Experiment No. 1

AIM :

Write a program to convert NFA to DFA.

THEORY:

DFA stands for Deterministic Finite Automaton. Its job is to read in a string of symbols (input) and move (transition) between its states based on the input and ultimately accept or reject the string of symbols. The transitions are deterministic: for a given symbol string and a given DFA, the DFA will go through the same states every time the same symbol string is fed into it. In other words, the path between the states is guaranteed to be unique for a given symbol string. The way this is enforced is by specifying that for every state, for a given input symbol, there's only one transition.

NFA is non-deterministic finite automaton. In an NFA:

- in a state, for one input, the automaton can transition to more than one states.
- automaton can transition using empty input.

ALGORITHM :

Suppose there is an NFA N: { Q, ∑, q0, δ, F } which recognizes a language L. Then the DFA D: { Q', ∑, q0, δ', F' } can be constructed for language L as:
Step 1: Initially Q' = ϕ.
Step 2: Add q0 to Q'.
Step 3: For each state in Q', find the possible set of states for each input symbol using transition function of NFA. If this set of states is not in Q', add it to Q'.
Step 4: Final state of DFA will be all states with contain F (final states of NFA).

CODE :

```
#include<iostream>

#include<bits/stdc++.h>

using namespace std;

void print(vector<vector<vector<int> > > table){
    cout<<"  STATE/INPUT  |";
    char a='a';
    for(int i=0;i<table[0].size()-1;i++){
        cout<<"   "<<a++<<"   |";
    }
    cout<<"   ^   "<<endl<<endl;
    for(int i=0;i<table.size();i++){
        cout<<"       "<<i<<"      ";
        for(int j=0;j<table[i].size();j++){
            cout<<" | ";
            for(int k=0;k<table[i][j].size();k++){
                cout<<table[i][j][k]<<" ";

            }
        }
        cout<<endl;
    }
}

void printdfa(vector<vector<int> > states, vector<vector<vector<int> > >
dfa){
    cout<<"  STATE/INPUT  ";
    char a='a';
    for(int i=0;i<dfa[0].size();i++){
        cout<<"|   "<<a++<<"   ";
    }
    cout<<endl;
    for(int i=0;i<states.size();i++){
          cout<<"{ ";
          for(int h=0;h<states[i].size();h++)
               cout<<states[i][h]<<" ";
          if(states[i].empty()){
               cout<<"^ ";
          }
          cout<<"} ";
          for(int j=0;j<dfa[i].size();j++){
               cout<<" | ";
               for(int k=0;k<dfa[i][j].size();k++){
                    cout<<dfa[i][j][k]<<" ";

               }
               if(dfa[i][j].empty()){
                    cout<<"^ ";
               }
          }
          cout<<endl;
     }
}
vector<int> closure(int s,vector<vector<vector<int> > > v){
     vector<int> t;
     queue<int> q;
     t.push_back(s);
     int a=v[s][v[s].size()-1].size();
     for(int i=0;i<a;i++){
          t.push_back(v[s][v[s].size()-1][i]);
```

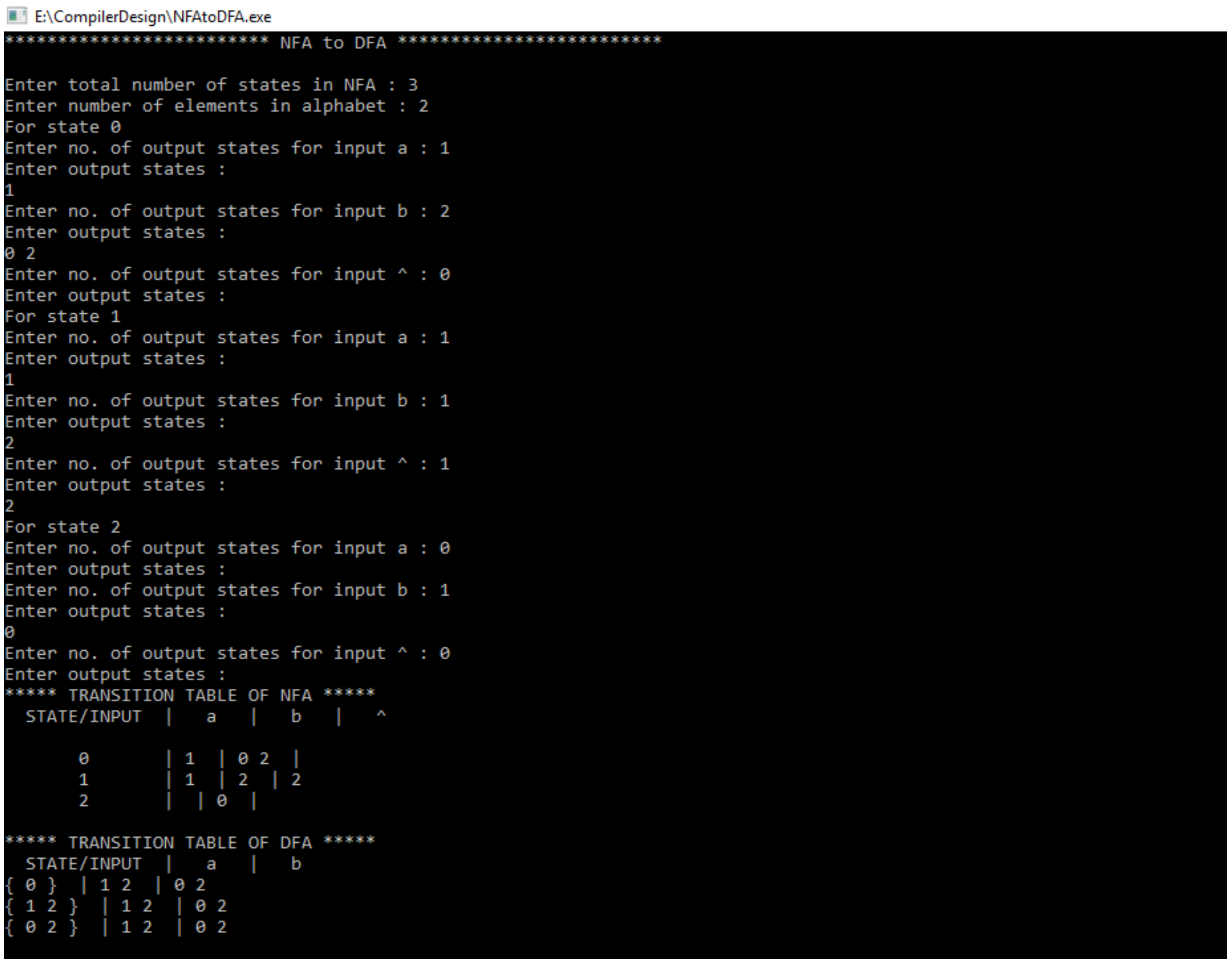
```
		//cout<<"t[i]"<<t[i]<<endl;
		q.push(t[i]);
	}
	while(!q.empty()){
		int f=q.front();
		q.pop();
		if(!v[f][v[f].size()-1].empty()){
			int u=v[f][v[f].size()-1].size();
			for(int i=0;i<u;i++){
				int y=v[f][v[f].size()-1][i];
				if(find(t.begin(),t.end(),y)==t.end()){
					//cout<<"y"<<y<<endl;
					t.push_back(y);
					q.push(y);
				}
			}
		}
	}
	return t;
}
int main(){
	int n,alpha;
	cout<<"************************* NFA to DFA 
*************************"<<endl<<endl;
	cout<<"Enter total number of states in NFA : ";
	cin>>n;
	cout<<"Enter number of elements in alphabet : ";
	cin>>alpha;
```

```
vector<vector<vector<int> > > table;
for(int i=0;i<n;i++){
      cout<<"For state "<<i<<endl;
      vector< vector< int > > v;
      char a='a';
      int y,yn;
      for(int j=0;j<alpha;j++){
            vector<int> t;
            cout<<"Enter no. of output states for input "<<a++<<" : ";
            cin>>yn;
            cout<<"Enter output states :"<<endl;
            for(int k=0;k<yn;k++){
                  cin>>y;
                  t.push_back(y);
            }
            v.push_back(t);
      }
      vector<int> t;
      cout<<"Enter no. of output states for input ^ : ";
      cin>>yn;
      cout<<"Enter output states :"<<endl;
      for(int k=0;k<yn;k++){
            cin>>y;
            t.push_back(y);
      }
      v.push_back(t);
      table.push_back(v);
```

```
}
cout<<"***** TRANSITION TABLE OF NFA *****"<<endl;
print(table);
cout<<endl<<"***** TRANSITION TABLE OF DFA *****"<<endl;
vector<vector<vector<int> > > dfa;
vector<vector<int> > states;
states.push_back(closure(0,table));
queue<vector<int> > q;
q.push(states[0]);
while(!q.empty()){
      vector<int> f=q.front();
      q.pop();
      vector<vector<int> > v;
      for(int i=0;i<alpha;i++){
            vector<int> t;
            set<int> s;
            for(int j=0;j<f.size();j++){

                  for(int k=0;k<table[f[j]][i].size();k++){
                        vector<int> cl= closure(table[f[j]][i][k],table);
                        for(int h=0;h<cl.size();h++){
                              if(s.find(cl[h])==s.end())
                              s.insert(cl[h]);
                        }
                  }
            }
            for(set<int >::iterator u=s.begin(); u!=s.end();u++)
```

```
                        t.push_back(*u);
                    v.push_back(t);
                    if(find(states.begin(),states.end(),t)==states.end())
                    {
                        states.push_back(t);
                        q.push(t);
                    }
                }
                dfa.push_back(v);
        }
        printdfa(states,dfa);
}
```

OUTPUT :



```
E:\CompilerDesign\NFAtoDFA.exe
************************* NFA to DFA *************************

Enter total number of states in NFA : 3
Enter number of elements in alphabet : 2
For state 0
Enter no. of output states for input a : 1
Enter output states :
1
Enter no. of output states for input b : 2
Enter output states :
0 2
Enter no. of output states for input ^ : 0
Enter output states :
For state 1
Enter no. of output states for input a : 1
Enter output states :
1
Enter no. of output states for input b : 1
Enter output states :
2
Enter no. of output states for input ^ : 1
Enter output states :
2
For state 2
Enter no. of output states for input a : 0
Enter output states :
Enter no. of output states for input b : 1
Enter output states :
0
Enter no. of output states for input ^ : 0
Enter output states :
***** TRANSITION TABLE OF NFA *****
  STATE/INPUT  |   a   |   b   |   ^

       0       | 1  | 0 2  |
       1       | 1  | 2  | 2
       2       |  | 0  |

***** TRANSITION TABLE OF DFA *****
  STATE/INPUT  |   a   |   b
{ 0 }  | 1 2  | 0 2
{ 1 2 }  | 1 2  | 0 2
{ 0 2 }  | 1 2  | 0 2
```

LEARNINGS :

We have learnt how to convert a given NFA to DFA. It is hard for a computer program to simulate an NFA because the transition function is multivalued. An algorithm, called the subset construction can convert an NFA for any language into a DFA that recognizes the same languages. This algorithm is closely related to an algorithm for constructing LR parser.

Experiment No. 2

AIM :

Write a program to check acceptance of string by a DFA.

THEORY:

DFA stands for Deterministic Finite Automaton. Its job is to read in a string of symbols (input) and move (transition) between its states based on the input and ultimately accept or reject the string of symbols. The transitions are deterministic: for a given symbol string and a given DFA, the DFA will go through the same states every time the same symbol string is fed into it. In other words, the path between the states is guaranteed to be unique for a given symbol string. The way this is enforced is by specifying that for every state, for a given input symbol, there's only one transition. To check acceptability, we have to check for every input and at last we have to check if it is a final state.

ALGORITHM :

```
1. Take DFA as input from user in form of transition table.
2. Input the final states in DFA.
3. While user inputs "y" :                    //If he wants to continue or not.
    a. Take a string as input from user to check for acceptance.
    b. Take 0 as initial/current state.
    c. Do following for length of string times :
        i. Move to next state by using current state and current alphabet as index in DFA
           table.
        ii. Store this next state as output state.
    d. If output state is a final state :
        i. Output : string is accepted.
    e. Else
        i. Output : string is rejected.
```

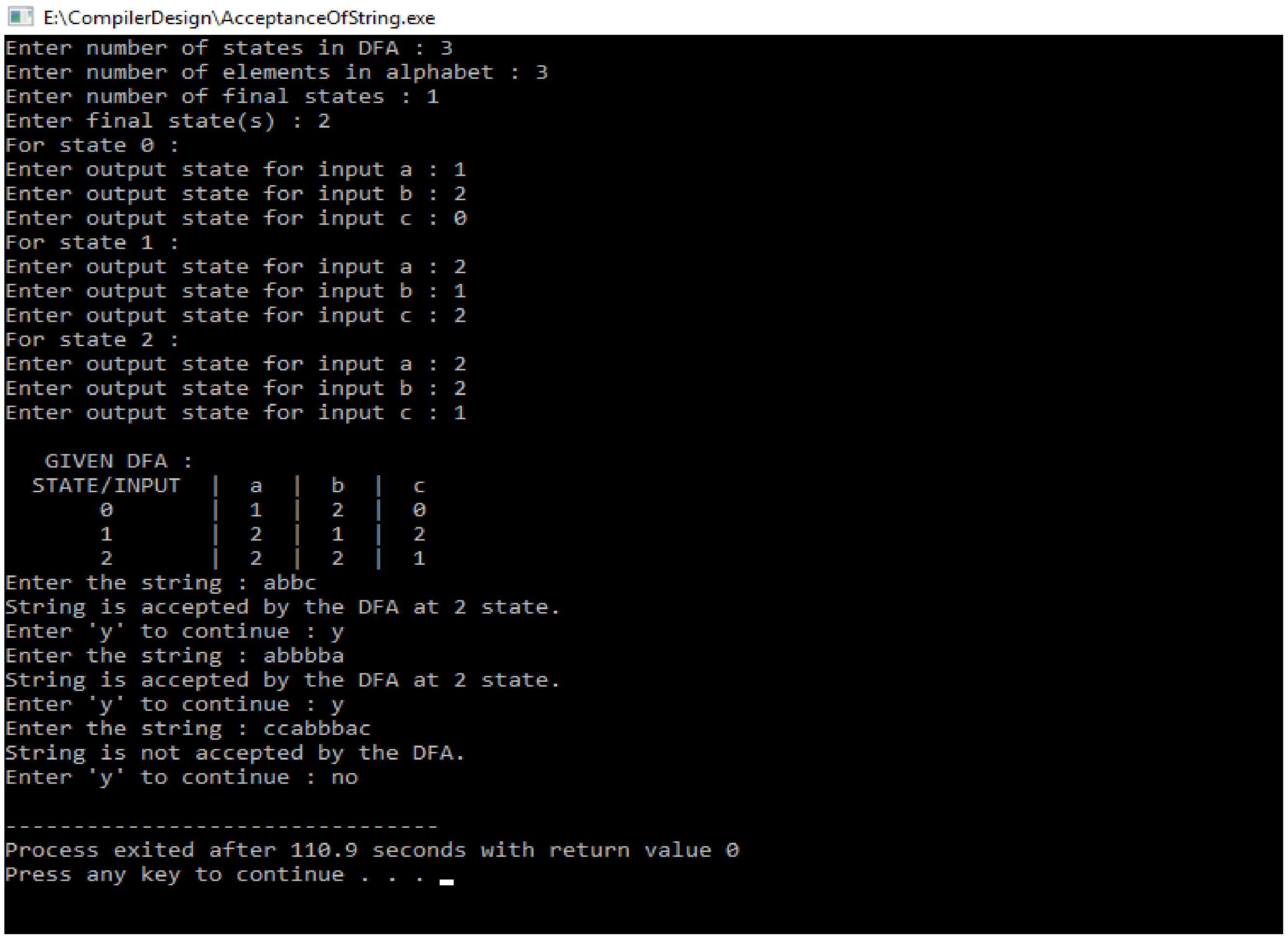
CODE :

```
#include<iostream>

#include<bits/stdc++.h>

using namespace std;

void print(vector<vector<int> > table){

      cout<<"  STATE/INPUT  ";

      char a='a';
```

```
	for(int i=0;i<table[0].size();i++){
		cout<<"|  "<<a++<<"  ";
	}
	cout<<endl;
	for(int i=0;i<table.size();i++){
		cout<<"       "<<i<<"      ";
		for(int j=0;j<table[i].size();j++){
			cout<<" |  ";
			cout<<table[i][j]<<" ";
		}
		cout<<endl;
	}
}
int main(){
	vector<vector<int> > dfa;
	int states,alpha,final;
	cout<<"Enter number of states in DFA : ";
	cin>>states;
	cout<<"Enter number of elements in alphabet : ";
	cin>>alpha;
	cout<<"Enter number of final states : ";
	cin>>final;
	vector<int> final_states(final);
	cout<<"Enter final state(s) : ";
	for(int i=0;i<final;i++){
		cin>>final_states[i];
	}
```

```
      for(int i=0;i<states;i++){
            cout<<"For state "<<i<<" :"<<endl;
            char a='a';
            vector<int> v(alpha);
            for(int j=0;j<alpha;j++){
                  cout<<"Enter output state for input "<<a<<" : ";
                  cin>>v[j];
                  a++;
            }
            dfa.push_back(v);
      }
      char ch='n';
      cout<<endl<<"   GIVEN DFA :"<<endl;
      print(dfa);
      do{
            cout<<"Enter the string : ";
            string s;
            cin>>s;
            int curr=0,out;
            for(int i=0;i<s.length();i++){
                  out=dfa[curr][s[i]-'a'];
                  curr=out;
            }

if(find(final_states.begin(),final_states.end(),out)!=final_states.end()){
                  cout<<"String is accepted by the DFA at "<<out<<"
state."<<endl;
            }else{
                    cout<<"String is not accepted by the DFA."<<endl;
                }
                cout<<"Enter 'y' to continue : ";
                cin>>ch;
        }while(ch=='y');
        return 0;
}
```

OUTPUT :



```
E:\CompilerDesign\AcceptanceOfString.exe
Enter number of states in DFA : 3
Enter number of elements in alphabet : 3
Enter number of final states : 1
Enter final state(s) : 2
For state 0 :
Enter output state for input a : 1
Enter output state for input b : 2
Enter output state for input c : 0
For state 1 :
Enter output state for input a : 2
Enter output state for input b : 1
Enter output state for input c : 2
For state 2 :
Enter output state for input a : 2
Enter output state for input b : 2
Enter output state for input c : 1

   GIVEN DFA :
  STATE/INPUT  |  a  |  b  |  c
       0       |  1  |  2  |  0
       1       |  2  |  1  |  2
       2       |  2  |  2  |  1
Enter the string : abbc
String is accepted by the DFA at 2 state.
Enter 'y' to continue : y
Enter the string : abbbba
String is accepted by the DFA at 2 state.
Enter 'y' to continue : y
Enter the string : ccabbbac
String is not accepted by the DFA.
Enter 'y' to continue : no

--------------------------------
Process exited after 110.9 seconds with return value 0
Press any key to continue . . .
```

LEARNINGS :

We have learnt how to check acceptance of a string by a given DFA. This can be particularly useful for decision problems, i.e, the problems in which we have to do something or take some decision about something after a condition is met. Such problems can be framed in this format so that when dfa accepts a string, some decision can be taken.

Experiment No. 3

AIM :

Write a program to count tokens in a given program .

THEORY:

As it is known that Lexical Analysis is the first phase of compiler also known as scanner. It converts the input program into a sequence of Tokens. A C++ program consists of various tokens and a token is either a keyword, an identifier, a constant, a string literal, or a symbol. A token is the smallest element of a program that is meaningful to the compiler. Tokens can be classified as follows:

1. Keywords
2. Identifiers
3. Constants
4. Strings
5. Special Symbols
6. Operators

ALGORITHM :

```
1. Open the program file using ifstream.
2. If file has been opened successfully do :
3. While it is not the end of file do :
    a. Parse the file line by line
    b. Check if the word in buffer currently is
        i. Keyword
        ii. Operator
        iii. Identifier
    c. Increase the count of corresponding token.
4. Close the file
5. Output the number of :
    a. Keywords
    b. Operators
    c. Identifiers
```

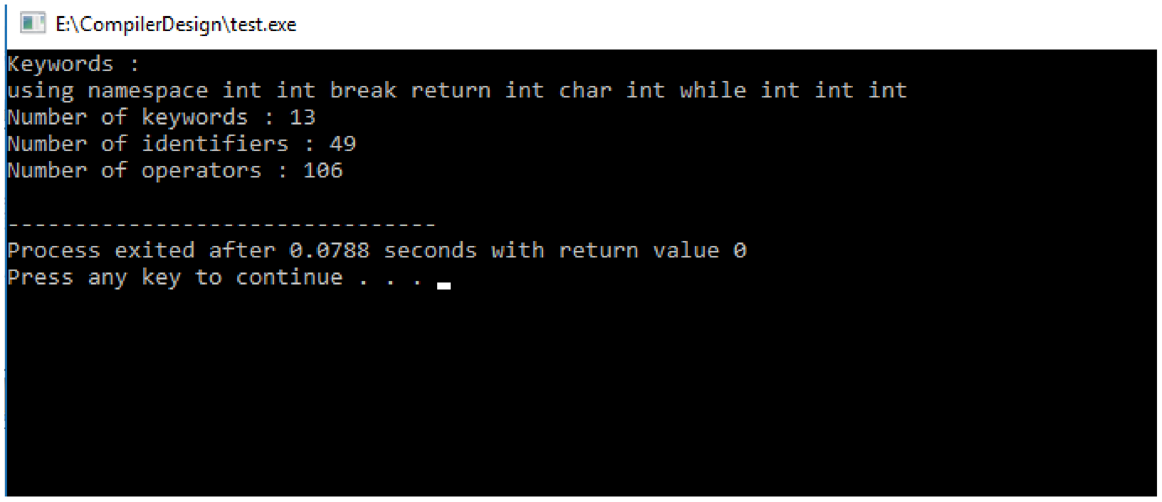
CODE :

```
#include<iostream>

#include<fstream>

#include<stdlib.h>

#include<string.h>

#include<ctype.h>

 using namespace std;

 int isKeyword(char buffer[]){

      char keywords[55][10] =
      {"auto","break","case","char","const","continue","default",

      "do", "double", "else", "enum", "extern",  "float","for","goto",

      "if", "int", "long", "register", "return", "short", "signed",
      "sizeof", "static", "struct", "switch", "typedef", "union",
      "unsigned", "void", "volatile", "while", "namespace", "bool",
      "explicit", "new",    "catch",   "false","operator", "template",

      "class",   "friend",  "private", "this","inline","public",  "throw",
      "delete",  "mutable", "protected",     "true", "try",   "typeid",
      "typename",     "using"

      };

      int i, flag = 0;

      for(i = 0; i < 55; ++i){

            if(strcmp(keywords[i], buffer) == 0){

                  flag = 1;

                  break;

            }

      }

      return flag;

}

 int main(){
```

```
char ch, buffer[15], operators[] = "+-*/%=(){,|?.><&}^~[]";

ifstream fin("test.cpp");

int i,j=0;

if(!fin.is_open()){

      cout<<"error while opening the file\n";

      exit(0);

}

cout<<"Keywords :"<<endl;

int op=0;

int id=0;

int key=0;

while(!fin.eof()){

      ch = fin.get();


      for(i = 0; i < 19; ++i){

            if(ch == operators[i])

                  op++;

      }


      if(isalnum(ch)){

            buffer[j++] = ch;

      }

      else if((ch == ' ' || ch == '\n') && (j != 0)){

                  buffer[j] = '\0';

                  j = 0;


                  if(isKeyword(buffer) == 1){
```

```
                                key++;
                                cout<<buffer<<" ";
                           }
                           else
                                id++;
                 }}
        fin.close();
        cout<<endl<<"Number of keywords : "<<key<<endl;
        cout<<"Number of identifiers : "<<id<<endl;
        cout<<"Number of operators : "<<op<<endl;
        return 0;
}
```

OUTPUT :



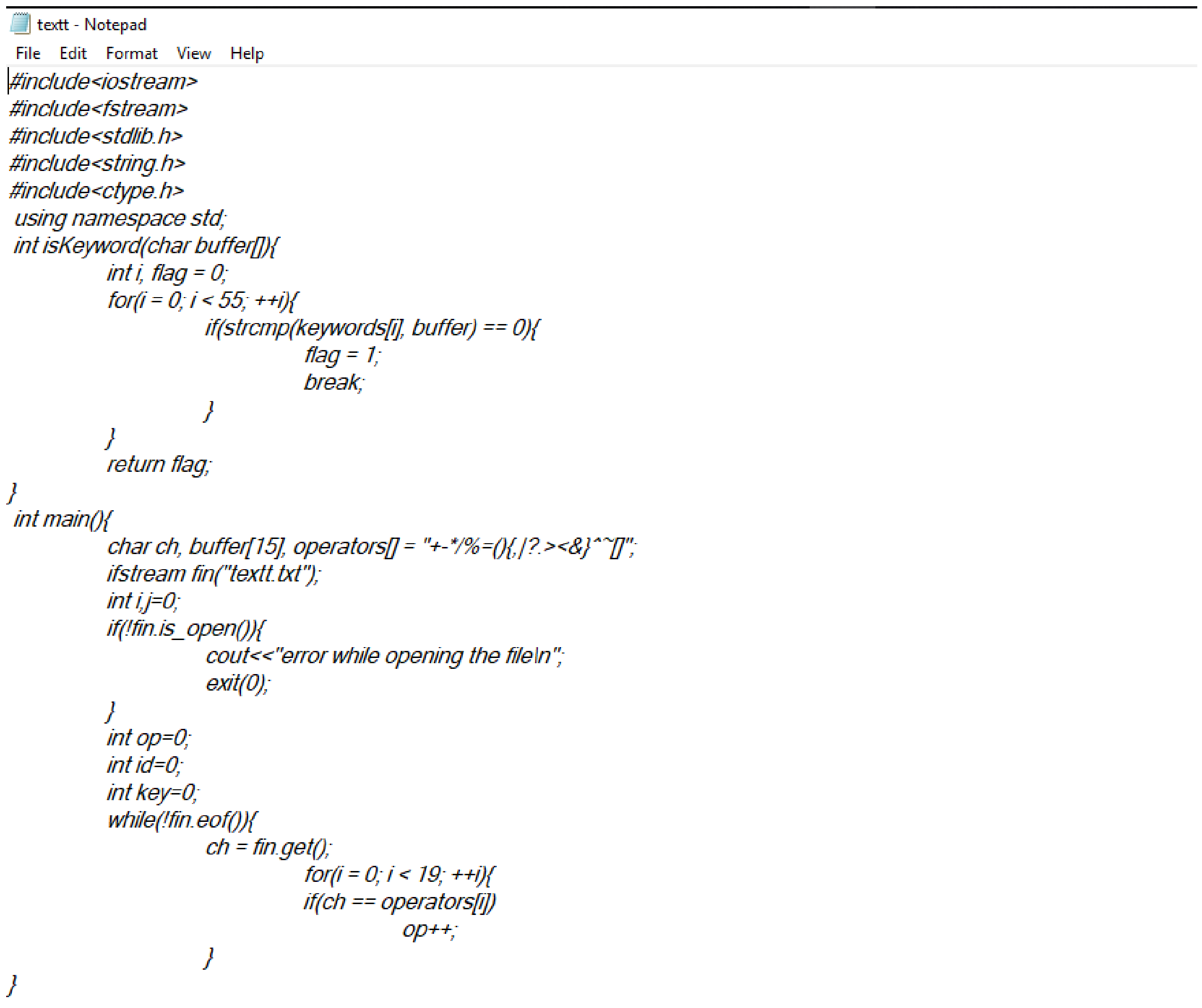
E:\CompilerDesign\test.exe

```
Keywords :
using namespace int int break return int char int while int int int
Number of keywords : 13
Number of identifiers : 49
Number of operators : 106

--------------------------------
Process exited after 0.0788 seconds with return value 0
Press any key to continue . . .
```

LEARNINGS :

We have learnt how to count tokes in a given program. We need to have the list of keywords contained in that language, C++ in the above program. We also need to have the rules required for the nomenclature of identifiers in that language.

FILE USED FOR COUNTING TOKENS IN C++ PROGRAM :



```
#include<iostream>
#include<fstream>
#include<stdlib.h>
#include<string.h>
#include<ctype.h>
 using namespace std;
 int isKeyword(char buffer[]){
		int i, flag = 0;
		for(i = 0; i < 55; ++i){
			if(strcmp(keywords[i], buffer) == 0){
				flag = 1;
				break;
			}
		}
		return flag;
}
 int main(){
		char ch, buffer[15], operators[] = "+-*/%=(){,|?.><&}^~[]";
		ifstream fin("textt.txt");
		int i,j=0;
		if(!fin.is_open()){
			cout<<"error while opening the file\n";
			exit(0);
		}
		int op=0;
		int id=0;
		int key=0;
		while(!fin.eof()){
			ch = fin.get();
				for(i = 0; i < 19; ++i){
				if(ch == operators[i])
					op++;
			}
}
```

Experiment No. 4

AIM :

Write a program to tokenize the given string.

THEORY:

As it is known that Lexical Analysis is the first phase of compiler also known as scanner. It converts the input program into a sequence of Tokens. A C++ program consists of various tokens and a token is either a keyword, an identifier, a constant, a string literal, or a symbol. A token is the smallest element of a program that is meaningful to the compiler. Tokens can be classified as follows:

1. Keywords
2. Identifiers
3. Constants
4. Strings
5. Special Symbols
6. Operators

ALGORITHM :

```
 7. Open the program file using ifstream.
 8. If file has been opened successfully do :
 9. While it is not the end of file do :
      a. Parse the file line by line
      b. If it a comment
           i. Output "COMMENT"
          ii. Break;
      c. Check if the word in buffer currently is
           i. Keyword
          ii. Operator
         iii. Identifier
      d. Output the shorthand word for that particular token.
10. Close the file
```

CODE :

```
#include<iostream>

#include<fstream>

#include<string.h>
```

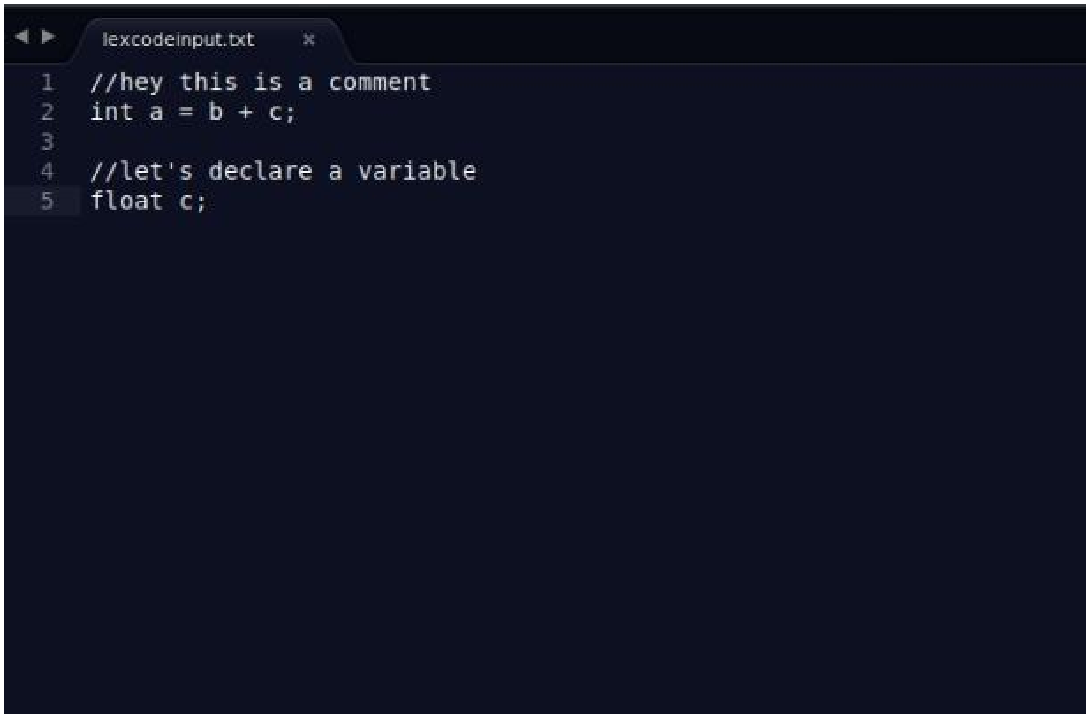
```
using namespace std;

bool isDelimiter(char* ch)
{
 if (ch[0] == ',' || ch[0] == ';' || ch[0] == '(' || ch[0] == ')'
     || ch[0] == '"' || ch[0] == '\'' || ch[0] == '[' || ch[0] ==
']' || ch[0] == '{' || ch[0] == '}')
 return (true);
 return (false);
}

bool isOperator(char* ch)
{
 if(!strcmp(ch, "<<") || !strcmp(ch, "<<") || !strcmp(ch, "endl"))
 return (true);
 if (ch[0] == '+' || ch[0] == '-' || ch[0] == '*' || ch[0] == '/'
     || ch[0] == '>' || ch[0] == '<' ||
 ch[0] == '=' )
 return (true);
 return (false);
}

bool validIdentifier(char str[])
{
 if (int(str[0]) >= int('0') && int(str[0]) <= int('9') || isDelimiter(str)
== true)
 return (false);
 return (true);
```

```
}

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

bool isInteger(char* str)
{
 int i, len = strlen(str);
 if (len == 0)
 return (false);
 for (i = 0; i < len; i++)
 {
```

```
 if (str[i] <= '0' || 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] >= '9') && str[i] != '.' || (str[i] == '-' &&
i > 0))
 return (false);
 if (str[i] == '.')
 hasDecimal = true;
 }
 return (hasDecimal);
}

bool isComment(char *str)
{
 return !strcmp(str, "//");
}
void printTokens(char *line)
```

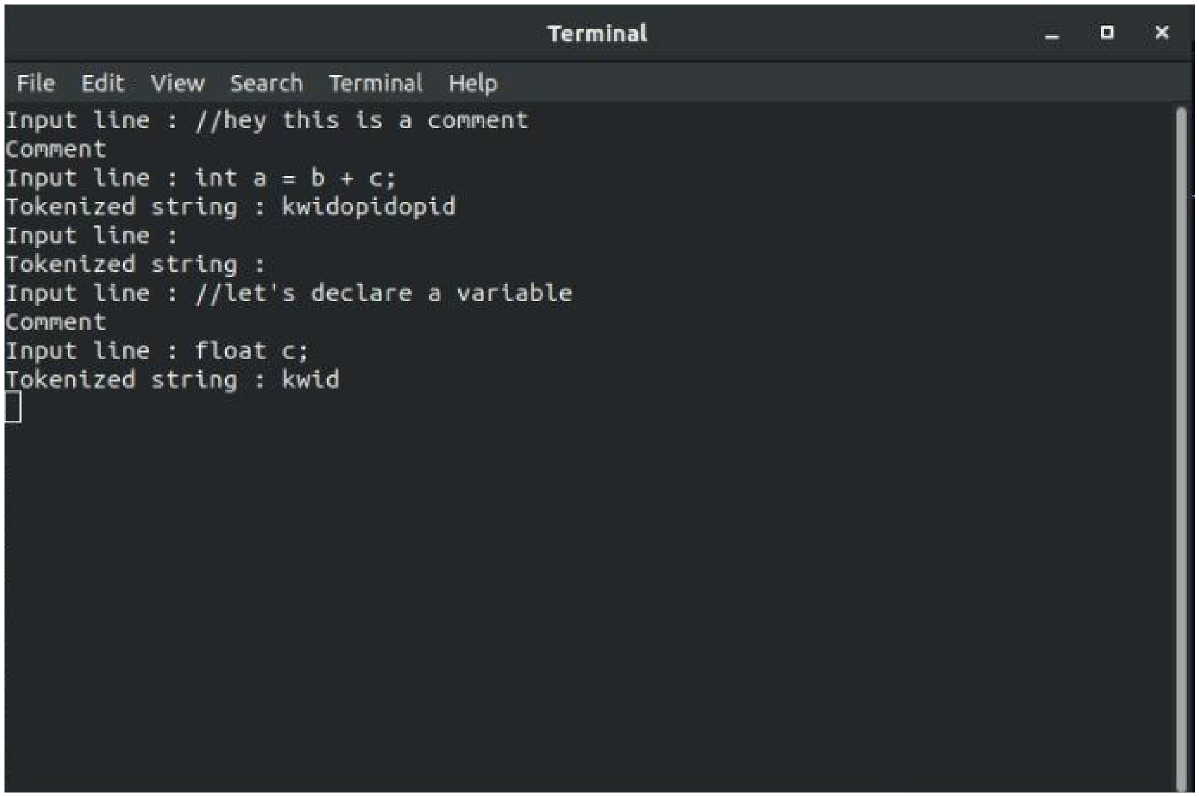
```
{
 char *token;
 token = strtok(line, " ");
 int k = 0;
 while(token != NULL)
 {
 if(isKeyword(token))
 {
 cout << "kw";
 }
 else if(isComment(token))
 {
 cout << "cmnt";
 return;
 }
 else if(isOperator(token))
 {
 cout << "op";
 }
 else if(isDelimiter(token))
 {
 cout << "dl";
 }
 else if(isInteger(token))
 {
 cout << "in";
 }
 else if(isRealNumber(token))

 {

 cout << "rn";

 }

 else if(validIdentifier(token))

 {

 cout << "id";

 }

 else if(!strcmp(token, " "))

 {

 cout << " ";

 }

 else

 {

 cout << "inv";

 }

 token = strtok(NULL, " ");

 }

}

int main()

{

 ifstream fin("lexcodeinput.txt", ios::in);

 ifstream file("lexcodeinput.txt", ios::in);

 char token[10];

 int kw = 0;

 int id = 0;

 int dl = 0;
 int op = 0;
 int rn = 0;
 int in = 0;
 int inv = 0;
 char line[100];
 while(file.getline(line, 100))
 {
 cout<< "Input line : ";
 cout << line << endl;
 if(line[0]=='/' && line[1]=='/'){
      cout<<"Comment"<<endl;
      continue;
 }
 cout<<"Tokenized string : ";
 printTokens(line);
 cout << endl;
 }
 return (0);
}
```

FILE :



```
//hey this is a comment
int a = b + c;

//let's declare a variable
float c;
```

OUTPUT :



```
Input line : //hey this is a comment
Comment
Input line : int a = b + c;
Tokenized string : kwidopidopid
Input line :
Tokenized string :
Input line : //let's declare a variable
Comment
Input line : float c;
Tokenized string : kwid
```

LEARNINGS :

We have learnt how to tokenize a given string. We need to have the list of keywords contained in that language, C++ in the above program. We also need to have the rules required for the nomenclature of identifiers in that language.

Experiment No. 5

AIM :

Write a program to remove left factoring in given grammar.

THEORY:

A grammar is said to be left factored when it is of the form –

A -> $\alpha\beta_1$ | $\alpha\beta_2$ | $\alpha\beta_3$ | …… | $\alpha\beta_n$ | γ

i.e, the productions start with the same terminal (or set of terminals). On seeing the input α we cannot immediately tell which production to choose to expand A. Left factoring is a grammar transformation that is useful for producing a grammar suitable for predictive or top down parsing. When the choice between two alternative A-productions is not clear, we may be able to rewrite the productions to defer the decision until enough of the input has been seen to make the right choice. Left factoring is removing the common left factor that appears in two productions of the same non-terminal. It is done to avoid back-tracing by the parser.

For the grammar A -> $\alpha\beta_1$ | $\alpha\beta_2$ | $\alpha\beta_3$ | …… | $\alpha\beta_n$ | γ
The equivalent left factored grammar will be –
A -> αA' | γ
A' -> β_1 | β_2 | β_3 | …… | β_n

ALGORITHM :

1. Take all productions as input and store them in hashmap.
2. Iterate through hashmap :
   a. For each RHS of production:
      i. Find the common prefix.
      ii. Replace all the RHS's having that prefix by modified RHS.
      iii. Add the new state to the production hashmap.
3. Print the modified grammar
4. This is the grammar after removing left factoring.

CODE :

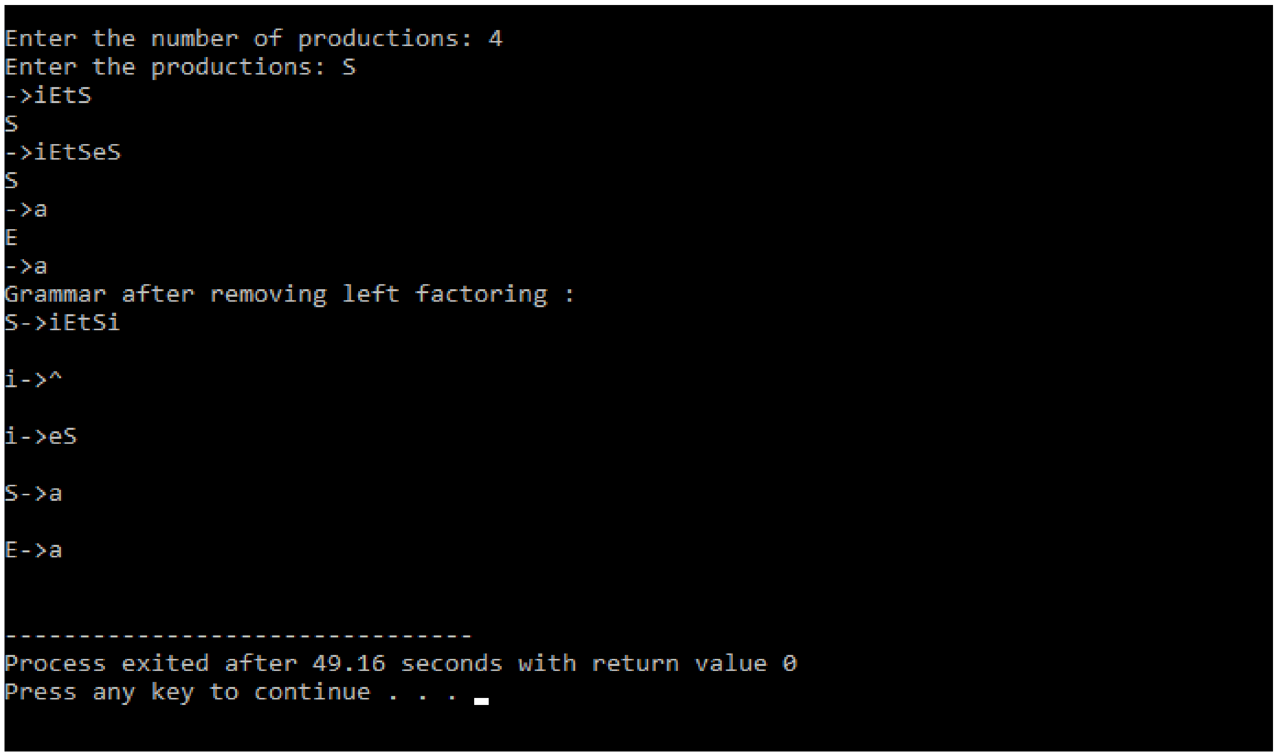
```
#include<bits/stdc++.h>

#include<iostream>
```

```
using namespace std;


int main(){

      map < char, vector<string> > prod;

      int n;

      cout<<"Enter number of productions : ";

      cin>>n;

      cout<<"Enter productions : "<<endl;

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

            char ch;

            string s;

            cin>>ch;

            cout<<" -> ";

            cin>>s;

            prod[ch].push_back(s);

      }


      cout<<"Given grammar is :"<<endl<<endl;

      for(map<char, vector<string> >::iterator i=prod.begin();
i!=prod.end(); i++){

            cout<<i->first<<" -> ";

            for(int j=0;j<i->second.size();j++){

                  cout<<i->second[j];

                  if(j!= i->second.size()-1)

                  cout<<" | ";

            }

            cout<<endl;

      }
      for(map<char, vector<string> >::iterator itr=prod.begin();
itr!=prod.end(); ++itr){



            vector<string> v= itr->second;

            //first find the maximum length that can be possible for a comman
prefix

            int min_len=INT_MAX;

            for(int j=0;j<v.size();j++){

                  if(v[j].length()<min_len)

                        min_len=v[j].length();

            }

            for(int i=0;i<v.size();i++){

                  int prev=0,curr=0,len=0;

                  //Iterate for all possible lengths of common prefix

                  for(int j= 1;j<=min_len;j++){

                        string s= v[i].substr(0,j);

                        curr=0;

                        for(int k=0;k<v.size();k++){

                              if(k==i)

                              continue;

                              if(v[k].substr(0,j).compare(s)==0)

                                    curr++;

                        }

                        //We know that previous prefix length covered more no.
of productions

                        //so that was the required min common prefix

                        if(curr<prev)
```

```
                    break;

                    prev=curr;

                    len=j;

                }

                        if(prev!=0){

                    //we have found the common prefix as :
v[i].substr(0,len)

                    //now replace the latter part with a new non-terminal

                    char ch= 'A' + prod.size();

                    string s= v[i].substr(0,len);

                    //Now see all the productions which have this prefix

                    for(int j=0;j<v.size();j++){

                        if(v[j].substr(0,len).compare(s)==0){

                            v[j]= s + ch;

                            itr->second[j]= v[j];

prod[ch].push_back(v[j].substr(len,v[j].size()-len));

                        }

                    }

                }

            }


        }

        cout<<"Grammar after removing left factoring is :"<<endl<<endl;

        for(map<char, vector<string> >::iterator i=prod.begin();
i!=prod.end(); i++){

            cout<<i->first<<" -> ";

            for(int j=0;j<i->second.size();j++){

                cout<<i->second[j];
```

```
                    if(j!= i->second.size()-1)

                    cout<<" | ";

            }

            cout<<endl;

        }

}
```

OUTPUT :



```
Enter the number of productions: 4
Enter the productions: S
->iEtS
S
->iEtSeS
S
->a
E
->a
Grammar after removing left factoring :
S->iEtSi

i->^

i->eS

S->a

E->a


--------------------------------
Process exited after 49.16 seconds with return value 0
Press any key to continue . . .
```

LEARNINGS :

We have learnt how to remove left factoring from a given grammar. Left factoring is a process by which the grammar with common prefixes is transformed to make it useful for Top down parsers. In left factoring, we make one production for each common prefixes and rest of the derivation is added by new productions. The grammar obtained after the process of left factoring is called as **left factored grammar**.

Experiment No. 6

AIM :

Write a program to remove left recursion in given grammar.

THEORY:

A production of grammar is said to have left recursion if the leftmost variable of its RHS is same as variable of its LHS. A grammar containing a production having left recursion is called as Left Recursive Grammar. For example :

$$S \rightarrow Sa \,/\, \in$$

Left recursion is considered to be a problematic situation for Top down parsers. Therefore, left recursion has to be eliminated from the grammar. Left recursion is eliminated by converting the grammar into a right recursive grammar. If we have the left-recursive pair of productions (where β does not begin with an A.) -

$$A \rightarrow A\alpha \,/\, \beta$$

Then, we can eliminate left recursion by replacing the pair of productions with-

$$A \rightarrow \beta A'$$

$$A' \rightarrow \alpha A' \,/\, \in$$

This right recursive grammar functions same as left recursive grammar.

ALGORITHM :

```
1. Take all productions as input and store them in hashmap.
2. Iterate through hashmap :
    a. For each of the  production:
        i. Initialize alpha and beta as empty vectors
        ii. If the first char in RHS equals LHS
```

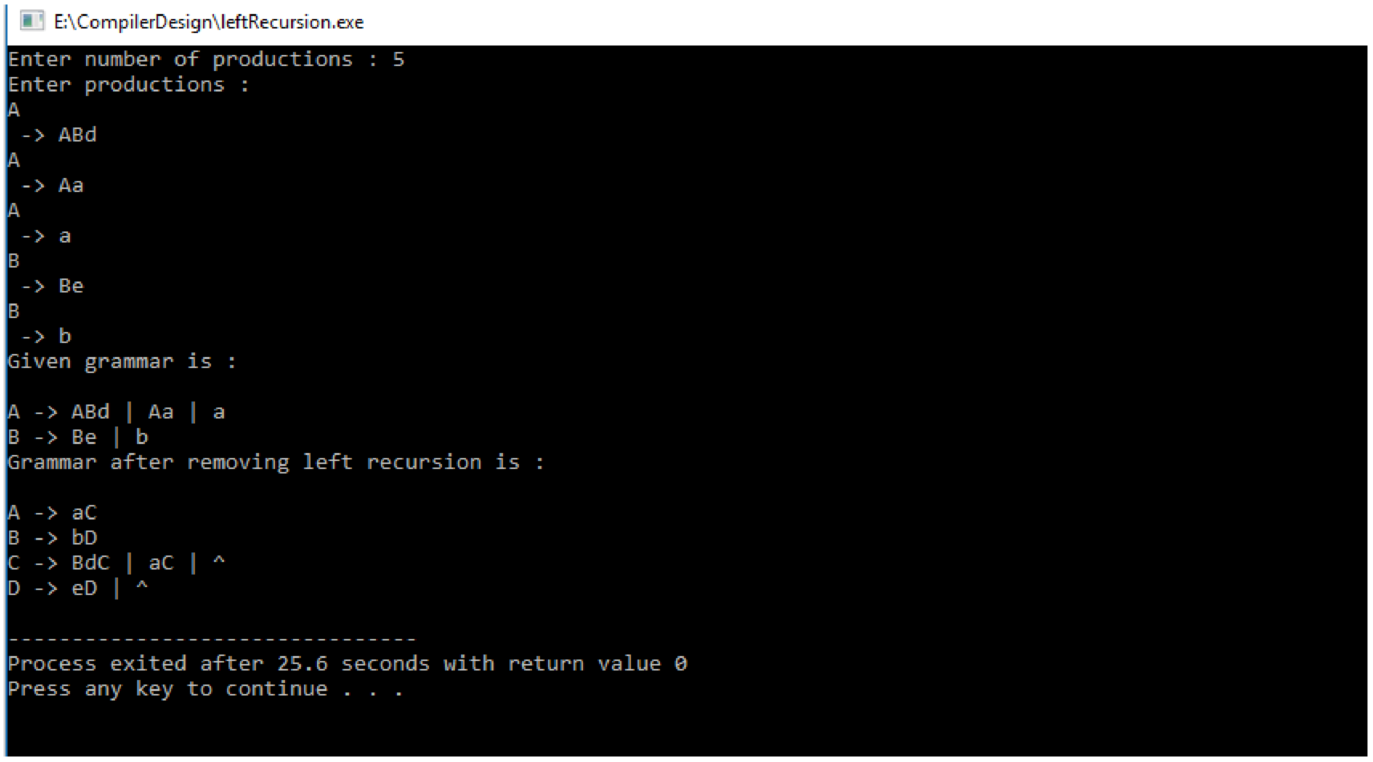
        1. Store RHS starting from 2nd char in alpha.
        2. Store remaining into beta.
        3. Delete this production from hashmap.
        4. Make new productions starting with a new character using alpha.
3. Print the modified grammar
4. This is the grammar after removing left recursion.

CODE :

```
#include<bits/stdc++.h>

#include<iostream>

using namespace std;

int main(){

      map < char, vector<string> > prod;

      int n;

      cout<<"Enter number of productions : ";

      cin>>n;

      cout<<"Enter productions : "<<endl;

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

            char ch;

            string s;

            cin>>ch;

            cout<<" -> ";

            cin>>s;

            prod[ch].push_back(s);

      }

      cout<<"Given grammar is :"<<endl<<endl;

      for(map<char, vector<string> >::iterator i=prod.begin();
i!=prod.end(); i++){

            cout<<i->first<<" -> ";

            for(int j=0;j<i->second.size();j++){
                cout<<i->second[j];

                if(j!= i->second.size()-1)

                cout<<" | ";}

           cout<<endl;}

      for(map<char, vector<string> >::iterator itr=prod.begin();
itr!=prod.end(); ++itr){

           vector<string> alpha,beta;

           for(int i= 0;i<itr->second.size();i++){

                if(itr->first==itr->second[i][0]){

alpha.push_back(itr->second[i].substr(1,itr->second[i].length()-1));

                }

                else{

                     beta.push_back(itr->second[i]);

                }

           }

           if(alpha.size()<1)

           continue;

          itr->second.clear();

           char ch= 'A' + prod.size();

           for(int i=0;i<beta.size();i++){

                itr->second.push_back(beta[i] + ch);

           }

           for(int i=0;i<alpha.size();i++){

                prod[ch].push_back(alpha[i]+ch);

           }

           prod[ch].push_back("^");

      }
```

```
	cout<<"Grammar after removing left recursion is :"<<endl<<endl;


	for(map<char, vector<string> >::iterator i=prod.begin();
i!=prod.end(); i++){

		cout<<i->first<<" -> ";

		for(int j=0;j<i->second.size();j++){

			cout<<i->second[j];

			if(j!= i->second.size()-1)

			cout<<" | ";

		}

		cout<<endl;

	}

}
```

OUTPUT :



```
E:\CompilerDesign\leftRecursion.exe
Enter number of productions : 5
Enter productions :
A
 -> ABd
A
 -> Aa
A
 -> a
B
 -> Be
B
 -> b
Given grammar is :

A -> ABd | Aa | a
B -> Be | b
Grammar after removing left recursion is :

A -> aC
B -> bD
C -> BdC | aC | ^
D -> eD | ^

--------------------------------
Process exited after 25.6 seconds with return value 0
Press any key to continue . . .
```

LEARNINGS :

We have learnt how to remove left recursion from a given grammar. Left recursion often poses problems for parsers because it leads them into infinite recursion (as in the case of most top-down parsers) . Therefore, a grammar is often preprocessed to eliminate the left recursion.

Experiment No. 7

AIM :

Write a program to compute FIRST and FOLLOW for a given grammar.

THEORY:

We need to find FIRST and FOLLOW sets for a given grammar, so that the parser can properly apply the needed rule at the correct position.

1) First(α) is a set of terminal symbols that begin in strings derived from α. Rules for calculating FIRST :
   i) For a production rule $X \rightarrow \in$
      First(X) = { ∈ }
   ii) For any terminal symbol 'a'
      First(a) = { a }
   iii) For a production rule $X \rightarrow Y1Y2Y3$, First(X) =
      a) If $\in \notin$ First(Y1), then First(X) = First(Y1)
      b) If $\in \in$ First(Y1), then First(X) = { First(Y1) – ∈ } ∪ First(Y2Y3)

2) Follow(α) is a set of terminal symbols that appear immediately to the right of α. Rules for calculating FOLLOW :
   i) For the start symbol S, place \$ in Follow(S).
   ii) For any production rule $A \rightarrow \alpha B$
      Follow(B) = Follow(A)
   iii) For any production rule $A \rightarrow \alpha B \beta$
      a) If $\in \notin$ First(β), then Follow(B) = First(β)
      b) If $\in \in$ First(β), then Follow(B) = { First(β) – ∈ } ∪ Follow(A)

ALGORITHM :

```
1. Take all productions as input and store them in hashmap.
2. Iterate through hashmap(char, vector<string>) :
    a. // Calculating FIRST
    b. For each of the char :
        i. Iterate through its all productions :
            1. If 1st element is terminal or null :
                a. Add it to first, continue.
            2. Else
                a. Compute FIRST of this element.
                b. Add it to FIRST.
                c. If FIRST contains NULL :
                    i. Compute first of next element
                    ii. Add it to FIRST.
        ii. Print FIRST for this char.
3. Iterate through hashmap(char, vector<string>) :
    a. //Calculating FOLLOW
    b. For each char ch:
        i. Iterate through its all productions :
```

```
        1. Iif RHS contains ch :
            a. Index = index of ch in RHS.
            b. Compute FIRST of substring starting at index.
            c. Add it to FOLLOW.
            d. If FIRST conatins NULL :
                i. Add FOLLOW of LHS to FOLLOW.
    ii. Print FOLLOW for this ch.
```

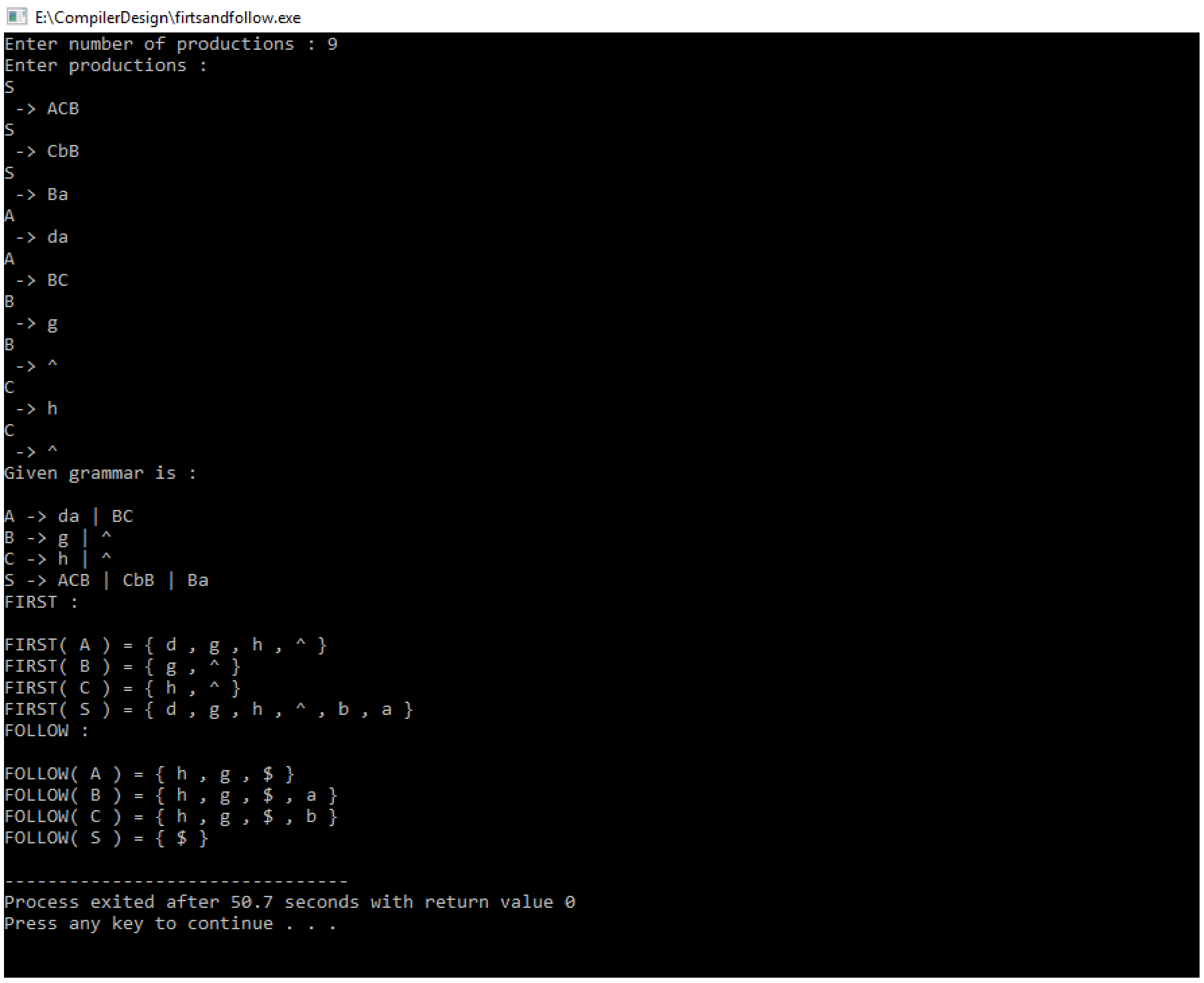
CODE :

```
#include<iostream>

using namespace std;

#include<bits/stdc++.h>


string first(map<char,vector<string> > m, char ch){

    if(ch=='^' || !(ch>='A' && ch<='Z'))

        return ch+"";

    string ans="";

    for(int i=0;i<m[ch].size();i++){

        string s = m[ch][i];

        bool checknext=true;

        for(int j=0;j<s.length() && checknext;j++){


            checknext = false;

            if(s[j]=='^' || !(s[j]>='A' && s[j]<='Z')){

                if(ans.find(s[j])==string::npos)

                    ans.push_back(s[j]);

            }

            else{

                string temp = first(m,s[j]);

                for(int k=0;k<temp.length();k++){
                    if(temp[k]=='^')
                        checknext=true;
                    else if(ans.find(temp[k])==string::npos)
                        ans.push_back(temp[k]);
                }
                if(checknext && j==s.length()-1)
                ans.push_back('^');
            }
        }
    }
    return ans;
}
string firstofstring(map<char,vector<string> > m, string s){
    string ans="";
    bool checknext = true;
    for(int j=0;j<s.length() && checknext;j++){

        checknext = false;
        if(s[j]=='^' || !(s[j]>='A' && s[j]<='Z')){
            if(ans.find(s[j])==string::npos)
                ans.push_back(s[j]);
        }
        else{
            string temp = first(m,s[j]);
            for(int k=0;k<temp.length();k++){
                if(temp[k]=='^')
                    checknext=true;
                    else if(ans.find(temp[k])==string::npos)
                         ans.push_back(temp[k]);
               }
               if(checknext && j==s.length()-1)
                    ans.push_back('^');
          }
     }
     if(ans=="")
          return "^";
     return ans;
}
string follow(map<char, vector<string> > prod, char start, char ch){
     string ans="";
     if(start==ch)
          ans.push_back('$');
     for(map<char, vector<string> >::iterator i=prod.begin();
i!=prod.end(); i++){

          for(int j=0;j<i->second.size();j++){

               if(i->second[j].find(ch)==string::npos)
                    continue;
               string temp="";
               bool parent = false;
               for(int a=0;a<i->second[j].length();a++){
                    parent=false;
                    if(i->second[j][a]==ch){
```

```
                              if(a==i->second[j].length()-1){
                              parent=true;


                              }else{
                                    //cout<<"substr
"<<i->second[j].substr(a,i->second[j].length()-a-1)<<endl;
                                    temp =
firstofstring(prod,i->second[j].substr(a+1,i->second[j].length()-a-1));
                                    //cout<<"temp "<<temp;
                                    for(int k=0;k<temp.length();k++){
                                          if(temp[k]=='^'){
                                                parent=true;
                                                continue;
                                          }
                                          if(ans.find(temp[k])==string::npos)
                                                ans.push_back(temp[k]);
                                    }
                              }
                              if(parent){
                                    //to tackle the case when parent is same as
'ch'
                                    if(ch==i->first)
                                          continue;
                                    temp=follow(prod,start,i->first);
                                    for(int k=0;k<temp.length();k++){
                                          if(ans.find(temp[k])==string::npos)
                                                ans.push_back(temp[k]);
                                    }
                              }
```

```
                        }
                  }
            }
      }
      return ans;
}
int main(){
      map <char, vector<string> > prod;
      int n;
      cout<<"Enter number of productions : ";
      cin>>n;
      cout<<"Enter productions : "<<endl;
      char start;
      for(int i=0;i<n;i++){
            char ch;
            string s;
            cin>>ch;
            if(i==0)
            start=ch;
            cout<<" -> ";
            cin>>s;
            prod[ch].push_back(s);
      }
      cout<<"Given grammar is :"<<endl<<endl;
      for(map<char, vector<string> >::iterator i=prod.begin();
i!=prod.end(); i++){
            cout<<i->first<<" -> ";
            for(int j=0;j<i->second.size();j++){
                cout<<i->second[j];
                if(j!= i->second.size()-1)
                cout<<" | ";
            }
            cout<<endl;
        }cout<<"FIRST : "<<endl<<endl;
        for(map<char, vector<string> >::iterator i=prod.begin();
i!=prod.end(); i++){
            cout<<"FIRST( "<<i->first<<" ) = { ";
            string s= first(prod,i->first);
            for(int j=0;j<s.length();j++)
                if(j!=s.length()-1)
                    cout<<s[j]<<" , ";
                else
                    cout<<s[j]<<" }";
            cout<<endl;
        }
        cout<<"FOLLOW :"<<endl<<endl;
        for(map<char, vector<string> >::iterator i=prod.begin();
i!=prod.end(); i++){
            cout<<"FOLLOW( "<<i->first<<" ) = { ";
            string s= follow(prod,start,i->first);
            for(int j=0;j<s.length();j++)
                if(j!=s.length()-1)
                    cout<<s[j]<<" , ";
                else
                    cout<<s[j]<<" ";
            cout<<"}"<<endl;
        }

}
```

OUTPUT :



E:\CompilerDesign\firtsandfollow.exe

```
Enter number of productions : 9
Enter productions :
S
 -> ACB
S
 -> CbB
S
 -> Ba
A
 -> da
A
 -> BC
B
 -> g
B
 -> ^
C
 -> h
C
 -> ^
Given grammar is :

A -> da | BC
B -> g | ^
C -> h | ^
S -> ACB | CbB | Ba
FIRST :

FIRST( A ) = { d , g , h , ^ }
FIRST( B ) = { g , ^ }
FIRST( C ) = { h , ^ }
FIRST( S ) = { d , g , h , ^ , b , a }
FOLLOW :

FOLLOW( A ) = { h , g , $ }
FOLLOW( B ) = { h , g , $ , a }
FOLLOW( C ) = { h , g , $ , b }
FOLLOW( S ) = { $ }

--------------------------------
Process exited after 50.7 seconds with return value 0
Press any key to continue . . .
```

LEARNINGS :

We have learnt how to compute FIRST and FOLLOW for a given grammar. They are useful in the further process of designing a parser.

Experiment No. 8

AIM :

Write a program to design LL(1) parser.

THEORY:

A top-down parser builds the parse tree from the top down, starting with the start non-terminal. This is a type of top down parser. 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 we going to see when you want to make a decision.

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.

ALGORITHM :

1. Input no. of terminals.
2. Store all the terminals with an index in a hashmap<char,int>.
3. Store all the productions in a hashmap<char,vector<string>>.
4. Define a parsing table as vector<vector<string>> .
5. Iterate through productions' hashmap :

   a. Compute FIRST of R.H.S.
   b. Enter this production in the block corresponding to LHS and each terminal in the FIRST obtained in step a.
   c. If FIRST contains null
      i. Compute FOLLOW of LHS.
      ii. Add the production : LHS -> null to each entry corresponding to LHS and terminals in FOLLOW computed in step i.
6. Print the parsing table.

CODE :

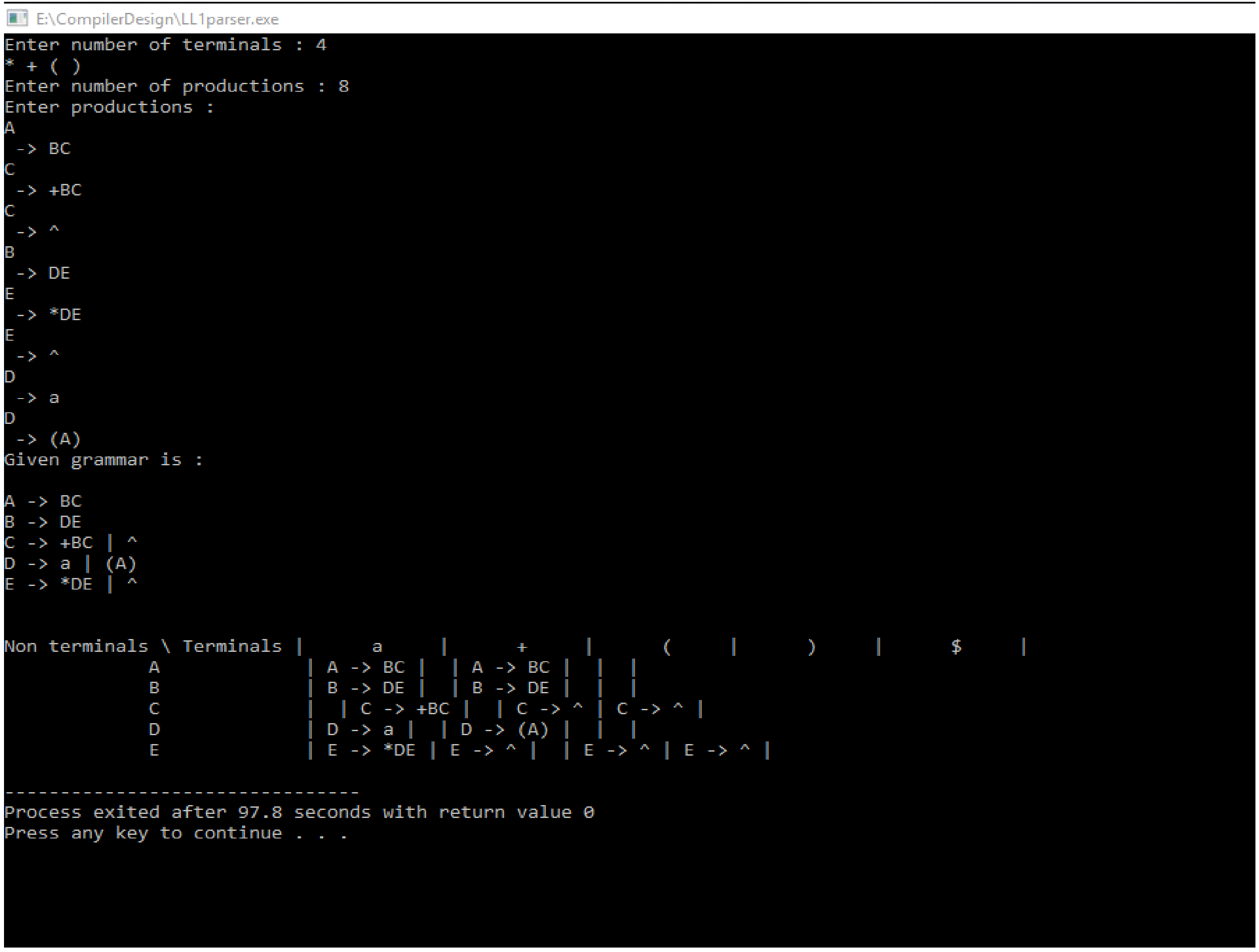
```
#include<iostream>
#include<bits/stdc++.h>
using namespace std;
string first(map<char,vector<string> > m, char ch){
     if(ch=='^' || !(ch>='A' && ch<='Z'))
          return ch+"";
     string ans="";
     for(int i=0;i<m[ch].size();i++){
          string s = m[ch][i];
          bool checknext=true;
          for(int j=0;j<s.length() && checknext;j++){

               checknext = false;
               if(s[j]=='^' || !(s[j]>='A' && s[j]<='Z')){
                    if(ans.find(s[j])==string::npos)
                         ans.push_back(s[j]);
               }
               else{
                    string temp = first(m,s[j]);
                    for(int k=0;k<temp.length();k++){
                         if(temp[k]=='^')
                              checknext=true;
                         else if(ans.find(temp[k])==string::npos)
                              ans.push_back(temp[k]);
                    }
                    if(checknext && j==s.length()-1)
                    ans.push_back('^');
               }
```

```
            }
        }
        return ans;
}
string firstofstring(map<char,vector<string> > m, string s){
        string ans="";
        bool checknext = true;
        for(int j=0;j<s.length() && checknext;j++){

            checknext = false;
            if(s[j]=='^' || !(s[j]>='A' && s[j]<='Z')){
                if(ans.find(s[j])==string::npos)
                    ans.push_back(s[j]);
            }
            else{
                string temp = first(m,s[j]);
                for(int k=0;k<temp.length();k++){
                    if(temp[k]=='^')
                        checknext=true;
                    else if(ans.find(temp[k])==string::npos)
                        ans.push_back(temp[k]);
                }
                if(checknext && j==s.length()-1)
                    ans.push_back('^');
            }
        }
        if(ans=="")
            return "^";
        return ans;
}
string follow(map<char, vector<string> > prod, char start, char ch){
        string ans="";
        if(start==ch)
            ans.push_back('$');
        for(map<char, vector<string> >::iterator i=prod.begin();
i!=prod.end(); i++){

            for(int j=0;j<i->second.size();j++){
                if(i->second[j].find(ch)==string::npos)
                     continue;
                string temp="";
                bool parent = false;
                for(int a=0;a<i->second[j].length();a++){
                     parent=false;
                     if(i->second[j][a]==ch){

                          if(a==i->second[j].length()-1){
                          parent=true;

                          }else{
                               //cout<<"substr
"<<i->second[j].substr(a,i->second[j].length()-a-1)<<endl;
                               temp =
firstofstring(prod,i->second[j].substr(a+1,i->second[j].length()-a-1)
);
                               //cout<<"temp "<<temp;
                               for(int k=0;k<temp.length();k++){
                                    if(temp[k]=='^'){
                                         parent=true;
                                         continue;
                                    }

if(ans.find(temp[k])==string::npos)
                                         ans.push_back(temp[k]);
                               }
                          }
                          if(parent){
                               //to tackle the case when parent is
same as 'ch'
                               if(ch==i->first)
                                    continue;
                               temp=follow(prod,start,i->first);
                               for(int k=0;k<temp.length();k++){

if(ans.find(temp[k])==string::npos)
                                         ans.push_back(temp[k]);
                               }
```

```
                              }
                         }
                    }
               }
          }
          return ans;
     }
     int main(){
          map <char, vector<string> > prod;
          int n;
          int t;
          cout<<"Enter number of terminals : ";
          cin>>t;
          map<char,int> terminals;
          for(int i=0;i<t;i++){
               char ch;
               cin>>ch;

               terminals[ch] = i;
          }
          terminals['$'] = t;
          cout<<"Enter number of productions : ";
          cin>>n;
          cout<<"Enter productions : "<<endl;
          char start;
          for(int i=0;i<n;i++){
               char ch;
               string s;
               cin>>ch;
               if(i==0)
               start=ch;
               cout<<" -> ";
               cin>>s;
               prod[ch].push_back(s);
          }

          cout<<"Given grammar is :"<<endl<<endl;
          for(map<char, vector<string> >::iterator i=prod.begin();
     i!=prod.end(); i++){
          cout<<i->first<<" -> ";
          for(int j=0;j<i->second.size();j++){
               cout<<i->second[j];
               if(j!= i->second.size()-1)
               cout<<" | ";
          }
          cout<<endl;
     }
     vector< vector< string > > table(prod.size(),
vector<string>(t+1, "") );
     for(map<char, vector<string> >::iterator i=prod.begin();
i!=prod.end(); i++){

          for(int j=0;j<i->second.size();j++){
               string s = firstofstring(prod, i->second[j]);
               bool hasnull=false;
               for(int k=0;k<s.length();k++){
                    if(s[k]=='^'){
                         hasnull=true;
                         continue;
                    }
                    string gg="";
                    gg= gg+i->first;
                    gg= gg + " -> ";
                    gg= gg +i->second[j];
                    table[i->first - 'A'][terminals[s[k]]] = gg;

               }
               if(hasnull){
                    string temp = follow (prod,start,i->first);
                    for(int k=0;k<temp.length();k++){
                         string ss= "";
                         ss+=i->first;
                         ss += " -> ^";
                         table[i->first - 'A'][terminals[temp[k]]] =
ss ;
                    }
               }
          }
```

```
    }
    cout<<endl<<endl;
    cout<<"Non terminals \\ Terminals | ";
    vector<char> vv(t+1);
    for(map<char,int>::iterator i= terminals.begin();
i!=terminals.end();i++)
        vv[i->second]=i->first;

    for(int i=0;i<vv.size();i++)
    cout<<"     "<<vv[i]<<"     | ";
    cout<<endl;
    int ii=0;
    for(map<char, vector<string> >::iterator i=prod.begin();
i!=prod.end(),ii<table.size(); i++,ii++){

        cout<<"             "<<i->first<<"             |";
        for(int j=0;j<table[ii].size();j++){
            cout<<" "<<table[ii][j]<<" |";
        }
        cout<<endl;
    }
}
```

OUTPUT :



E:\CompilerDesign\LL1parser.exe

```
Enter number of terminals : 4
* + ( )
Enter number of productions : 8
Enter productions :
A
 -> BC
C
 -> +BC
C
 -> ^
B
 -> DE
E
 -> *DE
E
 -> ^
D
 -> a
D
 -> (A)
Given grammar is :

A -> BC
B -> DE
C -> +BC | ^
D -> a | (A)
E -> *DE | ^


Non terminals \ Terminals |      a     |      +     |      (     |      )     |      $     |
             A             | A -> BC |  | A -> BC |  |  |
             B             | B -> DE |  | B -> DE |  |  |
             C             |  | C -> +BC |  | C -> ^ | C -> ^ |
             D             | D -> a |  | D -> (A) |  |  |
             E             | E -> *DE | E -> ^ |  | E -> ^ | E -> ^ |

--------------------------------
Process exited after 97.8 seconds with return value 0
Press any key to continue . . .
```

LEARNINGS :

We have learnt how to design parsing table for LL(1) parser by computing FIRST and FOLLOW of RHS of each production.