DELHI TECHNOLOGICAL UNIVERSITY

CO302: Compiler Design

Practical File

Submitted by: **Sayan Chatterjee**
2K19/CO/348

Submitted to: **Dr. Pawan Singh Mehra**
Assistant Professor
Department of Computer Science

INDEX

Experiment – 1

Aim:

Write a program to convert Non-Deterministic Finite Automata(NFA) to Deterministic Finite Automata (DFA)

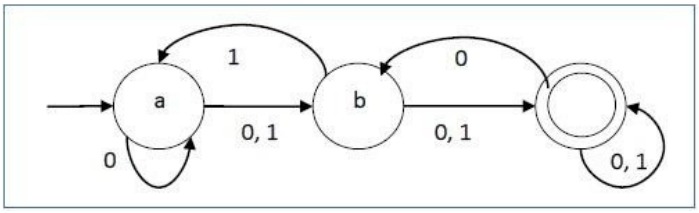
Theory:

Non Deterministic Finite Automata (NDFA)

In NDFA, for a particular input symbol, the machine can move to any combination of the states in the machine. In other words, the exact state to which the machine moves cannot be determined. Hence, it is called Non-deterministic Automaton. As it has finite number of states, the machine is called Non-deterministic Finite Machine or Non-deterministic Finite Automaton.

An NDFA can be represented by a 5-tuple $(Q, \sum, \delta, q_0, F)$ where,

- Q is a finite set of states.
- $\sum$ is a finite set of symbols called the alphabets.
- δ is the transition function where $\delta: Q \times \sum \rightarrow 2^Q$
- 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 ($F \subseteq Q$).

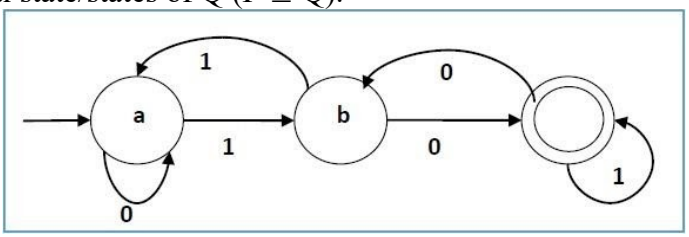


Example of NDFA

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 $(Q, \sum, \delta, q_0, F)$ where,

- Q is a finite set of states.
- $\sum$ is a finite set of symbols called the alphabet.
- δ is the transition function where $\delta: Q \times \sum \rightarrow Q$
- 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 ($F \subseteq Q$).



Example of DFA

Conversion from NFA to DFA

Suppose there is an NFA $N < Q, \sum, q0, \delta, F >$ which recognizes a language L. Then the DFA $D < Q', \sum, q0, \delta', F' >$ can be constructed for language L as:

1. Initially $Q' = \phi$.
2. Add q0 to Q'.
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'.
4. Final state of DFA will be all states with contain F (final states of NFA)

Code:

```
#include<iostream>
#include<cmath>
#include<map>
#include<set>
#include<vector>
#include<queue>
#include<iomanip>
#include<string>

using namespace std;

map<vector<int>,vector<int>> nfa_tran;
map<vector<int>,vector<vector<int>>> dfa_tran;
map<vector<int>,bool> vis;

int main()
{
    int nst, fin, in;
    int f[10];
    int inp[2] = {0,1};

    cout<<"Using the 0 based indexing\n";

    cout<<"\nEnter the number of states : ";
    cin>>nst;

    cout<<"\nEnter number of final states : ";
    cin>>fin;

    cout<<"\nEnter final states : ";
    for(int i=0;i<fin;i++){
        cin>>f[i];
    }
    cout<<"\nEnter initial state : ";
    cin>>in;
```

```
int p,q,r,ntran;
cout<<"\nEnter number of transition rules in NFA : ";
cin>>ntran;
cout<<"\n\nDefine transition rules in order : initial state input symbol final state\n : ";
for(int i=0; i<ntran; i++){
    cin>>p>>q>>r;
    nfa_tran[{p,q}].push_back(r);
}

queue<vector<int>> que;
que.push({in});
vis[{in}] = true;
while(!que.empty()){
   vector<int> curr = que.front();
   que.pop();
   vector<vector<int>> dfaout(int(sizeof(inp))/sizeof(int));
   for(int i:inp){
      set<int> out;
      for(int j:curr){
         if(nfa_tran.count({j,i})){
            for(int k:nfa_tran[{j,i}])
               out.insert(k);
         }
      }
      vector<int> outp(out.begin(),out.end());
      if(!vis[outp]){
         que.push(outp);
         vis[outp] = true;
      }
      dfaout[i] = outp;
   }
   dfa_tran[curr] = dfaout;
}
cout<<"Transition function of DFA is as follows\n";
cout<<setw(31)<<left<<"State";
for(int sym:inp)cout<<setw(31)<<left<<sym;
cout<<endl;
for(auto p:dfa_tran){
   cout<<"[";
   string input;
   for(int i:p.first){
      input+='q'; input+=to_string(i); input+= ",";
   }
   input+=']';
   cout<<setw(30)<<left<<input;

   for(auto arr:p.second){
      string output;
      cout<<"[";
```

```
            for(int j:arr){
                output+="q"; output+= to_string(j); output+= ",";
            }
            output += "]";
            cout<<setw(30)<<left<<output;
        }
        cout<<endl;
    }
    return 0;
}
```

Output:

```
Using the 0 based indexing

Enter the number of states : 3

Enter number of final states : 1

Enter final states : 2

Enter initial state : 0

Enter number of transition rules in NFA : 9


Define transition rules in order : initial state input symbol final state
 : 0 0 0
0 1 1
0 1 2
1 0 1
1 0 2
1 1 2
2 0 1
2 0 0
2 1 1
Transition function of DFA is as follows
State                          0                              1
[q0,]                          [q0,]                          [q1,q2,]
[q0,q1,q2,]                    [q0,q1,q2,]                    [q1,q2,]
[q1,q2,]                       [q0,q1,q2,]                    [q1,q2,]
```

Experiment – 2

Aim:

Write a program for acceptance of string by DFA.

Theory:

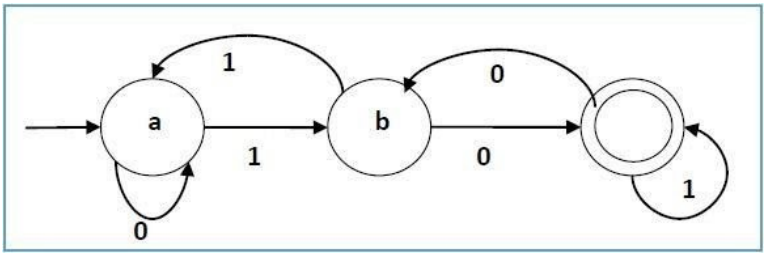
Deterministic Finite Automaton (DFA)

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, ∑, δ, q0, F) where −

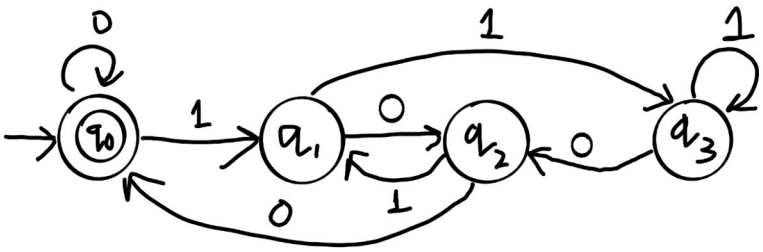
- Q is a finite set of states.
- ∑ is a finite set of symbols called the alphabet.
- δ is the transition function where δ: Q × ∑ → Q
- q0 is the initial state from where any input is processed (q0 ∈ Q).
- F is a set of final state/states of Q (F ⊆ Q).

An example is given below:



Implementation

For the purpose of this experiment, a DFA that accepts all strings divisible by 4 has been considered. The input for the program shall be in the form of a binary string (string containing only 0s and 1s), and the program shall determine whether the input string is accepted by the DFA or not. DFA accepting all strings divisible by 4 –



Initial State: q0, Final State: q0

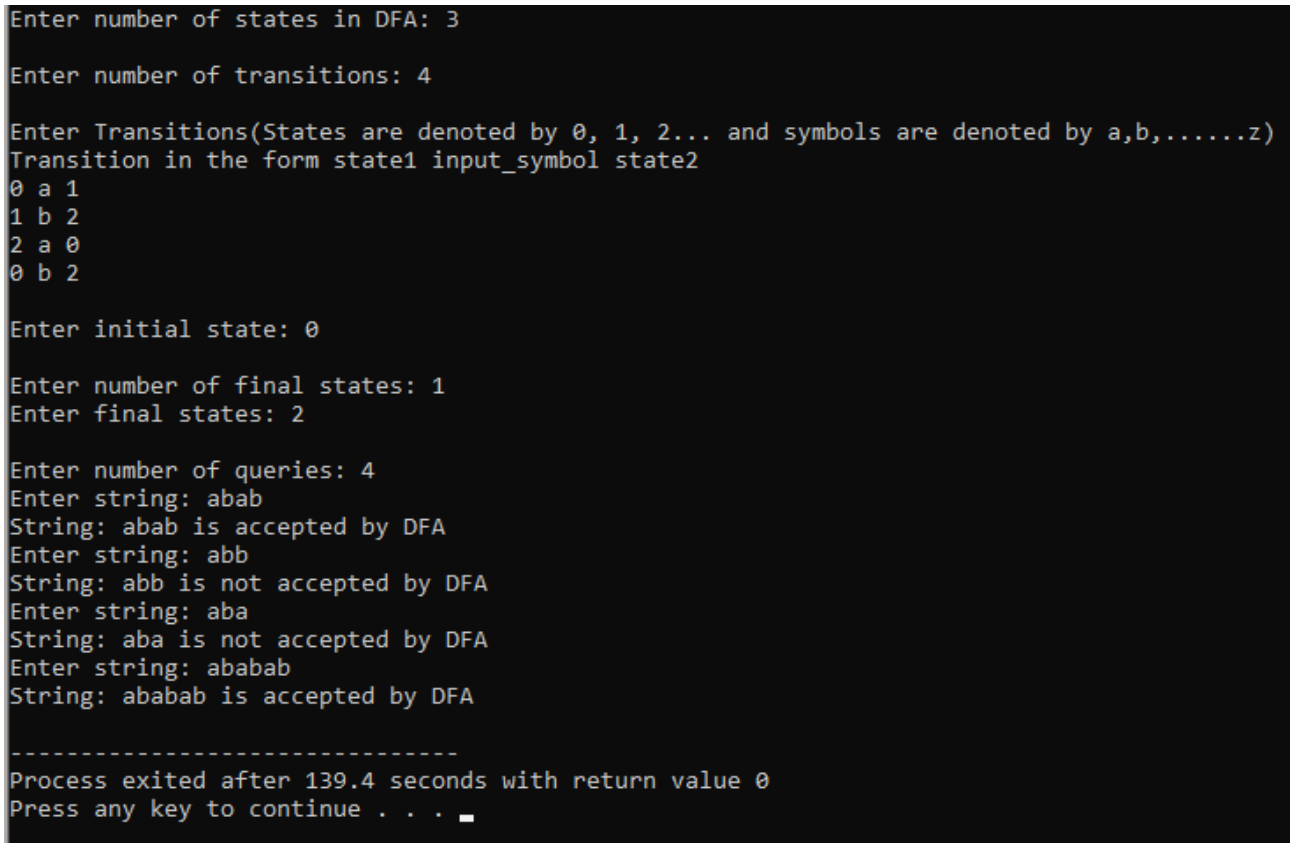
Code:

```
#include<iostream>
#include <bits/stdc++.h>
#define msize 500
using namespace std;

int main()
{
   int adj[msize][26];
   bool is_final[26];
   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:



Experiment – 3

Aim:

Write a program to find different tokens in a program.

Theory:

The initial phase of the compiler, often known as the scanner, is **lexical analysis**. It turns the supplied programme into a Token sequence.

Token

A token is a pair consisting of a token name and an optional attribute value. The token name is and abstract symbol representing a king of lexical unit, e.g., a particular keyword, or sequence of input characters denoting an identifier. The token names are the input symbols that the parser processes. Tokens in C are of mainly six types:

i. Keywords -> int, while, etc
ii. Identifiers -> main, total, etc
iii. Constants -> 10, 20, etc
iv. Strings -> “total”, “hello”, etc
v. Special characters -> (,), {, }, etc
vi. Operators -> +, /, -, *), etc

The program implemented below counts the number of various types of tokens found in C from an input text file.

Code:

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

using namespace std;

set<string> keywords = {"if", "else", "float", "int", "while", "case", "switch", "break",
                        "catch", "const", "do", "double", "false", "true", "goto", "long", "inline",
                        "new", "private", "protected", "public", "return", "static", "short",
                        "struct", "this", "union", "void", "volatile", "typedef", "unsigned",
                        "volatile", "cout", "cin", "printf", "getline", "scanf", "get", "ifstream", "ofstream"};
set<string> op = {"+", "=", "-", "*", "/", "!", "<", ">", "%", "&", "|", "~", "^"};
set<string> delimiters = {";", "{", "}", "(", ")", ",", ":", "[", "]"};

char type[1000];

void parse(string &s) {
   string token;
   int k = 0;
   for (auto &c: s) {
      if ((c >= 'a' and c <= 'z') or (c >= 'A' and c <= 'Z') or (c >= '0' and c <= '9')) token += c;
      else {
         if (!token.empty()) {
            if (keywords.count(token)) type[k++] = 'K';
```

```
            else type[k++] = isdigit(token[0]) ? 'N' : 'I';
        }
        token.clear();
        if (c == ' ') continue;
        if (op.count(string(1, c))) type[k++] = 'O';
        else if (delimiters.count(string(1, c))) type[k++] = 'S';
      }
   }
   map<char, int> cnt;
   for (int x = 0; x < s.length(); x++) {
      cnt[type[x]]++;
      if (type[x] == 'O' or type[x] == 'N') {
         while (x + 1 < s.length() and type[x + 1] == type[x]) x++;
      }
   }

   cout << "No. of Keywords : " << cnt['K'] << "\n";
   cout << "No. of Special Characters :" << cnt['S'] << "\n";
   cout << "No. of Identifiers : " << cnt['I'] << "\n";
   cout << "No. of Operators : " << cnt['O'] << "\n";
   cout << "No. of Numbers : " << cnt['N'] << "\n";
   cout << "No. of Total No. of Tokens : " << (cnt['K'] + cnt['S'] + cnt['I'] + cnt['O'] + cnt['N']);
}
int main() {
   ifstream t("input.txt");
   stringstream ss;
   ss << t.rdbuf();
   string s = ss.str();
   cout << "Program:\n";
   cout << s << "\n";
   for (int i = 0; i < 100; i++)cout << '.';
   cout << "\n";
   parse(s);
   return 0;
}
```

Output:

```
E:\Doc\Sem VI\CD\lab>exp3.exe < input.txt
Program:
#include<iostream>
using namespace std;
int main(){
cout<<"Hello Compiler Design\n";
int a = 1;
float b = 2.5;
double c = a + b;
cout<<c;
}
....................................................................................................
No. of Keywords : 6
No. of Special Characters :10
No. of Identifiers : 16
No. of Operators : 8
No. of Numbers : 2
No. of Total No. of Tokens : 42
E:\Doc\Sem VI\CD\lab>
```

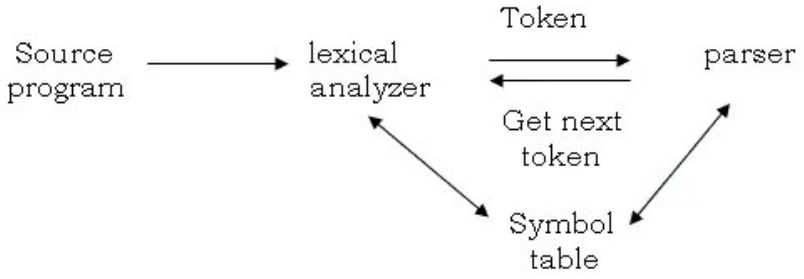
Experiment – 4

Aim:

Write a program to implement Lexical Analyser.

Theory:

Compiler is responsible for converting high level language in machine language. There are several phases involved in this and lexical analysis is the first phase.
Lexical analyser reads the characters from source code and convert it into tokens.



Different tokens or lexemes are:

- Keywords
- Identifiers
- Operators
- Constants

Code:

```
#include <bits/stdc++.h>
using namespace std;
void IO(string name) {
   freopen((name + ".txt").c_str(), "r", stdin);
}
set<string> keywords, op, delimiters;
bool isConst(string &s) {
   for (auto &c: s) {
      if (c >= '0' and c <= '9') continue;
      return false;
   }
   return true;
}
int main() {
   IO("inputProgramFile");
   keywords = {"if", "else", "float", "int", "while", "case", "switch", "break",
            "catch", "const", "do", "double", "false", "true", "goto", "long", "inline",
            "new", "private", "protected", "public", "return", "static", "short",
            "struct", "this", "union", "void", "volatile", "typedef", "unsigned", "namespace",
            "volatile", "cout", "cin", "printf", "getline", "scanf", "get", "ifstream", "ofstream",
"using"};
   op = {"+", "=", "-", "*", "/", "!", "<", ">", "%", "&", "|", "~", "^"};
   delimiters = {";", "{", "}", "(", ")", ",", ":", "[", "]"};
   map<string, string> lexicalAnalyser;
   while (!cin.eof()) {
```

```
        string s;
        cin >> s;
        while (!s.empty() and delimiters.count(string(1, s.back()))) s.pop_back();
        if (s.empty()) continue;
        if (isConst(s)) {
            lexicalAnalyser[s] = "Constant";
            continue;
        }
        bool is_op = true;
        for (auto &c: s) is_op &= op.count(string(1, c));
        string ans;
        if (keywords.count(s)) ans = "Keyword";
        else if (is_op) ans = "Operator";
        else ans = "Identifier";
        lexicalAnalyser[s] = ans;
    }
    for (auto &[a, b]: lexicalAnalyser) cout << a << " is " << b << endl;
    return 0;
}
```

Output:

inputProgramFile.txt

```
1
2   using namespace std;
3   int main(){
4       int a = 5, b = 10;
5       int c = b - a;
6       return 0;
7   }
```

C:\Users\DELL\Downloads\lexanalyze.exe

```
- is Operator
0 is Constant
10 is Constant
5 is Constant
= is Operator
a is Identifier
b is Identifier
c is Identifier
int is Keyword
main is Identifier
namespace is Keyword
return is Keyword
std is Identifier
using is Keyword

--------------------------------
Process exited after 10.54 seconds with return value 0
Press any key to continue . . .
```

Experiment – 5

Aim:

Write a program to implement a Recursive descent parser

Theory:

It is a kind of Top-Down Parser. A top-down parser builds the parse tree from the top to down, starting with the start non-terminal. A *Predictive Parser* is a special case of Recursive Descent Parser, where no Back Tracking is required. By carefully writing a grammar means eliminating left recursion and left factoring from it, the resulting grammar will be a grammar that can be parsed by a recursive descent parser.

Code:

```
#include <bits/stdc++.h>
using namespace std;
int i, error;
string inp;
struct RDP {
   void T() {
      F();
      Tprime();
   }
   void E() {
      T();
      Eprime();
   }
   void Eprime() {
      if (inp[i] == '+') {
         i++;
         T();
         Eprime();
      }
   }
   void F() {
      if (isalnum(inp[i]))i++;
      else if (inp[i] == '(') {
         i++;
         E();
         if (inp[i] == ')')
            i++;

         else error = 1;
      } else error = 1;
   }
   void Tprime() {
      if (inp[i] == '*') {
         i++;
         F(); Tprime();
      }
```

```
    }
} rdp;

int main() {
    cout << "Enter input string:\n";   cin >> inp;
    rdp.E();
    if (i == (int) inp.length() and !error) cout << "Accepted\n";
    else cout << "Rejected\n";
    return 0;
}
```

Input Grammar:

E -> E+T | T
T -> T*F | F
F -> (E) | id

After removing Left recursion:

E -> TE'
E' -> +TE' | ε
T -> FT'
T' -> *FT' | ε
F -> (E) | id

Output:

```
Enter input string:
a***a
Rejected
```

```
Enter input string:
a+(a*a)
Accepted
```

Experiment – 6

Aim:

Write a program to Left Factor the given grammar.

Theory:

If RHS of more than one production starts with the same symbol, then such a grammar is called as Grammar With Common Prefixes.
E.g.:

$$A \rightarrow \alpha\beta_1 / \alpha\beta_2 / \alpha\beta_3$$

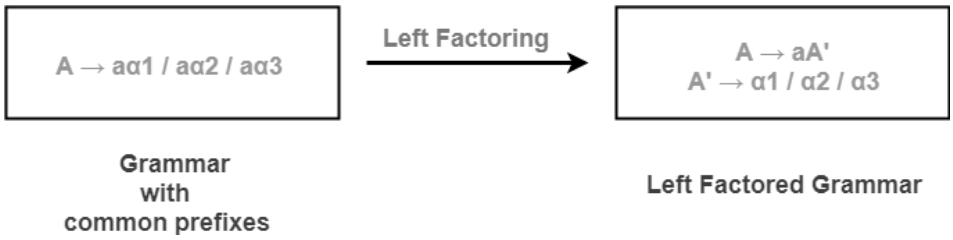
(Grammar with common prefixes)

This kind of grammar creates a problematic situation for Top down parsers.

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.
- The common prefix may be a terminal or a non-terminal or a combination of both.
- Rest of the derivation is added by new productions.

The grammar obtained after the process of left factoring is called as Left Factored Grammar.



Code:

```
#include <bits/stdc++.h>
using namespace std;
#define sz(v) (int) v.size()
const int N = 1e3 + 5;
string null;
struct Production {
   string lhs;
   vector<string> rhs;
} grammar[N];
vector<Production> leftFactoredGrammar;
void leftFactor(int i) {
   int j = 0;
   while (1) {
      bool can = true;
      vector<string> v = grammar[i].rhs;
      for (int x = 0; x < sz(v); x++) {
```

```
            if (sz(v[x]) == j or v[x][j] != v[0][j]) {
                can = false;
                break;
            }
        }
        if (!can) break;
        ++j;
    }
    if (!j) {
        leftFactoredGrammar.push_back(grammar[i]);
        return;
    }
    string nlhs = grammar[i].lhs;
    nlhs += "1";
    string nrhs = grammar[i].rhs[0].substr(0, j) + nlhs;
    leftFactoredGrammar.push_back({grammar[i].lhs, {nrhs}});
    vector<string> v;
    for (auto &s: grammar[i].rhs) {
        string cs = s.substr(j);
        if (cs.empty()) cs = null;
        v.push_back(cs);
    }
    leftFactoredGrammar.push_back({nlhs, v});
}
int main() {
    null = "∈";
    int n;
    cout << "Enter number of productions:\n";
    cin >> n;
    for (int x = 0; x < n; x++) {
        cout << "Enter LHS:\n";
        cin >> grammar[x].lhs;
        int y;
        cout << "Enter the number of rules in RHS:\n";
        cin >> y;
        cout << "Enter RHS (each rule separated by a space)\n";
        for (int z = 0; z < y; z++) {
            string rhs;
            cin >> rhs;
            grammar[x].rhs.push_back(rhs);
        }
    }
    for (int x = 0; x < n; x++) leftFactor(x);
    cout << "Left factored grammar:\n";
    for (auto &x: leftFactoredGrammar) {
        cout << x.lhs << " -> ";
        for (auto &s: x.rhs) {
            cout << s;
            if (s != x.rhs.back()) cout << " | ";
            else cout << '\n';
```

```
        }
    }
    return 0;
}
```

Input Grammar:

$$S \rightarrow aAd / aB$$

$$A \rightarrow a / ab$$

$$B \rightarrow ccd / ddc$$

Output:

```
Enter number of productions:
3
Enter LHS:
S
Enter the number of rules in RHS:
2
Enter RHS (each rule separated by a space)
aAd aB
Enter LHS:
A
Enter the number of rules in RHS:
2
Enter RHS (each rule separated by a space)
a ab
Enter LHS:
B
Enter the number of rules in RHS:
2
Enter RHS (each rule separated by a space)
ccd ddc
Left factored grammar:
S -> aS1
S1 -> Ad | B
A -> aA1
A1 -> ∈ | b
B -> ccd | ddc


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

Experiment 7

Aim: Write a program to convert Left Recursive grammar to Right Recursive grammar.

Theory:

Left Recursion:

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

Elimination of Left Recursion:

If we have the following left recursive production:

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

(where β does not begin with A)

Then we can eliminate the left recursion by replacing the production with:

$$A \rightarrow \beta A'$$

$$A' \rightarrow \alpha A' \mid \in$$

Code:

```
#include <bits/stdc++.h>
using namespace std;
#define sz(v) (int) v.size()
const int N = 1e3 + 5;

struct Production {
   char lhs;
   vector<string> rhs;
} grammar[N];

int n;
string null;
vector<pair<string, vector<string>>> rightRecursiveGrammar;

void removeLeftRecursion(int i) {
   vector<string> rem, withLHS;
   for (auto &rule: grammar[i].rhs) {
      if (rule[0] == grammar[i].lhs) {
         if (sz(rule) > 1) withLHS.push_back(rule.substr(1));
      } else rem.push_back(rule);
   }
   string nLHS;
```

```
    nLHS += grammar[i].lhs;
    if (rem.empty()) rightRecursiveGrammar.emplace_back(nLHS, grammar[i].rhs);
    else {
        nLHS += '1';
        for (auto &s: rem) s += nLHS;
        rightRecursiveGrammar.emplace_back(nLHS.substr(0, 1), rem);
        for (auto &s: withLHS) s += nLHS;
        withLHS.push_back(null);
        rightRecursiveGrammar.emplace_back(nLHS, withLHS);
    }
}

int main() {
    null = "∈";

    cout << "Enter number of productions:\n";
    cin >> n;

    for (int x = 0; x < n; x++) {
        cout << "Enter LHS:\n";
        cin >> grammar[x].lhs;
        int y;
        cout << "Enter the number of rules in RHS:\n";
        cin >> y;
        cout << "Enter RHS (each rule separated by a space)\n";
        for (int z = 0; z < y; z++) {
            string rhs;
            cin >> rhs;
            grammar[x].rhs.push_back(rhs);
        }
        removeLeftRecursion(x);
    }

    cout << "Right Recursive Grammar\n";
    for (auto &[lhs, rhs]: rightRecursiveGrammar) {
        cout << lhs << " -> ";
        for (auto &s: rhs) {
            cout << s;
            if (s != rhs.back()) cout << " | ";
            else cout << '\n';
        }
    }
    return 0;
}
```

Input Grammar:

$$A \rightarrow Ad / Ae / aB / ac$$

$$B \rightarrow Be / b$$

Output:

```
Enter number of productions:
2
Enter LHS:
A
Enter the number of rules in RHS:
4
Enter RHS (each rule separated by a space)
Ad Ae aB ac
Enter LHS:
B
Enter the number of rules in RHS:
2
Enter RHS (each rule separated by a space)
Be b
Right Recursive Grammar
A -> aBA1 | acA1
A1 -> dA1 | eA1 | ∈
B -> bB1
B1 -> eB1 | ∈
```

Experiment 8

Aim: Write a program to compute First and Follow.

Theory:

First(α) is a set of terminal symbols that begin in strings derived from α.

Follow(α) is a set of terminal symbols that appear immediately to the right of α.

Computation of FIRST

FIRST (α) is defined as the collection of terminal symbols which are the first letters of strings derived from α.

FIRST (α) = {α |α $\rightarrow*$ αβ for some string β }

If X is Grammar Symbol, then First (X) will be −

- If X is a terminal symbol, then FIRST(X) = {X}
- If X → ε, then FIRST(X) = {ε}
- If X is non-terminal & X → a α, then FIRST (X) = {a}
- If X → Y_1, Y_2, Y_3, then FIRST (X) will be

(a) If Y is terminal, then

FIRST (X) = FIRST (Y_1, Y_2, Y_3) = {Y_1}

(b) If Y_1 is Non-terminal and

If Y_1 does not derive to an empty string i.e., If FIRST (Y_1) does not contain ε then, FIRST (X) = FIRST (Y_1, Y_2, Y_3) = FIRST(Y_1)

(c) If FIRST (Y_1) contains ε, then.

FIRST (X) = FIRST (Y_1, Y_2, Y_3) = FIRST(Y_1) − {ε} ∪ FIRST(Y_2, Y_3)

Similarly, FIRST (Y_2, Y_3) = {Y_2}, If Y_2 is terminal otherwise if Y_2 is Non-terminal then

- FIRST (Y_2, Y_3) = FIRST (Y_2), if FIRST (Y_2) does not contain ε.
- If FIRST (Y_2) contain ε, then
- FIRST (Y_2, Y_3) = FIRST (Y_2) − {ε} ∪ FIRST (Y_3)

Similarly, this method will be repeated for further Grammar symbols, i.e., for Y_4, Y_5, Y_6 … . Y_K.

Computation of FOLLOW

Follow (A) is defined as the collection of terminal symbols that occur directly to the right of A.

$$FOLLOW(A) = \{a | S \Rightarrow^* \alpha A a \beta \text{ where } \alpha, \beta \text{ can be any strings}\}$$

Rules to find FOLLOW

- If S is the start symbol, FOLLOW (S) ={$}
- If production is of form $A \rightarrow \alpha B \beta$, $\beta \neq \varepsilon$.

(a) If FIRST (β) does not contain ε then, FOLLOW (B) = {FIRST (β)}

Or

(b) If FIRST (β) contains ε (i. e. , $\beta \Rightarrow^* \varepsilon$), then

FOLLOW (B) = FIRST (β) − {ε} ∪ FOLLOW (A)

∵ when β derives ε, then terminal after A will follow B.

- If production is of form $A \rightarrow \alpha B$, then Follow (B) ={FOLLOW (A)}.

Code:

```
#include <bits/stdc++.h>
using namespace std;
#define sz(v) (int) v.size()
const int N = 1e3 + 5;
struct Production {
   char lhs;
   vector<string> rhs;
} grammar[N];
int n; char null; map<char, set<char>> first, follow; map<char, int> idx;
void getFirst(int i, set<char> &cf) {
   for (auto &rule: grammar[i].rhs) {
      for (auto &c: rule) {
         if (!(c >= 'A' and c <= 'Z')) {
            cf.insert(c);
            if (c != null) break;
            continue;
         }
         set<char> cur_first;
         getFirst(idx[c], cur_first);
         for (auto &ci: cur_first) cf.insert(ci);
         if (!cur_first.count(null)) break;
      }
   }
}
```

```
void getFollow(char C, set<char> &cf) {
   if (C == grammar[0].lhs) {
      cf.insert('$');
   }
   for (int x = 0; x < n; x++) {
      for (auto &rule: grammar[x].rhs) {
         int j = 0;
         bool b = false;
         for (auto &c: rule) {
            if (c != C and !b) {
               ++j;
               continue;
            }
            if (j + 1 < sz(rule)) {
               char nxt = rule[j + 1];
               if (!(nxt >= 'A' and nxt <= 'Z')) {
                  cf.insert(nxt);
                  if (nxt != null) break;
                  cf.erase(nxt);
                  continue;
               }
               set<char> nxt_first = first[nxt];
               for (auto &ci: nxt_first) cf.insert(ci);
               if (!nxt_first.count(null)) break;
               cf.erase(null), b = true;
            } else if (grammar[x].lhs != C) {
               set<char> cur_follow;
               getFollow(grammar[x].lhs, cur_follow);
               for (auto &ci: cur_follow) cf.insert(ci);
            }
            ++j;
         }
      }
   }
}

int main() {
   null = '&';
   cout << "Enter number of productions:\n";
   cin >> n;
   for (int x = 0; x < n; x++) {
      cout << "Enter LHS:\n";
      cin >> grammar[x].lhs;
      int y;
      cout << "Enter the number of rules in RHS:\n";
      cin >> y;
      cout << "Enter RHS (each rule separated by a space)\n";
      for (int z = 0; z < y; z++) {
         string rhs;
```

```
            cin >> rhs;
            grammar[x].rhs.push_back(rhs);
        }
        idx[grammar[x].lhs] = x;
    }
    for (int i = 0; i < n; i++) {
        set<char> cf;
        getFirst(i, cf);
        first[grammar[i].lhs] = cf;
    }
    for (int i = 0; i < n; i++) {
        set<char> cf;
        getFollow(grammar[i].lhs, cf);
        follow[grammar[i].lhs] = cf;
    }
    cout << "First\n";
    for (auto &[c, st]: first) {
        cout << c << ": ";
        for (auto &ci: st) cout << ci << " ";
        cout << endl;
    }
    cout << "Follow\n";
    for (auto &[c, st]: follow) {
        cout << c << ": ";
        for (auto &ci: st) cout << ci << " ";
        cout << endl;
    }
    return 0;
}
```

Input Grammar:

S → aBDh D → EF

B → cC E → g / ∈

C → bC / ∈ F → f / ∈

Output:

```
Enter number of productions:
6
Enter LHS:
S
Enter the number of rules in RHS:
1
Enter RHS (each rule separated by a space)
aBDh
Enter LHS:
B
Enter the number of rules in RHS:
1
Enter RHS (each rule separated by a space)
cC
Enter LHS:
C
Enter the number of rules in RHS:
2
Enter RHS (each rule separated by a space)
bC &
Enter LHS:
D
Enter the number of rules in RHS:
1
Enter RHS (each rule separated by a space)
EF
```

```
Enter LHS:
E
Enter the number of rules in RHS:
2
Enter RHS (each rule separated by a space)
g &
Enter LHS:
F
Enter the number of rules in RHS:
2
Enter RHS (each rule separated by a space)
f &
First
B: c
C: & b
D: & f g
E: & g
F: & f
S: a
Follow
B: f g h
C: f g h
D: h
E: f h
F: h
S: $
```

Experiment – 9

Aim:

Write a program to construct LL(1) parsing table.

Theory:

Algorithm to construct LL(1) Parsing Table:

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

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

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

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

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

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

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

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

To construct the parsing table, we have two functions: In the table, rows will contain the Non-Terminals and the column will contain the Terminal Symbols. All the Null Productions of the Grammars will go under the Follow elements and the remaining productions will lie under the elements of the First set.

Code:

```
#include <iostream>
#include<vector>
#include<map>
#include<set>
#include<sstream>
#define outv(x)      \
   for (auto i : x) \
   cout << i << " "
using namespace std;
#define str(x) string(1,x)
map<string,vector<string>> gram;
map<string,set<char>> First;
map<string,set<char>> Follow;
```

```
map<string,int> nT;
map<char,int> T;
string start;
string generateFirst(string var){
   if(First.count(var)) return string(First[var].begin(),First[var].end());
   if(var[0]<'A' or var[0]>'Z' ) return str(var[0]);
   vector<string> prods = gram[var];
   for(int i = 0;i<prods.size();i++){
      bool isNull;
      int ind = 0;
      do{
         isNull = false;
         string first = generateFirst(str(prods[i][ind]));
         for(auto j: first){
            if(j!='#')First[var].insert(j);
            else isNull = true;
         };
         if(ind==prods[i].size()-1){
            if(isNull)First[var].insert('#');
            break;
         }
         ind++;
      }while(ind<prods[i].size() and isNull);
   }
   return string(First[var].begin(),First[var].end());
}
string generateFollow(string var){
   if(Follow.count(var))
      return string(Follow[var].begin(),Follow[var].end());
   if(var==start)Follow[var].insert('$');
   for(auto prod:gram){
      for(auto rule:prod.second){
         for(int i = 0;i<rule.size();i++){
            if(rule[i]==var[0]){
               int ind = i+1;
               bool isNull = false;
               do{
                  isNull = false;
                  string follow="";
                  if(ind==rule.size()){
                     if(prod.first!=var)
                        follow = generateFollow(prod.first);
                  }else{
                     follow = generateFirst(str(rule[ind]));
                  }
                  for(auto j:follow){
                     if(j!='#')Follow[var].insert(j);
                     else isNull = true;
                  }
```

```
                ind++;
            }while(ind<=rule.size() and isNull);
        }
    }
  }
  return string(Follow[var].begin(),Follow[var].end());
}
char getT(int x){
  for(auto p:T)
    if(p.second == x)return p.first;
}
string getnT(int x){
  for(auto p:nT)
    if(p.second == x)return p.first;
}
int main() {
  int n,curr = 0;
  cout<<"Enter the no. of productions : ";
  cin>>n;
  for(int i = 0;i<n;i++){
    string rhs, lhs, prod;
    cout<<"Enter Lhs and Rhs of Each Production(Lhs Rhs) : ";
    cin>>lhs>>rhs;
    if(i==0)start = lhs;
    nT[lhs] = i;
    stringstream ss(rhs);
    while(getline(ss,prod,'/')){
      for(auto ch:prod){
        if((ch<'A' or ch>'Z') and ch!='#')
          if(!T.count(ch))T[ch] = curr++;
      }
      gram[lhs].push_back(prod);
    }
    T['$'] = curr;
  }
  vector<vector<int>> table(nT.size(),vector<int>(T.size()));
  int seq=1;
  for(auto prod:gram){
    for(auto rule:prod.second){
      cout<<seq<<". "<<prod.first<<" -> "<<rule<<endl;
      bool isNull = true;
      if(rule[0]!='#')
      for(int i = 0;i<rule.size();i++){
        isNull = false;
        string first = generateFirst(str(rule[i]));
        for(char ch:first){
          if(ch!='#') table[nT[prod.first]][T[ch]] = seq;
          else isNull = true;
```

```
            }
            if(!isNull)break;
        }
        if(isNull){
            for(char ch:generateFollow(prod.first)){
                table[nT[prod.first]][T[ch]] = seq;
            }
        }
        seq++;
    }
  }
  cout<<"LL1 Parsing Table : \n\n";
  cout<<"\t ";
  for(int i = 0;i<T.size();i++){
    cout<<getT(i)<<"\t";
  }cout<<endl;
  cout<<"  ____________________________________\n";
  for(int i = 0;i<nT.size();i++){
    cout<<getnT(i)<<"\t|";
    for(int j:table[i])
      if(j!=0)cout<<j<<"\t";
      else cout<<" "<<"\t";
    cout<<endl;
  }
  return 0;
}
```

Output:

```
Enter the no. of productions : 5
Enter Lhs and Rhs of Each Production(Lhs Rhs) : E TG
Enter Lhs and Rhs of Each Production(Lhs Rhs) : G +TG/#
Enter Lhs and Rhs of Each Production(Lhs Rhs) : T FU
Enter Lhs and Rhs of Each Production(Lhs Rhs) : U *FU/#
Enter Lhs and Rhs of Each Production(Lhs Rhs) : F i/(E)
1. E -> TG
2. F -> i
3. F -> (E)
4. G -> +TG
5. G -> #
6. T -> FU
7. U -> *FU
8. U -> #
LL1 Parsing Table :

        +       *       i       (       )       $
   ------------------------------------------------
E       |               1       1
G       |4                                5       5
T       |               6       6
U       |8      7                         8       8
F       |               2       3
```