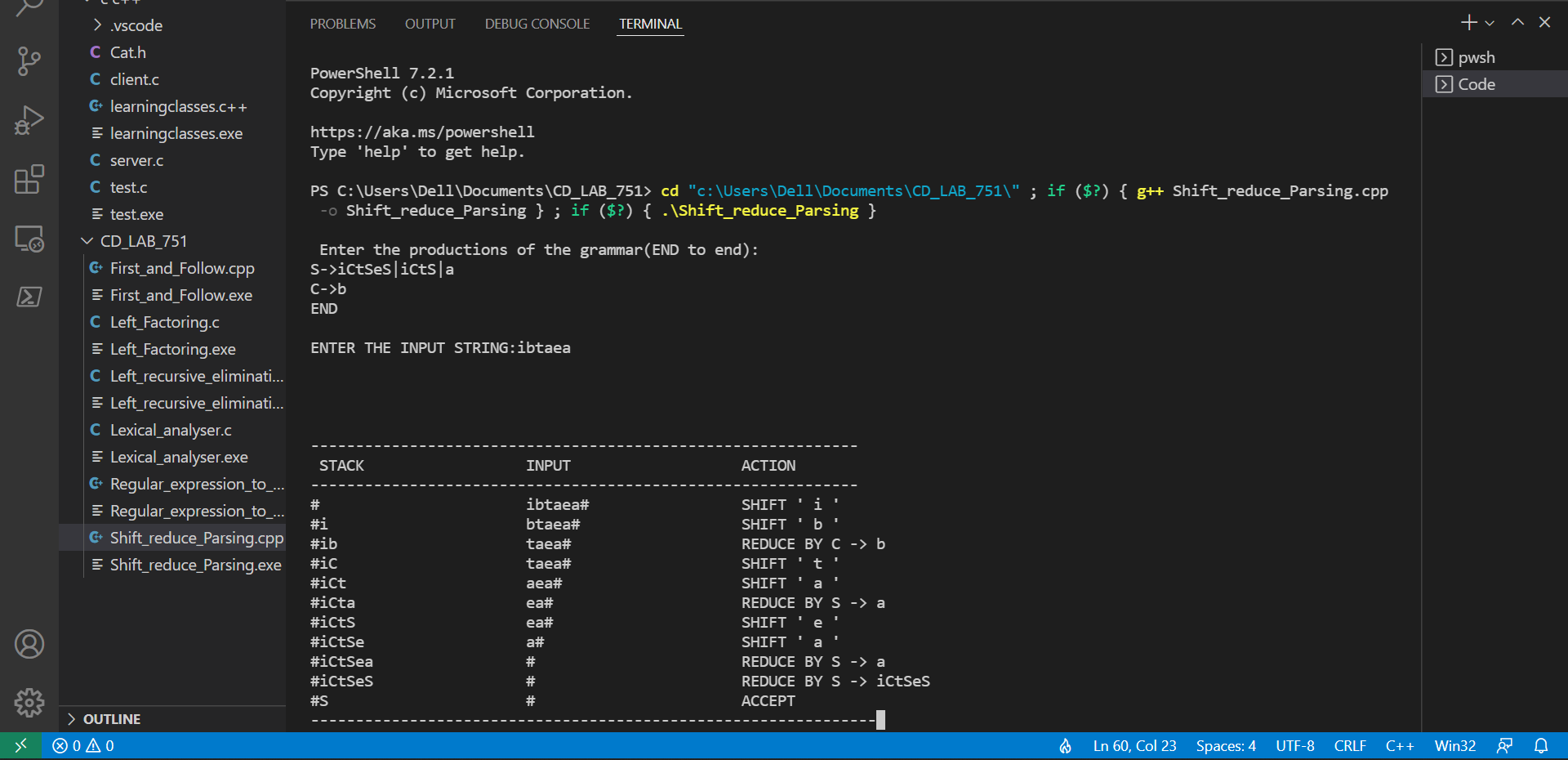








**Output:**

****

**Result:** Hence, the Shift reduce Parsing is computed in C++.

EXPERIMENT 7

LEADING AND TRAILING

**M. Sai Krishna**

**RA1911003010751**

**Aim:** Study and implement Leading and Trailing in CPP

# Code:

#include<iostream>

#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')

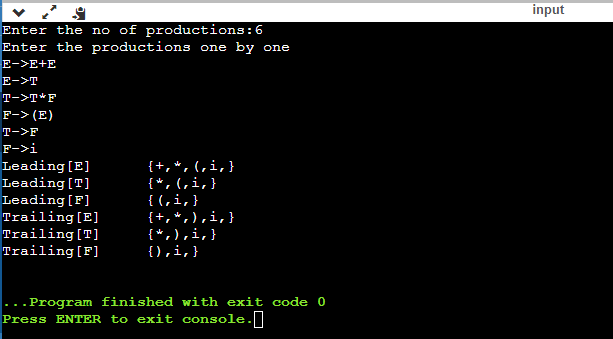
                cout<<T[j]<<","; }

        cout<<"}\n";

    }

    return 0;

**Output:**



**Result:**

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

SLR parsing

M.Sai krishna

RA1911003010751

Code-

#include<iostream>

#include<string.h>

#include<stdlib.h>

#include<stdio.h>

using namespace std;

char terminals[100]={};int no\_t;

char non\_terminals[100]={};int no\_nt;

char goto\_table[100][100];

char reduce[20][20];

char follow[20][20];char fo\_co[20][20];

char first[20][20];

struct state{

int prod\_count;

char prod[100][100]={{}};

};

void add\_dots(struct state \*I){

for(int i=0;i<I->prod\_count;i++){

for (int j=99;j>3;j--)

I->prod[i][j] = I->prod[i][j-1];

I->prod[i][3]='.';

}

}

void augument(struct state \*S,struct state \*I){

if(I->prod[0][0]=='S')

strcpy(S->prod[0],"Z->.S");

else{

strcpy(S->prod[0],"S->.");

S->prod[0][4]=I->prod[0][0];}

S->prod\_count++;

}

void get\_prods(struct state \*I){

cout<<"Enter the number of productions:\n";

cin>>I->prod\_count;

cout<<"Enter the number of non terminals:"<<endl;

cin>>no\_nt;

cout<<"Enter the non terminals one by one:"<<endl;

for(int i=0;i<no\_nt;i++)

cin>>non\_terminals[i];

cout<<"Enter the number of terminals:"<<endl;

cin>>no\_t;

cout<<"Enter the terminals (single lettered) one by one:"<<endl;

for(int i=0;i<no\_t;i++)

cin>>terminals[i];

cout<<"Enter the productions one by one in form (S->ABc):\n";

for(int i=0;i<I->prod\_count;i++){

cin>>I->prod[i];

}

}

bool is\_non\_terminal(char a){

if (a >= 'A' && a <= 'Z')

return true;

else

return false;

}

bool in\_state(struct state \*I,char \*a){

for(int i=0;i<I->prod\_count;i++){

if(!strcmp(I->prod[i],a))

return true;

}

return false;

}

char char\_after\_dot(char a[100]){

char b;

for(int i=0;i<strlen(a);i++)

if(a[i]=='.'){

b=a[i+1];

return b;}

}

char\* move\_dot(char b[100],int len){

char a[100]={};

strcpy(a,b);

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

if(a[i]=='.'){

swap(a[i],a[i+1]);

break;

}

}

return &a[0];

}

bool same\_state(struct state \*I0,struct state \*I){

if (I0->prod\_count != I->prod\_count)

return false;

for (int i=0; i<I0->prod\_count; i++)

{

int flag = 0;

for (int j=0; j<I->prod\_count; j++)

if (strcmp(I0->prod[i], I->prod[j]) == 0)

flag = 1;

if (flag == 0)

return false;

}

return true;

}

void closure(struct state \*I,struct state \*I0){

char a={};

for(int i=0;i<I0->prod\_count;i++){

a=char\_after\_dot(I0->prod[i]);

if(is\_non\_terminal(a)){

for(int j=0;j<I->prod\_count;j++){

if(I->prod[j][0]==a){

if(!in\_state(I0,I->prod[j])){

strcpy(I0->prod[I0->prod\_count],I->prod[j]);

I0->prod\_count++;

}

}

}

}

}

}

void goto\_state(struct state \*I,struct state \*S,char a){

int time=1;

for(int i=0;i<I->prod\_count;i++){

if(char\_after\_dot(I->prod[i])==a){

if(time==1){

time++;

}

strcpy(S->prod[S->prod\_count],move\_dot(I->prod[i],strlen(I->prod[i])));

S->prod\_count++;

}

}

}

void print\_prods(struct state \*I){

for(int i=0;i<I->prod\_count;i++)

printf("%s\n",I->prod[i]);

cout<<endl;

}

bool in\_array(char a[20],char b){

for(int i=0;i<strlen(a);i++)

if(a[i]==b)

return true;

return false;

}

char\* chars\_after\_dots(struct state \*I){

char a[20]={};

for(int i=0;i<I->prod\_count;i++){

if(!in\_array(a,char\_after\_dot(I->prod[i]))){

a[strlen(a)]=char\_after\_dot(I->prod[i]);

}

}

return &a[0];

}

void cleanup\_prods(struct state \* I){

char a[100]={};

for(int i=0;i<I->prod\_count;i++)

strcpy(I->prod[i],a);

I->prod\_count=0;

}

int return\_index(char a){

for(int i=0;i<no\_t;i++)

if(terminals[i]==a)

return i;

for(int i=0;i<no\_nt;i++)

if(non\_terminals[i]==a)

return no\_t+i;

}

void print\_shift\_table(int state\_count){

cout<<endl<<"\*\*\*\*\*\*\*\*Shift Actions\*\*\*\*\*\*\*\*\*"<<endl<<endl;

cout<<"\t";

for(int i=0;i<no\_t;i++)

cout<<terminals[i]<<"\t";

for(int i=0;i<no\_nt;i++)

cout<<non\_terminals[i]<<"\t";

cout<<endl;

for(int i=0;i<state\_count;i++){

int arr[no\_nt+no\_t]={-1};

for(int j=0;j<state\_count;j++){

if(goto\_table[i][j]!='~'){

arr[return\_index(goto\_table[i][j])]= j;

}

}

cout<<"I"<<i<<"\t";

for(int j=0;j<no\_nt+no\_t;j++){

if(i==1&&j==no\_t-1)

cout<<"ACC"<<"\t";

if(arr[j]==-1||arr[j]==0)

cout<<"\t";

else{

if(j<no\_t)

cout<<"S"<<arr[j]<<"\t";

else

cout<<arr[j]<<"\t";

}

}

cout<<"\n";

}

}

int get\_index(char c,char \*a){

for(int i=0;i<strlen(a);i++)

if(a[i]==c)

return i;

}

void add\_dot\_at\_end(struct state\* I){

for(int i=0;i<I->prod\_count;i++){

strcat(I->prod[i],".");

}

}

void add\_to\_first(int n,char b){

for(int i=0;i<strlen(first[n]);i++)

if(first[n][i]==b)

return;

first[n][strlen(first[n])]=b;

}

void add\_to\_first(int m,int n){

for(int i=0;i<strlen(first[n]);i++){

int flag=0;

for(int j=0;j<strlen(first[m]);j++){

if(first[n][i]==first[m][j])

flag=1;

}

if(flag==0)

add\_to\_first(m,first[n][i]);

}

}

void add\_to\_follow(int n,char b){

for(int i=0;i<strlen(follow[n]);i++)

if(follow[n][i]==b)

return;

follow[n][strlen(follow[n])]=b;

}

void add\_to\_follow(int m,int n){

for(int i=0;i<strlen(follow[n]);i++){

int flag=0;

for(int j=0;j<strlen(follow[m]);j++){

if(follow[n][i]==follow[m][j])

flag=1;

}

if(flag==0)

add\_to\_follow(m,follow[n][i]);

}

}

void add\_to\_follow\_first(int m,int n){

for(int i=0;i<strlen(first[n]);i++){

int flag=0;

for(int j=0;j<strlen(follow[m]);j++){

if(first[n][i]==follow[m][j])

flag=1;

}

if(flag==0)

add\_to\_follow(m,first[n][i]);

}

}

void find\_first(struct state \*I){

for(int i=0;i<no\_nt;i++){

for(int j=0;j<I->prod\_count;j++){

if(I->prod[j][0]==non\_terminals[i]){

if(!is\_non\_terminal(I->prod[j][3])){

add\_to\_first(i,I->prod[j][3]);

}

}

}

}

}

void find\_follow(struct state \*I){

for(int i=0;i<no\_nt;i++){

for(int j=0;j<I->prod\_count;j++){

for(int k=3;k<strlen(I->prod[j]);k++){

if(I->prod[j][k]==non\_terminals[i]){

if(I->prod[j][k+1]!='\0'){

if(!is\_non\_terminal(I->prod[j][k+1])){

add\_to\_follow(i,I->prod[j][k+1]);

}

}

}

}

}

}

}

int get\_index(int \*arr,int n){

for(int i=0;i<no\_t;i++){

if(arr[i]==n)

return i;

}

return -1;

}

void print\_reduce\_table(int state\_count,int \*no\_re,struct state \*temp1){

cout<<"\*\*\*\*\*\*\*\*\*\*Reduce actions\*\*\*\*\*\*\*\*\*\*"<<endl<<endl;

cout<<"\t";

int arr[temp1->prod\_count][no\_t]={-1};

for(int i=0;i<no\_t;i++){

cout<<terminals[i]<<"\t";

}

cout<<endl;

for(int i=0;i<temp1->prod\_count;i++){

int n=no\_re[i];

for(int j=0;j<strlen(follow[return\_index(temp1->prod[i][0])-no\_t]);j++){

for(int k=0;k<no\_t;k++){

if(follow[return\_index(temp1->prod[i][0])-no\_t][j]==terminals[k])

arr[i][k]=i+1;

}

}

cout<<"I"<<n<<"\t";

for(int j=0;j<no\_t;j++){

if(arr[i][j]!=-1&&arr[i][j]!=0&&arr[i][j]<state\_count)

cout<<"R"<<arr[i][j]<<"\t";

else

cout<<"\t";

}

cout<<endl;

}

}

int main(){

struct state init;

struct state temp;struct state temp1;

int state\_count=1;

get\_prods(&init);

temp=init;

temp1=temp;

add\_dots(&init);

for(int i=0;i<100;i++)

for(int j=0;j<100;j++)

goto\_table[i][j]='~';

struct state I[50];

augument(&I[0],&init);

closure(&init,&I[0]);

cout<<"\nI0:\n";

print\_prods(&I[0]);

char characters[20]={};

for(int i=0;i<state\_count;i++){

char characters[20]={};

for(int z=0;z<I[i].prod\_count;z++)

if(!in\_array(characters,char\_after\_dot(I[i].prod[z])))

characters[strlen(characters)]=char\_after\_dot(I[i].prod[z]);

for(int j=0;j<strlen(characters);j++){

goto\_state(&I[i],&I[state\_count],characters[j]);

closure(&init,&I[state\_count]);

int flag=0;

for(int k=0;k<state\_count-1;k++){

if(same\_state(&I[k],&I[state\_count])){

cleanup\_prods(&I[state\_count]);flag=1;

cout<<"I"<<i<<" on reading the symbol "<<characters[j]<<" goes to I"<<k<<".\n";

goto\_table[i][k]=characters[j];;

break;

}

}

if(flag==0){

state\_count++;

cout<<"I"<<i<<" on reading the symbol "<<characters[j]<<" goes to I"<<state\_count-1<<":\n";

goto\_table[i][state\_count-1]=characters[j];

print\_prods(&I[state\_count-1]);

}

}

}

int no\_re[temp.prod\_count]={-1};

terminals[no\_t]='$';no\_t++;

add\_dot\_at\_end(&temp1);

for(int i=0;i<state\_count;i++){

for(int j=0;j<I[i].prod\_count;j++)

for(int k=0;k<temp1.prod\_count;k++)

if(in\_state(&I[i],temp1.prod[k]))

no\_re[k]=i;

}

find\_first(&temp);

for(int l=0;l<no\_nt;l++){

for(int i=0;i<temp.prod\_count;i++){

if(is\_non\_terminal(temp.prod[i][3])){

add\_to\_first(return\_index(temp.prod[i][0])-no\_t,return\_index(temp.prod[i][3])-no\_t);

}

}}

find\_follow(&temp);

add\_to\_follow(0,'$');

for(int l=0;l<no\_nt;l++){

for(int i=0;i<temp.prod\_count;i++){

for(int k=3;k<strlen(temp.prod[i]);k++){

if(temp.prod[i][k]==non\_terminals[l]){

if(is\_non\_terminal(temp.prod[i][k+1])){

add\_to\_follow\_first(l,return\_index(temp.prod[i][k+1])-no\_t);}

if(temp.prod[i][k+1]=='\0')

add\_to\_follow(l,return\_index(temp.prod[i][0])-no\_t);

}

}l

}

}

print\_shift\_table(state\_count);

cout<<endl<<endl;

print\_reduce\_table(state\_count,&no\_re[0],&temp1);

}

**18CSC304J- Compiler Design**

**Predictive parsing Table**

# Sai Krishna Movva

RA1911003010751

## 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()

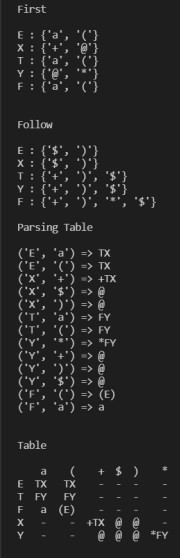
follow\_dict[start] = follow\_dict[start].union('$') for lhs in productions:

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:



**Result:** The Predictive Parsing Table was successfully constructed for the given grammar.

**Compiler Design**

**Experiment - 10**

**Sai Krishna Movva**

**RA1911003010751**

**Postfix Evaluation**

**Aim:**

To evaluate a postfix expression.

**Algorithm:**

1) Create a stack to store operands (or values).

2) Scan the given expression and do the following for every scanned element.

…..a) If the element is a number, push it into the stack

…..b) If the element is an operator, pop operands for the operator from the stack. Evaluate the operator and push the result back to the stack

3) When the expression is ended, the number in the stack is the final answer

**Program:**

#include <bits/stdc++.h>

using namespace std;

int evaluatePostfix(string exp) {

stack<int> st;

for(int i = 0; exp[i] != '\0'; i++) {

if(isdigit(exp[i]))

st.push(exp[i]-'0');

else{

int op2 = st.top();

st.pop();

int op1 = st.top();

st.pop();

switch (exp[i]) {

case '+': st.push(op1+op2); break;

case '-': st.push(op1-op2); break;

case '\*': st.push(op1\*op2); break;

case '/': st.push(op1/op2); break;

case '^': st.push(op1^op2); break;

}

}

}

return st.top();

}

int main()

{

string exp;

cout << "Postfix expression: ";

cin >> exp;

cout << "Result: ";

cout << evaluatePostfix(exp);

return 0;

}

**Output:**



**Result:**

Hence, the given postfix expression is evaluated.

**Prefix Evaluation**

**Aim:**

To evaluate a postfix expression.

**Algorithm:**

Step 1: Put a pointer P at the end of the end

Step 2: If character at P is an operand push it to Stack

Step 3: If the character at P is an operator pop two

elements from the Stack. Operate on these elements

according to the operator, and push the result

back to the Stack

Step 4: Decrement P by 1 and go to Step 2 as long as there

are characters left to be scanned in the expression.

Step 5: The Result is stored at the top of the Stack,

return it

Step 6: End

**Program:**

#include <bits/stdc++.h>

using namespace std;

int evaluatePrefix(string exp) {

stack<int> st;

for(int i = exp.size()-1; i >= 0; i--) {

if(isdigit(exp[i]))

st.push(exp[i]-'0');

else {

int op1 = st.top();

st.pop();

int op2 = st.top();

st.pop();

switch (exp[i]) {

case '+': st.push(op1+op2); break;

case '-': st.push(op1-op2); break;

case '\*': st.push(op1\*op2); break;

case '/': st.push(op1/op2); break;

case '^': st.push(op1^op2); break;

}

}

}

return st.top();

}

int main()

{

string exp;

cout << "Prefix expression: ";

cin >> exp;

cout << "Result: ";

cout << evaluatePrefix(exp);

return 0;

}

**Output:**



**Result:**

Hence, the given prefix expression is evaluated.

**Experiment 11 - Intermediate code generation- Quadruple, Triple, Indirect Triple**

Sai Krishna MovvaRA1911003010751

Aim:

A program to implement Intermediate code generation – Quadruple, Triple, Indirect Triple.

Algorithm:

The algorithm takes a sequence of three-address statements as input. For each three address statements of the form a:= b op c perform the various actions. These are as follows:

* Invoke a function getreg to find out the location L where the result of computation b op c should be stored.
* 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 MOV y' , L to place a copy of y in L.
* Generate the instruction 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.
* 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 op z those register will no longer contain y or z.

Code:

OPERATORS = set(['+', '-', '\*', '/', '(', ')']) PRI = {'+': 1, '-': 1, '\*': 2, '/': 2} def infix\_to\_postfix(formula):

stack = [] 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()

else:

while stack and stack[-1] != '(' and PRI[ch] <= PRI[stack[-1]]: output += stack.pop()

stack.append(ch)

while stack:

output += stack.pop()

return output

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()

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)

while op\_stack:

op = op\_stack.pop() a = exp\_stack.pop() b = exp\_stack.pop() exp\_stack.append(op+b+a)

return exp\_stack[-1] 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

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

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

def IndirectTriple(pos):

Stack=[]

Op=[]x=

0 c=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} | {3:^5d}".format(i, op1, "(-)", c)) x = x+1 if stack != []:

op2 = stack.pop() op1 = stack.pop()

print("{0:^4s} | {1:^4s} | {2:^4s} | {3:^5d}".format(

"+", op1, op2, c)) stack.append("(%s)" % x) x = x+1 c = c+1 elif i == '=':

op2 = stack.pop() op1 = stack.pop()

print("{0:^4s} | {1:^4s} | {2:^4s} | {3:^5d}".format(i, op1, op2, c)) c = c+1

else:

op1 = stack.pop() if stack != []:

op2 = stack.pop()

print("{0:^4s} | {1:^4s} | {2:^4s} | {3:^5d}".format(

i, op2, op1, c))

stack.append("(%s)" % x) x = x+1 c = c+1

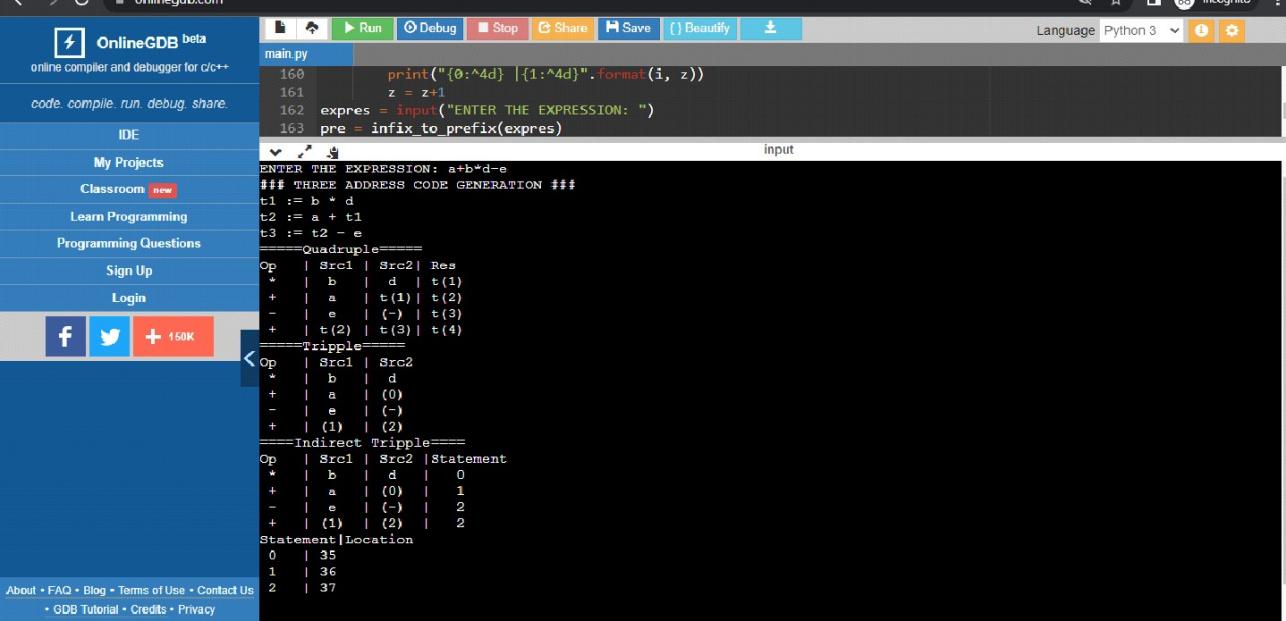
z = 35 print("Statement|Location") for i in range(0, c): print("{0:^4d} |{1:^4d}".format(i, z)) z = z+1

expres = input("ENTER THE EXPRESSION: ") pre = infix\_to\_prefix(expres) pos = infix\_to\_postfix(expres) generate3AC(pos) print("=====Quadruple=====")

print("Op | Src1 | Src2| Res") Quadruple(pos) print("=====Tripple=====") print("Op | Src1 | Src2") Triple(pos) print("====Indirect Tripple====") print("Op | Src1 | Src2 |Statement")

IndirectTriple(pos)

**Output:**



Result:

The program was successfully compiled and run.