

# DELHI TECHNOLOGICAL UNIVERSITY

## DEPARTMENT OF COMPUTER SCIENCE



---

## COMPILER DESIGN (CO-302)

### LAB FILE

---

#### Submitted to:

Mr. Sanjay Patidar  
Assistant Professor  
Department of Computer Science  
Delhi Technological University

#### Submitted by:

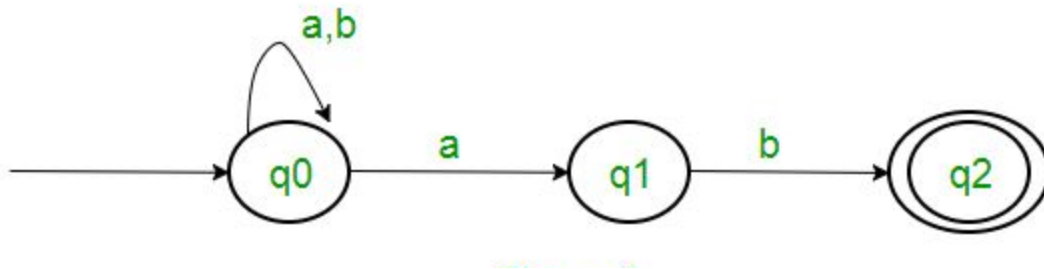
Kunal Sinha  
**2K17/CO/164**  
Computer Science(A3)  
**Batch A3 Group G1**

# INDEX

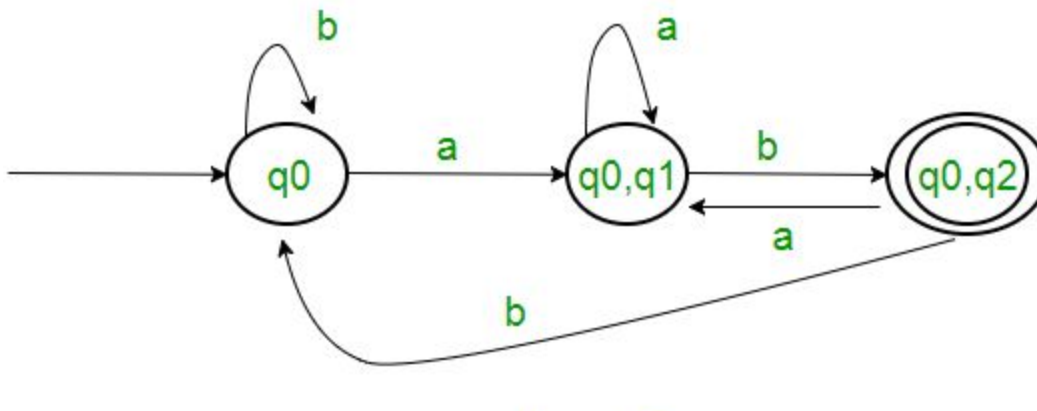
S.No	Experiment	Date	Signature	Remarks
1	Write a program to convert given NFA to DFA	09/01/2020		
2	Write a program for acceptance of string by DFA	09/01/2020		
3	Write a program to find different tokens in a program	16/01/2020		
4	Write a program to implement Lexical Analyser	16/01/2020		
5	Write a program to implement recursive descent parser	23/01/2020		
6	Write a program to left factor the given grammar	30/01/2020		
7	Write a program to convert left recursive grammar to right recursive grammar	06/02/2020		
8	Write a program to compute FIRST and FOLLOW	20/02/2020		
9	Write a program to construct LL(1) parsing table	27/02/2020		
10	Write a program to implement non recursive predictive parsing.	09/04/2020		
11	Write a study of an error handler	16/04/2020		
12	Write a study of one pass compiler	23/04/2020		

## EXPERIMENT 1

- **OBJECTIVE** : Write a program to convert given NFA to DFA
- **THEORY** : An NFA can have zero, one or more than one move from a given state on a given input symbol. An NFA can also have NULL moves (moves without input symbol). On the other hand, DFA has one and only one move from a given state on a given input symbol.  
NFA



Equivalent DFA



The algorithm consist of following steps:

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

- **CODE :**

```
#include <iostream>
#include <fstream>
#include <bitset>
#include <vector>
#include <set>
#include <queue>
#define MAX_NFA_STATES 10
#define MAX_ALPHABET_SIZE 10

using namespace std;
// Representation of an NFA state
class NFAsate {
public:
int transitions[MAX_ALPHABET_SIZE][MAX_NFA_STATES];
NFAsate()
{
for (int i = 0; i < MAX_ALPHABET_SIZE; i++)
for (int j = 0; j < MAX_NFA_STATES; j++)
transitions[i][j] = -1;
}
} * NFAsates;
// Representation of a DFA state
struct DFAsate {
bool finalState;
bitset<MAX_NFA_STATES> constituentNFAsates;
bitset<MAX_NFA_STATES> transitions[MAX_ALPHABET_SIZE];
int symbolicTransitions[MAX_ALPHABET_SIZE];
};
set<int> NFA_finalStates;
vector<int> DFA_finalStates;
vector<DFAsate*> DFAsates;
queue<int> incompleteDFAsates;
int N, M; // N -> No. of states, M -> Size of input alphabet
{
for (int i = 0; i < N && NFAsates[state].transitions[0][i] != -1; i++)
if (closure[NFAsates[state].transitions[0][i]] == 0) {
closure[NFAsates[state].transitions[0][i]] = 1;
epsilonClosure(NFAsates[state].transitions[0][i], closure);
}}

void epsilonClosure(bitset<MAX_NFA_STATES> state,
bitset<MAX_NFA_STATES>& closure)
{
```

```

for (int i = 0; i < N; i++)
if (state[i] == 1)
epsilonClosure(i, closure);
}
// returns a bitset representing the set of states the NFA could be in after moving
// from state X on input symbol A
void NFAMove(int X, int A, bitset<MAX_NFA_STATES>& Y)
{
for (int i = 0; i < N && NFAsates[X].transitions[A][i] != -1; i++)
Y[NFAsates[X].transitions[A][i]] = 1;
}

void NFAMove(bitset<MAX_NFA_STATES> X, int A, bitset<MAX_NFA_STATES>& Y)
{
for (int i = 0; i < N; i++)
if (X[i] == 1)
NFAMove(i, A, Y);
}
int main()
{
int i, j, X, Y, A, T, F, D;
// read in the underlying NFA
ifstream fin("/Users/a/Ios/Compiler Design/Compiler Design/input_nfa.txt");
fin >> N >> M;
NFAsates = new NFAsate[N];
fin >> F;
for (i = 0; i < F; i++) {
fin >> X;
NFA_finalStates.insert(X);
}
fin >> T;
while (T--) {
fin >> X >> A >> Y;
for (i = 0; i < Y; i++) {
fin >> j;
NFAsates[X].transitions[A][i] = j;
}
}
fin.close();
// construct the corresponding DFA
D = 1;
DFAstates.push_back(new DFAstate);
DFAstates[0]->constituentNFAsates[0] = 1;
epsilonClosure(0, DFAstates[0]->constituentNFAsates);
for (j = 0; j < N; j++)

```

```

if (DFAstates[0]->constituentNFAstates[j] == 1 && NFA_finalStates.find(j) !=
NFA_finalStates.end()) {
DFAstates[0]->finalState = true;
DFA_finalStates.push_back(0);
break;
}
incompleteDFAstates.push(0);
while (!incompleteDFAstates.empty()) {
X = incompleteDFAstates.front();
incompleteDFAstates.pop();
for (i = 1; i <= M; i++) {
NFAmove(DFAstates[X]->constituentNFAstates, i,
DFAstates[X]->transitions[i]);
epsilonClosure(DFAstates[X]->transitions[i],
DFAstates[X]->transitions[i]);
for (j = 0; j < D; j++)
if (DFAstates[X]->transitions[i] == DFAstates[j]->constituentNFAstates) {
DFAstates[X]->symbolicTransitions[i] = j;
break;
}
if (j == D) {
DFAstates[X]->symbolicTransitions[i] = D;
DFAstates.push_back(new DFAstate);
DFAstates[D]->constituentNFAstates
= DFAstates[X]->transitions[i];
for (j = 0; j < N; j++)
if (DFAstates[D]->constituentNFAstates[j] == 1
&& NFA_finalStates.find(j) != NFA_finalStates.end()) {
DFAstates[D]->finalState = true;
DFA_finalStates.push_back(D);
break;
}
incompleteDFAstates.push(D);
D++;
}}
}
// write out the corresponding DFA
ofstream fout("/Users/samarthgupta/Ios/Compiler Design/Compiler Design/dfa.txt");
fout << D << " " << M << "\n" << DFA_finalStates.size();
for (vector<int>::iterator it = DFA_finalStates.begin(); it
!= DFA_finalStates.end();
it++)
fout << " " << *it;
fout << "\n";
for (i = 0; i < D; i++) {
for (j = 1; j <= M; j++)

```

```

fout << i << " " << j << " "
<< DFAstates[i]->symbolicTransitions[j] << "\n";
}
fout.close();
return 0;
}

```

- **OUTPUT :**

```

Enter the no of states and no of inputs
3 4
0 # 1 2 1,2
1 0 2 0,2 #
2 # # # #
Enter the initial state and final state
0 2
0      0,1,2    1,2      0,1,2
0,1,2  0,1,2    1,2      0,1,2
1,2    0,1,2    2        0,1,2
2      #        #        #

Process returned 0 (0x0)   execution time : 52.457 s
Press any key to continue.

```

- **FINDINGS AND LEARNINGS :**

We learnt the process of converting an NFA to equivalent DFA and how this can be useful in various aspects of Computer Science. We also came to know that sometimes, it is not easy to convert regular expression to DFA. We can convert regular expression to NFA and then NFA to DFA. NFA can use empty string transition while DFA cannot use empty string transition. NFA is easier to construct while it is more difficult to construct DFA.

## EXPERIMENT 2

- **OBJECTIVE :** Write a program for acceptance of a string by DFA.

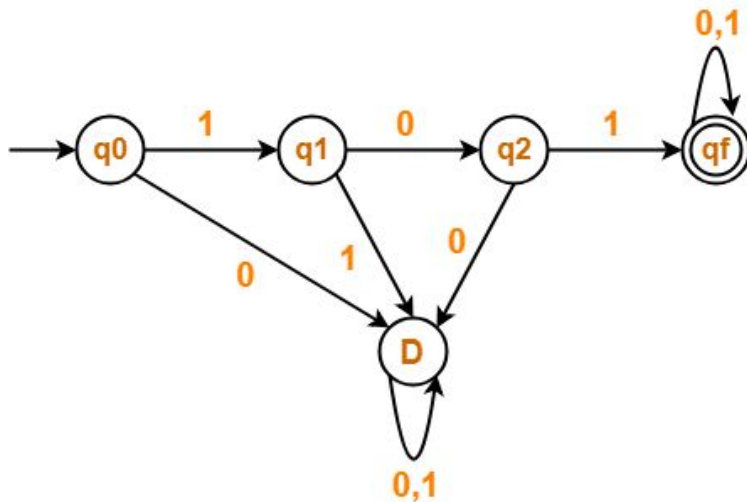
- **THEORY :** Deterministic Finite Automaton (DFA)

In DFA, for each input symbol, one can determine the state to which the machine will move. Hence, it is called Deterministic Automaton. As it has a finite number of states, the machine is called Deterministic Finite Machine or Deterministic Finite Automaton.

A DFA can be represented by a 5-tuple  $(Q, \Sigma, \delta, q_0, F)$  where –

- $Q$  is a finite set of states.
- $\Sigma$  is a finite set of symbols called the alphabet.
- $\delta$  is the transition function where  $\delta: Q \times \Sigma \rightarrow Q$
- $q_0$  is the initial state from where any input is processed ( $q_0 \in Q$ ).
- $F$  is a set of final state/states of  $Q$  ( $F \subseteq Q$ ).

An example is given below:



- **CODE :**

```
#include<iostream>
#include <bits/stdc++.h>
# define msize 500
using namespace std;
int adj[msize][26];
bool is_final[26];
int main()
{
    int n, m;
    cout<<"Enter number of states in DFA: ";
    cin>>n;
    cout<<"\nEnter number of transitions: ";
    cin>>m;
    cout<<"\nEnter Transitions(States are denoted by 0, 1, 2... and symbols are denoted by a,b,.....z)";
    cout<<"\nTransition in the form state1 input_symbol state2\n";
```



```

int i, x, y, j, init;
char sym;
memset(adj, -1, sizeof adj); // initialize all the elements to -1
for(i=0; i<m; i++)
{
    cin>>x;
    cin>>sym;
    while(sym==' '||sym=='\n')
    {cin>>sym;
    }
    cin>>y;
    adj[x][sym-97]=y;
}
cout<<"\nEnter initial state: ";
cin>>init;
int f;
cout<<"\nEnter number of final states: ";
cin>>f;
cout<<"Enter final states: ";
for(i=0; i<f; i++){
    cin>>x;
    is_final[x]=1; }
cout<<"\nEnter number of queries: ";
int q;
cin>>q;
while(q--){
    cout<<"Enter string: ";
    string s;
    cin>>s;
    int cur=init;
    for(i=0; i<s.size(); i++){
        if(adj[cur][s[i]-97]==-1){
            break; }
        cur=adj[cur][s[i]-97]; }
    if(i==s.size() && is_final[cur])
        cout<<"String:"<<" "<<s<<" "<<"is accepted by DFA\n";
    else
        cout<<"String:"<<" "<<s<<" "<<"is not accepted by DFA\n"; } }

```

- **OUTPUT :**

```
Enter number of states in DFA: 3
Enter number of transitions: 4
Enter Transitions(States are denoted by 0, 1, 2... and symbols are denoted by a,b,.....z)
Transition in the form state1 input_symbol state2
0 a 1
1 b 2
2 a 0
0 b 2

Enter initial state: 0

Enter number of final states: 1
Enter final states: 2

Enter number of queries: 4
Enter string: ab
String: ab is accepted by DFA
Enter string: b
String: b is accepted by DFA
Enter string: abb
String: abb is not accepted by DFA
Enter string: aba
String: aba is not accepted by DFA
```

- **FINDINGS AND LEARNINGS :**

In this experiment we learnt the method of the acceptance of a string by a DFA and which strings can be accepted by the DFA and which are not accepted by it and also we saw various examples regarding it. Only those Strings will be accepted by the DFA in which the termination is at one of the final states and this fact is helpful in generating Regular expressions for a DFA and vice versa.

## EXPERIMENT 3

- **OBJECTIVE :** Write a program to count the tokens in a c++ program.
- **THEORY :** Lexemes are said to be a sequence of characters (alphanumeric) in a token. There are some predefined rules for every lexeme to be identified as a valid token. These rules are defined by grammar rules, by means of a pattern. A pattern explains what can be a token, and these patterns are defined by means of regular expressions. In programming language, keywords, constants, identifiers, strings, numbers, operators and punctuations symbols can be considered as tokens.

C tokens are of six types mainly

1. Keywords (eg: int, while),
2. Identifiers (eg: main, total),
3. Constants (eg: 10, 20),
4. Strings (eg: "total", "hello"),
5. Special symbols (eg: (), {}),
6. Operators (eg: +, /, -, \*)

- **CODE :**

```
#include <iostream>
#include <fstream>
using namespace std;
int checkKeyword(char buffer[]){
char keywords[32][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"
};
int i, flag = 0;
for(i = 0; i < 32; ++i){
if(strcmp(keywords[i], buffer) == 0){
flag = 1;
break;
}
}
return flag;
}
bool checkDelimiter(char ch)
{
if (ch == '/' || ch == ';' || ch == ':' || ch == '>' ||
ch == '<' || ch == '(' || ch == ')')
ch == '[' || ch == ']' || ch == '{' || ch == '}' )
```

```

return (true);
return (false);
}
int main() {
ifstream readFile;
readFile.open("/Users/samarthgupta/Ios/Compiler Design/Compiler Design/code.cpp");
int keywords = 0,delimiters = 0,identifiers = 0,operators_count = 0;
int flag = 0;
bool isComment = false;
while(!readFile.eof()){
char out[100];
readFile.getline(out,100);
isComment=false;
for( int i=0; out[i]!='\0'; i++) {
if (out[i] == '/' && out[i + 1] == '/' && flag == 0) {
isComment = true;
break;
} else if (out[i] == '/' && out[i + 1] == '*' && flag == 0) {
flag = 1;
isComment=true;
}
if(flag == 1){
isComment = true;
}
if(out[i] == '*' && out[i+1] == '/' && flag == 1){
flag = 0;
isComment = true;
break;
}
}
if(!isComment){
char ch, buffer[15], operators[] = "+-*/%=&|^";
int j=0,i=0;
for(i=0;out[i]!='\0';i++){
ch=out[i];
for(int s = 0; s<9; s++){
if(ch == operators[s])
{
operators_count++;
//cout<<ch<<" is operator\n";
break;
}
}
}
if(checkDelimiter(ch)){
delimiters++;

```

```

//cout<<ch<<" is delimiter\n";
}
if(isalnum(ch)){
buffer[j++] = ch;
} else if((ch == ' ' || ch == '\n') && (j != 0)){
buffer[j] = '\0';
j = 0;
if(checkKeyword(buffer) == 1){
//cout<<buffer<<" is keyword\n";
keywords++;
} else{
//cout<<buffer<<" is identifier\n";
identifiers++;
}
}
}
if(out[i]=='\0' && (j!=0)){
buffer[j] = '\0';
j = 0;
if(checkKeyword(buffer) == 1){
//cout<<buffer<<" is keyword\n";
keywords++;
}
else{
//cout<<buffer<<" is identifier\n";
identifiers++;
}}}
}
readFile.close();
cout<<"Keywords : "<<keywords<<endl;
cout<<"Identifiers : "<<identifiers<<endl;
cout<<"Operators : "<<operators_count<<endl;
cout<<"Delimiters : "<<delimiters<<endl;
return 0;

```

- **OUTPUT :**

```
Input
int a=10,b=2,c;
c=a+b;
cout<<c;

<Keyword> : 2
<Special Character> : 5
<Identifier> : 7
<Operator> : 5
<Number> :2
Tokens :21
Process returned 0 (0x0)   execution time : 1.540 s
Press any key to continue.
```

- **FINDINGS AND LEARNINGS :**

In this experiment we learnt how to count the number of tokens in a code and also certain things which are not explicitly mentioned. For example the code between the string quotes are counted as one token. Counting the number of tokens in a program code is important as this will be useful for the parser to parse the code and generate the parse tree and then compiling the code.

## EXPERIMENT 4

- **OBJECTIVE :** Write a program to implement a lexical analyzer.
- **THEORY :** Lexical analysis is the first phase of a compiler. It takes the modified source code from language preprocessors that are written in the form of sentences. The lexical analyzer breaks these syntaxes into a series of tokens, by removing any whitespace or comments in the source code. If the lexical analyzer finds a token invalid, it generates an error. The lexical analyzer works closely with the syntax analyzer. It reads character streams from the source code, checks for legal tokens, and passes the data to the syntax analyzer when it demands.



Since the function of the lexical analyzer is to scan the source program and produce a stream of tokens as output, the issues involved in the design of lexical analyzer are:

1. Identifying the tokens of the language for which the lexical analyzer is to be built, and to specify these tokens by using suitable notation, and
2. Constructing a suitable recognizer for these tokens.

- **CODE :**

```
#include <iostream>
#include <fstream>
using namespace std;
int checkKeyword(char buffer[]){
char keywords[32][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"};
};
int i, flag = 0;
for(i = 0; i < 32; ++i){
if(strcmp(keywords[i], buffer) == 0){
flag = 1;
break;
}
}
return flag;
```

```

}
int main() {
ifstream readFile;
readFile.open("/Users/samarthgupta/Ios/Compiler Design/Compiler Design/code.cpp");
int keywords = 0,delimiters = 0,identifiers = 0,operators_count = 0;
// int count=1;
int flag = 0;
bool isComment = false;
while(!readFile.eof()){
char out[100];
readFile.getline(out,100);
isComment=false;
for( int i=0; out[i]!='\0'; i++) {
if (out[i] == '/' && out[i + 1] == '/' && flag == 0) {
isComment = true;
break;
} else if (out[i] == '/' && out[i + 1] == '*' && flag == 0) {
flag = 1;
isComment=true;
}
if(flag == 1){
isComment = true;
}
if(out[i] == '*' && out[i+1] == '/' && flag == 1){
flag = 0;
isComment = true;
break;
}}
if(!isComment){
char ch, buffer[15], operators[] = "+-*/%=&|^";
int j=0,i=0;
for(i=0;out[i]!='\0';i++){
ch=out[i];
if(isalnum(ch)){
buffer[j++] = ch;
} else if((ch == ' ' || ch == '\n') && (j != 0)){
buffer[j] = '\0';
j = 0;
if(checkKeyword(buffer) == 1){
cout<<buffer<<" is keyword\n";
keywords++;
} else{
cout<<buffer<<" is indentifier\n";
identifiers++;
}}}
}

```



```

if(out[i]=='\0' && (j!=0)){
buffer[j] = '\0';
j = 0;
if(checkKeyword(buffer) == 1){
cout<<buffer<<" is keyword\n";
keywords++;
}
else{
cout<<buffer<<" is identifier\n";
identifiers++;
}}}}
readFile.close();
return 0;
}

```

- **OUTPUT :**

```

Input
int a=10,b=2,c;
c=a+b;
cout<<c;

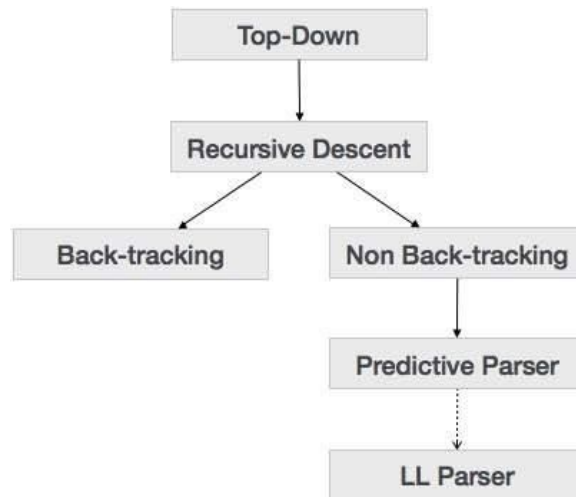
<Keyword>
<Identifier> <Operator> <Number> <Special Character> <Identifier>
<Operator> <Number> <Special Character> <Identifier> <Special Character>
<Identifier> <Operator> <Identifier> <Operator> <Identifier>
<Special Character> <Keyword> <Operator> <Identifier>
<Special Character>
Process returned 0 (0x0)   execution time : 1.044 s
Press any key to continue.

```

- **FINDINGS AND LEARNINGS :** Lexical analyser separates the delimiters, operators, identifiers, keywords, literals and convert them into lexemes and those lexemes are used in the compilation of code. In fact, many compiler tools allow the user to write a lexical analyzer and call it from the generated parser or to make changes to a lexical analyzer provided by the tool.

## EXPERIMENT 5

- **OBJECTIVE :** Write a program to implement recursive descent parser.
- **THEORY :** Recursive descent is a top-down parsing technique that constructs the parse tree from the top and the input is read from left to right. It uses procedures for every terminal and nonterminal entity. This parsing technique recursively parses the input to make a parse tree, which may or may not require backtracking. But the grammar associated with it (if not left factored) cannot avoid back-tracking. A form of recursive-descent parsing that does not require any back-tracking is known as predictive parsing.



Consider the grammar used before for simple arithmetic expressions:

$$\begin{aligned} P &\rightarrow E \\ E &\rightarrow E + T \mid E - T \mid T \\ T &\rightarrow T * S \mid T / S \mid S \\ S &\rightarrow F \wedge S \mid F \\ F &\rightarrow ( E ) \mid \text{char} \end{aligned}$$

The above grammar won't work for recursive descent because of the left recursion in the second and third rules. (The recursive function for E would immediately call E recursively, resulting in an indefinite recursive regression.)

In order to eliminate left recursion, one simple method is to introduce new notation: curly brackets, where {xx} means "zero or more repetitions of xx", and parentheses () used for grouping, along with the or-symbol: |. Because of the many metasymbols, it is a good idea to enclose all terminals in single quotes. Also put a '\$' at the end. The resulting grammar looks as follows:

$$\begin{aligned} P &\rightarrow E \$ \\ E &\rightarrow T \{ (' | '-' ) T \} \\ T &\rightarrow S \{ (' * | '/' ) S \} \\ S &\rightarrow F '^' S \mid F \\ F &\rightarrow '(' E ')' \mid \text{char} \end{aligned}$$

Now the grammar is suitable for creation of a recursive descent parser. Notice that this is a different grammar that describes the same language, that is the same sentences or strings of terminal symbols. A given sentence will have a similar parse tree to one given by the previous grammar, but not necessarily the same parse tree.

- **CODE :**

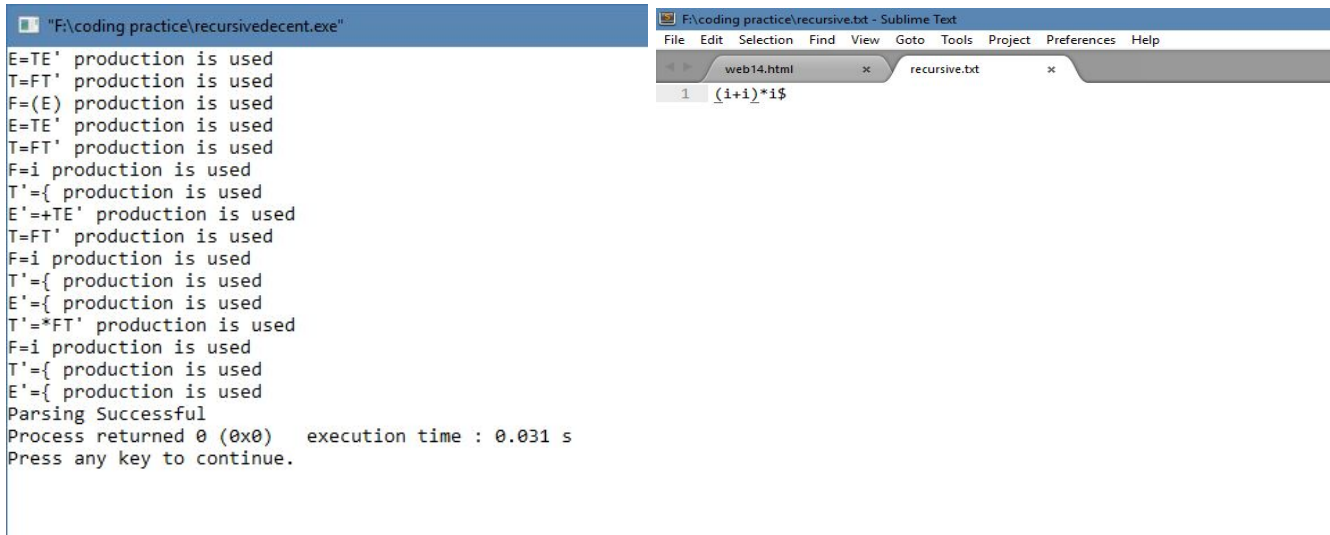
```
#include<bits/stdc++.h>
using namespace std;
fstream file1;
char l;
void T();
void E();
void T1();
void E1();
void F();
void match(char t){
    if(l==t)
        file1.get(l);
    else
        cout<<"Error in parsing";}
void E(){
    cout<<"E=TE' production is used"<<endl;
    T();
    E1();}
void T(){
    cout<<"T=FT' production is used"<<endl;
    F();
    T1();}
void E1(){
    if(l=='+'){
        cout<<"E'=+TE' production is used"<<endl;
        match('+');
        T();
        E1();}
    else{
        cout<<"E'={ production is used"<<endl;
        return ;}}
void F(){
    if(l=='i'){
        cout<<"F=i production is used"<<endl;
        match('i');}
    else if(l=='('){
        cout<<"F=(E) production is used"<<endl;
        match('(');
        E();
        match(')'); }}
void T1(){
    if(l=='*')
    {
```

```

        cout<<"T'=*FT' production is used"<<endl;
        match('*');
        F();
        T1();
    }
else
{
    cout<<"T'={ production is used"<<endl;
    return;
}
}
int main()
{
    file1.open("recursive.txt",ios::in);
    file1.get(l);
    E();
    if(l=='$')
        cout<<"Parsing Successful";
}

```

## ● OUTPUT :



```

F:\coding practice\recursivedecent.exe
E=TE' production is used
T=FT' production is used
F=(E) production is used
E=TE' production is used
T=FT' production is used
F=i production is used
T'={ production is used
E'="+TE' production is used
T=FT' production is used
F=i production is used
T'={ production is used
E'={ production is used
T'=*FT' production is used
F=i production is used
T'={ production is used
E'={ production is used
Parsing Successful
Process returned 0 (0x0)   execution time : 0.031 s
Press any key to continue.

```

```

F:\coding practice\recursive.txt - Sublime Text
File Edit Selection Find View Goto Tools Project Preferences Help
web14.html x recursive.txt x
1 (i+i)*i$

```

- **FINDINGS AND LEARNINGS :** In this experiment we learnt that Recursive descent is a simple parsing algorithm that is very easy to implement. It is a top-down parsing algorithm because it builds the parse tree from the top (the start symbol) down. The main limitation of recursive descent parsing (and all top-down parsing algorithms in general) is that they only work on grammars with certain properties. For example, if a grammar contains any left recursion, recursive descent parsing doesn't work.

## EXPERIMENT 6

- **OBJECTIVE :** Write a program to left factor the given grammar.
- **THEORY :** Left factoring is another useful grammar transformation used in parsing. The general Idea is to replace the production

$A \rightarrow X \mid X Y Z$

After performing Left Factoring we get:

$A \rightarrow X B$

$B \rightarrow Y Z \mid \epsilon$

Algorithm

Let the given grammar:  $A \rightarrow ab1 \mid ab2 \mid ab3$

- 1). We can see that, for every production, there is a common prefix & if we choose any production here, it is not confirmed that we will not need to backtrack.
- 2). It is nondeterministic, because we cannot choose any production and be assured that we will reach our desired string by making the correct parse tree. But if we rewrite the grammar in a way that is deterministic and also leaves us to be flexible enough to make it any string that may be possible without backtracking.... it will be:

$A \rightarrow aA', A' \rightarrow b1 \mid b2 \mid b3$

now if we are asked to make the parse tree for string ab2.... we don't need backtracking. Because we can always choose the correct production when we get A' thus we will generate the correct parse tree. Left factoring is required to eliminate non-determinism of a grammar. Suppose a grammar,

$S \rightarrow abS \mid aSb$

Here, S is deriving the same terminal a in the production rule (two alternative choices for S), which follows non-determinism.

We can rewrite the production to defer the decision of S as-

$S \rightarrow aS'$

- **CODE :**

```
#include<bits/stdc++.h>
using namespace std;
set<string> prod[26], ans [52];
bool used[26], vis[26];
vector<int> fr;
int fc;
int h(char ch){
    return ch-65;}
char rh(int x){
    return x+65;}
bool isTerminal(char ch){
    if(ch=='^'||(ch>=65 && ch<=90))
        return false;
    return true;}
void left_factor(){
```

```

int i, j;
queue<int> Q;
for(i=0;i<26;i++){
    if(!prod[i].empty())
        Q.push(i);}
while(!Q.empty()){
    i=Q.front();
    Q.pop();
    set<string> S=prod[i], cur;
    while(1){
        cur.clear();
        bool b=0;
        set<string>::iterator it=S.begin(), f;
        it=S.begin();
        while(it!=S.end()){
            string x=*it;
            f=it;
            f++;
            if(f==S.end())/*it is last production in this set{
                cur.insert(x);
                break; }
            string match="";
            string y=*f;
            for(j=0;j<x.size() && j<y.size();j++){
                if(x[j]==y[j]) {
                    match+=x[j];
                    b=1; }
            else
                break;}
            if(match=="")//No prefix matching {
                cur.insert(x);
                it=f;
                continue; }
            //Now at-least 2 strings have common prefix for sure
            f++;
            //search all productions with match as common prefix
            while(f!=S.end()) {
                y=*f;
                if(match.size()>y.size())
                    break;
                for(j=0;j<match.size();j++) {
                    if(match[j]!=y[j])
                        break; }
                if(j!=match.size())break;
                f++; }

```

```

//add matchX production to cur i.e i->matchX
int k=fr[fc];
cur.insert(match+rh(k));
//add X->remaining part that did not match from it to f
while(it!=f) {
    if((*it).size()>match.size())
        prod[k].insert((*it).substr(match.size()));
    else//add null production
        prod[k].insert("^");
    it++; }
//add k to queue
Q.push(k);
//increment free terminals counter
fc++; }
S=cur;
if(!b)
    break; }
//set ans[i] to cur
ans[i]=cur; }}
int main(){
    int m;
    cout<<"Denote non-terminals by uppercase alphabets(A-Z) and terminals with lowercase
    alphabets(a-z) and digits from(0-9). Epsilon is denoted by '^'\n";
    cout<<"\nEnter number of productions: ";
    cin>>m;
    cout<<"\nEnter each production in a separate line(format S->aB|cd):\n";
    int i, st;
    string s;
    for(i=0;i<m;i++){
        cin>>s;
        int x=h(s[0]);
        used[x]=1;
        int j=1;
        while(s[j]!='-'||s[j]!='>')
            j++;
        while(j<s.size()){
            string v="";
            while(s[j]!=' ')j++;
            while(j<s.size()&&s[j]!='|') {
                v+=s[j];
                j++; }
            prod[x].insert(v);
            j++; } }
    for(i=0;i<26;i++)
        if(!used[i])fr.push_back(i);

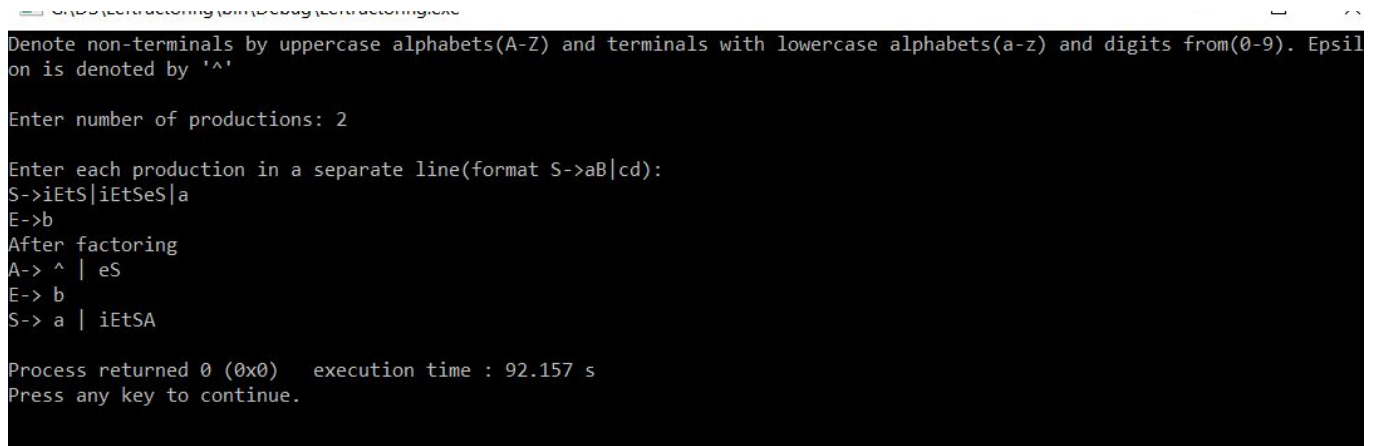
```

```

left_factor();
cout<<"After factoring\n";
for(i=0;i<26;i++) {
    if(ans[i].empty())continue;
    cout<<rh(i)<<"-> ";
    for(set<string>::iterator j=ans[i].begin();j!=ans[i].end();j++) {
        cout<<*j;
        set<string>::iterator k=j;
        k++;
        if(k!=ans[i].end())
            cout<<" | "; }
    cout<<endl; }}

```

## ● OUTPUT :



The screenshot shows a terminal window with the following text:

```

Denote non-terminals by uppercase alphabets(A-Z) and terminals with lowercase alphabets(a-z) and digits from(0-9). Epsilon is denoted by '^'

Enter number of productions: 2

Enter each production in a separate line(format S->aB|cd):
S->iEtS|iEtSe|a
E->b
After factoring
A-> ^ | eS
E-> b
S-> a | iEtSA

Process returned 0 (0x0)   execution time : 92.157 s
Press any key to continue.

```

## ● FINDINGS AND LEARNINGS:

In this experiment we learnt how to remove the left factoring from the grammar and the difference between left recursion and left factoring and how this can be helpful in converting a grammar to suitable form. Difference between Left Factoring and Left Recursion

1. Left recursion: when one or more productions can be reached from themselves with no tokens consumed in-between.
2. Left factoring: a process of transformation, turning the grammar from a left-recursive form to an equivalent non-left-recursive form.



## EXPERIMENT 7

- **OBJECTIVE :** Write a program to convert left recursive grammar to right recursion grammar.
- **THEORY :** The production is left-recursive if the leftmost symbol on the right side is the same as the non terminal on the left side. For example,  $\text{expr} \rightarrow \text{expr} + \text{term}$ .

Algorithm

For each rule which contains a left-recursive option,

$A \rightarrow A\alpha \mid \beta$

introduce a new nonterminal  $A'$  and rewrite the rule as

$A \rightarrow \beta A'$

$A' \rightarrow \alpha A' \mid \epsilon$

Thus the production:

$E \rightarrow E + T \mid T$

is left-recursive with "E" playing the role of "A", "+" T" playing the role of  $\alpha$ , and "T" playing the role of  $\beta$ . Introducing the new nonterminal  $E'$ , the production can be replaced by:

$E \rightarrow T E'$

$E' \rightarrow + T E' \mid \epsilon$

- **CODE:**

```
#include <bits/stdc++.h>
#include <iostream>
#define pb push_back
using namespace std;
set<string> prod[26], ans[52];
bool used[26];
int h(char ch)
{
    return ch-65;
}
char rh(int x)
{
    return x+65;
}
bool isTerminal(char ch)
{
    if(ch=='^' || (ch>=65 && ch<=90))
        return false;
    return true;
}
void convert_to_right(int i, set<string> conv, set<string> S)
{
    for(set<string>::iterator j=S.begin();j!=S.end();j++)
```

```

    {
        string x= *j+rh(i);
        x+="";
        ans[i].insert(x);
    }
    for(set<string>::iterator j=conv.begin();j!=conv.end();j++)
    {
        string x=(*j).substr(1)+rh(i);
        x+="";
        ans[i+26].insert(x);
    }
    ans[i+26].insert("^");

}
int main()
{
    int m;
    cout<<"Denote non-terminals by uppercase alphabets(A-Z) and terminals with lowercase
    alphabets(a-z) and digits from(0-9). Epsilon is denoted by '^'\n";
    cout<<"\nEnter number of productions: ";
    cin>>m;
    cout<<"\nEnter each production in a separate line(format S->aB|cd):\n";
    int i, st;
    string s;
    for(i=0;i<m;i++)
    {
        cin>>s;
        int x=h(s[0]);
        used[x]=1;
        int j=1;
        while(s[j]!='-'||s[j]!='>')
            j++;
        while(j<s.size())
        {
            string v="";
            while(s[j]!=' ')j++;
            while(j<s.size()&& s[j]!='|')
            {
                v+=s[j];
                j++;
            }
            prod[x].insert(v);
            j++;
        }
    }
    for(i=0;i<26;i++)

```

```

{
    if(prod[i].empty())continue;
    set<string> S=prod[i], conv, aux;
    set<string>::iterator it;
    for(it=S.begin();it!=S.end();it++)
    {
        string v=*it;
        int j=h(v[0]);
        if(j>=0&&j<i)
        {
            //modify this production
            set<string>::iterator k=ans[j].begin();
            while(k!=ans[j].end())
            {
                aux.insert(*k+v.substr(1));
                k++;
            }
        }
        else
            aux.insert(v);
    }
    for(it=aux.begin();it!=aux.end();it++)
    {
        string v=*it;
        int j=h(v[0]);
        if(j==i)
            conv.insert(v);
        else ans[i].insert(v);
    }
    if(conv.size()==0)continue;
    S=ans[i];
    ans[i].clear();
    convert_to_right(i, conv, S);
}
cout<<"Productions after removal of left Recursion\n";
for(i=0;i<26;i++)
{
    if(ans[i].empty())continue;
    cout<<rh(i)<<"-> ";
    for(set<string>::iterator j=ans[i].begin();j!=ans[i].end();j++)
    {
        cout<<*j;
        set<string>::iterator k=j;
        k++;
        if(k!=ans[i].end())
            cout<<" | ";
    }
}

```

```

    }
    cout<<endl;
    if(ans[i+26].empty())continue;
    cout<<rh(i)<<"-> ";
    for(set<string>::iterator j=ans[i+26].begin();j!=ans[i+26].end();j++)
    {
        cout<<*j;
        set<string>::iterator k=j;
        k++;
        if(k!=ans[i+26].end())
            cout<<" | "; }
    cout<<endl;
}
}

```

- **OUTPUT:**

```

Denote non-terminals by uppercase alphabets(A-Z) and terminals with lowercase alphabets(a-z) and digits from(0-9). Epsilon is denoted by '^'

Enter number of productions: 1

Enter each production in a separate line(format S->aB|cd):
A->Ac|Aad|bd|^
Productions after removal of left Recursion
A-> ^A' | bdA'
A'-> ^ | adA' | cA'

Process returned 0 (0x0)   execution time : 40.655 s
Press any key to continue.

```

- **FINDINGS AND LEARNINGS:**

In this experiment we learnt how to convert the left recursive grammar to right recursive grammar and benefits of it over the former one. Although the above transformations preserve the language generated by a grammar, they may change the parse trees that witness strings' recognition. With suitable bookkeeping, tree rewriting can recover the originals, but if this step is omitted, the differences may change the semantics of a parse.

## EXPERIMENT 8

- **OBJECTIVE :** Write a program to compute FIRST and FOLLOW.
- **THEORY :** FIRST( If  $a$  is any string of grammar symbols, let  $FIRST(a)$  be the set of terminals that begin the strings derived from  $a$ . If  $a \vdash e$  then  $e$  is also in  $FIRST(a)$ .  
FOLLOW( $A$ ) : for nonterminal  $A$ , to be the set of terminals  $a$  that can appear immediately to the right of  $A$  in some sentential form, that is, the set of terminals  $a$  such that there exists a derivation of the form  $S \vdash aAab$  for some  $a$  and  $b$ . Note that there may, at some time during the derivation, have been symbols between  $A$  and  $a$ , but if so, they derived  $e$  and disappeared. If  $A$  can be the rightmost symbol in some sentential form, then  $\$,$  representing the input right end marker, is in  $FOLLOW(A)$ .

Algorithm:

To compute  $FIRST(X)$  for all grammar symbols  $X$ , apply the following rules until no more terminals or  $e$  can be added to any  $FIRST$  set:

1. If  $X$  is terminal, then  $FIRST(X)$  is  $\{X\}$ .
2. If  $X \rightarrow e$  is a production, then add  $e$  to  $FIRST(X)$ .
3. If  $X$  is nonterminal and  $X \rightarrow Y_1 Y_2 \dots Y_k$  is a production, then place  $a$  in  $FIRST(X)$  if for some  $i$ ,  $a$  is in  $FIRST(Y_i)$ , and  $e$  is in all of  $FIRST(Y_1), \dots, FIRST(Y_{i-1})$ ; that is,  $Y_1, \dots, Y_{i-1} \vdash e$ . If  $e$  is in  $FIRST(Y_j)$  for all  $j = 1, 2, \dots, k$ , then add  $e$  to  $FIRST(X)$ . For example, everything in  $FIRST(Y_1)$  is surely in  $FIRST(X)$ . If  $Y_1$  does not derive  $e$ , then we add nothing more to  $FIRST(X)$ , but if  $Y_1 \vdash e$ , then we add  $FIRST(Y_2)$  and so on.

To compute  $FOLLOW(A)$  for all nonterminals  $A$ , apply the following rules until nothing can be added to any  $FOLLOW$  set:

1. Place  $\$$  in  $FOLLOW(S)$ , where  $S$  is the start symbol and  $\$$  is the input right end marker.
2. If there is a production  $A \vdash aBb$ , then everything in  $FIRST(b)$ , except for  $e$ , is placed in  $FOLLOW(B)$ .
3. If there is a production  $A \vdash aB$ , or a production  $A \vdash aBb$  where  $FIRST(b)$  contains  $e$  (i.e.,  $b \vdash e$ ), then everything in  $FOLLOW(A)$  is in  $FOLLOW(B)$ .

- **CODE :**

```
#include <bits/stdc++.h>
#define pb push_back
using namespace std;
vector<vector<char>> > prod[26];
set<char> first[26], follow[26];
int h(char ch)
{
    return ch-65;
}
char rh(int x)
{
    return x+65;
}
bool isTerminal(char ch)
```

```

{
    if(ch=='^'||(ch>=65&&ch<=90))
        return false;
    return true;
}
int n, m;
bool visited[26];
void findFirst(int x)
{
    int y, i, j;
    visited[x]=1;
    for(i=0;i<prod[x].size();i++)
    {
        vector<char> v=prod[x][i];
        for(j=0;j<v.size();j++)
        {
            char ch=v[j];
            if(isTerminal(ch)||ch=='^')
            {
                first[x].insert(ch);
                break;
            }
            int y=h(ch);
            if(!visited[y])
                findFirst(y);

            set<char> s=first[y];
            set<char>::iterator it;
            for(it=s.begin();it!=s.end();it++)
            {
                if(*it!='^')
                    first[x].insert(*it);
            }
            if(s.find('^')==s.end())break;//no epsilon in first
        }
        if(j==v.size())//add epsilon to first[x] as string may be empty
            first[x].insert('^');
    }
}
void findFollow(char ch)
{
    int i, j, k, x, y;
    set<char>::iterator it;
    x=h(ch);
    visited[x]=1;

```

```

//check each production for occurrence of x
for(i=0;i<26;i++)
{
    if(prod[i].size()==0)
        continue;
    for(j=0;j<prod[i].size();j++)
    {
        vector<char> s=prod[i][j];
        bool b=0;
        for(k=0;k<s.size();k++)
        {
            if(s[k]==ch)
            {
                b=1;
                continue;
            }
        }
        if(b)
        {
            if(isTerminal(s[k]))
            {
                follow[x].insert(s[k]);
                break;
            }
            if(s[k]=='^')
                continue;
            y=h(s[k]);
            for(it=first[y].begin();it!=first[y].end();it++)
            {
                if(*it!='^')
                    follow[x].insert(*it);
            }
            if(first[y].find('^')==first[y].end())break;
        }
    }
    if(b==0)//No occurrence found in this production
        continue;
    if(k==s.size())//Nothing on right of ch...add follow of parent
    {
        if(!visited[i])
            findFollow(i+65);
        for(it=follow[i].begin();it!=follow[i].end();it++)
            follow[x].insert(*it);
    }
}
}
}

```

```

}
int main()
{
    cout<<"Note: Denote non-terminals by uppercase alphabets(A-Z) and terminals with lowercase
    alphabets(a-z) and digits from(0-9). Epsilon is denoted by '^'\n";
    cout<<"\nEnter number of productions: ";
    cin>>m;
    cout<<"\nEnter each production in a separate line(format S->aB):\n";
    int i, j, st;
    string s;
    for(i=0;i<m;i++)
    {
        cin>>s;
        int x=h(s[0]);
        j=1;
        while(s[j]!='-'||s[j]!='>')
            j++;
        vector<char> v;
        while(j<s.size())
        {
            v.pb(s[j]);
            j++;
        }
        prod[x].pb(v);
    }
    char ch;
    cout<<"\nEnter start symbol: ";
    cin>>ch;
    while(ch==' '||ch=='\n')
        cin>>ch;
    st=h(ch);
    findFirst(st);
    for(i=0;i<26;i++)
    {
        if(!visited[i]&& prod[i].size()>0)
            findFirst(i);
    }
    cout<<"First:\n";
    for(i=0;i<26;i++)
    {
        if(first[i].size()==0)
            continue;
        cout<<rh(i)<<": {";
        set<char>::iterator it;
        for(it=first[i].begin();it!=first[i].end();it++)

```



```

    {
        cout<<*it;
        it++;
        if(it!=first[i].end())
            cout<<" ";
        it--;
    }
    cout<<"}\n";
}
for(i=0;i<26;i++)
    visited[i]=0;
follow[st].insert('$');
findFollow(st+65);
for(i=0;i<26;i++)
{
    if(!visited[i])
        findFollow((char)(i+65));
}
cout<<"Follow: \n";
for(i=0;i<26;i++)
{
    if(follow[i].size()==0)
        continue;
    cout<<rh(i)<<": {";
    set<char>::iterator it;
    for(it=follow[i].begin();it!=follow[i].end();it++)
    {
        cout<<*it;
        it++;
        if(it!=follow[i].end())
            cout<<" ";
        it--;
    }
    cout<<"}\n";
}
return 0;
}

```

- **OUTPUT :**

```
Note: Denote non-terminals by uppercase alphabets(A-Z) and terminals with lowercase alphabets(a-z) and digits from(0-9).
Epsilon is denoted by '^'

Enter number of productions: 5

Enter each production in a separate line(format S->aB):
E->TP
P->+TP
T->FQ
Q->*FQ
F->a

Enter start symbol: E
First:
E: {a}
F: {a}
P: {+}
Q: {*}
T: {a}
Follow:
E: {$}
F: {*}
P: {$}
Q: {+}
T: {+}
```

- **FINDINGS AND LEARNINGS :** In this experiment we learnt how to find the first and follow of a grammar and how this can be used to make different parsers such as LL1, SR1, and CLR1.

## EXPERIMENT 9

- **OBJECTIVE :** Write a program to construct LL(1) parsing table.
- **THEORY :** The first L means the input string is processed from left to right. The second L means the derivation will be a leftmost derivation (the leftmost variable is replaced at each step). 1 means that one symbol in the input string is used to help guide the parse.

Algorithm:

For every production A in the grammar:

1. If A can derive a string starting with a (i.e., for all a in FIRST(A) ,

Table [A, a] = A  $\rightarrow \alpha$

2. If A can derive the empty string,  $\epsilon$ , then, for all b that can follow a string derived from A (i.e., for all b in FOLLOW(A) ,

Table [A,b] = A  $\rightarrow \epsilon$

To explain an LL(1) parser workings we will consider the following small LL(1) grammar:

1.  $S \rightarrow F$
2.  $S \rightarrow ( S + F )$
3.  $F \rightarrow a$

and parse the following input: ( a + a )

The parsing table is given by:

	)	(	a	b	\$
S	2	-	1	-	-
F	-	-	3	-	-

- **CODE:**

```
#include<bits/stdc++.h>
using namespace std;
string char2string(char c){
    string s="";
    s=s+c;
    return s;
}
set<string> setunion(set<string> a,set<string> b){
    set<string>::iterator it;
    for(it=b.begin();it!=b.end();it++)
    {
        if(*it=="{")
        {
            continue;
        }
    }
}
```

```

    }
    a.insert(*it);
}
return a;}
set<string> fos(string s, map<char,set<string>> m){
    set<string> result;
    for(int i=0;i<s.length();i++)
    {
        if(!isupper(s[i]))
        {
            result.insert(char2string(s[i]));
            return result;
        }
        else
        {
            result = setunion(result,m[s[i]]);
            if(m[s[i]].find("{}")==m[s[i]].end())
            {
                return result; } } }
    result.insert("{}");
    return result;
}
set<string> firststr(map<char,set<string>> t,string s1){
    char c=s1[0];
    set<string> s;
    string temp="";
    if(islower(c)) {
        s.insert(char2string(c));
        return s;}
    set<string> b=t[c];
    int i=0;
    for(set<string>::iterator it1=b.begin();it1!=b.end();it1++){
        temp=*it1;
        int n=temp.length();
        if(isupper(temp[i])){
            set<string> temp2=firststr(t,char2string(temp[n-1]));
            temp2.clear();
            while(1){
                if(i<n) {
                    set<string> temp1=firststr(t,char2string(temp[i]));
                    set<char>::iterator it;
                    if(temp1.find("{}")!=temp1.end()){
                        i++;
                        s=setunion(s,temp1);
                        temp1.clear();}
                }
            }
        }
    }
}

```

```

        else{
            s=setunion(s,temp1);
            break; }
    }
    else{
        break;}
    if(temp2.find("{}")!=temp2.end()&&i==n-1){
        s.insert("{}"); } } }
else if(temp[i]=='{'){
    s.insert("{}"); }
else{
    s.insert(char2string(temp[i])); }
i=0;}
return s;}
void follow(map<char,set<string>> &result, multimap<char,string> prod, map<char,set<string>>
first_res,char c,map<char,int> &encountered){
    encountered[c]=1;
    multimap<char,string>::iterator mmit;
    for(mmit=prod.begin();mmit!=prod.end();mmit++) {
        if((mmit->second).find(char2string(c))!=(-1)) {
            int p=(mmit->second).find(char2string(c));
            if(p==(mmit->second).length()-1 && (mmit->first)==c)
            {
                continue;
            }
            string ss=(mmit->second).substr(p+1,(mmit->second).length()-p);
            set<string> s =fos(ss,first_res);
            result[c]=setunion(result[c],s);
            if(s.find("{}")!=s.end()) {
                if(encountered[mmit->first]==1 && result[mmit->first].size()==0){
                    continue;}
                follow(result,prod,first_res,mmit->first,encountered);
                result[c]= setunion(result[c],result[mmit->first]); } } } }
int main(){
    int n;
    cout<<"Enter the no of productions\n";
    cin>>n;
    char starting;
    map<char,set<string>> m;
    set<string> s;
    for(int i=0;i<n;i++){
        char p;
        cin>>p;
        if(i==0) {
            starting = p;}

```

```

string s1="";
while(1) {
    cin>>s1;
    if(s1=="-1") {
        break; }
    s.insert(s1); }
s1="";
m.insert(make_pair(p,s));
s.clear(); }
set<string> ss;
multimap<char,string> mm;
map<char,int> en;
for (auto it=m.begin(); it!=m.end(); ++it){
    en.insert(make_pair(it->first,0)); }
for (map<char,set<string>>::iterator it=m.begin(); it!=m.end(); ++it){
    ss=it->second;
    set<string>::iterator it1;
    for(it1=ss.begin();it1!=ss.end();it1++){
        mm.insert(make_pair(it->first,*it1));}}
map<char,set<string>> first_res;
set<string> q;
for (map<char,set<string>>::iterator it=m.begin(); it!=m.end(); ++it){
    q=firststr(m,char2string(it->first));
    first_res.insert(make_pair(it->first,q));
    q.clear();}
set<string> p;
map<char,set<string>> results;
set<string> temporary;
temporary.insert("$");
results.insert(make_pair(starting,temporary));
for(auto it=m.begin();it!=m.end();it++){
    follow(results,mm,first_res,it->first,en);}
set<string> temp;
map<char,map<char,string>> pt_result;
for(auto it=mm.begin();it!=mm.end();it++){
    temp=fos(it->second,first_res);
    for(auto it1=temp.begin();it1!=temp.end();it1++) {
        string temp1=char2string(it->first)+"="+it->second;
        if(*it1==""){
            continue;}
        pt_result[it->first].insert(make_pair((*it1)[0],temp1));}
if(temp.find("{}")!=temp.end()) {
    p=results[it->first];
    for(auto it1=p.begin();it1!=p.end();it1++){
        string temp1=char2string(it->first)+"="+it->second;

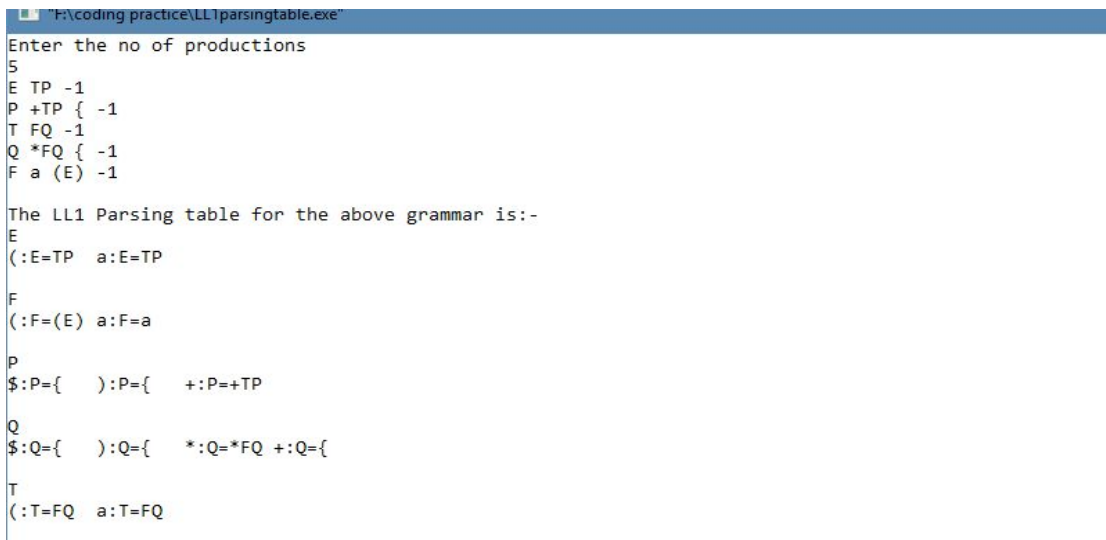
```

```

        pt_result[it->first].insert(make_pair((*it1)[0],temp1)); } } }
map<char,string> z;
cout<<endl;
cout<<"The LL1 Parsing table for the above grammar is:-";
cout<<endl;
for(auto it=pt_result.begin();it!=pt_result.end();it++){
    cout<<it->first<<endl;
    z=it->second;
    for(auto it1=z.begin();it1!=z.end();it1++){
        {
            cout<<it1->first<<":"<<it1->second<<"\t"; }
        cout<<endl;
        cout<<endl; }
    return 0;
}

```

## ● OUTPUT:



```

F:\coding practice\LL1parsingtable.exe
Enter the no of productions
5
E TP -1
P +TP { -1
T FQ -1
Q *FQ { -1
F a (E) -1

The LL1 Parsing table for the above grammar is:-
E
(:E=TP a:E=TP

F
(:F=(E) a:F=a

P
$:P={ } :P={ +:P=+TP

Q
$:Q={ } :Q={ *:Q=*FQ +:Q={

T
(:T=FQ a:T=FQ

```

## ● FINDINGS AND LEARNINGS :

We have used map data structure for inputting grammar. A function of follow, is made which takes the map and the non-terminal and the result map (which stores Follow of each non-terminal) as parameter. The program demands First of string which is implemented using a function which makes use of the First of grammar. The parsing table is generated using First and Follow and the algorithm mentioned above.

## EXPERIMENT 10

- **OBJECTIVE :** Write a program to implement non recursive predictive parsing.
- **THEORY :** Predictive parsing can be performed using a pushdown stack, avoiding recursive calls.
  - a. Initially the stack holds just the start symbol of the grammar.
  - b. At each step a symbol X is popped from the stack:
    - if X is a terminal symbol then it is matched with lookahead and lookahead is advanced,
    - if X is a nonterminal, then using lookahead and a parsing table (implementing the FIRST sets) a production is chosen and its right hand side is pushed onto the stack.
  - c. This process goes on until the stack and the input string become empty. It is useful to have an end\_of\_stack and an end\_of\_input symbols. We denote them both by \$.
- **CODE :**

```
#include <iostream.h>
#include <conio.h>
#include <string.h>
#include <stdio.h>
#include <stdlib.h>
void main()
{
    clrscr();
    int i=0,j=0,k=0,m=0,n=0,o=0,p=0,q=0,r=0,s=0,t=0,u=0,v=0,w=0,x=0,y=0,z=0;
    char str[30],str1[40]="E",temp[20],temp1[20],temp2[20],tt[20],t3[20];
    strcpy(temp1,"\0");
    strcpy(temp2,"\0");
    char t[10];
    char array[6][5][10] = {
        "NT", "<id>", "+", "*", ";", ",",
        "E", "Te", "Error", "Error", "Error",
        "e", "Error", "+Te", "Error", "\0",
        "T", "Vt", "Error", "Error", "Error",
        "t", "Error", "\0", "*Vt", "\0",
        "V", "<id>", "Error", "Error", "Error"
    };
    cout << "\n\tLL(1) PARSER TABLE \n";
    for(i=0;i<6;i++)
    {
        for(j=0;j<5;j++)
        {
            cout.setf(ios::right);
            cout.width(10);
            cout<<array[i][j];
```



```

    }
    cout<<endl;
}
cout << endl;
cout << "\n\tENTER THE STRING :";
gets(str);
if(str[strlen(str)-1] != ';')
{
    cout << "END OF STRING MARKER SHOULD BE ' ';";
    getch();
    exit(1);
}
cout << "\n\tCHECKING VALIDATION OF THE STRING ";
cout << "\n\t" << str1;
i=0;

while(i<strlen(str))
{
    again:
    if(str[i] == ' ' && i<strlen(str))
    {
        cout << "\n\tSPACES IS NOT ALLOWED IN SOURCE STRING ";
        getch();
        exit(1);
    }
    temp[k]=str[i];
    temp[k+1]='\0';
    fl=0;
    again1:
    if(i>=strlen(str))
    {
        getch();
        exit(1);
    }
    for(int l=1;l<=4;l++)
    {
        if(strcmp(temp,array[0][l])==0)
        {
            fl=1;
            m=0,o=0,var=0,o1=0;
            strcpy(temp1,"\0");
            strcpy(temp2,"\0");
            int len=strlen(str1);
            while(m<strlen(str1) && m<strlen(str))
            {

```

```

        if(str1[m]==str[m])
        {
            var=m+1;
            temp2[o1]=str1[m];
            m++;
            o1++;
        }
        else
        {
            if((m+1)<strlen(str1))
            {
                m++;
                temp1[o]=str1[m];
                o++;
            }
            else
                m++;
        }
    }

    temp2[o1] = '\0';
    temp1[o] = '\0';
    t[0] = str1[var];
    t[1] = '\0';
    for(n=1;n<=5;n++)
    {
        if(strcmp(array[n][0],t)==0)
            break;
    }
    strcpy(str1,temp2);
    strcat(str1,array[n][1]);
    strcat(str1,temp1);
    cout << "\n\t" <<str1;
    getch();

    if(strcmp(array[n][1],'\0')==0)
    {
        if(i==(strlen(str)-1))
        {
            int len=strlen(str1);
            str1[len-1]='\0';
            cout << "\n\t"<<str1;
            cout << "\n\n\tENTERED STRING IS  VALID";
            getch();
            exit(1);
        }
    }

```

```

    }
    strcpy(temp1, "\0");
    strcpy(temp2, "\0");
    strcpy(t, "\0");
    goto again1;
}
if(strcmp(array[n][1], "Error")==0)
{
    cout << "\n\tERROR IN YOUR SOURCE STRING";
    getch();
    exit(1);
}
strcpy(tt, "\0");
strcpy(tt, array[n][1]);
strcpy(t3, "\0");
f=0;
for(c=0; c<strlen(tt); c++)
{
    t3[c]=tt[c];
    t3[c+1]='\0';
    if(strcmp(t3, temp)==0)
    {
        f=0;
        break;
    }
    else
        f=1;
}

if(f==0)
{
    strcpy(temp, "\0");
    strcpy(temp1, "\0");
    strcpy(temp2, "\0");
    strcpy(t, "\0");
    i++;
    k=0;
    goto again;
}
else
{
    strcpy(temp1, "\0");
    strcpy(temp2, "\0");
    strcpy(t, "\0");
    goto again1;
}

```

```

    }
  }
}
i++;
k++;
}
if(f1==0)
  cout << "\nENTERED STRING IS INVALID";
else
  cout << "\n\n\tENTERED STRING IS VALID";
getch();
}

```

## • OUTPUT:

```

OUTPUT
*****

LL(1)  PARSER  TABLE

NT      <id>      +      *      ;
E       Te       Error   Error   Error
e       Error     +Te     Error   Error
T       Vt       Error   Error   Error
t       Error     *Vt     Error   Error
V       <id>      Error   Error   Error

ENTER THE STRING :<id>+<id>*<id>;

CHECKING VALIDATION OF THE STRING
E
Te
Vte
<id>te
<id>e
<id>+Te
<id>+Vte
<id>+<id>te
<id>+<id>*Vte
<id>+<id>*<id>te
<id>+<id>*<id>e
<id>+<id>*<id>
ENTERED STRING IS VALID
[ /Code]

```

## • FINDINGS AND LEARNINGS :

A predictive parser attempts to match the nonterminals and the terminals in the stack with the remaining input. Therefore two types of conflicts can occur.

- 1) T-conflict : A terminal appearing on top of the stack does not match the following input token.
- 2) N-conflict : For a nonterminal B on top of the stack and the lookahead token b the entry M[B, b] of the parsing table is empty.

## EXPERIMENT 11

- **OBJECTIVE :** Write a study of an error handler.
- **THEORY :** The tasks of the **Error Handling** process are to detect each error, report it to the user, and then make some recover strategy and implement them to handle error. During this whole process processing time of program should not be slow. An Error is the blank entries in the symbol table.  
Types or Sources of Error – There are two types of error: run-time and compile-time error:
  1. A run-time error is an error which takes place during the execution of a program, and usually happens because of adverse system parameters or invalid input data. The lack of sufficient memory to run an application or a memory conflict with another program and logical error are example of this. Logic errors, occur when executed code does not produce the expected result. Logic errors are best handled by meticulous program debugging.
  2. Compile-time errors rises at compile time, before execution of the program. Syntax error or missing file reference that prevents the program from successfully compiling is the example of this.

### Classification of Compile-time error –

- 1) Lexical : This includes misspellings of identifiers, keywords or operators
- 2) Syntactical : missing semicolon or unbalanced parenthesis
- 3) Semantical : incompatible value assignment or type mismatches between operator and operand
- 4) Logical : code not reachable, infinite loop.

**Finding error or reporting an error –** Viable-prefix is the property of a parser which allows early detection of syntax errors.

- Goal: detection of an error as soon as possible without further consuming unnecessary input
- How: detect an error as soon as the prefix of the input does not match a prefix of any string in the language.

Example: for(;), this will report an error as for have two semicolons inside braces

### Error Recovery

The basic requirement for the compiler is to simply stop and issue a message, and cease compilation. There are some common recovery methods that are follows.

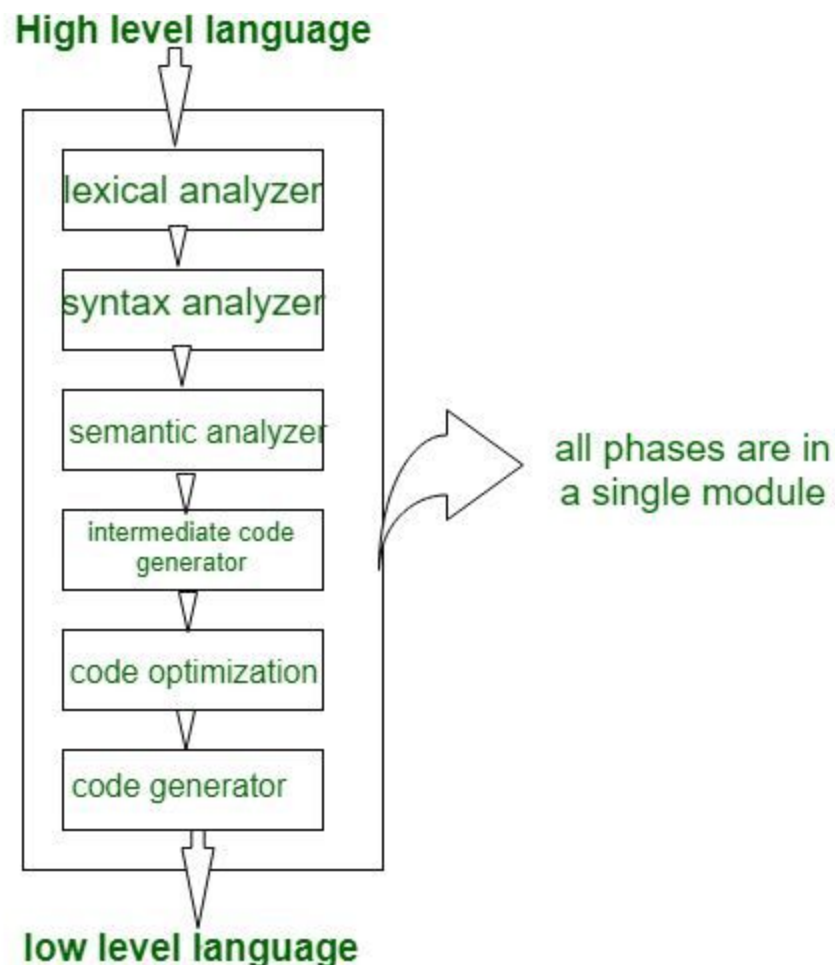
- 1) **Panic mode recovery:** This is the easiest way of error-recovery and also, it prevents the parser from developing infinite loops while recovering error. The parser discards the input symbol one at a time until one of the designated (like end, semicolon) set of synchronizing tokens (are typically the statement or expression terminators) is found. This is adequate when the presence of multiple errors in same statement is rare. Example: Consider the erroneous expression-  $(1 + + 2) + 3$ .
- 2) **Phase level recovery:** Perform local correction on the input to repair the error. But error correction is difficult in this strategy.

- 3) **Error productions:** Some common errors are known to the compiler designers that may occur in the code. Augmented grammars can also be used, as productions that generate erroneous constructs when these errors are encountered. Example: write  $5x$  instead of  $5*x$
- 4) **Global correction:** Its aim is to make as few changes as possible while converting an incorrect input string to a valid string. This strategy is costly to implement.

- **FINDINGS AND LEARNINGS :** In this experiment we learnt about the different kinds of errors that can occur in a compiler or during the design of a compiler. We also learnt how an error handler works and what are the different error recovery methods using which we can recover from any error.

## EXPERIMENT 12

- **OBJECTIVE :** Write a study of a one pass compiler.
- **THEORY :** A Compiler pass refers to the traversal of a compiler through the entire program. Compiler passes are two types: Single Pass Compiler, and Two Pass Compiler.  
**Single Pass Compiler:** If we combine or group all the phases of compiler design in a single module known as single pass compiler.
  - 1) A one pass/single pass compiler is that type of compiler that passes through the part of each compilation unit exactly once.
  - 2) Single pass compiler is faster and smaller than the multi pass compiler.
  - 3) Single pass compiler is one that processes the input *exactly once*, so going directly from lexical analysis to code generator, and then going back for the next read.
  - 4) when the line source is processed, it is scanned and the token is extracted.
  - 5) Then the syntax of each line is analyzed and the tree structure is build. After the semantic part, the code is generated.
  - 6) The same process is repeated for each line of code until the entire program is compiled.



**Disadvantages:**

1. We can not optimize very well due to the context of expressions are limited.
2. As we can't backup and process, it again so grammar should be limited or simplified.
3. Command interpreters such as *bash/sh/tcsh* can be considered as Single pass compiler, but they also execute entry as soon as they are processed
4. As a disadvantage of a single pass compiler is that it is less efficient in comparison with a multipass compiler.

- **FINDINGS AND LEARNINGS :** We learnt about passes in a compiler. We also learnt about the one pass compiler, its architecture, working , advantages and disadvantages.